Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingeniería de la Computación

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Teoría de la Computación

Sobre la corrección de algoritmos de enumeración de n-adas para la enumeración de programas

T E S I S

QUE PARA OPTAR POR EL GRADO DE:
M A E S T R O E N C I E N C I A S E I N G E N I E R Í A
D E L A C O M P U T A C I Ó N

P R E S E N T A:
L U I S F E L I P E B E N Í T E Z L L U I S

Directores de tesis:

Dr. Francisco Hernández Quiroz
Facultad de Ciencias

Dr. Héctor Zenil Chávez
Oxford Immune Algorithmics

Ciudad Universitaria, Cd. Mx., Enero 2022

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Sobre la corrección de algoritmos de enumeración de n-adas para la enumeración de programas

por

Luis Felipe Benítez Lluis

Tesis presentada para optar por el grado de

Maestro en Ciencias e Ingeniería de la Computación

en el

Posgrado de Ciencias e Ingeniería de la Computación

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Universidad Nacional Autónoma de México

Ciudad Universitaria, Cd. Mx., Enero 2022

Agradecimientos

A las instituciones: Agradezco a la *Universidad Nacional Autónoma de México* por ser mi hogar durante los últimos diez años, y darme in sinfín de experiencias tanto en el campo de juego como en sus aulas y en sus espacios. Por darme trabajo y apoyos que simplificaron el proceso de estudio y de realización de este proyecto. Al *IIMAS* por albergarme en estos dos últimos años. Agradezco a *CONACYT* por el gran apoyo económico que me otorgó así como el apoyo parcial por parte del proyecto *PAPIIT* IN119920.

En la maestría: Quiero agradecer a todas las personas de la maestría que formaron parte de este capítulo en mi vida. Un agradecimiento a Eduardo, Zoé, Mario y Dani, quienes fueron excelentes compañeros de equipo. A todos los profesores y profesoras que fueron una parte importante de mi aprendizaje. Agradezco particularmente a Favio, quién fue mi maestro en más de una ocasión y sin el cual no habría logrado completar este proceso. Desde las clases de oyente que tomé, como los apuntes de materias, hasta las clases en la maestría, las enseñanzas fueron monumentales.

En el proyecto: Comienzo agradeciendo el difícil trabajo por parte del sínodo (Favio, Armando y Jorge) en la revisión de este trabajo. Agradezco a todos los miembros del proyecto en conjunto. A los doctores Quiróz y Zenil, quienes apoyaron con el asesoramiento de este proyecto. A Vladimir, Victor y Eduardo sin los cuales este proyecto no se habría concretado de forma satisfactoria. Un especial agradecimiento a Eduardo por su gran trabajo y cuya marca en este proyecto es evidente.

A la familia: A todos y todas mis amigos y amigas que han figurado en mi vida. Agradezco a toda mi familia por el apoyo que he tenido en todos estos años. A Liz que siempre estuvo conmigo durante todo el proceso y a mi madre cuyo apoyo incondicional fue indispensable para concretar este trabajo.

A todos y todas quienes no alcancé a nombrar,
¡muchas gracias!
LL11

Resumen

El proyecto desarrollado por Francisco Hernández Quiroz, Héctor Zenil y Vladimir Lemus Yáñez [1], busca aproximar la distribución universal mediante la ejecución exhaustiva de una cantidad finita de programas en un modelo de cómputo fijo. Entre las metodologías desarrolladas para la generación de programas se requiere, entre otros procedimientos, de una pareja de funciones Rank y Unrank entre n-adas de naturales y el conjunto de naturales cuyo cálculo sea rápido y n-adas de suma de entradas pequeña correspondan a naturales pequeños y de suma de entradas grandes a naturales grandes. Esta tesis se dedica a plantear y probar la corrección y complejidad en tiempo y espacio de los algoritmos planteados para enumeración de n-adas de naturales.

Palabras Clave: Funciones de emparejamiento, enumeración, Rank y Unrank de n-adas números simplex, composiciones de enteros.

Abstract

The project developed by Francisco Hernández Quiroz, Héctor Zenil and Vladimir Lemus Yáñez [1] attempts to approximate the universal distribution by exhaustively running a fixed number of programs, in a determined model of computation. Some of the methods developed for the generation of programs require, among other procedures, a pair of functions Rank and Unrank between the set of natural numbers and n-tuples that are quick to compute and such that n-tuples with small sum over their components map to small naturals and n-tuples with a large sum over their components map to large naturals. This thesis is dedicated to present and proof the correctness and calculate the time and space complexity of the algorithms stated for this enumeration of n-tuples.

Keywords: Pairing functions, enumeration, Rank and Unrank of n-tuples, simplex numbers, integer compositions.

Índice general

Introducción **1**

1. Antecedentes **5**

2. Números Simplex **19**
2.1. Números poligonales, politopo y simplex 19
2.1.1. Algoritmos para cálculo de números Simplex 25
2.2. Índice del k-simplex que antecede a un natural 27
2.2.1. Algoritmo FloorSimplex 33
2.2.2. Versiones alternativas de FloorSimplex 40

3. Enumeración de Composiciones **45**
3.1. Composiciones de enteros 45
3.1.1. Representación por barras y estrellas 47
3.1.2. Algoritmos de transformación de representaciones 50
3.2. k-descomposiciones 52
3.3. Enumeración de composiciones 58
3.3.1. Algoritmos Rank y Unrank para composiciones 61
3.3.2. Complejidades para enumeración de composiciones 65

4. Enumeración de k-tuplas **68**
4.1. Los algoritmos Rank y Unrank de k-tuplas 68
4.2. Versiones compactas para la enumeración de k-tuplas 71
4.3. Propiedades y complejidades de enumeración de k-tuplas 75
4.4. Algoritmos Rank y Unrank de tuplas multidimensionales 79
4.5. Pruebas de escritorio 85

Conclusiones **92**

A. Más pruebas de corrección y complejidad **95**
A.1. Algoritmos de cálculo de simplex 95
A.2. Versiones compactas de enumeración de composiciones 98

Índice de algoritmos **99**

Referencias **101**

Introducción

La complejidad de Kolmogorov propuesta en [2, 3] para una cadena x se define como la longitud más corta de un programa que imprime tal cadena dada una máquina universal U, esto es

$$K(x) := \text{mín}\{|p| \mid U \text{ devuelve } x \text{ al ejecutar } p\}.$$

Este concepto intenta diferenciar cadenas de naturaleza aleatoria, de aquellas cadenas a las que se les puede generar siguiendo solo una serie de reglas, un patrón dado o en otras palabras un proceso algorítmico que genere la cadena. Resulta natural pensar en patrones como programas cortos que especifican la cadena. Se puede denominar aleatorias a cadenas cuya especificación más corta sea de tamaño semejante al tamaño de cadena, y podemos denominar a cadenas como no-aleatorias cuando su especificación es de tamaño significativamente menor al tamaño de la cadena.

Podemos notar que las cadenas pueden tener múltiples programas que las representan, esto es múltiples programas que la generan. Resulta natural buscar una métrica de las cadenas para medir qué tanto se prestan a ser especificadas mediante programas de longitudes cortas. Se define la *probabilidad algorítmica* M de una cadena x como

$$M(x) := \sum_{p \in S(x)} \frac{1}{2^{|p|}} \quad \text{donde } S(x) := \{p \text{ programas} \mid \text{ la salida de } U \text{ con } p \text{ es } x\}$$

una semi-medida de probabilidad que pondera los programas p que generan la cadena x, de manera que, mientras más corto sea el programa, mayor peso aportará sobre la medida de esta cadena. De esta manera, cadenas con múltiples representaciones con programas cortos, tendrán una mayor probabilidad. Esta medida propuesta por Solomonoff [4] se conoce como la distribución universal y está íntimamente relacionada con la complejidad de Kolmogorov [5].

En [6] Solomonoff comenta nociones de la incomputabilidad de la complejidad de Kolmogorov, idea también compartida por Kolmogorov [2]. Una prueba formal de la incomputabilidad se puede consultar en [7]. A pesar de la incomputabilidad de la complejidad de Kolmogorov y por tanto de la distribución universal, es posible aproximar esta distribución para una cantidad finita de programas, como se propone en [8] (ver también [9]).

El proyecto de doctorado de Vladimir Lemus Yáñez [1] asesorado por Francisco Hernández Quiróz y Héctor Zenil, propone aproximar acumulativamente la distribución universal a través de la generación sistemática de programas cortos y la ejecución de estos en un modelo de computación fijo. Esta idea es semejante a la propuesta en [8] pero con la diferencia de usar un lenguaje de programación de alto nivel.

El modelo de computación propuesto es un lenguaje imperativo denominado IMP [10], el cual define programas a partir de asignaciones de memoria, condicionales `if`, ciclos `while`, el programa `skip`, así como secuencias de éstos, como se plantea en la siguiente gramática. Una

$$\mathbf{P} \to \texttt{skip} \mid (\mathbf{X} := \mathbf{A} \mid)(\texttt{if}\ \mathbf{B}\ \texttt{then}\ \mathbf{P}\ \texttt{else}\ \mathbf{P}) \mid (\texttt{while}\ \mathbf{B}\ \texttt{do}\ \mathbf{P}) \mid (\mathbf{P}\ ;\ \mathbf{P})$$
$$\mathbf{A} \to \mathbf{N} \mid \mathbf{X} \mid (\mathbf{A}\ +\ \mathbf{A}) \mid (\mathbf{A}\ -\ \mathbf{A}) \mid (\mathbf{A}\ \times\ \mathbf{A})$$
$$\mathbf{B} \to \texttt{true} \mid \texttt{false} \mid (\mathbf{A}\ =\ \mathbf{A}) \mid (\mathbf{A}\ <\ \mathbf{A}) \mid (\mathbf{B}\ \vee\ \mathbf{B}) \mid (\mathbf{B}\ \wedge\ \mathbf{B}) \mid \neg\mathbf{B}$$
$$\mathbf{X} \to x_{\mathbf{N}}$$
$$\mathbf{N} \to 0 \mid 1S \mid 2S \mid \ldots \mid 9S$$
$$\mathbf{S} \to \varepsilon \mid 0S \mid 1S \mid \ldots \mid 9S$$

Figura 1: Gramática del lenguaje imperativo Imp.

importante cuestión es mostrar que el algoritmo con el que se obtienen los programas que se usarán para esta aproximación genere de forma exhaustiva todos los programas para el modelo de cómputo propuesto. Dada la naturaleza recursiva del modelo de computación propuesto, se enfrenta con la tarea de generar programas más complejos a partir de programas previos.

Hay varias propuestas para realizar la tarea de generación, pero a grandes rasgos estas propuestas se basan en deducir la estructura de los programas a través de árboles de sintaxis, los cuales son codificados mediante números naturales. En este sentido, el peso más grande de cómo se obtienen estos programas yace en la forma en que codificamos las reglas recursivas de construcción de estos.

Dado que un programa se puede construir recursivamente a partir de expresiones booleanas, expresiones aritméticas y otros programas previamente construidos, surge la necesidad de codificar en un solo natural a programas constituidos por varias subestructuras. Si asumimos que cada subestructura ya fue codificada previamente se busca colapsar estos fragmentos de información en un solo natural de forma que se pueda revertir el proceso. De aquí que se busque dar funciones biyectivas Rank de $\mathbb{N}^3 \longrightarrow \mathbb{N}$, $\mathbb{N}^2 \longrightarrow \mathbb{N}$ y en general $\mathbb{N}^n \longrightarrow \mathbb{N}$.

Requerimos que estas funciones no sólo sean biyectivas, sino rápidas de calcular y más aún, de inversa rápida de calcular Unrank, ya que el objetivo primordial es generar los programas de forma ordenada. Es decir, el algoritmo Unrank servirá para decodificar un natural en un programa y así generar todos los programas. A estas funciones las llamamos enumeraciones.

Resulta indispensable que este algoritmo sea rápido y que el cálculo de un valor de este no requiera conocer el valor de programas previos. Lo anterior es necesario para obtener un buen rendimiento a la hora de hacer la aproximación a la distribución universal. En este trabajo se definirán dichas funciones y serán utilizadas en el proyecto a cargo de Eduardo Acuña Yeomans [11] para la enumeración de programas, entre otras labores.

Para el caso de parejas ordenadas, se pueden solucionar de manera razonablemente eficiente utilizando la función biyectiva propuesta por Cantor, pero el caso general precisa de una solución rápida de calcular. Se va a desarrollar una enumeración que generaliza idea de Cantor, cuya primera propuesta fue ideada por Skolem en [12]. Se desarrollará más profundamente esta idea y se probarán las propiedades correspondientes de esta generalización. Particularmente se mostrará la biyección de esta enumeración y se mostrará que esta función es estructuralmente creciente. Se entiende que una enumeración es estructuralmente creciente si la función es monótona respecto a un ordenamiento. El ordenamiento para los naturales será el usual y para las n-adas se dará uno que se inspira en el heredado por la enumeración de Cantor de parejas.

Se dedica el siguiente trabajo para desarrollar los conceptos para plantear esta enumeración así como para computarla con el objetivo de que tengan un buen desempeño tanto en espacio

como en tiempo. Además, se buscará demostrar su corrección y complejidad en tiempo y en espacio. Los algoritmos mencionados no se restringen a esta tarea y pueden ser aplicables en otros problemas.

Objetivos de este trabajo

Objetivos Generales

1. Plantear algoritmos de enumeración Rank y Unrank para parejas, ternas de naturales y en general k-tuplas de naturales, así como una enumeración de tuplas multidimensional que cumplan:

   a) El cálculo de un valor en la enumeración no dependa de valores previos en la enumeración.
   b) La enumeración sea estructuralmente creciente.
   c) Los algoritmos sean rápidos de computar.

2. Dar pruebas matemáticas de su corrección y complejidad en tiempo y espacio.

Objetivos Particulares

3. Plantear un algoritmo para el cálculo del índice de un simplex que antecede a un natural y mostrar su corrección y complejidad en tiempo y espacio.

4. Realizar pruebas de escritorio para medir el desempeño de los algoritmos.

5. Incorporar los algoritmos al sistema de enumeración de programas planteado en el proyecto de Acuña Yeomans [11].

Contenido por capítulos

El trabajo se divide en tres capítulos de contenido, así como dos secciones para los antecedentes y las conclusiones al principio y al final del escrito respectivamente.

En la sección de antecedentes se trabaja un poco más a detalle el concepto de la complejidad de Kolmogorov y distribución universal, así como los conceptos de funciones de emparejamiento que dan pie a la enumeración propuesta. También se estudian los aspectos relevantes de combinatoria algorítmica, los cuales se retoman al justificar las propiedades de la enumeración. Asimismo, se plantean las convenciones para la expresión de los algoritmos y se establece la forma en que se analizan.

En el segundo capítulo, se expresan las propiedades mínimas de combinatoria y teoría de números que se utilizarán en los algoritmos. Se definen los números simplex, se muestran sus propiedades y se plantean algoritmos para calcularlos. Se define el operador FloorSimplex y se proponen variantes para algoritmos que calculan este operador. Se muestra su corrección y analizan sus complejidades.

El tercer capítulo plantea cómo se enumeran composiciones, idea que se utiliza en la enumeración de diagonales. Se desarrollan los conceptos de representación por separadores y k-descomposición. Estos conceptos son la esencia de la enumeración de tuplas y se usan fuertemente en las pruebas de corrección de estos algoritmos. Se plantean algoritmos Rank y Unrank para composiciones, se muestra su corrección y complejidad en tiempo y espacio.

El cuarto capítulo, retoma las ideas planteadas y se enuncian algoritmos RANK y UNRANK para k-tuplas. Se muestra su corrección y complejidad en tiempo y espacio. Con base en los anteriores, se trabajan algoritmos análogos para tuplas multidimensionales. Finalmente, se enuncian resultados de pruebas de escritorio para la implementación de los algoritmos trabajados. Se puede consultar las implementaciones en [1] y [2].

Se adjunta una sección de apéndice con algunas pruebas que se delegaron y algunas versiones alternativas de los algoritmos planteados.

En [12] es el primer trabajo que propone la idea de la enumeración planteada, la cual se extiende en este trabajo. Los algoritmos diseñados en este trabajo así como varias de las propiedades demostradas son originales y a conocimiento del autor no se han planteado trabajos bajo este enfoque.

[1] `https://colab.research.google.com/drive/1s5_ydQWwm2OcDOeZ7VgXoTOgNNnjp9UF?usp=sharing`

[2] `https://colab.research.google.com/drive/1XZe2LSLXPS23qXb5WzVsn8BG-l7SRuRT?usp=sharing`

Capítulo 1

Antecedentes

Comencemos planteando la siguiente pregunta: ¿cómo será la forma más adecuada de compactar información? Para siquiera aspirar a responder esta difícil pregunta debemos aclarar lo que significa "compactar información". Podemos compactar información de muchas maneras, algunas de las cuales pueden sobrepasar procesos realistas, como dotar de un símbolo a toda posible colección de información. Si limitamos estos medios para compactar a procesos algorítmicos en un lenguaje fijo, podemos codificar este proceso en un programa cuya salida sea la información que buscamos compactar. Nótese que este programa debe carecer de entrada ya que buscamos poder recuperar la información sin dependencias.

Es evidente que la información puede venir en distintas presentaciones, pero en general podemos codificar esta información en cadenas binarias. Si elegimos una máquina de Turing universal U, y codificamos el proceso de compresión de la información en un programa p que sirva como entrada de U y codificamos la información en una cadena x, decimos que p es una descripción de x si al tomar p como la entrada de U devuelve x. En otras palabras, usamos la máquina U para realizar el proceso algorítmico con el cual recuperar x. Esto depende de la elección de U, pero la diferencia de usar otras elecciones de U no debe de dar alteraciones significativas. Siendo así, podemos reformular la pregunta a ¿cuál es la descripción p más corta de una cadena x? Dejando fijo U sabemos que debe haber tal descripción mínima. Más aún, este concepto sirve para comparar la complejidad de distintas cadenas, observando cual tiene descripción de menor tamaño. Este concepto se conoce como la *complejidad algoritmica de Kolmogorov* y fue desarrollado por personajes como Chaitin [3] y Kolmogorov [2].

$$K(x) := \text{mín}\{|p| \mid U \text{ devuelve } x \text{ al ejecutar } p\}. \tag{1.1}$$

Si observamos cadenas de información como 1111111111111111, 1010101010101010, se puede observar cierta regularidad. Estas cadenas se pueden codificar con especificaciones como *repite* 1 *16 veces*, *repite* 10 *8 veces* respectivamente. Por otro lado, cadenas del estilo 0011010100001100, 0011011010100000, al menos de forma intuitiva no muestran una regularidad clara y puede que su especificación más corta sea del estilo de *imprime* 0011010100001100. La aparente aleatoriedad de las dos últimas cadenas puede que nos complique encontrar una descripción que sea más corta que dar la cadena misma. Aquí, la noción de "aleatoriedad" es justamente el punto clave y la cual resultaría conveniente tener una definición clara. Podemos proponer una definición de cadenas aleatorias como justamente aquellas cuya descripción más corta sea aproximadamente el tamaño de la cadena misma.

De hecho, en [4], Solomonoff utiliza esta idea de cadenas que se pueden describir con programas cortos como probabilidades *a priori* en una inferencia inductiva cuando se emplea la regla de Bayes.

Si tenemos datos D y conjunto de hipótesis H (potenciales causas de los datos) buscamos saber cuáles de las hipótesis $h \in H$ hacen más probables a los datos D. Esto es, buscamos $h \in H$ que haga que la probabilidad de las hipótesis dadas los datos, en símbolos $P(h \mid D)$, sea lo más grande posible. La regla de Bayes se puede plantear como

$$P(h \mid D) = \frac{P(D \mid h)P(h)}{P(D)},$$

donde $P(D \mid h)$ representa la probabilidad de que se generen los datos dada una hipótesis h y $P(H)$ y $P(D)$ la probabilidad de una hipótesis y de los datos respectivamente. Por la regla anterior, hallar qué hipótesis hace más verosímil a los datos se reduce a encontrar qué hipótesis maximiza $P(D \mid h)P(h)$. A pesar de que el computo de $P(D \mid h)$ no suele ser complicado, calcular $P(h)$ conlleva un reto. A este valor se le suele conocer como probabilidad *a priori*[1] y representa la probabilidad inicial para algún fenómeno para el cual no siempre tenemos información de su distribución [7].

La idea de Solomonoff es proponer una distribución de probabilidad de las hipótesis que sea sensible a qué tan complejas son. Codificando esto en cadenas, se busca que cadenas que puedan ser descritas con múltiples representaciones cortas sean más probables. A esta distribución se le conoce como *distribución universal*, y se puede definir como

$$M(x) := \sum_{p \in S(x)} \frac{1}{2^{|p|}} \quad \text{donde } S(x) := \{p \text{ programas} \mid \text{ la salida de } U \text{ con } p \text{ es } x\}. \tag{1.2}$$

La complejidad de Kolmogorov de la ecuación (1.1) es incomputable, pues se pueden reducir al problema del paro [9, 7]. La distribución definida en la ecuación (1.2) tiene esta misma inconveniencia al ser incomputable como Solomonoff comenta en [5]. En la anterior referencia Solomonoff justifica que, a pesar de la incomputabilidad de la distribución universal, esta se puede aproximar. En [13], se aproxima esta distribución al enumerar y ejecutar todas las máquinas de Turing de cinco estados, adicionando las probabilidades para cada cadena que sea la salida de una de estas máquinas.

El objetivo del proyecto [1] es dar una aproximación a la distribución universal al ejecutar programas cortos. A diferencia de [13], se plantea usar como modelo de cómputo un lenguaje de alto nivel denominado Imp planteado en [10] (ver figura 1). Una de las tareas para esta aproximación consiste en la generación sistemática de programas en este lenguaje.

Generación de programas

Buscando aproximar la distribución universal con el lenguaje Imp, se necesita un procedimiento para generar todos los programas descritos por este lenguaje para posteriormente ejecutarlos y registrar su salida. Para un programa p con salida x se añade $\frac{1}{2^{|p|}}$. Entre más corto sea p, la aportación que éste ofrece es más significativa, por lo que resulta indispensable que la metodología de generar programas comience por programas cortos.

[1] A $P(h \mid D)$ se le llama probabilidad *a posteriori* y a $P(D \mid H)$ la *verosimilitud*.

Correr todos los programas en la muestra finita generada tiene el potencial de ser muy costosa computacionalmente, por el tamaño de la muestra y posibles programas que no terminen. Debido a esto, se busca que la generación sea lo más rápida y barata posible. El proyecto [1] contempla la posibilidad de repartir la gran carga de ejecutar los programas en diversos usuarios, ofreciendo una criptomoneda *automacoin*[14] como incentivo, de forma muy semejante a como se mencionó en [15]. Esto añade la necesidad adicional de que el proceso de generación sea paralelizable, de forma que este mecanismo debe de ser reproducible por distintos usuarios de forma independiente y rápida.

Particularmente, la tarea de la generación de programas en Imp es trabajada en [11]. En este proyecto no sólo se propone un protocolo para generar objetos de Imp(mediante una función $R : \text{Imp} \longrightarrow \mathbb{N}$), sino también un protocolo general de enumeración de un conjunto numerable X de objetos sintácticos descritos por alguna gramática libre de contexto (a través de $R : X \longrightarrow \mathbb{N}$). Estos protocolos cumplen con la propiedades buscadas antes mencionadas.

Demos un poco de intuición a cómo funciona este proceso de generación para un programa describiendo la función R. Observemos que un programa de Imp puede ser: el programa trivial `skip`, una asignación de la forma $X := A$, un condicional `if`, un ciclo `while`, o una secuencia de programas $(P; P)$. Todas salvo la primera comprenden una cantidad infinita de posibilidades. La idea será recorrer cada una de las posibilidades un paso a la vez , alternando entre cada una. Esto propone una correspondencia que se mapea como

$$\begin{array}{rcl}
\text{Imp} & \longrightarrow & \mathbb{N} \\
\texttt{skip} & \longmapsto & 0, \\
(\mathbf{X} := \mathbf{A}) & \longmapsto & 1, \quad 5, \quad 9, \quad \ldots \quad 4m+1, \quad \ldots \\
(\texttt{if}\ \mathbf{B}\ \texttt{then}\ \mathbf{P}\ \texttt{else}\ \mathbf{P}) & \longmapsto & 2, \quad 6, \quad 10, \quad \ldots \quad 4m+2, \quad \ldots \\
(\texttt{while}\ \mathbf{B}\ \texttt{do}\ \mathbf{P}) & \longmapsto & 3, \quad 6, \quad 11, \quad \ldots \quad 4m+3, \quad \ldots \\
(\mathbf{P}\ ;\ \mathbf{P}) & \longmapsto & 4, \quad 8, \quad 12, \quad \ldots \quad 4m+4, \quad \ldots
\end{array}$$

El algoritmo de Euclides nos garantiza que podemos descomponer cualquier natural n de forma única respecto a un divisor a como $n = am + r$. Considerando que tenemos que ofrecer un desfase $t_0 = 1$ para enumerar `skip`, se puede representar a n con un divisor a y desfase t_0 de forma única como $n = am + r + t_0$. Esto nos garantiza que si P corresponde con n identificando r sabremos si P es una asignación de una localidad, un condicional `if`, un ciclo `while` o una secuencia de programas (P_0, P_1). El valor m nos indicará cuál de los anteriores será.

Por ejemplo, si tuviésemos el programa $\mathbf{P} \equiv (X_0 := 0; X_1 := 0)$ y queremos saber qué natural n asignarle, sabremos que debe de tener la forma $n = 4m + 4$, por lo que el problema se reduce a encontrar tal m. Supongamos inductivamente que ya tenemos los valores correspondientes $R(X_0 := 0) = m_0$ y $R(X_1 := 0) = m_1$, obtener $R(\mathbf{P})$ se reduce a dar una correspondencia R_2 de la pareja de naturales en un sólo natural $R_2(m_0, m_1) = m$. Dado que buscamos que R sea rápida de calcular, invertible e independiente de valores previos, la correspondencia R_2 debe tener también estas propiedades.

El proceso anterior requirió saber inductivamente que $R(X_0 := 0) = m_0$. Resolver cómo debe de mapear R a las asignaciones requiere a su vez enumeraciones $R_{\text{loc}}(X_0) = m_{0,0}$ y $R_{\text{arit}}(0) = m_{0,1}$ de las localidades y expresiones aritméticas respectivamente, donde podemos hacer $R(X_0 := 0) = R_2\Big(R_{\text{loc}}(X_0), R_{\text{arit}}(0)\Big) = R_2(m_{0,0}, m_{0,1}) = m_0$.

Enumerar expresiones aritméticas se puede manejar de forma semejante al mapeo con los programas, salvo que con $a = 5$ y $t_0 = 0$ pues hay cinco distintas formas que una expresión aritmética puede tomar y no requiere trasladar para casos finitos. De forma similar, una expresión aritmética requiere a su vez enumeraciones de las localidades y los naturales las cuales son sencillas de definir.

Si el programa tiene la forma (`if` B_0 `then` P_0 `else` P_1), entonces requeriremos de una nueva función R_3 que tome ternas de naturales y devuelva un natural. Buscamos que esta función cumpla también con las propiedades antes mencionadas por razones semejantes a con R_2. Por supuesto, las expresiones booleanas también requerirán un mapeo que se puede construir de forma semejante, con $t_0 = 2$ y $a = 5$.

Las deducciones de estas enumeraciones se trabajan a detalle en [11] y particularmente requieren de las funciones R_2 y R_3 para poder compactar grupos de información en un sólo número que las represente. En general, se trabajan enumeraciones de estructuras sintácticas con k-hijos, cada uno con algún valor en una enumeración previamente obtenida, valores que se buscan compactar en un sólo natural. Recordando que queremos que R comience con programas cortos para llegar a los largos (a lo cual denominaremos como que R sea estructuralmente creciente) se busca que R_2, R_3 y en general R_k también lo sean.

Combinatoria Algorítmica

Como se esbozó en la sección anterior, generar los programas es en gran parte un problema de carácter combinatorio. El estudio de algoritmos que manejen diversas estructuras combinatorias es amplio y se puede encontrar mucha literatura al respecto, [16, 17, 18, 19] por mencionar algunos.

La estructura combinatoria a la que nos enfocaremos serán las parejas, ternas y en general n-adas ordenadas de naturales. Una n-ada ordenada es una función de un segmento inicial de naturales en los naturales a la que denotamos $(a_1, \ldots, a_n)$. A una n-ada también se le llama n-tupla, y convendremos usar este último término pues es más práctico cuando no queremos hacer explícito el tamaño de esta.

Como se menciona en [16], los problemas combinatorios se pueden dividir en cinco categorías: *existencia, construcción, enumeración o conteo, generación* y *optimización*. Nuestro problema cae en la categoría de *generación*, en donde se busca un procedimiento sistemático para construir todas las estructuras combinatorias de cierto tipo. Es importante señalar que la categoría de *enumeración o conteo* sólo refiere a la cantidad de estructuras combinatorias de cierto tipo sin necesariamente mostrarlas. Es claro que tener la generación implica el de conteo, pero el recíproco no es necesariamente cierto.

El orden de la generación de estructuras combinatorias $a \in \mathcal{A}$, obedece a una función inyectiva $R : \mathcal{A} \longrightarrow \mathbb{N}$ a la cual llamaremos *enumeración* (sin pérdida de generalidad la imagen de esta función es un segmento inicial de naturales o bien todos los naturales). Como se menciona en [17, 19] la generación de estructuras se suele obtener mediante dos formas.

Iterativa. A partir de una estructura inicial a_0 (la cual suele ser sencilla de obtener y cumpla $R(a_0) = 0$) y un algoritmo sucesor que recibe como entrada una estructura a que cumpla que $R(a) = n$ y devuelve la estructura a' tal que $R(a') = n + 1$.

Por *ranking*. Mediante un algoritmo que calcule la función U, la cual es inversa a R y corriéndola sobre su dominio.

Para ambos casos es deseable tener un algoritmo que compute la función R, pero no es realmente necesario. A un algoritmo para computar la función R se le suele llamar Rank. De forma semejante, un algoritmo para computar U se le suele llamar Unrank. Para el caso iterativo, calcular R es bastante directo pues se puede llevar un contador por cada vez que aplique la función sucesor.

Los algoritmos que obtienen el sucesor suelen ser muy rápidos y a menudo más fáciles de plantear. Las versiones iterativas son bastante útiles cuando se hace una búsqueda secuencial de una estructura que cumpla con alguna propiedad buscada (por ejemplo, en búsquedas a soluciones de problemas por fuerza bruta).

Una desventaja de usar las versiones iterativas es que suelen estar atadas a la secuencialidad. Esto es, para obtener la estructura a que cumpla que $R(a) = n$, se debe, ya sea hacer un recorrido de todas las estructuras previas en la enumeración (lo cual puede ser costoso en tiempo), tener almacenada una estructura previa y cercana donde retomar la enumeración (lo cual puede ser costoso en memoria) o bien tener un procedimiento para obtener el estado de memoria suficiente para retomar la ejecución secuencial (tal procedimiento es en el fondo una forma de calcular U). Esto hace que algoritmos iterativos no se presten a paralelizar pues los costos subyacentes pueden sobrepasar las ventajas de hacer cómputos paralelos.

La generación mediante un algoritmo Unrank no tiene estas desventajas pues no requiere de valores previos en la enumeración para generar la estructura, favoreciendo su capacidad de paralelizarse. Adicionalmente, esta función puede usarse de forma iterativa aplicando Unrank sucesivamente sobre su dominio e incluso puede usarse para la generación aleatoria, al sólo tener que obtener aleatoriamente un natural en su dominio [19]. Dependiendo del problema, cada llamada a Unrank puede ser un poco más costosa en tiempo que una llamada a un sucesor, por lo que es posible que en la tarea enlistar todas las estructuras, resulte más rápido la versión iterativa. A pesar de ello, las ventajas de un algoritmo Unrank son evidentes, por lo que será la aproximación empleada para este trabajo.

Enumeración de cantor

Resolver la enumeración de 1-tuplas se logra con la función identidad en una envoltura de 1-tupla. Para 2-tuplas, la propuesta de Cantor [20], comúnmente utilizada para enumerar números racionales, satisface las propiedades buscadas. En esencia, esta propuesta acomoda todas las parejas ordenadas de naturales, posicionándolas en una tabla convencional y enumera a las parejas ordenadas recorriendo exhaustivamente cada diagonal en el orden del tamaño de éstas (véase la tabla 1.1). Este recorrido de diagonales puede comenzar desde la pareja inferior izquierda para proceder hacia la pareja superior derecha o viceversa (nos mantendremos con la primera de éstas).

	0	1	2	3	4	5	6	⋯
0	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	⋯
1	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)		
2	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)			
3	(0,3)	(1,3)	(2,3)	(3,3)				
4	(0,4)	(1,4)	(2,4)					
5	(0,5)	(1,5)						
6	(0,6)							
⋮	⋮							

(a) Tabla de parejas ordenadas

	0	1	2	3	4	5	6	⋯
0	**0**	2	5	9	14	20	27	⋯
1	**1**	4	8	13	19	26		
2	**3**	7	12	18	25			
3	**6**	11	17	24				
4	**10**	16	23					
5	**15**	22						
6	**21**							
⋮	⋮							

(b) Enumeración de las parejas ordenadas

Tabla 1.1: Esquema de enumeración de parejas ordenadas

La primera observación que podemos hacer es que la primera columna de la tabla 1.1b de enumeración constituye de puros números triangulares, los cuales son números que se pueden acomodar perfectamente en forma de triángulos cuando se expresan a éstos por unidades.

Este patrón es esperable ya que las tuplas se pueden ver como unidades o fichas y al hacer cortes diagonales en el arreglo de parejas, literalmente se pintan triángulos en la enumeración. En otras palabras, el valor en la enumeración del inicio de una diagonal debe constituir la suma de los tamaños de cada diagonal previa y de aquí que el inicio tenga la forma $1+2+3+\cdots+n = \frac{n(n+1)}{2} = S_2(n)$.

La segunda observación es que las diagonales en $\mathbb{N}\times\mathbb{N}$ representan parejas ordenadas que suman siempre el mismo número, y este número es justo el número de diagonal al acomodarlas por tamaño (a este número le llamaremos *índice*). Mas formalmente, cada diagonal representa todas las composiciones en 2 partes del índice de la diagonal.

Finalmente, el orden interno de cada diagonal está dotado por la primera entrada de la pareja, como se puede observar en la tabla 1.1a.

La imagen en $\mathbb{N}$ de las diagonales representan subconjuntos finitos ajenos que empiezan siempre con un número triangular y terminan en el siguiente número triangular. Denominaremos a estos intervalos de naturales como *gnomons*[2] y definimos a su índice como el índice de la diagonal que le corresponde. Si n es el índice de un gnomon, entonces el n-ésimo número triangular es el inicio del gnomon.

En las tablas 1.2b y 1.2a vemos ejemplos para los primeros tres gnomons y diagonales respectivamente.

Para la función $R:\mathbb{N}^2\longrightarrow\mathbb{N}$, recordamos que el índice de la diagonal a la que pertenece (x,y) es $x+y$, por lo que su imagen debe pertenecer al $x+y$-ésimo gnomon. Equivalentemente, la imagen de (x,y) debe ser mayor o igual al $x+y$-ésimo número triangular, en símbolos $S_2(x+y)$. Considerando que, en el orden de las diagonales, la entrada x ofrece su colocación, se define

$$R_2(x,y) := x + S_2(x+y) = x + \frac{(x+y)(x+y+1)}{2}. \tag{1.3}$$

Para definir la función inversa $U:\mathbb{N}\longrightarrow\mathbb{N}^2$, requerimos saber a qué gnomon pertenece un natural m. Si definimos a $\lfloor\!\lfloor m\rfloor\!\rfloor_2$ el índice del gnomon al que pertenece m, sabemos que la pareja (x,y) que cumple que $R(x,y)=m$ satisface $x+y=\lfloor\!\lfloor m\rfloor\!\rfloor_2$ y también $S_2(x+y)+x=m$ por lo que podemos definir

$$U_2(m) = \Big(m - S_2(\lfloor\!\lfloor m\rfloor\!\rfloor_2),\ \lfloor\!\lfloor m\rfloor\!\rfloor_2 - m + S_2(\lfloor\!\lfloor m\rfloor\!\rfloor_2)\Big). \tag{1.4}$$

Cabe mencionar que si $n=\lfloor\!\lfloor m\rfloor\!\rfloor_2$ entonces $S_2(n)\leq m < S_2(n+1)$, es decir n es el índice del número triangular más grande que antecede a m, lo cual corresponde con el gnomon al que pertenece. En el siguiente capítulo se mostrará que $\lfloor\!\lfloor m\rfloor\!\rfloor_2 = \left\lfloor \frac{\sqrt{1+8m}-1}{2}\right\rfloor$. No es complicado mostrar que las funciones R y U definidas en (1.3) y (1.4) son inversas.

A una función biyectiva entre $\mathbb{N}^2$ y $\mathbb{N}$ se le denomina una *función de emparejamiento ("pairing function")* [22]. El encaje de Cantor de las ecuaciones (1.3) y (1.4) es un ejemplo de este tipo de funciones, pero existen otros ejemplos de funciones de empearejamientos [22]. Rosenberg plantea en [23] la siguiente función de emparejamiento así como su inversa [3]:

$$R_r(x,y) := \big(\,\text{máx}(x,y)\big)^2 + \text{máx}(x,y) + x - y, \tag{1.5}$$

[2] La intuición para este término proviene de trabajar a los números figurados, los cuales abordaremos más adelante. Se puede consultar [21] para más información.

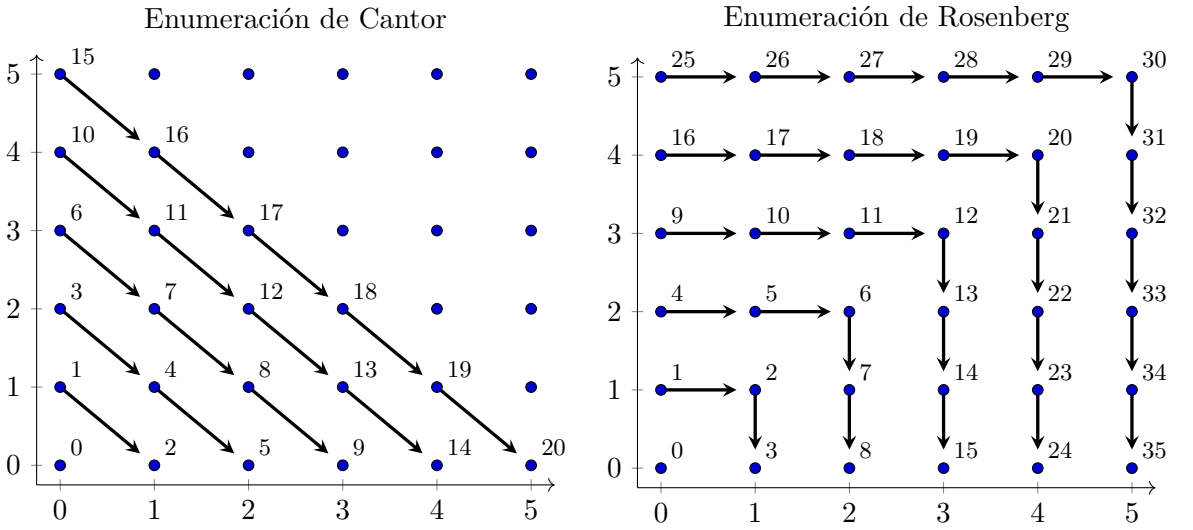
[3] Las versiones propuestas en las ecuaciones 1.5 y 1.6 corresponden a las variantes enunciadas en [22] que plantean enumeraciones sobre los naturales en lugar de sobre los enteros positivos como en [23].

$$U_r(m) = \begin{cases} \Big(m - \lfloor\sqrt{m}\rfloor^2, \lfloor\sqrt{m}\rfloor\Big) & \text{si } m - \lfloor\sqrt{m}\rfloor^2 < \lfloor\sqrt{m}\rfloor \\ \Big(\lfloor\sqrt{m}\rfloor, \lfloor\sqrt{m}\rfloor^2 + 2\lfloor\sqrt{m}\rfloor - m\Big) & \text{en otro caso.} \end{cases} \tag{1.6}$$

La prueba de que las funciones definidas en (1.5) y (1.6) son inversas se puede hacer con una doble inducción [24]. Esta enumeración tiene muchas similitudes con la de Cantor, ya que divide a las parejas de naturales en conjuntos finitos para ir enumerando exhaustivamente sobre cada uno de ellos como se puede apreciar en las figuras 1.1a y 1.1b.

Estos conjuntos finitos se les denomina *cascarones ("shells")* y cada uno de estos conjuntos representa una colección de parejas que tienen una misma medida de tamaño [23, 24]. En el caso de la enumeración de Cantor, los cascarones son las diagonales que corresponden con las parejas (x, y) cuya medida de tamaño definida por $x+y$ es la misma. Esta se asemeja a la norma ℓ_1 conocida como norma del taxista o de Manhattan. Para la enumeración de Rosenberg, los cascarones corresponden con parejas (x, y) cuya medida de tamaño es máx(x, y). Esta última se asemeja con la norma ℓ_∞ conocida norma uniforme o del supremo (de aquí que los cascarones sean cuadrados).

Análogamente a como se hizo con la enumeración de Cantor, en la enumeración de Rosenberg se pueden definir los *gnomons*, como los intervalos de $\mathbb{N}$ que se mapean a un cascarón específico. Semejante a como los gnomons para la enumeración de Cantor comienzan con un número triangular, los gnomons de la enumeración de Rosenberg comienzan con un número cuadrado.



(a) Cascarones diagonales que comienzan con números triangulares.

(b) Cascarones cuadrados que comienzan con números cuadrados.

Figura 1.1

La forma de Cantor de enumerar mediante diagonales, hace a esta enumeración estructuralmente creciente al comenzar por parejas cuya medida de tamaño es pequeña y eventualmente llegar a parejas con medida grande. En otras palabras, se cumple que para cualesquiera $(x_1, y_1), (x_2, y_2) \in \mathbb{N}^2$ si $x_1 + y_1 < x_2 + y_2$ entonces $R_2(x_1, y_1) < R_2(x_1, y_2)$.

La enumeración de Rosenberg cumple también con ser una enumeración estructuralmente creciente si modificamos nuestra noción de medida de tamaño al máximo de las entradas. Esto es, para $(x_1, y_1), (x_2, y_2) \in \mathbb{N}^2$ si máx$(x_1, y_1) <$ máx(x_2, y_2) entonces $R_r(x_1, y_1) < R_r(x_1, y_2)$.

De hecho, si planteamos otras funciones de paridad con distintos cascarones que correspondan con una medida de tamaño para las parejas, podemos a su vez tener enumeraciones de parejas estructuralmente crecientes y que cumplan las propiedades deseables (por ejemplo, enumeraciones cuyos cascarones sean diagonales o cuadrados pero que la forma de enumerar cada cascarón varíe). En [22] se mencionan algunas funciones de paridad populares con cascarones cuadrados, diagonales y otros tipos de cascarones. Para nuestros propósitos, varias de estas propuestas nos resultan igual de útiles y con el potencial de escalar bien a otras dimensiones, pero nos mantendremos con la versión de Cantor, por ser sencilla y armoniosa a intuiciones geométricas.

Usando la función R definida en la ecuación (1.3) podemos definir una función $R_3 : \mathbb{N}^3 \longrightarrow \mathbb{N}$ de la siguiente manera

$$R_3(x, y, z) := R_2\Big(R_2(x, y), z\Big) \tag{1.7}$$

Esta definición a pesar de que se puede escalar de forma muy sencilla a dimensiones mayores, induce un desbalance en el crecimiento de las entradas pues $R_2(x, y)$ puede crecer de forma cuadrática. Esto corrompe con nuestra intención de que si $x_1 + y_1 + z_1 < x_2 + y_2 + z_3$ entonces $R_3(x_1, y_1, z_1) < R_3(x_2, y_2, z_2)$. Por ejemplo, tomando $\bar{x} = (1, 1, 0)$ y $\bar{y} = (0, 0, 3)$ se tiene que $\sum \bar{x} = 2 < 3 = \sum \bar{y}$, pero $R_3(\bar{x}) = 14 > 6 = R_3(\bar{y})$.

Enseguida generalizaremos esta enumeración a tres dimensiones y posteriormente a k dimensiones basándonos en la generalización de la enumeración de Cantor planteada por Skolem en [12].

Generalización a tres dimensiones

Para generalizar la función de Cantor a tres dimensiones, comenzamos por tomar la noción de diagonal en $\mathbb{N}^3$ como las ternas que sumen un mismo valor. Es decir, la n-ésima 3-diagonal constará de todas las composiciones de n en 3 partes, lo cual dota inmediatamente de un orden total a las diagonales. Naturalmente, la enumeración recorrerá cada diagonal exhaustivamente prosiguiendo con el orden entre las diagonales. De forma muy semejante, la noción de gnomon se puede entender como los intervalos de naturales que serán la imagen de una diagonal.

$\text{Diag}_2(0)$	(0,0)			
$\text{Diag}_2(1)$	(0,1)	(1,0)		
$\text{Diag}_2(2)$	(0,2)	(1,1)	(2,0)	
$\text{Diag}_2(3)$	(0,3)	(1,2)	(2,1)	(3,0)

(a) Diagonales en $\mathbb{N}^2$

$\text{Gnom}_2(0)$	**0**			
$\text{Gnom}_2(1)$	**1**	2		
$\text{Gnom}_2(2)$	**3**	4	5	
$\text{Gnom}_2(3)$	**6**	7	8	9

(b) 2-gnomons en $\mathbb{N}$

$\text{Diag}_3(0)$	(0,0,0)											
$\text{Diag}_3(1)$	(0,0,1)	(0,1,0)	(1,0,0)									
$\text{Diag}_3(2)$	(0,0,2)	(0,1,1)	(1,0,1)	(0,2,0)	(1,1,0)	(2,0,0)						
$\text{Diag}_3(3)$	(0,0,3)	(0,1,2)	(1,0,2)	(0,2,1)	(1,1,1)	(2,0,1)	(0,3,0)	(1,2,0)	(2,1,0)	(3,0,0)		

(c) Diagonales en $\mathbb{N}^3$

$\text{Gnom}_3(0)$	**0**											
$\text{Gnom}_3(1)$	**1**	2	3									
$\text{Gnom}_3(2)$	**4**	5	6	7	8	9						
$\text{Gnom}_3(3)$	**10**	11	12	13	14	15	16	17	18	19		

(d) 3-gnomons en $\mathbb{N}$

Tabla 1.2: Esquemas de enumeración de parejas (a),(b) y ternas (c),(d)

Se puede observar que la diferencia entre los tamaños de diagonales adyacentes ya no es 1 como en caso para parejas, sino que las diagonales crecen un poco más rápido, como se puede ver

en la tabla 1.2c. De hecho, la cardinalidad de las diagonales crece como un número tetragonal, el cual se podría considerar como los números triangulares en tres dimensiones. El n-ésimo número tetragonal es $S_3(n) := \frac{n(n+1)(n+2)}{6}$ y como su nombre sugiere, estos son números que se pueden arreglar en tetraedros perfectos.

Así como los números triangulares son sucesión de un segmento inicial de números consecuentes, los números tetragonales son sucesión de un segmento inicial de números triangulares. Por ejemplo **0**=0, **1**=0+1, **4**=0+1+3, **10**=0+1+3+6, **20**=0+1+3+6+10, etcétera.

Los números simplex son la generalización a dimensión k de un número triangular. Es interesante notar que los números consecutivos (o lineales) se pueden pensar como un número triangular en una sola dimensión. Más adelante abordaremos a mayor detalle esta clase de números y sus propiedades.

Debido a que un número tetragonal es una sucesión de números triangulares, podemos seccionar a cada número tetragonal en piezas de tamaño correspondiente a un número triangular como se hace en la tabla 1.2c. Esto muestra que las diagonales en las ternas tienen el tamaño de un número triangular y por tanto las podemos partir en segmentos de tamaño $1, 2, \ldots, n$ respectivamente. Si acomodamos las 3-tuplas de esta diagonal de forma adecuada, podemos hacer que la última entrada en cada una de las ternas de estas secciones coincidan como en la tabla 1.2c. Así, en todas las secciones, las primeras dos entradas de las ternas en tales secciones codifican todas las composiciones en 2 partes del resultado de restar el índice de la diagonal con la última entrada.

Esto induce un orden en la diagonal de ternas al acomodarlas por su última entrada (de mayor a menor) y para ternas que coincidan en la última entrada, se ordenan de acuerdo con el orden de las diagonales de parejas ordenadas, para las primeras dos entradas de la terna. Para hacer más claro este proceso, se trabajará con otra representación para las ternas mediante parejas que al ordenarlas antilexicográficamente inducirá el orden buscado.

Si nos encontramos en la n-ésima 3-diagonal, sabemos que toda terna $(x, y, z) \in \mathrm{Diag}_3(n)$ es tal que $x+y+z = n$. Siendo así, conociendo n, podemos saber el valor de z ya que $n-x+y = z$. Definamos $a_1 = x$ y $a_2 = x + y$. De esta manera podemos codificar la terna (x, y, z) en la pareja $(a_1, a_2) = (x, x + y)$. Este renombramiento es muy sencillo de calcular y de revertir, ya que $x = a_1$, $y = a_2 - a_1$ y $z = n - a_2$. La razón de usar este renombramiento es que al ordenar estas parejas (a_1, a_2) en orden antilexicográfico los valores a_2 y a_1 crecen como el índice de un número triangular y lineal respectivamente ofreciendo una manera de seccionar la diagonal de forma muy natural y predecible. Es esta representación la que induce un orden, el cual se hereda a las ternas, como se puede apreciar en la tabla 1.3.

d	(a_1, a_2)	(x, y, z)	d	(a_1, a_2)	(x, y, z)	d	(a_1, a_2)	(x, y, z)
0	(0,0)	(0,0,4)	5	(2,2)	(2,0,2)	10	(0,4)	(0,4,0)
1	(0,1)	(0,1,3)	6	(0,3)	(0,3,1)	11	(1,4)	(1,3,0)
2	(1,1)	(1,0,3)	7	(1,3)	(1,2,1)	12	(2,4)	(2,2,0)
3	(0,2)	(0,2,2)	8	(2,3)	(2,1,1)	13	(3,4)	(3,1,0)
4	(1,2)	(1,1,2)	9	(3,3)	(3,0,1)	14	(4,4)	(4,0,0)

Tabla 1.3: Ordenación de $\mathrm{Diag}_3(4)$.

Los valores (a_1, a_2) correspondientes a la terna (x, y, z) codifican la representación por separadores o representación por barras y estrellas. La intuición a estas representaciones se abordarán en el tercer capítulo.

(x,y,z)	(a_1,a_2)	$\overbrace{x+y+z}^{n}$	a_1	$S_2(a_2)$	$\overbrace{a_1+S_2(a_2)}^{d}$	$S_3(n)$	$\overbrace{d+S_3(n)}^{m}$
(0,0,4)	(0,0)	4	0	0	0	20	20
(0,1,3)	(0,1)	4	0	1	1	20	21
(1,0,3)	(1,1)	4	1	1	2	20	22
(0,2,2)	(0,2)	4	0	3	3	20	23
(1,1,2)	(1,2)	4	1	3	4	20	24
(2,0,2)	(2,2)	4	2	3	5	20	25
(0,3,1)	(0,3)	4	0	6	6	20	26
(1,2,1)	(1,3)	4	1	6	7	20	27
(2,1,1)	(2,3)	4	2	6	8	20	28
(3,0,1)	(3,3)	4	3	6	9	20	29
(0,4,0)	(0,4)	4	0	10	10	20	30
(1,3,0)	(1,4)	4	1	10	11	20	31
(2,2,0)	(2,4)	4	2	10	12	20	32
(3,1,0)	(3,4)	4	3	10	13	20	33
(4,0,0)	(4,4)	4	4	10	14	20	34

Tabla 1.4: RANK de las ternas en $\mathrm{Diag}_3(4)$.

Para definir una función $R_3 : \mathbb{N}^3 \longrightarrow \mathbb{N}$, basándonos en la idea de enumerar exhaustivamente cada diagonal, sabemos que para una terna $(x,y,z) \in \mathrm{Diag}_3(n)$, la imagen $R_3(x,y,z) = m \in \mathrm{Gnom}_3(n)$, lo que significa m es mayor o igual al n-ésimo número tetragonal $S_3(n)$. De esta manera, $m = d + S_3(n)$, donde d corresponde con el valor de la enumeración de la diagonal $\mathrm{Diag}_3(n)$. Recordando que una diagonal la podemos particionar en secciones de tamaño $1, 2, \ldots, n$ respectivamente. Cuando transformamos a la terna en su representación (a_1, a_2), el valor a_2 nos dice en qué sección estamos, y por tanto cuántos valores previos a la sección debieron estar antes en la enumeración de la diagonal. Por los tamaños de cada sección, debemos tener al menos $1 + 2 + \cdots + a_2 = S_2(a_2)$ valores previos. Ahora bien, a_1 nos indica el ordinal de la terna dentro de la sección, resultando que, en la enumeración interna de la diagonal, la terna tiene valor $d = a_1 + S_2(a_2)$. Estando en la diagonal $n = x + y + z$, se debieron haber recorrido todas las diagonales previas, cuyo valor en la enumeración es al menos $S_3(n)$, es decir el valor debe ser $m = S_3(n) + d$. En la tabla 1.4 se puede ver apreciar como se secciona la diagonal y cómo cada valor a_1 y a_2 muestran la ubicación de la terna en estas secciones para $\mathrm{Diag}_3(4)$.

$$R_3(x,y,z) = S_3(x+y+z) + S_2(x+y) + x \tag{1.8}$$

Para definir la función $U_3 : \mathbb{N} \longrightarrow \mathbb{N}^3$, requerimos saber en qué 3-gnomon pertenece un natural m. Si denotamos por $\lVert m \rVert_3$ al natural n que cumple que $m \in \mathrm{Gnom}_3(n)$ sabremos que la terna (x,y,z) tal que $R_3(x,y,z) = m$ cumple que $(x,y,z) \in \mathrm{Diag}_3(n)$. Más aún, si $d = m - S_3(n)$ sabemos que d es el valor en la enumeración de la diagonal que le corresponde a (x,y,z). Recordando como se planteó R_3 dividiendo la diagonal en secciones, requerimos saber a qué sección pertenece la d-ésima terna. Dado que cada sección tiene un elemento más que la anterior, saber a qué sección pertenece d, se reduce a ver a cuál 2-gnomon pertenece. Con el índice del gnomon $\lVert d \rVert_2$ podemos saber en qué sección se encuentra y así, el valor $d - S_2(\lVert d \rVert_2)$ refleja cual elemento en la sección se refiere.

Estos valores refieren a la representación por separadores (a_1, a_2) con la cual se puede reconstruir la terna (x,y,z) dado que conocemos n. Podemos ver este proceso en la tabla 1.5.

m	$\overbrace{\lVert m \rVert_3}^{n}$	$S_3(n)$	$\overbrace{m - S_3(n)}^{d}$	$\overbrace{\lVert d \rVert_2}^{a_2}$	$S_2(\lVert d \rVert_2)$	$\overbrace{d - S_2(\lVert d \rVert_2)}^{a_1}$	(a_1, a_2)	(x, y, z)
20	4	20	0	0	0	0	(0,0)	(0,0,4)
21	4	20	1	1	1	0	(0,1)	(0,1,3)
22	4	20	2	1	1	1	(1,1)	(1,0,3)
23	4	20	3	2	3	0	(0,2)	(0,2,2)
24	4	20	4	2	3	1	(1,2)	(1,1,2)
25	4	20	5	2	3	2	(2,2)	(2,0,2)
26	4	20	6	3	6	0	(0,3)	(0,3,1)
27	4	20	7	3	6	1	(1,3)	(1,2,1)
28	4	20	8	3	6	2	(2,3)	(2,1,1)
29	4	20	9	3	6	3	(3,3)	(3,0,1)
30	4	20	10	4	10	0	(0,4)	(0,4,0)
31	4	20	11	4	10	1	(1,4)	(1,3,0)
32	4	20	12	4	10	2	(2,4)	(2,2,0)
33	4	20	13	4	10	3	(3,4)	(3,1,0)
34	4	20	14	4	10	4	(4,4)	(4,0,0)

Tabla 1.5: UNRANK de las ternas en $\text{Diag}_3(4)$.

Siendo así, podemos definir la función U como

$$U_3(m) := \Big(d - S_2(\lVert d \rVert_2),\, \lVert d \rVert_2 - \big(d - S_2(\lVert d \rVert_2)\big),\, \lVert m \rVert_3 - \lVert d \rVert_2\Big), \tag{1.9}$$

donde $d = m - S_3(\lVert m \rVert_3)$. Se va a mostrar en el siguiente capítulo que

$$\lVert m \rVert_3 = \left\lfloor \sqrt[3]{3m + \sqrt[2]{9m^2 - 3^{-3}}} + \sqrt[3]{3m - \sqrt[2]{9m^2 - 3^{-3}}} - 1 \right\rfloor. \tag{1.10}$$

Mostrar que las funciones R_3 y U_3 definidas en (1.8) y (1.9) son inversas puede ser algo engorroso, pero intuitivamente este hecho es claro por construcción. Generalizar las funciones R y U a k-tuplas requerirá plantar con más detalle conceptos como números simplex y sus índices. Particularmente obtener el índice semejante a como en (1.10) puede no ser viable para dimensiones grandes.

La utilidad de la representación por separadores y sus propiedades se generalizarán y se mostrará como marcan el camino dentro de una diagonal para encontrar una k-tupla dentro de la enumeración. Este concepto se denominó como k-descomposición (ver [22, 25, 26, 27]) y es la clave para definir la función UNRANK la cual es la de mayor interés.

Modelo de Computación

Debido a que este escrito se centra en el planteamiento de algoritmos y las pruebas de su corrección, es relevante plantear aclaraciones sobre cómo se enunciarán los mismos y el comportamiento esperado. Todos los algortimos planteados estarán enunciados en *pseudocódigo* que se asemeja a muchos lenguajes de programación modernos. La convención de usar *pseudocódigo* abunda en la literatura que aborda el estudio de los algoritmos (por ejemplo, en [28, 29, 30, 25], por mencionar algunos) pues ofrece una forma adecuada de definir el comportamiento del algoritmo. Como se menciona en [29], se busca la manera más clara y concisa de especificar un algoritmo, y por esta razón se omiten detalles de modularidad y manejo de errores, entre otras cuestiones que le conciernen a la ingeniería de software.

El *pseudocódigo* que se usará es muy parecido a la especificación en [29]. Se utilizarán estructuras como el bloque if-else, ciclos for, while y loop, etcétera. Estas estructuras se delimitarán con las palabras reservadas antes mencionadas como inicio y la variante End correspondiente, así como un sangrado adecuado. A continuación, enunciamos los puntos claves en la interpretación que se da al *pseudocódigo* en este escrito.

- La conjunción y disyunción se denotan con los símbolos $\wedge$ y $\vee$ respectivamente y el cálculo de su valor emplea corto circuito como es usual. La negación se denota con $\neg$ o bien $\neq$ para el caso de la negación de la igualdad.
- El ciclo for itera el bloque con un contador que se inicializa en el valor provisto y cuya última iteración se realiza cuando el contador es exactamente el límite de detención dado. Se usa la palabra reservada upto cuando el contador incremente en una unidad y la palabra reservada downto cuando el contador decrementa en una unidad. Si el límite de detención es menor (mayor) estrictamente que el valor de iniciación para el contador incremental (decremental) el bloque no se ejecuta y si este límite es exactamente igual, sólo se ejecuta una vez.
- El ciclo while se ejecuta mientras la guardia tenga valor True y finaliza cuando esta guardia tenga valor False al inicio de una iteración o bien, se ejecute ya sea el comando return o break.
- El ciclo loop se comporta igual al ciclo while con la salvedad de que carece de una guardia antes de cada iteración, por lo que precisa de uno de los comandos previamente mencionados para evitar caer en un ciclo infinito.
- La asignación de un valor a en una variable v se escribe como $v \leftarrow a$. Permitimos múltiples asignaciones en una sola línea, por ejemplo $v_1, v_2 \leftarrow a_1, a_2$ asigna a v_1 el valor a_1 y v_2 el valor a_2.
- El comando return finaliza el procedimento y devuelve el valor provisto. Permitimos devolver múltiples valores los cuales se podrán desempacar con una asignación múltiple. Por ejemplo, si $\textsc{Foo}(x)$ devuelve dos valores, podemos hacer la asignación $a, b \leftarrow \textsc{Foo}(x)$ para recuperar estos valores.
- Los parámetros se pasan por valor, de forma que si se llama una función con parámetro x, se usa una copia de este valor.
- En los algoritmos planteados se usarán arreglos bidimensionales para almacenar valores numéricos los cuales llamaremos *tablas*. Estos arreglos poseerán dos atributos que reflejan sus dimensiones en cantidad de renglones y cantidad de columnas. Por ejemplo, si A es una tabla, el atributo correspondiente a la cantidad de renglones se obtienen con la notación usual de punto $A.rowLen$. De forma análoga $A.colLen$, corresponde con la cantidad de columnas. La numeración de los elementos en la tabla se empieza en 0 como es usual, por lo que el valor que se encuentra primer renglón y primera columna se denota como $A[0, 0]$ y si $i = A.rowLen - 1$ y $j = A.colLen - 1$ el elemento en el último renglón y última columna será $A[i, j]$. Para los propósitos de este trabajo, no se requieren muchas operaciones sobre esta estructura más allá del proceso de asignación y consulta de valores.
- Dado que varios de los algoritmos que se presentan trabajarán con tuplas de naturales, se harán explícitos los valores de la tupla mediante la notación $(x_1, x_2, x_3, \ldots, x_k)$, la cual

podrá ser parte de los parámetros del algoritmo o bien la salida de este. El valor k que corresponde con la longitud de la tupla será un atributo accesible cuando se tenga a la tupla como parámetro. Dado que los valores de la tupla se hacen explícitos, por simplicidad se podrá omitir la notación punto para acceder a la longitud de la tupla y usaremos el valor k dentro del algoritmo.

Durante las pruebas de corrección será común enunciar propiedades respecto a las variables. Dado que estas variables pueden cambiar su estado dentro de un ciclo, algunas de las propiedades enunciadas dependerán del número de iteración del ciclo. En la demostración se distinguirá el estado de una variable al terminar una iteración y se denotará mediante un subíndice que indique el número de iteración. Por ejemplo, si la variable a yace dentro de un ciclo, definiremos a_i como el estado de la variable a al terminar la i-ésima iteración, y también definiremos a_0 como el estado de a previo a entrar al ciclo. En esta forma, muchas propiedades se podrán enunciar de forma precisa al estado de las variables. Si una variable tiene más de un estado durante una iteración de algún ciclo, se podrán usar super-índices entre paréntesis para enumerar cada estado que ésta haya tenido. No obstante, las notaciones usadas para distinguir los estados de una variable en las pruebas de corrección se harán explícitos previo a su uso para evitar confusión.

Modelo RAM

Como es común en la literatura de análisis de algoritmos, usaremos como modelo de computación la *máquina de acceso aleatorio ("random acces machine") RAM*. Este modelo logra un balance adecuado al ser suficientemente sencillo para trabajar con él y suficientemente robusto para dar resultados significativos que se asemejen a aplicaciones verdaderas en computadoras. Otra ventaja que ofrece es que permite que el análisis del desempeño de un algoritmo sea independiente a la computadora donde se ejecute [28].

Este modelo mide el tiempo de ejecución de un algoritmo mediante la cantidad de pasos que ejecuta y ofrece una variedad de operaciones aritméticas, de manipulación de información y de control usualmente encontradas en computadoras modernas las cuales se asume que toman un paso en realizar. Algunas de las operaciones que se tienen son las siguientes.

Aritméticas. Las operaciones suma, resta, producto, división (entera y de punto flotante), módulo, piso y techo toman un paso.

Booleanas. Verificar igualdad o relación de orden (reflexiva y estricta) toman un paso, así como resolver una conjunción o disyunción binaria.

Manejo de datos. La creación y asignaciones de variables, toman un paso y no se distingue entre memoria caché o en disco.

Dar una cantidad exacta de pasos que un algoritmo va a tomar para una entrada arbitraria, puede ser una tarea bastante complicada. Como se menciona en [28], entender con absoluta precisión la cantidad de pasos, implica manejar detalles poco relevantes respecto al algoritmo y que no aportan información significativa adicional. Se analizarán las complejidades de los algoritmos de la forma usual mediante análisis asintótico del escenario que toma mayor tiempo de ejecución, ofreciendo así una cota superior al tiempo de ejecución. Para esto tomaremos la usual notación asintótica con la "O grande". A continuación, definimos la clase de funciones $O(f(n))$ cuyo manejo y propiedades son las usuales a las que se presentan en la literatura (por ejemplo, ver [28, 29, 30])

Definición 1.0.1. Sea $f : \mathbb{N} \longrightarrow \mathbb{N}$. Definimos la clase de funciones $O(f(n))$ como las funciones $g : \mathbb{N} \longrightarrow \mathbb{N}$ tales que existen constantes $c, N \in \mathbb{N}$ de forma que para toda $n > N$ se cumpla que $g(n) \leq cf(n)$.

Esta noción simplifica lo suficiente la cantidad de pasos de un algoritmo logrando encapsular la esencia de su complejidad en tiempo, además de permitir hacer comparaciones adecuadas entre distintos algoritmos.

Capítulo 2

Números Simplex

En este capítulo definiremos los números k-simplex así como demostrar las propiedades que usaremos de éstos para los algoritmos de enumeración de tuplas. Plantearemos también el algoritmo FloorSimplex que resuelve la tarea de hallar el índice de un simplex que antecede a un natural dado.

2.1. Números poligonales, politopo y simplex

Los números poligonales son números naturales que se pueden arreglar en forma de algún polígono dado cuando se expresa al número como fichas o puntos, donde cada ficha vale una unidad. Se tienen registros del estudio de estos números desde los tiempos de la escuela pitagórica en el siglo VI A.C., quienes estaban en busca de relacionar la aritmética con la geometría [21]. Su estudio yace de generalizar números que se pueden arreglar en forma de un triángulo o cuadrado a aquellos que se puedan arreglar en forma de cualquier polígono regular (números poligonales) para incluso en poliedros regulares de tres dimensiones (números poliédricos) o figuras regulares de mayores dimensiones (números politopo).

La forma usual para obtener un número poligonal a partir de uno anterior es extender dos aristas adyacentes en un punto, y agregar los lados correspondientes, cada uno con la cantidad de puntos establecida al hacer esta extensión. Ejemplos de esto se pueden ver en la figura 2.1. Siguiendo esta pauta se puede concluir que para cualquier polígono regular hay una infinidad de números que se arreglen en tal forma.

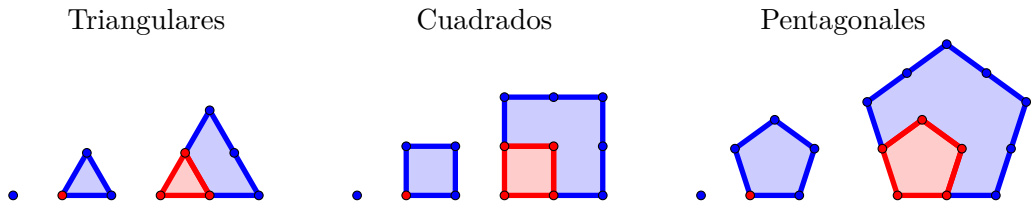


Figura 2.1: Representación geométrica de los primeros tres números poligonales

En otras palabras, para generar un número poligonal, se precisa un número poligonal previamente obtenido y una cantidad de puntos adecuada que añadiremos para obtener el siguiente

número poligonal. A esta cantidad se le denomina *gnomon* aludiendo a su cualidad geométrica. En geometría, un *gnomon* de una figura dada, es otra figura que, al agregarla a ésta, se obtiene una figura semejante a la inicial (ver figura 2.2). El *gnomon* de un cuadrilátero se parece a un reloj de sol, que de hecho es el significado de esta palabra.

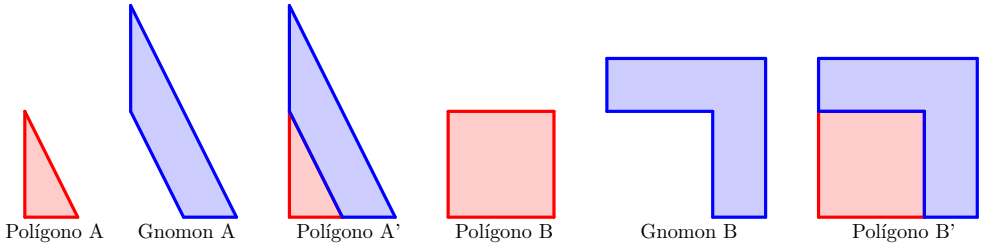


Figura 2.2: Concepto geométrico de Gnomon

Los números politopos son la generalización de los números poligonales a mayores dimensiones que, de forma semejante, se generan a través de un número politopo anterior y un gnomon.

Los números triangulares son los números poligonales en los que nos enfocaremos. Consecuentemente, nos enfocaremos en los números politopo correspondientes a esta figura. La figura denominada "simplex" es la generalización de un triángulo a varias dimensiones. Por ejemplo, un 1-simplex es un simple segmento, un 2-simplex es un triangulo equilátero, 3-simplex un tetraedro, 4-simplex un pentácoro, etcétera. Naturalmente los números simplex son números que se pueden arreglar en forma de las figuras del mismo nombre.

Al ser estos números politopo, comparten la cualidad de que el $n+1$-ésimo número $k+1$-simplex resulta del n-ésimo número $k+1$-simplex más un *gnomon*. Algo que los destaca es que este *gnomon* resulta ser el $n+1$-ésimo número k-simplex como se puede apreciar en la tabla 2.3.

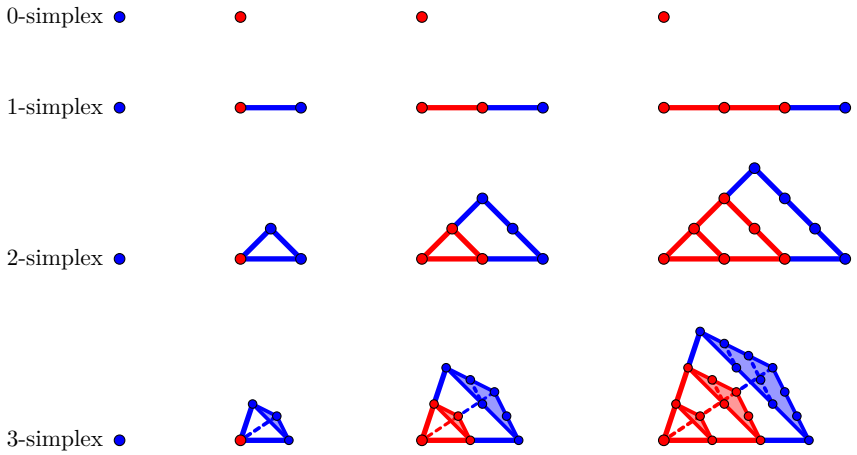


Figura 2.3: Representación geométrica de números simplex. La sección azul representa el gnomon.

Esta cualidad sobre su gnomon describe una propiedad de recurrencia que se puede utilizar para calcular de forma rápida estos números simplex mediante técnicas de memorización. Más aún, esta propiedad es compartida en la forma en que crece la cantidad de composiciones de un natural, que como se verá más adelante, será una estructura combinatoria que se emplee para el algoritmo de enumeración.

Los números triangulares surgen frecuentemente en diversos problemas combinatorios, por ejemplo, en la conocida suma de Gauss $\sum_{i\leq n} i = \frac{n(n+1)}{2}$. Este sencillo teorema describe una fórmula para obtener un número triangular sin necesidad de computar todas las sumas además de exhibir la propiedad de que todo simplex es suma de un segmento inicial de simplex de dimensión menor. Esta fórmula se generaliza para mayores dimensiones y será exactamente como definamos un número k-simplex. Naturalmente, la propiedad de suma de Gauss se va a heredar a raíz de esta definición.

Definición 2.1.1. Sea $x \in \mathbb{R}$. Definimos recursivamente sobre $\mathbb{N}$ el *factorial ascendente* $x^{\bar{k}}$ como $0^{\bar{0}} := 0$, $x^{\bar{0}} := 1$ para $x \neq 0$ y $x^{\overline{k+1}} := (x+k)\cdot x^{\bar{k}}$. De forma semejante, definimos el *factorial descendente* $x^{\underline{k}}$ como: $0^{\underline{0}} := 0$, $x^{\underline{0}} := 1$ para $x \neq 0$ y $x^{\overline{k+1}} := (x-k)\cdot x^{\bar{k}}$.

Sea $n \in \mathbb{N}$. Definimos recursivamente el *factorial* de n como $0! = 1$ y $(n+1)! = (n+1)n!$.

Observación 2.1.2. No es difícil ver que $x^{\bar{1}} = x = x^{\underline{1}}$, $1^{\bar{k}} = k! = k^{\underline{k}}$, $x^{\bar{k}} = \prod_{i=0}^{k-1}(x+i)$ y $x^{\underline{k}} = \prod_{i=0}^{k-1}(x-i)$.

Definición 2.1.3. Sean $n, k \in \mathbb{N}$. Se define el n-ésimo número *k-simplex* $S_k(n)$ como:

$$S_k(n) := \frac{n^{\bar{k}}}{k!}, \text{ donde } n^{\bar{k}} \text{ es el factorial ascendente.}$$

Diremos que n es el índice de $S_k(n)$ así como k su dimensión. Si $k > 0$ es fácil ver que

$$S_k(n) = \frac{n(n+1)(n+2)\cdots(n+(k-1))}{k!}.$$

Con esta definición podemos computar los números simplex como se aprecian en la tabla 2.1, además se puede observar que estos números no son nada más que coeficientes binomiales acomodados en una forma distinta su presentación usual mediante el triángulo de Pascal. Las diagonales de esta tabla, de hecho, corresponden con los renglones de este triángulo. La ventaja primordial por la cual estudiamos los números simplex en lugar de conformarnos con su equivalente como coeficientes binomiales, es que ofrecen una intuición más sencilla y clara al problema combinatorio.

		n										
	$S_k(n)$	0	1	2	3	4	5	6	7	8	9	10
k	0	0	1	1	1	1	1	1	1	1	1	1
	1	0	1	2	3	4	5	6	7	8	9	10
	2	0	1	3	6	10	15	21	28	36	45	55
	3	0	1	4	10	20	35	56	84	120	165	220
	4	0	1	5	15	35	70	126	210	330	495	715
	5	0	1	6	21	56	126	252	462	792	1287	2002

Tabla 2.1: Primeros 10 números simplex de dimensión 0-5

Por lo anterior, toda propiedad de un número simplex, tiene su homóloga como coeficiente binomial, pero a lo largo de este escrito se optará por la perspectiva del número simplex. Mostremos a continuación la similitud entre un coeficiente binomial y un número simplex.

Definición 2.1.4. Sean $n, m \in \mathbb{N}$ con $m \leq n$. Definimos el coeficiente binomial $\binom{n}{m} := \frac{n!}{m!(n-m)!}$.

Observación 2.1.5. Las siguientes propiedades son directas de la definición.

1. Para cualquier $n \in \mathbb{N}$ pasa $\binom{n}{0} = \binom{n}{n} = 1$.
2. Para cualesquiera $i, n \in \mathbb{N}$ con $i \leq n$ pasa que $\binom{n}{i} = \binom{n}{n-i}$.
3. Para todo k, n se cumple que $S_k(n+1) = \binom{k+n}{n} = \binom{k+n}{k} = S_n(k+1)$.
4. $S_0(0) = 0$ y $S_0(n) = 1$ si $n \neq 0$.
5. Para todo n se tiene que $S_1(n) = n$.
6. Para todo k se tiene que $S_k(0) = 0$, $S_k(1) = 1$ y $S_k(2) = k + 1$.

Proposición 2.1.6 (Pascal)**.** *Sea $n \geq 1$. Para cualquier $i \in \mathbb{N}$ que cumpla que $0 \leq i < n$ se satisface*

$$\binom{n}{i+1} + \binom{n}{i} = \binom{n+1}{i+1}.$$

Demostración. Tenemos que:

$$\begin{aligned}
\binom{n}{i+1} + \binom{n}{i} &= \frac{n!}{(i+1)!(n-i-1)!} + \frac{n!}{i!(n-i)!} &&= \frac{n!(n-i) + n!(i+1)}{(i+1)!(n-i)!} \\
&= \frac{n!(n-i+1+i)}{(i+1)!(n-i)!} &&= \frac{n!(n+1)}{(i+1)!(n-i)!} \\
&= \frac{(n+1)!}{(i+1)!((n+1)-(i+1))!} &&= \binom{n+1}{i+1}
\end{aligned}$$

QED

Lema 2.1.7. *Para cualesquiera $n, k \in \mathbb{N}$ con $m \leq n$ se cumple que $\binom{n}{k} \in \mathbb{N}$ y en particular $S_k(n) \in \mathbb{N}$.*

Demostración. Procedamos por inducción sobre n. Si $n = 0$ entonces $k = 0$ y entonces $\binom{0}{0} = 1 \in \mathbb{N}$. Supongamos que para toda $m \leq n$ se cumple que $\binom{n}{k} \in \mathbb{N}$, veamos que se cumple para $n + 1$.

Si $k = 0$ o bien $k = n + 1$, entonces $\binom{n+1}{k} = 1 \in \mathbb{N}$. Si $0 < k < n + 1$, entonces $k = m + 1$ para alguna m y $m < n$. Por la proposición 2.1.6 se sigue que $\binom{n+1}{m+1} = \binom{n}{m+1} + \binom{n}{m}$ y por hipótesis de inducción $\binom{n}{m+1} \in \mathbb{N}$ y $\binom{n}{m} \in \mathbb{N}$, por lo que $\binom{n+1}{m+1} \in \mathbb{N}$.

Finalmente, si $n = 0$ entonces $S_k(n) = 0 \in \mathbb{N}$. Si $n = \hat{n} + 1$ para alguna $\hat{n} \in \mathbb{N}$, entonces $S_k(n) = S_k(\hat{n} + 1) = \binom{k+\hat{n}}{k} \in \mathbb{N}$. QED

Como consecuencia del teorema de Pascal 2.1.6, se tiene el siguiente corolario. Esta propiedad se puede ver como una propiedad de recurrencia la cual se puede explotar para hacer cálculos más rápidos de un número simplex provisto que se hayan calculado el inmediato antecesor tanto en índice como en dimensión. Esta propiedad se puede ver muy claramente en la representación geométrica de los simplex que se mencionó en la figura 2.3.

Corolario 2.1.8 (Propiedad de Pascal)**.** *Para cualquier $k, n \in \mathbb{N}$ se cumple que $S_{k+1}(n+1) = S_{k+1}(n) + S_k(n+1)$ y por tanto $S_{k+1}(n+1) - S_{k+1}(n) = S_k(n+1)$*

Demostración. La demostración es resultado directo de la observación 2.1.5 y el teorema de Pascal 2.1.6. QED

La siguiente propiedad es una generalización de la suma Gauss. Tanto esta propiedad como la propiedad de Pascal pueden usarse para definir a los números simplex y resultan en definiciones equivalentes. Se optó por su representación combinatoria mediante factoriales ascendentes para mayor simplicidad. A lo largo del escrito usaremos las propiedades de Gauss y Pascal frecuentemente.

Proposición 2.1.9 (Suma de Gauss)**.** *Para cualesquiera $n, k \in \mathbb{N}$ se cumple que $S_{k+1}(n) = \sum_{i=0}^{n} S_k(i)$.*

Demostración. Mostremos este resultado por inducción sobre n. Para $n = 0$ se tiene que por la observación 2.1.5 $S_{k+1}(0) = 0 = \sum_{i=0}^{0} 0 = \sum_{i=0}^{0} S_k(0)$. Supongamos que el resultado se tiene para n y veamos que se sostiene para $n+1$.

Por el corolario 2.1.8 se tiene que

$$S_{k+1}(n+1) = S_{k+1}(n) + S_k(n+1) \overset{\text{HI}}{=} \Big(\sum_{i=0}^{n} S_k(i)\Big) + S_k(n+1) = \sum_{i=0}^{n+1} S_k(i).$$

QED

Proposición 2.1.10 (Suma Telescópica)**.** *Para cualesquiera $n, k \in \mathbb{N}$ con $k \geq 1$ se cumple que*

$$S_k(n+1) = \Big(\sum_{i=1}^{k} S_i(n)\Big) + 1.$$

Demostración. Procedamos por inducción sobre k. Para $k = 1$, por la observación 2.1.5 se sigue $S_1(n+1) = n + 1 = S_1(n) + 1$. Supongamos que el resultado se cumple para k y veamos que se cumple para $k+1$. Empleando el corolario 2.1.8 se tiene que

$$S_{k+1}(n+1) = S_{k+1}(n) + S_k(n+1) \overset{\text{HI}}{=} S_{k+1}(n) + \Big(\sum_{i=1}^{k} S_i(n)\Big) + 1 = \Big(\sum_{i=1}^{k+1} S_i(n)\Big) + 1.$$

Esto último era lo buscado.

QED

Mostremos que los simplex son monótonos respecto a su índice y también su dimensión. Para esto requeriremos un par de resultados.

Proposición 2.1.11. *Sean $x, y, r_1, \ldots, r_k \in \mathbb{R}$ tales que $0 \leq x, y, r_1, \ldots, r_k$. Se siguen las siguientes:*

1. *$x \cdot \prod_{i=1}^{k}(x + r_i) < y \cdot \prod_{i=1}^{k}(y + r_i)$ si y sólo si $x < y$,*
2. *$x \cdot \prod_{i=1}^{k}(x + r_i) \leq y \cdot \prod_{i=1}^{k}(y + r_i)$ si y sólo si $x \leq y$.*

Demostración. **(1)** El regreso es sencillo pues dado que $0 \leq x, y$, se tiene entonces $0 \leq (x + r_i) < (y + r_i)$ para toda $i \in \{1, \ldots, k\}$, y por tanto $x \cdot \prod_{i=1}^{k}(x + r_i) < y \cdot \prod_{i=1}^{k}(y + r_i)$. Para la ida, si $x = 0$ es resultado es trivial, por lo que tomemos $0 < x < y$. Observemos que

$$x \cdot \Big(\prod_{i=1}^{k}(x + r_i)\Big) = x \cdot \Big(\sum_{i=0}^{k} a_i x^i\Big) = \sum_{i=0}^{k} a_i x^{i+1} \qquad \text{y también}$$

$$y \cdot \Big(\prod_{i=1}^{k}(y + r_i)\Big) = y \cdot \Big(\sum_{i=0}^{k} a_i y^i\Big) = \sum_{i=0}^{k} a_i y^{i+1}, \qquad \text{donde } a_i \geq 0 \text{ y } a_k = 1.$$

Tenemos que $x \cdot \prod_{i=1}^{k}(x+r_i) < y \cdot \prod_{i=1}^{k}(y+r_i)$, lo que implica que $0 < y \cdot \prod_{i=1}^{k}(y+r_i) - x \cdot \prod_{i=1}^{k}(x+r_i)$. De esto último tenemos que

$$\begin{aligned} y \cdot \prod_{i=1}^{k}(y+r_i) - x \cdot \prod_{i=1}^{k}(x+r_i) = & \sum_{i=0}^{k} a_i y^{i+1} - \sum_{i=0}^{k} a_i x^{i+1} = \sum_{i=0}^{k} a_i (y^{i+1} - x^{i+1}) \\ = & \sum_{i=0}^{k} a_i \Big((y-x) \sum_{j=0}^{i} x^j y^{i-j} \Big) = (y-x) \sum_{i=0}^{k} \sum_{j=0}^{i} a_i x^j y^{i-j}. \end{aligned}$$

Esto es, $0 < (y-x) \sum_{i=0}^{k} \sum_{j=0}^{i} a_i x^j y^{i-j}$. Como $a_k = 1$ y $0 < x < y$ entonces $\sum_{j=0}^{i} a_i x^j y^{i-j} > 0$ y así necesariamente $0 < y - x$ y por tanto $x < y$.

(2) Esta demostración es completamente análoga a la del inciso anterior.

QED

Corolario 2.1.12. *Si $x, y \in \mathbb{R}$ y $k \in \mathbb{N}$ con $0 \leq x, y$ y $k \geq 1$, entonces*

1. *$x^{\bar{k}} < y^{\bar{k}}$ si y sólo si $x < y$,*
2. *$x^{\bar{k}} \leq y^{\bar{k}}$ si y sólo si $x \leq y$.*

Demostración. Resultado directo de la proposición 2.1.11 tomando $r_i = i$ con $i \in \{0, \ldots, k-1\}$. QED

Proposición 2.1.13. *Para cualesquiera $n, m, k, i, j \in \mathbb{N}$ se siguen las siguientes:*

1. *$n < m$ si y sólo si $S_k(n) < S_k(m)$,*
2. *$n = m$ si y sólo si $S_k(n) = S_k(m)$,*
3. *$n \leq m$ si y sólo si $S_k(n) \leq S_k(m)$,*
4. *$i < j$ y $n > 1$ si y sólo si $S_i(n) < S_j(n)$,*
5. *si $i < j$, entonces $S_i(n) \leq S_j(n)$,*
6. *si $i < j$ y $n < m$, entonces $S_i(n) < S_j(m)$, y*
7. *si $S_i(n) < S_j(m)$, entonces $i < j$ o $n < m$.*

Demostración. **(1)** Por el corolario 2.1.12 se tiene que

$$n < m \quad \text{si y sólo si} \quad n^{\bar{k}} < m^{\bar{k}} \quad \text{si y sólo si} \quad \frac{n^{\bar{k}}}{k!} < \frac{m^{\bar{k}}}{k!} \quad \text{si y sólo si} \quad S_k(n) < S_k(m).$$

(2) La ida es evidente. El regreso se puede probar por contrapuesta. Si $n \neq m$, entonces, por tricotomía y sin pérdida de generalidad tomemos $n < m$. Por el inciso (1) se tiene que $S_k(n) < S_k(m)$ lo que implica que $S_k(n) \neq S_k(m)$.

(3) Resultado directo de los incisos (1) y (2).

(4) Supongamos primero que $i < j$ y $n > 1$. Se tiene que existe $m \in \mathbb{N}$ tal que $n = m + 1$. Por la observación 2.1.5 (2) y el inciso (1) se siguen las siguientes equivalencias:

$$\begin{array}{ccccc} S_i(n) < S_j(n) \text{ si y sólo si} & S_i(m+1) < S_j(m+1) & \text{si y sólo si} & S_m(i+1) < S_m(j+1) \\ \text{si y sólo si} & i+1 < j+1 & \text{si y sólo si} & i < j \end{array}$$

Así, $i < j$ implica que $S_i(n) < S_j(n)$. Para la conversa supongamos que $S_i(n) < S_j(n)$. Tiene que ser el caso de que $n > 1$ pues si tomamos $n = 0$ o bien $n = 1$ por la observación 2.1.5(5) se sigue que $S_i(n) = S_j(n)$. Siendo así, por las equivalencias planteadas previamente se sigue que $i < j$.

(5) Como visto anteriormente, si $n = 0$ o $n = 1$ se tiene $S_i(n) = S_j(n)$. Si $n > 1$ por el inciso (4) se sigue que $S_i(n) \leq S_j(n)$.

(6) De los incisos (1) y (5) se sigue $S_i(n) < S_i(m) \leq S_j(m)$ y por tanto $S_i(n) < S_j(m)$.

(7) Por contrapuesta utilizando los incisos (3), (5) y (6). QED

Proposición 2.1.14. *Para cualesquiera $m, k \in \mathbb{N}$ con $k > 0$ sucede $m \leq S_k(m)$.*

Demostración. Si $m = 0$, es resultado es claro. Tomando $m \geq 1$, para toda $i \in \{1, \ldots, k-1\}$ se cumple que $i + 1 \leq m + i$, es decir $1 \leq \frac{m+i}{i+1}$. Multiplicando sobre i se sigue que

$$\prod_{i=1}^{k-1} \frac{m+i}{i+1} \geq 1 \quad \text{y por tanto} \quad m \cdot \prod_{i=1}^{k-1} \frac{m+i}{i+1} = \frac{m^{\bar{k}}}{k!} \geq m.$$

QED

2.1.1. Algoritmos para cálculo de números Simplex

El siguiente algoritmo toma dos naturales n y k y devuelve el n-ésimo k-simplex $S_k(n)$. Este algoritmo es bastante directo al computar $\frac{n(n+1)\cdots(n+k-1)}{k!}$. Resulta importante destacar que hay distintos órdenes en que se puede hacer el cálculo ya sea, por ejemplo, calculando el numerador y el denominador de la fracción de forma separada, para proceder a dividir los valores obtenidos. Si bien esto es correcto, el numerador y denominador crecen muy rápido y para números grandes puede acarrear problemas pues hacer la división entre ellos puede traer errores de redondeo en implementaciones en algún lenguaje.

Se opta por multiplicar paulatinamente cada factor del numerador y posteriormente hacer la división para asegurar no tener que laborar con enteros innecesariamente grandes. Un último detalle es que la división se hace con la operación de división entera, la cual suele ser más eficiente y permite laborar con números de tipo entero sin recurrir a flotantes. Hacer divisiones mediante divisiones enteras puede ser riesgoso si no se garantiza que los valores a operar tengan residuo cero. Afortunadamente, para este procedimiento será este el caso. No obstante, si no se requieren calcular números muy grandes estas pequeñas optimizaciones pueden ser despreciables.

Algoritmo 1 SIMPLEX(n, k)

Entrada: Una pareja de números $n, k \in \mathbb{N}$.
Salida: El valor $S_k(n)$.

```
1: if n = 0 ∨ n = 1 then
2:     return n
3: end if
4: A ← 1
5: for i = 0 upto k − 1 do          ▷ Si k = 0 no se entra en ciclo y se regresa 1.
6:     A ← A · (n + i)
7:     A ← A//(i + 1)               ▷ División entera de A entre i + 1.
8: end for
9: return A
```

El algoritmo anterior es bastante claro y las siguientes proposiciones referentes a su corrección y complejidad en tiempo y espacio son evidentes. No obstante, las pruebas de las proposiciones 2.1.15 y 2.1.16 se pueden consultar en *Dem.*2.1.15 y *Dem.*2.1.16 respectivamente.

Proposición 2.1.15. *Si la entrada al algoritmo* Simplex *(1) es* (n, k)*, la salida es el número* $S_k(n)$*.*

Proposición 2.1.16. *La complejidad en tiempo del algoritmo* Simplex(n, k) *(1) es* $O(k)$ *y la complejidad en espacio es* $O(1)$*.*

Ahora bien, en el proceso de Rank y Unrank de tuplas se requiere calcular varios números simplex en múltiples dimensiones. Para optimizar este proceso a costa de un poco de memoria podemos tener precomputados una cantidad de números simplex. Si estos números yacen en esta tabla, podemos ahorrarnos una cantidad significativa de cálculos. Más adelante veremos que para la enumeración planteada podemos saber qué tan grande tiene que ser nuestra tabla para garantizar la ejecución de Unrank o Rank. Esta versión será práctica si se espera que se hagan múltiples llamadas a Unrank o Rank.

El proceso para rellenar esta tabla es de hecho muy rápido y numéricamente estable ya que se explotará el corolario 2.1.8 obteniendo el simplex $S_{k+1}(n+1)$ mediante simplex previamente calculados $S_k(n+1)$ y $S_{k+1}(n)$.

Algoritmo 2 Tabla-Simplex(k, n)

Entrada: Dimensiones del arreglo $k, n \in \mathbb{Z}^+$.
Salida: Arreglo TS de $k \times n$ tal que $TS[i, j] = S_i(j)$.

```
1: Incializa el arreglo TS de k × n en cero.
2: for j = 1 upto n − 1 do
3:     TS[0, j] ← 1                          ▷ Asigna primer renglón de simplex
4: end for
5: for i = 0 upto k − 2 do                   ▷ Iteración sobre renglones
6:     for j = 0 upto n − 2 do               ▷ Iteración sobre columnas
7:         TS[i + 1, j + 1] ← TS[i, j + 1] + TS[i + 1, j]   ▷ Propiedad de Pascal
8:     end for
9: end for
10: return TS
```

De nuevo, el algoritmo anterior es bastante directo por lo que delegamos las pruebas de la corrección y complejidad en tiempo y espacio al apéndice. Las pruebas de las proposiciones 2.1.17 y 2.1.18 se pueden consultar en *Dem:* 2.1.17 y *Dem:* 2.1.18 respectivamente.

Proposición 2.1.17. *Sean* $k, n \in \mathbb{Z}^+$ *y* Tabla-Simplex$(k, n) = TS$*. Si* $a < k$ *y* $b < n$ *entonces* $TS[a, b] = S_a(b)$*.*

Proposición 2.1.18. *Si* $k, n \in \mathbb{Z}^+$*, el algoritmo* Tabla-Simplex(k, n) *(2) tiene complejidad en tiempo y en espacio* $O(kn)$ *(contando la salida).*

Esta tabla puede ser útil para encontrar el índice del k-simplex que antecede a un natural m, provisto que este simplex se encuentre en la tabla. Bastará con hacer búsqueda binaria en el renglón correspondiente a la dimensión del simplex que se busca. Esta idea es de hecho como funcionará la versión del algoritmo FloorSimplex que trabaja con la tabla de simplex como se verá en la siguiente sección.

2.2. Índice del k-simplex que antecede a un natural

El algoritmo Unrank de tuplas requerirá conocer el índice del k-simplex que antecede a un natural m arbitrario pues esta tarea es la piedra angular de su funcionamiento. En esta sección mostraremos varios algoritmos para obtener este índice para $k > 0$ además de mostrarse las soluciones exactas para el caso de $k = 1, 2, 3$ y 4. El que este algoritmo sea rápido será lo que haga que Unrank sea rápido.

Notación 2.2.1. Sea $m, k \in \mathbb{N}$ con $k > 0$. Denotemos el índice del k-simplex que antecede a m como $\lfloor\!\lfloor m \rfloor\!\rfloor_k$. Esto es, $\lfloor\!\lfloor m \rfloor\!\rfloor_k := \max\{n \in \mathbb{N} \mid S_k(n) \leq m\}$. Al operador $\lfloor\!\lfloor _ \rfloor\!\rfloor_k$ lo llamaremos también FloorSimplex.

Observación 2.2.2. Para cualquier $m, k \in \mathbb{N}$ con $k > 0$, si $A = \{n \in \mathbb{N} \mid S_k(n) \leq m\}$, entonces $0 \in A$, por lo que $A \neq \varnothing$. Más aún, por la proposición 2.1.14 sabemos que para toda $n \in A$ se sigue que $n \leq m$, por lo que el conjunto A tiene máximo al ser un conjunto de naturales no vacío y acotado superiormente. Esto muestra que el operador FloorSimplex está bien definido al ser el elemento máximo único.

De la definición de FloorSimplex se sigue que si $\lfloor\!\lfloor m \rfloor\!\rfloor_k = n$ entonces $S_k(n) \leq m < S_k(n+1)$.

		m									
	$\lfloor\!\lfloor m \rfloor\!\rfloor_k$	0	1	2	3	4	5	6	7	8	9	10
k	1	0	1	2	3	4	5	6	7	8	9	10
	2	0	1	1	2	2	2	3	3	3	3	4
	3	0	1	1	1	2	2	2	2	2	2	3
	4	0	1	1	1	1	2	2	2	2	2	2
	5	0	1	1	1	1	1	2	2	2	2	2
	6	0	1	1	1	1	1	1	2	2	2	2

Tabla 2.2: FloorSimplex de dimensión 1-6 para los primeros 10 naturales

Proposición 2.2.3. *Para toda $m, n, k, i, j \in \mathbb{N}$ con $k, i, j > 0$ se cumplen las siguientes.*

1. *Si $n < m$, entonces $\lfloor\!\lfloor n \rfloor\!\rfloor_k \leq \lfloor\!\lfloor m \rfloor\!\rfloor_k$.*
2. *Si $\lfloor\!\lfloor n \rfloor\!\rfloor_k < \lfloor\!\lfloor m \rfloor\!\rfloor_k$, entonces $n < m$.*
3. *Si $j < i$, entonces $\lfloor\!\lfloor m \rfloor\!\rfloor_i \leq \lfloor\!\lfloor m \rfloor\!\rfloor_j$.*
4. *Si $\lfloor\!\lfloor m \rfloor\!\rfloor_i < \lfloor\!\lfloor m \rfloor\!\rfloor_j$, entonces $j < i$.*
5. *Si $n < m$ y $j < i$, entonces $\lfloor\!\lfloor n \rfloor\!\rfloor_i \leq \lfloor\!\lfloor m \rfloor\!\rfloor_j$.*

Demostración. **(1)** Sean $A_n := \{x \in \mathbb{N} \mid S_k(x) \leq n\}$ y $A_m = \{x \in \mathbb{N} \mid S_k(x) \leq m\}$. Supongamos primero que $n < m$, entonces $A_n \subseteq A_m$ y por tanto $\max A_n \leq \max A_m$, es decir $\lfloor\!\lfloor n \rfloor\!\rfloor_k \leq \lfloor\!\lfloor m \rfloor\!\rfloor_k$.

(2) Resolvamos equivalentemente por contrapuesta. Por tricotomía basta mostrar que si $n \leq m$, entonces $\lfloor\!\lfloor n \rfloor\!\rfloor_k \leq \lfloor\!\lfloor m \rfloor\!\rfloor_k$. Si $n = m$ es resultado es directo. Si $n < m$ el resultado se sigue del inciso (1).

(3) Sea $a := \lfloor\!\lfloor m \rfloor\!\rfloor_i$ y $b := \lfloor\!\lfloor m \rfloor\!\rfloor_j$. Supongamos que $j < i$. Por definición $S_i(a) \leq m < S_i(a+1)$ y $S_j(b) \leq m < S_j(b+1)$ y supongamos por contradicción que $b < a$, entonces $b + 1 \leq a$, y por la proposición 2.1.13 (3), se tiene que $m < S_j(b+1) \leq S_i(a) \leq m$. Esto último implica $m < m$, que es una contradicción.

(4) Mostremos esto por contrapuesta. Por tricotomía basta mostrar que si $j \leq i$, entonces $\llbracket m \rrbracket_i \leq \llbracket m \rrbracket_j$. Si $j = i$ el resultado es directo. Si $j < j$ por el inciso (3) se sigue el resultado.
(5) Por los incisos (1) y (3) se sigue que $\llbracket n \rrbracket_i \leq \llbracket m \rrbracket_i \leq \llbracket m \rrbracket_j$. QED

Si $m, n \in \mathbb{N}$ son tales que $S_k(n) \leq m < S_k(n+1)$, entonces $\frac{n^{\bar{k}}}{k!} \leq m < \frac{(n+1)^{\bar{k}}}{k!}$. El objetivo es poner a n en términos de m.

Para $k = 1$, esta desigualdad no es complicada de resolver. En efecto, dado que $\frac{n^{\bar{1}}}{1!} = n$ para toda n, se sigue que

$$n \leq m < n+1 \quad \text{lo que implica} \quad n = m \quad \text{y así} \quad \llbracket m \rrbracket_1 = m.$$

Para $k = 2$ tenemos que $\frac{n^{\bar{k}}}{k!} = \frac{n(n+1)}{2}$ y por tanto de la primera desigualdad se sigue

$$\begin{aligned} \frac{n(n+1)}{2} &\leq m \\ n(n+1) - 2m &\leq 0 \\ n^2 + n - 2m &\leq 0 \qquad (2.1) \\ (n - r_1)(n - r_2) &\leq 0 \qquad (2.2) \end{aligned}$$

donde r_1, r_2 son las raíces de la ecuación en (2.1). Resolviendo la ecuación de segundo grado en (2.1) es fácil obtener los valores $r_1 = \frac{-1+\sqrt{1+8m}}{2}$ y $r_2 = \frac{-1-\sqrt{1+8m}}{2}$. Para que el producto en (2.2) sea menor a cero hay dos casos: que o bien $(n - r_1) \geq 0$ y $(n - r_2) \leq 0$, o bien $(n - r_1) \leq 0$ y $(n - r_2) \geq 0$. Dado que n, m son enteros positivos, sólo el segundo de estos casos es posible. Si nos enfocamos en la raíz positiva r_1 podemos notar que

$$n - r_1 = n - \frac{-1 + \sqrt{1+8m}}{2} \leq 0, \quad \text{esto es,} \quad n \leq \frac{-1 + \sqrt{1+8m}}{2}. \qquad (2.3)$$

Esto nos dice que la raíz r_1 es una cota de n. De hecho, es una buena cota, en el sentido de que se cumple $r_1 < n+1$. Con lo anterior, gracias al lema enunciado enseguida podemos calcular n en términos de m como

$$n = \left\lfloor \frac{-1 + \sqrt{1+8m}}{2} \right\rfloor.$$

Lema 2.2.4. *Para cualesquiera $x \in \mathbb{R}$ y $n \in \mathbb{Z}$ se cumplen las siguientes.*

1. *$\lfloor x \rfloor = n$ si y sólo si $n \leq x < n+1$.*
2. *$n + \lfloor x \rfloor = \lfloor n + x \rfloor$.*
3. *Si $\lfloor x \rfloor = n$, entonces existe $p \in [0, 1)$ tal que $n + p = x$.*

Demostración. **(1)** Para la ida supongamos que $n = \max\{m \in \mathbb{Z} \mid n \leq x\}$. Por definición ocurre que $n \leq x$. No puede ser el caso de que $n + 1 \leq x$ ya que n es el número máximo con esta propiedad.

Para el regreso, supongamos que $n \leq x < n+1$. Sea $m = \lfloor x \rfloor$, entonces por ser máximo se cumple que $n \leq m \leq x$. No puede ser el caso de que $n < m$, ya que se ser así, $n + 1 \leq m$ lo que implicaría $x < n + 1 \leq m$ lo que es una contradicción. Por tanto $n = m = \lfloor x \rfloor$.

(2) Sea $a \in \mathbb{Z}$ tal que $a = \lfloor x \rfloor$. Del inciso (1) se sigue que $a \leq x < a + 1$. Sumando n se obtiene $n + a \leq n + x < n + a + 1$, por lo que $n + a = \lfloor n + x \rfloor$, es decir $n + \lfloor x \rfloor = \lfloor n + x \rfloor$.

(3) Sea $p = x - n$ y por tanto $x = n + p$. Por definición $n \leq x < n + 1$ por lo que restando n se tiene $0 \leq p < 1$, así $p \in [0, 1)$.

QED

Como se pudo apreciar en el procedimiento anterior, resolver la desigualdad se reduce a factorizar un polinomio adecuado. Definamos el polinomio $p_k^m(x) = \frac{x^{\bar{k}}}{k!} - m$. Observemos que $p_k^m(0) = -m < 0$ y claramente $p_k^m(m) > 0$ por la proposición 2.1.14. Al ser p_k^m continuo, por el teorema del valor intermedio, hay una raíz de p_k^m en el intervalo $[0, m]$.

Si desarrollamos este polinomio podremos ver que la expresión correspondiente a $\frac{x^{\bar{k}}}{k!}$ consta de puros coeficientes positivos además de no tener coeficiente constante. Siendo así el coeficiente constante de p_k^m es $-m$, lo que hace que este polinomio sólo tenga un cambio de signo en sus coeficientes. La regla de cambio signos de Descartes nos garantiza que la diferencia entre la cantidad de cambios de signo entre los términos no cero de un polinomio y la cantidad de raíces positivas es par además de que la primera cantidad acota superiormente a la segunda. Particularmente, si hay exactamente un cambio de signo entonces hay exactamente una raíz positiva (estos teoremas se pueden ver en [31]).

Por lo tanto, sabemos que p_k^m tiene exactamente una raíz positiva $r(m)$. Este valor debe de cumplir que $\frac{r(m)^{\bar{k}}}{k!} = m$. Nótese que es posible que este valor no sea un número natural (caso que ocurre cuando m no es un k-simplex).

Proposición 2.2.5. *Sean $m, k \in \mathbb{N}$ con $k > 0$ y $r(m)$ la raíz del polinomio p_k^m, entonces $\lfloor r(m) \rfloor = \llbracket m \rrbracket_k$.*

Demostración. Sea $n = \llbracket m \rrbracket_k$, se sigue que

$$\frac{n^{\bar{k}}}{k!} \leq \frac{r(m)^{\bar{k}}}{k!} < \frac{(n+1)^{\bar{k}}}{k!} \qquad \text{lo que equivale a} \qquad n^{\bar{k}} \leq r(m)^{\bar{k}} < (n+1)^{\bar{k}}.$$

Aplicando el corolario 2.1.12 se tiene que $n \leq r(m) < n + 1$, por lo que por el lema 2.2.4 $\lfloor r(m) \rfloor = n = \llbracket m \rrbracket_k$. QED

En general, encontrar la raíz del polinomio p_k^m puede ser una tarea compleja y el teorema de Abel-Ruffini nos dice que para polinomios de grado mayor o igual a 5 no existe una solución general por radicales. No obstante, dado que buscamos el mayor entero menor a ésta, bastará con tener una aproximación adecuada a la raíz para resolver la tarea.

Los polinomios p_k^m tienen segunda derivada positiva en $\mathbb{R}^+$ y por tanto son convexos, lo que hace que sean unas funciones muy amenas a distintos algoritmos de aproximación de raíces para valores m no demasiado grandes.

Sin embargo, los polinomios p_k^m crecen muy rápido en coeficientes a lo largo que crece k. Considerando que los algoritmos clásicos de aproximación de raíces manejan números de tipo flotante durante su ejecución, para valores de k grandes, la estabilidad numérica se ve afectada considerablemente. Considerando que sólo se necesita de un entero, se plantea un algoritmo que labore con puras operaciones de tipo entero.

Ayuda tener las raíces exactas para los polinomios p_k^m con los cuales tenemos soluciones exactas para $\llbracket m \rrbracket_k$. A continuación enunciamos estas soluciones para $k = 1, 2, 3$ y 4. Resulta de interés tener más soluciones exactas para dimensiones mayores para estos polinomios, pero puede ser una labor considerablemente complicada. Estas soluciones se obtuvieron mediante las soluciones generales a los polinomios de grado 1,2,3 y 4.

Proposición 2.2.6. *Para toda $m \in \mathbb{N}$ se siguen las siguientes:*

1. $[\![m]\!]_1 = m$,
2. $[\![m]\!]_2 = \left\lfloor \frac{\sqrt{1+8m}-1}{2} \right\rfloor$,
3. $[\![m]\!]_3 = \left\lfloor \sqrt[3]{3m + \sqrt[2]{9m^2 - \frac{1}{27}}} + \sqrt[3]{3m - \sqrt[2]{9m^2 - \frac{1}{27}}} - 1 \right\rfloor$,
4. $[\![m]\!]_4 = \left\lfloor \frac{\sqrt{5+4\sqrt{24m+1}}-3}{2} \right\rfloor$.

Demostración. Por la proposición 2.2.5 basta mostrar que las expresiones en los incisos 1,2,3 y 4 son las raíces de los polinomios p_1^m, p_2^m, p_3^m y p_4^m respectivamente.

(1) En efecto se tiene que $p_1^m(x) = x - m$ que tiene m como raíz claramente.

(2) Sea $r := \frac{-1+\sqrt{1+8m}}{2}$ mostremos que $r(r+1) = 2m$. En efecto,

$$\begin{aligned}\left(\frac{\sqrt{1+8m}-1}{2}\right)\left(\frac{\sqrt{1+8m}-1}{2}+1\right) &= \left(\frac{\sqrt{1+8m}-1}{2}\right)\left(\frac{\sqrt{1+8m}+1}{2}\right)\\ &= \frac{(1+8m)-1}{4} = \frac{8m}{4} = 2m\end{aligned}$$

Así $p_2^m(r) = \frac{r(r+1)}{2} - m = \frac{2m}{2} - m = 0$.

(3) Sea $r := \sqrt[3]{3m + \sqrt[2]{9m^2 - 3^{-3}}} + \sqrt[3]{3m - \sqrt[2]{9m^2 - 3^{-3}}} - 1$. Veamos $r(r+1)(r+2) = 6m$. Sean $a = 3m$, $b = \sqrt[2]{9m^2 - 3^{-3}}$, $c = \sqrt[3]{a+b}$ y $d = \sqrt[3]{a-b}$. Entonces $r = c + d - 1$. Tenemos que

$$\begin{aligned}r(r+1)(r+2) &= (c+d-1)(c+d)(c+d+1)\\ &= \left((c+d)^2 - 1\right)(c+d)\\ &= \left[c^2 + 2cd + d^2 - 1\right](c+d)\\ &= c^3 + 3c^2d + 3cd^2 + d^3 - c - d\\ &= (c^3 + d^3) + [3c^2d + 3cd^2] - c - d\\ &= a + b + a - b + [3c^2d + 3cd^2] - c - d\\ &= 2a + [3c^2d + 3cd^2] - c - d \qquad (2.4)\end{aligned}$$

Ahora bien, tenemos que

$$3c^2d = 3\sqrt[3]{(a+b)^2(a-b)} = 3\sqrt[3]{(a^2-b^2)(a+b)}$$

De donde $a^2 - b^2 = 9m^2 - 9m^2 + 3^{-3} = 3^{-3}$, por lo que

$$3\sqrt[3]{(a^2-b^2)(a+b)} = 3\sqrt[3]{3^{-3}(a+b)} = \sqrt[3]{(a+b)} = c$$

De forma muy parecida tenemos que

$$3cd^2 = 3\sqrt[3]{(a+b)(a-b)^2} = 3\sqrt[3]{(a^2-b^2)(a-b)}$$

Reescribiendo $a^2 - b^2$,

$$3\sqrt[3]{(a^2-b^2)(a-b)} = 3\sqrt[3]{3^{-3}(a-b)} = \sqrt[3]{(a-b)} = d$$

Reescribiendo esto en (2.4),

$$r(r+1)(r+2) = 2a + [c+d] - c - d = 2a = 2(3m) = 6m$$

Así $p_3^m(r) = \frac{r(r+1)(r+2)}{6} - m = \frac{6m}{6} - m = 0$.

(4) Sea $r := \frac{\sqrt{5+4\sqrt{24m+1}}-3}{2}$ y $a := \sqrt{24m+1}$. Mostremos que $r(r+1)(r+2)(r+3) = 24m$.

$$r+1 = \frac{\sqrt{5+4\sqrt{24m+1}}-1}{2}, \quad r+2 = \frac{\sqrt{5+4\sqrt{24m+1}}+1}{2}, \quad r+3 = \frac{\sqrt{5+4\sqrt{24m+1}}+3}{2}$$

Tenemos que

$$r(r+3) = \frac{\sqrt{5+4a}-3}{2} \cdot \frac{\sqrt{5+4a}+3}{2} = \frac{1}{4}(5+4a-9) = \frac{1}{4}(4a-4) = a-1$$
$$(r+1)(r+2) = \frac{\sqrt{5+4a}-1}{2} \cdot \frac{\sqrt{5+4a}+1}{2} = \frac{1}{4}(5+4a-1) = \frac{1}{4}(4a+4) = a+1$$

Multiplicando estas dos últimas se tiene

$$r(r+1)(r+2)(r+3) = (a-1)(a+1) = a^2 - 1 = (24m+1) - 1 = 24m.$$

Así $p_4^m(r) = \frac{r(r+1)(r+2)(r+3)}{24} - m = \frac{24m}{24} - m = 0$. QED

Como mencionado antes, no se puede garantizar que haya soluciones por radicales a los polinomios p_k^m, pero uno puede intuir que estas funciones se han de comportar como una raíz k-ésima. La siguiente proposición da cotas que justo muestran este crecimiento en FloorSimplex.

Siendo que todo coeficiente binomial es un número simplex, estas cotas también lo son para números binomiales. Particularmente son útiles para hacer una búsqueda binaria. Sin embargo, calcular raíces k-ésimas puede ser una desventaja por el tiempo que puede tomar este cálculo además de dar pie a estabilidad numérica para números grandes. Más adelante trabajaremos con algoritmos que usen estas cotas y otros que no.

Proposición 2.2.7. *Para toda $m, k \in \mathbb{N}$ con $k > 0$ se cumple que*

$$\sqrt[k]{m} - k \leq \sqrt[k]{k!m} - k \leq \lVert m \rVert_k \leq \sqrt[k]{k!m} \leq k\sqrt[k]{m}.$$

Demostración. Si $m = 0$ el resultado es claro, así que tomemos $m > 0$. Sea $n = \lVert m \rVert_k$, entonces por definición se cumple que $\frac{n^{\bar{k}}}{k!} \leq m < \frac{(n+1)^{\bar{k}}}{k!}$. Como $m > 0$ entonces $n > 0$, es decir $n \geq 1$. De esta manera $n + j \geq j + 1$ para toda $j \in \{0, \ldots, k-1\}$. Por lo tanto $\frac{n+1}{j+1} \geq 1$ para toda $j \in \{0, \ldots, k-1\}$. Esto último implica

$$\prod_{j=0}^{k-1} \frac{n+j}{j+1} = \frac{n^{\bar{k}}}{k!} \leq m \quad \text{y entonces} \quad \frac{n+j}{j+1} \leq m \text{ para toda } j \in \{0, \ldots, k-1\}.$$

Se tiene que $n \leq n+j$ por lo que $\frac{n}{j+1} \leq \frac{n+j}{j+1}$ para toda $j \in \{0,\ldots,k-1\}$. De lo anterior se sigue

$$\prod_{j=0}^{k-1} \frac{n}{j+1} = \frac{n^k}{k!} \leq \prod_{j=0}^{k-1} \frac{n+j}{j+1} = \frac{n^{\bar{k}}}{k!} \leq m \quad \text{así} \quad \sqrt[k]{\frac{n^k}{k!}} = \frac{n}{\sqrt[k]{k!}} \leq \sqrt[k]{m}. \tag{2.5}$$

Por otro lado, $\frac{n+k}{j} \geq \frac{j(n+j)}{j}$ para toda $j \in \{1,\ldots,k\}$. Se sigue que

$$\frac{(n+k)^k}{k!} = \prod_{j=1}^{k} \frac{n+k}{j} \geq \prod_{j=1}^{k} \frac{n+j}{j} = \frac{(n+1)^{\bar{k}}}{k!} \geq m \quad \text{que implica}$$

$$\sqrt[k]{\frac{(n+k)^k}{k!}} = \frac{n+k}{\sqrt[k]{k!}} \geq \sqrt[k]{m}. \tag{2.6}$$

Despejando n de las ecuaciones (2.5) y (2.6) se obtiene $\sqrt[k]{k!m} - k \leq n \leq \sqrt[k]{k!m}$, que equivale a $\sqrt[k]{k!m} - k \leq \lfloor\!\lfloor m \rfloor\!\rfloor_k \leq \sqrt[k]{k!m}$. Finalmente, es claro que $k! \leq k^k$ de donde $\sqrt[k]{k!} \leq \sqrt[k]{k^k} = k$ por lo que se sigue $\sqrt[k]{m} - k \leq \sqrt[k]{k!m} - k$ y $\sqrt[k]{k!m} \leq k\sqrt[k]{m}$, que es lo que se quería demostrar. QED

Observación 2.2.8. Por la proposición 2.2.7, el comportamiento de la función $\lfloor\!\lfloor m \rfloor\!\rfloor_k$ tiene una descripción un poco más amena a las funciones que estamos acostumbrados. Esto es, $O(\lfloor\!\lfloor m \rfloor\!\rfloor_k) \subseteq O(\sqrt[k]{k!m}) \subseteq O(k\sqrt[k]{m})$. Particularmente sucede que $O\big(\log(\lfloor\!\lfloor m \rfloor\!\rfloor_k)\big) \subseteq O\big(\log(\sqrt[k]{k!m})\big) \subseteq O\big(\log(k\sqrt[k]{m})\big)$, lo cual será de utilidad durante el cálculo de la complejidad del algoritmo FLOOR-SIMPLEX, que se verá en la siguiente sección.

Proposición 2.2.9. *Se tienen los siguientes límites.*

1. *Para toda $k \in \mathbb{N}$ con $k > 0$,* $\lim_{m\to\infty} \frac{\lfloor\!\lfloor m \rfloor\!\rfloor_k}{\sqrt[k]{m}} = \sqrt[k]{k!}$.
2. *Para toda $m \in \mathbb{N}$ con $m > 0$,* $\lim_{k\to\infty} \frac{\lfloor\!\lfloor m \rfloor\!\rfloor_k}{\sqrt[k]{m}} = 1$.

Demostración. **(1)** Primero observemos que $\sqrt[k]{m} \to \infty$ cuando $m \to \infty$, por lo que $\frac{1}{\sqrt[k]{m}} \to 0$ cuando $m \to \infty$. Es decir, tenemos que $\lim_{m\to\infty} \frac{1}{\sqrt[k]{m}} = 0$.

Ahora bien, por la proposición 2.2.7, para toda $m \in \mathbb{N}$ se sigue que

$$\frac{\lfloor\!\lfloor m \rfloor\!\rfloor_k}{\sqrt[k]{m}} \leq \frac{\sqrt[k]{k!}\sqrt[k]{m}}{\sqrt[k]{m}} = \sqrt[k]{k!} \quad \text{lo que implica} \quad \lim_{m\to\infty} \frac{\lfloor\!\lfloor m \rfloor\!\rfloor_k}{\sqrt[k]{m}} \leq \lim_{m\to\infty} \sqrt[k]{k!} = \sqrt[k]{k!}. \tag{2.7}$$

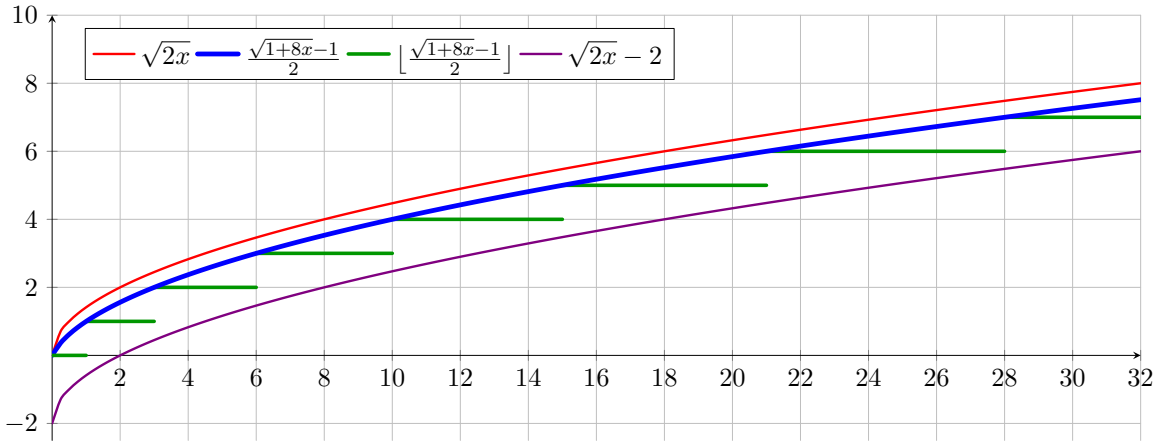
Por otro lado, de nuevo de la proposición 2.2.7, para toda $m \in \mathbb{N}$ se sigue que

$$\begin{array}{ccccccc} \frac{\lfloor\!\lfloor m \rfloor\!\rfloor_k}{\sqrt[k]{m}} & \geq & \frac{\sqrt[k]{k!}\sqrt[k]{m} - k}{\sqrt[k]{m}} & = & \sqrt[k]{k!} - \frac{k}{\sqrt[k]{m}} & & \text{lo que implica} \\ \lim_{m\to\infty} \frac{\lfloor\!\lfloor m \rfloor\!\rfloor_k}{\sqrt[k]{m}} & \geq & \lim_{m\to\infty} \left(\sqrt[k]{k!} - \frac{k}{\sqrt[k]{m}}\right) & = & \sqrt[k]{k!} - k \cdot \lim_{m\to\infty} \frac{1}{\sqrt[k]{m}} & = & \sqrt[k]{k!} \end{array} \tag{2.8}$$

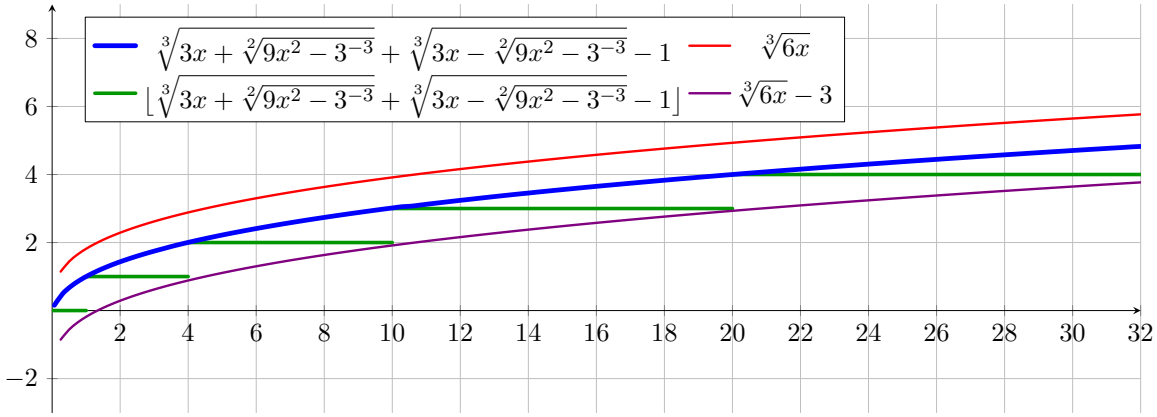
De las ecuaciones (2.7) y (2.8) se sigue que $\lim_{m\to\infty} \frac{\lfloor\!\lfloor m \rfloor\!\rfloor_k}{\sqrt[k]{m}} = \sqrt[k]{k!}$, que es lo buscado.

(2) Por la observación 2.1.5 sucede que $S_k(2) = k+1$ y $S_k(1) = 1$. Para $k > m$ sucede que $S_k(1) = 1 \leq m < k+1 = S_k(2)$ lo que implica que $\lfloor\!\lfloor m \rfloor\!\rfloor_k = 1$.

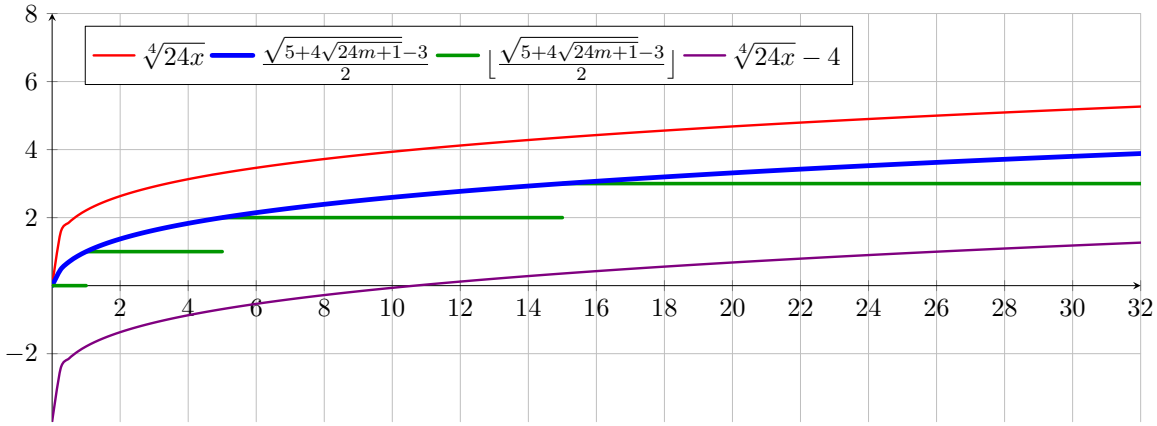
Para k suficientemente grande, esto es $k > m$ sucede que $\frac{\lfloor\!\lfloor m \rfloor\!\rfloor_k}{\sqrt[k]{m}} = \frac{1}{\sqrt[k]{m}}$. Sea $x = \frac{-1}{k}$, entonces es claro que $x \to 0$ cuando $k \to \infty$, por lo que $\lim_{k\to\infty} \frac{\lfloor\!\lfloor m \rfloor\!\rfloor_k}{\sqrt[k]{m}} = \lim_{k\to\infty} \frac{1}{\sqrt[k]{m}} = \lim_{k\to\infty} m^{\frac{-1}{k}} = \lim_{x\to 0} m^x = 1$. QED



(a) Función floorSimplex para dimensión 2 y sus cotas.



(b) Función floorSimplex para dimensión 3 y sus cotas.



(c) Función floorSimplex para dimensión 4 y sus cotas.

Figura 2.4: Gráficas de las funciones floorSimplex para dimensión 2,3 y 4 así como sus cotas.

2.2.1. Algoritmo FloorSimplex

Procedamos a plantear la versión estándar del algoritmo FloorSimplex que obtiene el índice del simplex que antecede a un número m. Esto se logra calculando cotas superiores e inferiores al índice mediante una búsqueda exponencial. Una vez obtenidas estas cotas, se hace búsque-

da binaria hasta hallar el índice. Más adelante plantearemos otras versiones de este algoritmo donde algunas usan las cotas provistas por la proposición 2.2.7 y otras laboran sobre una tabla precomputada de números simplex que se puede obtener con el algoritmo 2. Estas versiones son en esencia el mismo algoritmo con distintas modificaciones.

Algoritmo 3 FloorSimplex(m, k)

Entrada: $m, k \in \mathbb{N}$ con $k > 0$.
Salida: $\llbracket m \rrbracket_k$.

```
1: if m = 0 then
2:     return 0
3: end if
4: a ← 1
5: b ← 2
       ▷ Búsqueda de cotas
6: while S_k(b) ≤ m do
7:     a ← b
8:     b ← 2 · b
9: end while
10: loop                ▷ Búsqueda binaria para índice
11:     d ← b − a
12:     if d = 1 then
13:         return a
14:     end if
15:     c ← a + ⌊d/2⌋
16:     if m < S_k(c) then
17:         b ← c
18:     else
19:         a ← c
20:     end if
21: end loop
```

Observación 2.2.10. Si $m = 0$ es la entrada al algoritmo FloorSimplex (3) la salida es $0 = \llbracket 0 \rrbracket_k = \llbracket m \rrbracket_k$ por el condicional if de la línea 1.

Afirmaciones del ciclo while de las líneas 6-9

Sean a_0, b_0 los valores de las variables a, b antes de entrar al ciclo while de las líneas 6-9. Sean a_i, b_i sus valores correspondientes al terminar la i-ésima iteración de este ciclo.

Afirmación 2.2.11. Dadas las definiciones de a_i, b_i propuestas y supongamos que $m > 0$, se tienen las siguientes sobre el ciclo while de las líneas 6-9:

1. $a_i = 2^i$ y $b_i = 2^{i+1}$ para toda $i \in \mathbb{N}$.
2. El ciclo while termina.
3. Al terminar el ciclo while, se sigue que $S_k(a) \leq m < S_k(b)$.

Demostración. **(1)** Procedamos por inducción sobre i. Para $i = 0$ se tiene que por las líneas $a_0 = 1 = 2^0$ y $b_0 = 2 = 2^{0+1}$.

Supongamos que el resultado se tiene para i y veamos que se sostiene para $i+1$. Supongamos que estamos en la $i+1$-iteración del ciclo. Por las líneas 7 y 8 se tiene que

$$a_{i+1} = b_i \overset{\text{HI}}{=} 2^{i+1} \qquad \text{y además} \qquad b_{i+1} = 2 \cdot b_i \overset{\text{HI}}{=} 2 \cdot 2^{i+1} = 2^{i+2}.$$

(2) Sabemos que $i < 2^i$ para toda i y considerando el principio arquimedeano se sigue que existe $i_0 \in \mathbb{N}$ de forma que $m < i_0 < 2^{i_0+1}$. Por la proposición 2.1.14 se tiene que $S_k(2^{i_0+1}) > m$ por lo que en la iteración i_0 necesariamente $S_k(b_{i_0}) > m$ pues $b_{i_0} = 2^{i_0+1}$ por (1). De aquí que la condición del while de la línea 6 necesariamente se rompa finalizando este ciclo.

(3) Por el inciso (2) el ciclo while termina. Sea i_0 la iteración en la que se cumple $S_k(b_{i_0}) > m$. Si $i_0 = 0$, entonces $S_k(2) > m$ es decir $S_k(b) > m$. Dado que $m > 0$ se sigue que $S_k(a) = S_k(1) \leq m$.

Si $i_0 = j + 1$ para alguna j, entonces, $S_k(b_{j+1}) = S_k(2^{j+2}) > m$, dado que el ciclo termina en la $j + 1$ iteración necesariamente sucede que $S_k(b_j) = S_k(2^{j+1}) \leq m$. Por el inciso (1) se sigue que $a_{j+1} = b_j$ y por tanto $S_k(a_{j+1}) \leq m < S_k(b_{j+1})$. Así, al finalizar el ciclo, las variables a, b cumplen que $S_k(a) \leq m < S_k(b)$. QED

Afirmación 2.2.12. Si $n = \lVert m \rVert_k$ entonces se sostienen las siguientes.

1. Si $i = \lfloor \log_2 n \rfloor$, entonces $2^i \leq n < 2^{i+1}$, es decir $a_i \leq n < b_i$.
2. Si $i = \lfloor \log_2 n \rfloor$, entonces $S_k(a_i) \leq m < S_k(b_i)$.
3. El ciclo while de las líneas 6-9 termina en exactamente $\lfloor \log_2 n \rfloor$ iteraciones.

Demostración. **(1)** Si $i = \lfloor \log_2 n \rfloor$ entonces por definición se sigue que $i \leq \log_2 n < i + 1$. De la desigualdad anterior se sigue que $2^i \leq n < 2^{i+1}$. De la afirmación 2.2.11 se tienen que $a_i \leq n < b_i$.

(2) Del inciso (1) se tiene que $a_i \leq n < b_i$, esto es, $n + 1 \leq b_i$. Dado que $n = \lVert m \rVert_k$ y por la proposición 2.1.13(3) se sigue que $S_k(a_i) \leq S_k(n) \leq m < S_k(n + 1) \leq S_k(b_i)$.

(3) El inciso (2) implica que en la iteración $i = \lfloor \log_2 n \rfloor$, se rompe el condicional del ciclo while. Dado que existe un único i que satisface $2^i \leq n < 2^{i+1}$ para toda $n \in \mathbb{N}$, entonces necesariamente tiene que ser el caso de que $i = \lfloor \log_2 n \rfloor$ para que se termine. Por lo tanto toma exactamente$\lfloor \log_2 n \rfloor$ iteraciones para terminar. QED

Afirmaciones del ciclo loop de las líneas 10-21

Sean a_0, b_0 los valores de las variables a, b previo de comenzar la iteración de ciclo loop de las líneas 10-21, esto es, su valor al concluir el ciclo while de las líneas 6-9. Sean también a_i, b_i, c_i, d_i el valor de estas variables tras la i-ésima iteración. Obsérvese que las variables c, d se asignan por primera vez dentro de este ciclo.

Afirmación 2.2.13. Dadas las definiciones previas de a_i, b_i, c_i, d_i, se siguen las siguientes afirmaciones correspondientes al ciclo loop de las líneas 10-21 para toda $i \in \mathbb{N}$.

1. $d_{i+1} = b_i - a_i$ y $d_{i+1} \in \mathbb{N}$ con $0 < d_{i+1}$.
2. Si se concreta la iteración $i + 1$, se tiene $c_{i+1} = a_i + \lfloor \frac{d_{i+1}}{2} \rfloor$ y además $a_i < c_{i+1} < b_i$.
3. Si se concreta la iteración $i + 1$, entonces $a_{i+1}, b_{i+1} \in \mathbb{N}$ con $a_{i+1} < b_{i+1}$ y además sucede $a_{i+1} = c_{i+1}$, $b_{i+1} = b_i$ y $S_k(c_{i+1}) \leq m$ o bien $a_{i+1} = a_i$, $b_{i+1} = c_{i+1}$ y $S_k(c_{i+1}) > m$.

Demostración. Comencemos observando que la única detención al ciclo puede ocurrir sólo en la línea 13 que ocurre antes de la definición de c_i (línea 15) y para esto necesariamente $d_i = 1$, por lo que si se alcanza a definir c_i necesariamente $d_i \neq 1$.

Probemos simultáneamente los incisos 1,2 y 3, esto es, la conjunción de éstos por inducción sobre i.

Caso Base. Para $i = 0$, dado que a_0, b_0 tendrán los valores al salir del ciclo while de las líneas 6-9, se sigue de la afirmación 2.2.11 que $a_0 = 2^j$ y $b_0 = 2^{j+1}$ ambos elementos de $\mathbb{N}$ para alguna $j \in \mathbb{N}$.

1. De la línea 11 se obtiene $d_1 = b_0 - a_0 = 2^{j+1} - 2^j = 2^j(2-1) = 2^j \in \mathbb{N}$. Más aún, $d_1 > 0$ por ser una potencia de 2.

2. Supongamos que se concreta la $i+1$-ésima iteración, entonces por nuestra observación inicial, $d_1 \neq 0$, de donde por el inciso (1) se tiene que $d_1 \geq 2$, es decir, $\frac{d_1}{2} \geq 2$ y así $\lfloor \frac{d_1}{2} \rfloor \geq 1$.

   De la línea 15 se sigue que

   $$c_1 = a_0 + \left\lfloor \frac{d_1}{2} \right\rfloor \geq a_0 + 1 > a_0 \quad \text{y claramente} \quad c_1 \in \mathbb{N}.$$

   De hecho, por el inciso (1) y el lema 2.2.4 las siguientes igualdades toman lugar

   $$c_1 = a_0 + \left\lfloor \frac{d_1}{2} \right\rfloor = \left\lfloor a_0 + \frac{d_1}{2} \right\rfloor = \left\lfloor a_0 + \frac{b_0 - a_0}{2} \right\rfloor = \left\lfloor \frac{a_0 + b_0}{2} \right\rfloor.$$

   Basta mostrar que $\frac{a_0+b_0}{2} < b_0$. Claramente $a_0 < b_0$, entonces sumando b_0 de ambos lados se sigue $a_0 + b_0 < 2b_0$ por lo que $\frac{a_0+b_0}{2} < b_0$. Así $a_0 < c_1 < b_0$.

3. Siguiendo el cómputo, si en el condicional de la línea 16 se cumple $m < S_k(c_1)$, entonces por la línea 17 se obtiene que $b_1 = c_1$, y se concreta la iteración, de donde $a_1 = a_0$. Por el inciso (2) se tiene que $a_1 = a_0 < c_1 = b_1$.

   Si en el condicional de la línea 16 se tiene el caso $S_k(c_1) \leq m$ se tiene, por la línea 19 se sigue $a_1 = c_1$ y finaliza la iteración por lo que $b_1 = b_0$. Por el inciso (2) se tiene que $a_1 = c_1 < b_0 = b_1$.

   Sea cual fuere el caso, es evidente que $a_1, b_1 \in \mathbb{N}$ y además $a_1 < b_1$.

Caso Inductivo. Supongamos que los incisos 1,2 y 3 se satisfacen para la iteración i y veamos que estos incisos se sostienen para la iteración $i+1$.

1. Por hipótesis inductiva en el inciso (3) se obtiene $b_{i+1} > a_{i+1}$, ambos naturales. Conjuntando esto, al considerar la línea 11 se sigue que $d_{i+2} = b_{i+1} - a_{i+1} > 0$ y también $d_{i+2} \in \mathbb{N}$.

2. Supongamos que se concreta la iteración $i+2$. De la observación inicial, necesariamente $d_{i+2} > 2$ y por tanto $\lfloor \frac{d_{i+2}}{2} \rfloor \geq 1$. De nuevo, de la línea 15 se tiene

   $$c_{i+2} = a_{i+1} + \left\lfloor \frac{d_{i+2}}{2} \right\rfloor \geq a_{i+1} + 1 > a_{i+1} \quad \text{y claramente} \quad c_{i+2} \in \mathbb{N}.$$

   Análogamente al caso base, por el inciso (1) y el lema 2.2.4 se tiene

   $$c_{i+2} = a_{i+1} + \left\lfloor \frac{d_{i+2}}{2} \right\rfloor = \left\lfloor a_{i+1} + \frac{d_{i+2}}{2} \right\rfloor = \left\lfloor a_{i+1} + \frac{b_{i+1} - a_{i+1}}{2} \right\rfloor = \left\lfloor \frac{a_{i+1} + b_{i+1}}{2} \right\rfloor.$$

   Mostremos que $\frac{a_{i+1}+b_{i+1}}{2} < b_{i+1}$. Por hipótesis inductiva, se tiene $a_{i+1} < b_{i+1}$, entonces sumando b_{i+1} de ambos lados se sigue $a_{i+1} + b_{i+1} < 2b_{i+1}$ por lo que $\frac{a_{i+1}+b_{i+1}}{2} < b_{i+1}$. Así $a_{i+1} < c_{i+2} < b_{i+1}$.

3. Supongamos que se concreta la iteración $i+2$. De nuevo, en el condicional de la línea 16, si se satisface $m < S_k(c_{i+1})$, de la línea 17 se tiene que $b_{i+2} = c_{i+2}$ y $a_{i+2} = a_{i+1}$. Del inciso (2) se sigue que $a_{i+2} = a_{i+1} < c_{i+2} = b_{i+2}$.

Si en el condicional de la línea 16 ocurriera que $S_k(c_{i+1}) \leq m$, entonces por la línea 19 se sigue que $a_{i+2} = c_{i+2}$ y $b_{i+2} = b_{i+1}$. Nuevamente, por el inciso (2) se tiene que $a_{i+2} = c_{i+2} < b_{i+1} = b_{i+2}$.

Sea cual fuere el caso, ocurre que $a_{i+2}, b_{i+2} \in \mathbb{N}$ y $a_{i+2} < b_{i+2}$.

QED

Afirmación 2.2.14. Para cualquier $i \in \mathbb{N}$ se siguen las siguientes:

1. $a_i < a_{i+1}$ y $b_{i+1} = b_i$ o bien $a_i = a_{i+1}$ y $b_{i+1} < b_i$,
2. $d_{i+2} < d_{i+1}$ y
3. $d_{i+2} = d_{i+1} - \lfloor \frac{d_{i+1}}{2} \rfloor$ o bien $d_{i+2} = \lfloor \frac{d_{i+1}}{2} \rfloor$.

Demostración. Sea $i \in \mathbb{N}$.

(1) Por el inciso (2) de la afirmación 2.2.13 se sigue que

$$a_i < c_{i+1} < b_i \tag{2.9}$$

Del inciso (3) de esta misma afirmación se sigue que $a_{i+1} = a_i$ y $b_{i+1} = c_{i+1}$ o bien $a_{i+1} = c_{i+1}$ y $b_{i+1} = b_i$. Supongamos el primer caso disyuntiva. Substituyendo en la ecuación (2.9) se tiene que $a_i < a_{i+1}$ y $b_{i+1} = b_i$. Para el segundo caso, se tiene $a_{i+1} = a_i$ y $b_{i+1} = c_{i+1}$, y substituyendo en la ecuación (2.9) se obtiene que $b_{i+1} < b_i$.

(2) Por el inciso (1) de la afirmación 2.2.13 se tiene que $d_{j+1} = b_j - a_j$ para toda $j \in \mathbb{N}$. Por el inciso (1) se tienen dos casos. Si $a_i < a_{i+1}$ y $b_{i+1} = b_i$ entonces multiplicando por -1 la desigualdad y sumando b_{i+1} se obtiene $b_i - a_i > b_{i+1} - a_{i+1}$, es decir, $d_{i+2} < d_{i+1}$.

Si fuera el caso de que $a_i = a_{i+1}$ y $b_{i+1} < b_i$, entonces restando a_{i+1} a la desigualdad se obtiene $b_{i+1} - a_{i+1} > b_i - a_i$, es decir, $d_{i+2} < d_{i+1}$. Sea cual fuere el caso se llega a que $d_{i+2} < d_{i+1}$.

(3) Sea $i \in \mathbb{N}$. Si la variable d_{i+2} tiene valor asignado quiere decir que se concretó la iteración $i+1$, lo que implica que los valores a_{i+1},a_i, b_{i+1},b_i, d_{i+1} y c_{i+1} están definidos.

Por la afirmación 2.2.13 se tiene que $c_{i+1} = a_i + \left\lfloor \frac{d_{i+1}}{2} \right\rfloor$ y

$$d_{i+2} = \begin{cases} b_i - c_{i+1} & \text{si } S_k(c_{i+1}) \leq m, \\ c_{i+1} - a_i & \text{si } S_k(c_{i+1}) > m. \end{cases}$$

Para el primer caso se tiene que

$$d_{i+2} = b_i - c_{i+1} = b_i - a_i - \left\lfloor \frac{d_{i+1}}{2} \right\rfloor = d_{i+1} - \left\lfloor \frac{d_{i+1}}{2} \right\rfloor.$$

Para el segundo caso

$$d_{i+2} = c_{i+1} - a_i = a_i + \left\lfloor \frac{d_{i+1}}{2} \right\rfloor - a_i = \left\lfloor \frac{d_{i+1}}{2} \right\rfloor.$$

QED

Afirmación 2.2.15. Para toda $i \in \mathbb{N}$ se sostienen las siguientes.

1. $S_k(a_i) \leq m < S_k(b_i)$.

2. Si $d_{i+1} = 1$ entonces $a_i = \llbracket m \rrbracket_k$.

Demostración. **(1)** Procedamos por inducción sobre i. Para $i = 0$ se tiene que por la afirmación 2.2.11(3) se cumple que $S_k(a_0) \leq m < S_k(b_0)$.

Supongamos que el resultado se tiene para i y veamos que se sostiene para $i + 1$. Por la afirmación 2.2.14(3) hay dos casos.

Si fuera el caso de que $a_{i+1} = a_i$, $c_{i+1} = b_{i+1}$ y $m < S_k(c_{i+1})$. Por hipótesis inductiva se sigue que $S_k(a_i) \leq m$, es decir $S_k(a_{i+1}) \leq m$. Como $m < S_k(c_{i+1})$, entonces $m < S_k(b_{i+1})$ Conjuntando lo anterior se tiene que $S_k(a_{i+1}) \leq m < S_k(b_{i+1})$.

Si fuera el caso de que $b_{i+1} = b_i$, $c_{i+1} = a_{i+1}$ y $S_k(c_{i+1}) \leq m$, entonces $S_k(a_{i+1}) \leq m$. Por hipótesis de inducción se tiene que $m < S_k(b_i)$. Esto es $m < S_k(b_{i+1})$ y por tanto $S_k(a_{i+1}) \leq m < S_k(b_{i+1})$.

En ambos casos llegamos a lo buscado.

(2) Sea $i \in \mathbb{N}$. De la afirmación 2.2.13(1) se sigue que $d_{i+1} = b_i - a_i$. Si $d_{i+1} = 1$ entonces $b_i = a_i + 1$ y se entra el condicional de la línea 12 y devuelve el valor a_i por la línea 13.

Sabemos que por el inciso (1) sucede que $S_k(a_i) \leq m < S_k(b_i)$, lo que implica que $S_k(a_i) \leq m < S_k(a_i + 1)$. Esto último implica que $a_i = \llbracket m \rrbracket_k$. QED

Lema 2.2.16. *Para cualquier $n \in \mathbb{N}$ sucede que $n - \lfloor \frac{n}{2} \rfloor \leq \frac{n+1}{2}$.*

Demostración. Si $n = 2k$ para alguna $k \in \mathbb{N}$, entonces

$$n - \left\lfloor \frac{n}{2} \right\rfloor = 2k - \left\lfloor \frac{2k}{2} \right\rfloor = 2k - \lfloor k \rfloor = k \leq k + \frac{1}{2} = \frac{2k+1}{2} = \frac{n+1}{2}.$$

Si $n = 2k + 1$, entonces

$$n - \left\lfloor \frac{n}{2} \right\rfloor = 2k + 1 - \left\lfloor \frac{2k+1}{2} \right\rfloor = 2k + 1 - \left\lfloor k + \frac{1}{2} \right\rfloor = k + 1 = \frac{2k+1+1}{2} = \frac{n+1}{2}.$$

Por lo tanto $n - \lfloor \frac{n}{2} \rfloor \leq \frac{n+1}{2}$. QED

Afirmación 2.2.17. Si tomamos d_i como previamente definidos, entonces para toda $i \in \mathbb{N}$ se cumplen las siguientes:

1. $d_{i+2} \leq \frac{d_{i+1}+1}{2}$,
2. $d_{i+1} < \frac{b_0 - a_0}{2^{i-1}} + 1$,
3. $2d_{i+2} - 1 \leq d_{i+1} < 4d_{i+2}$,
4. si $d_{i+1} = 1$ entonces $i + 1 \leq \log_2(b_0 - a_0) + 3$, y
5. el ciclo loop de las líneas 10-21 termina en a lo más $\log_2(b_0 - a_0) + 3$ iteraciones.

Demostración. **(1)** Sea $i \in \mathbb{N}$. Si la variable d_{i+2} tiene valor asignado quiere decir que se concretó la iteración $i + 1$, lo que implica que los valores a_{i+1},a_i, b_{i+1},b_i, d_{i+1} y c_{i+1} están definidos. De la afirmación 2.2.14(3) hay dos casos.

Si $d_{i+2} = d_{i+1} - \left\lfloor \frac{d_{i+1}}{2} \right\rfloor$, entonces por el lema 2.2.16 se tiene $d_{i+1} - \left\lfloor \frac{d_{i+1}}{2} \right\rfloor \leq \frac{d_{i+1}+1}{2}$.

Por otro lado, si $d_{i+2} = \left\lfloor \frac{d_{i+1}}{2} \right\rfloor$, entonces $\left\lfloor \frac{d_{i+1}}{2} \right\rfloor \leq \frac{d_{i+1}}{2} \leq \frac{d_{i+1}+1}{2}$. Sin importar el caso se concluye que $d_{i+2} \leq \frac{d_{i+1}+1}{2}$.

(2) Procedamos por inducción sobre i. Para $i = 0$ se sigue que $d_1 = b_0 - a_0 < 2(b_0 - a_0) = \frac{b_0-a_0}{2^{-1}}$. Supongamos que el resultado es válido para i y mostremos que es válido para $i+1$, esto es, mostremos que $d_{i+2} \leq \frac{b_0-a_0}{2^i} + 1$. Usando el inciso (1) y la hipótesis inductiva se tiene que

$$d_{i+2} \leq \frac{d_{i+1}+1}{2} < \frac{\left[\frac{b_0-a_0}{2^{i-1}} + 1\right] + 1}{2} = \frac{b_0-a_0}{2^i} + 1 \qquad \text{pues } d_{i+1} < \frac{b_0-a_0}{2^{i-1}} + 1.$$

(3) Del inciso (1) se sigue que $2d_{i+2} - 1 \leq d_{i+1}$. Mostremos que $d_{i+1} < 4d_{i+2}$. Por la afirmación 2.2.14(2) hay dos casos: Si $d_{i+2} = d_{i+1} - \left\lfloor \frac{d_{i+1}}{2} \right\rfloor$, observemos que $\left\lfloor \frac{d_{i+1}}{2} \right\rfloor \leq \frac{d_{i+1}}{2}$ y por tanto $-\frac{d_{i+1}}{2} \leq -\left\lfloor \frac{d_{i+1}}{2} \right\rfloor$. De esto último se tiene que

$$d_{i+2} = d_{i+1} - \left\lfloor \frac{d_{i+1}}{2} \right\rfloor \geq d_{i+1} - \frac{d_{i+1}}{2} = \frac{d_{i+1}}{2} \quad \text{ es decir, } \quad \frac{d_{i+1}}{2} \leq d_{i+2}.$$

De la desigualdad anterior se sigue que $d_{i+1} \leq 2d_{i+1} < 4d_{i+2}$ ya que $d_{i+1} \geq 1$ por la afirmación 2.2.13(1).

Si fuera el caso de que $d_{i+2} = \left\lfloor \frac{d_{i+1}}{2} \right\rfloor$, por definición se sigue que

$$\frac{d_{i+1}}{2} < d_{i+2} + 1, \quad \text{ y por tanto } \quad d_{i+1} < 2d_{i+2} + 2 \leq 2d_{i+2} + 2d_{i+2} = 4d_{i+2}.$$

Sea cual fuere el caso, se concluye que $d_{i+1} < 4d_{i+2}$.

(4) Hay dos casos. Si $i = 0$, esto es, en la primera iteración sucede que $d_1 = 1$, entonces por la afirmación 2.2.13 sucede que $1 = d_1 = b_0 - a_0$, y así $\log_2(b_0 - a_0) = 0$ lo que implica que $i + 1 = 1 < \log_2(b_0 - a_0) + 3$.

Si $i > 0$, entonces existe un j tal que $i + 1 = j + 2$. Esto es, si en la iteración $j + 2$ se cumple que $d_{j+2} = 1$ la cual no es el primera iteración. Por el inciso (3), el valor de la iteración inmediata anterior $j + 1$ cumple que $2d_{j+2} - 1 \leq d_{j+1} < 4d_{j+2}$, es decir, $1 \leq d_{j+1} < 4$. No puede ser el caso de que $d_{j+1} = 1$ por la afirmación 2.2.14.

Si $d_{j+1} = 2$ por el inciso (2) se cumple que

$$2 < \frac{b_0-a_0}{2^{j+1}} + 1 \quad \text{ lo que implica } \quad 2^{j-1} < b_0 - a_0 \quad \text{ es decir, } \quad j - 1 < \log_2(b_0 - a_0). \tag{2.10}$$

Si $d_{j+1} = 3$, de nuevo por el inciso (2) se sigue

$$3 < \frac{b_0-a_0}{2} + 1 \quad \text{ lo que implica } \quad 2^j < b_0 - a_0 \quad \text{ finalmente } \quad j < \log_2(b_0 - a_0). \tag{2.11}$$

En ambas ecuaciones (2.10) y (2.11) se obtiene directamente que $i+1 = j+2 < \log_2(b_0-a_0)+3$, por lo que sin importar que caso sea la desigualdad $i + 1 < \log_2(b_0 - a_0) + 3$ se sostiene.

(5) De las afirmaciones 2.2.13 y 2.2.14 sabemos que $0 < d_{i+1} \in \mathbb{N}$ y $d_{i+2} < d_{i+1}$ para toda iteración $i \in \mathbb{N}$. Es decir, la variable d desciende estrictamente y es siempre un natural positivo. Necesariamente hay una $i \in \mathbb{N}$ donde $d_{i+1} = 1$. En esta iteración, el ciclo loop de las líneas 10-21 se rompe pues se entra en el condicional de la línea 12 y se ejecuta la línea 13.

Por el inciso (4), para esta i debe de ocurrir que $i + 1 < \log_2(b_0 - a_0) + 3$, por lo que este ciclo termina en a lo más $\log_2(b_0 - a_0) + 3$ iteraciones. QED

Proposición 2.2.18. *El algoritmo* FloorSimplex*(3) termina.*

Demostración. Es claro que las operaciones de las líneas 1-5 terminan pues constan de meras asignaciones y un condicional. Por la afirmación 2.2.11, el ciclo while de las líneas 6-9 termina.

Por la afirmación 2.2.17 el ciclo loop de las líneas 10-21 termina, lo que finaliza la ejecución del algoritmo.

QED

Proposición 2.2.19. *Si la entrada al algortimo* FloorSimplex *(3) es $m, k \in \mathbb{N}$ con $k > 0$, entonces la salida es $\llbracket m \rrbracket_k$.*

Demostración. Si $m = 0$ el resultado se sigue directamente. Si $m > 0$ por la afirmación 2.2.14 el algoritmo alcanza el valor $d = 1$ en alguna iteración. Por la afirmación 2.2.15 se tiene que el valor de la variable $a = \llbracket m \rrbracket_k$ que corresponde con la salida por la línea 13. Esto último es lo buscado. QED

Complejidad en tiempo y espacio

Proposición 2.2.20. *Sean $m, k \in \mathbb{N}$ con $k > 0$. Si el cálculo de un simplex se hace con el algoritmo* Simplex*(1), entonces la complejidad en tiempo de* FloorSimplex(m, k) *(3) es* [1] $O(k \log_2(\llbracket m \rrbracket_k))$ *y la complejidad en espacio es $O(1)$.*

Demostración. Sea $n = \llbracket m \rrbracket_k$. Si $m = 0$, claramente se alcanza la salida en tiempo $O(1)$.

Si $m > 0$ las líneas 1-5 toman tiempo $O(1)$. Por la afirmación 2.2.12 sabemos que el ciclo while de las líneas 6-9 toma a lo más $\lfloor \log_2(n) \rfloor$ iteraciones. Por lo tanto, la complejidad en tiempo de este ciclo será la cantidad de operaciones que subtienden a este multiplicado por la cantidad de iteraciones. El condicional del ciclo while (línea 6) toma $O(k)$ pues es el cálculo de un simplex y las líneas 7 y 8 toman tiempo $O(1)$. Por tanto el ciclo while toma tiempo $O(k \log_2(n))$.

Si a_0 y b_0 son los valores de las variables a y b al terminar el ciclo while, por la afirmación 2.2.12, se cumple que $a_0 = 2^i \leq n < 2^{i+1} = b_0$ donde $i = \lfloor \log_2(n) \rfloor$. Siendo así,

$$b_0 - a_0 = 2^{i+1} - 2^i = 2^i(2 - 1) = 2^i \leq n \tag{2.12}$$

Por otro lado, por la afirmación 2.2.17, el ciclo loop de las líneas 10-21 toma a lo más $\log_2(b_0 - a_0) + 3$ iteraciones. Por la ecuación (2.12), el ciclo toma a lo más $\log_2(n) + 3$ iteraciones. Las líneas 11-15 y 17-20 toman tiempo $O(1)$ y sólo el condicional de la línea 16 toma tiempo $O(k)$ ya que se calcula un k-simplex. Debido a esto, este ciclo toma tiempo $O(k \log_2(n) + 3k)$. Como $n = \llbracket m \rrbracket_k$, el tiempo de ejecución del algoritmo es la suma de tiempos

$$O\big(k \log_2(n) + k \log_2(n) + 3k\big) = O\big(2k \log_2(n) + 3k\big) = O\big(k \log_2(n)\big) = O\big(k \log_2(\llbracket m \rrbracket_k)\big).$$

Para la complejidad en espacio, es sencillo observar que se utilizan solo las variables a, b, c, d además del espacio que tome calcular un simplex en las líneas 6 y 16. Ambos cálculos toman espacio $O(1)$, por lo que el algoritmo toma espacio $O(1)$. QED

2.2.2. Versiones alternativas de FloorSimplex

Utilizando las cotas provistas por la proposición 2.2.7 y una tabla de simplex precomputados obtenida con el algoritmo 2 tenemos las siguientes versiones de FloorSimplex. La corrección de estos algoritmos se debe a la similaridad con el algoritmo 3. Muchas de las afirmaciones

[1] Por la observación 2.2.8, esta complejidad también es $O(k \log_2(k \sqrt[k]{m}))$.

planteadas en la demostración de la corrección del algoritmo 3 se siguen sosteniendo con pocas o nulas modificaciones. Nos enfocaremos principalmente en su complejidad en tiempo.

La siguiente versión de FloorSimplex utiliza la tabla de simplex como medio para ahorrarse el cómputo de estos números. Particularmente, si esta tabla es suficientemente grande como para contener el índice buscado, se puede hacer búsqueda binaria en ella para obtener el índice. Cabe mencionar que, gracias a la búsqueda exponencial de cotas, la búsqueda binaria siempre se hará sobre un subarreglo cuya longitud no sobrepasa el valor del índice. Esta cualidad es ventajosa pues hace que el procedimiento no se vuelva contraproducente si la tabla de valores precomputados es muy grande.

Algoritmo 4 FloorSimplex(m, k, TS)(Con tabla precomputada)

Entrada: $m, k \in \mathbb{N}$ tales que $\llbracket m \rrbracket_k < TS.colLen$ y $0 < k < TS.rowLen$.
Salida: $\llbracket m \rrbracket_k$.

1: **if** $m = 0$ **then**
2: **return** 0
3: **end if**
4: $a \leftarrow 1$
5: $b \leftarrow 2$
6: **while** $TS[k, b] \leq m$ **do**
7: $a \leftarrow b$
8: $b \leftarrow \text{mín}\{2b, TS.\text{colLen} - 1\}$
9: **end while**
10: **loop** ▷ Búsqueda binaria para índice
11: $d \leftarrow b - a$
12: **if** $d = 1$ **then**
13: **return** a
14: **end if**
15: $c \leftarrow a + \lfloor \frac{d}{2} \rfloor$
16: **if** $m < TS[k, c]$ **then** ▷ $m < S_k(c)$
17: $b \leftarrow c$
18: **else**
19: $a \leftarrow c$
20: **end if**
21: **end loop**

Esta versión añade la restricción de que el índice buscado se encuentre en la tabla provista. Se necesita primero que se tenga la dimensión de simplex para los cuales se busca el índice. Finalmente, dado que necesitamos de una cota superior e inferior para este índice basta pedir que el índice sea menor estrictamente a la cantidad de columnas menos una, para que haya al menos una cota superior.

Proposición 2.2.21. *Si la entrada al algoritmo (4) es $m, k, TS \in \mathbb{N}$ con TS una tabla suficientemente grande, $\llbracket m \rrbracket_k < TS.colLen - 1$ y $0 < k < TS.rowLen$, entonces termina y la salida es $\llbracket m \rrbracket_k$.*

Demostración. Para el ciclo while de las líneas 6-9, la afirmación 2.2.11 es idéntica con la salvedad de que $b_i = \text{mín}\{2^{i+1}, TS.colLen - 1\}$ y la prueba es análoga. Conjuntando esto con el hecho de que $\llbracket m \rrbracket_k < TS.colLen - 1$ por hipótesis, la prueba de la afirmación 2.2.12 es análoga.

No es complicado ver que las pruebas de las afirmaciones 2.2.13, 2.2.14, 2.2.15 y 2.2.17 del ciclo loop de las líneas 10-21 son completamente iguales.

Nótese que al solicitar $k < TS.rowLen$ y $\llbracket m \rrbracket_k < TS.colLen - 1$ garantizamos que no se elijan valores fuera de la tabla TS. Por todo anterior las proposiciones 2.2.18 y 2.2.19 se siguen sosteniendo, por lo que se tiene el resultado. QED

Al usar la tabla con números simplex, surge el problema de saber qué dimensiones debe de tener para poder garantizar el cómputo de un algoritmo utilice números simplex o bien llame a FloorSimplex.

Definición 2.2.22. Sea TABLA-SIMPLEX$(k,n) = TS$. Decimos que TS *es suficientemente grande para computar* algún algoritmo dado con una entrada dada si: todo simplex $S_i(j)$ llamado en el cómputo del algoritmo cumple que $0 \leq i < k$; y toda llamada a $\lfloor\!\lfloor m \rfloor\!\rfloor_r$ cumple que $1 \leq r < k$ y $\lfloor\!\lfloor m \rfloor\!\rfloor_r < n-1$.

Proposición 2.2.23. *Sean $m, k \in \mathbb{N}$ con $k > 0$ y TS una tabla de simplex suficientemente grande. La complejidad en tiempo del algoritmo (4) con la entrada (m, k, TS) es*[2] $O(\log_2(\lfloor\!\lfloor m \rfloor\!\rfloor_k))$ *y la complejidad en espacio es $O(1)$.*

Demostración. Sea $n = \lfloor\!\lfloor m \rfloor\!\rfloor_k$ con $m > 0$ pues para $m = 0$ es resultado es claro. Procediendo de forma análoga a la proposición 2.2.20, el ciclo while de las líneas 6-9 hace a lo más $\log_2(n)$ iteraciones (la afirmación 2.2.12 se sigue sosteniendo). La llamada del condicional del while de la línea 6 así como las líneas 7 y 8 toman tiempo constante. Por lo tanto, el ciclo while de las líneas 6-9 toma tiempo $O(\log_2 n)$

Ahora bien, la distancia entre las variables a, b satisface que al final del ciclo while es

$$b - a = 2^i - \text{mín}\{2^{i+1}, TS.colLen\} = \text{mín}\{2^i, 2^i - TS.colLen\} \leq 2^i \leq n, \tag{2.13}$$

donde $i = \lfloor \log_2(n) \rfloor$. Por la afirmación 2.2.17, utilizando la ecuación (2.13) concluimos que el ciclo loop de las líneas 10-21 realiza a lo más $\log_2(n)$ iteraciones. Dado que consultar un simplex toma tiempo constante pues se encuentra en la tabla TS, no es difícil verificar que cada iteración toma tiempo constante. El tiempo de este ciclo será entonces $O(\log_2(n))$.

Por lo anterior, el algoritmo toma tiempo $O(2\log_2(n)) = O(\log_2(n)) = O(\log_2(\lfloor\!\lfloor m \rfloor\!\rfloor_k))$, como se quería mostrar. No es difícil ver que el algoritmo sólo usa las variables a, b, c y d. Sin considerar la entrada m, k y TS, la complejidad en espacio es $O(1)$. QED

Como se puede apreciar en las proposiciones 2.2.20 y 2.2.23, el procedimiento se puede dividir en dos fases: una búsqueda de cotas superiores e inferiores para el índice $\lfloor\!\lfloor m \rfloor\!\rfloor_k$, y una búsqueda binaria en entre estas cotas. La complejidad en tiempo no es nada más que el tiempo que toma encontrar estas cotas más $\log_2(d_{m,k})$, donde $d_{m,k}$ es la distancia entre las cotas. La búsqueda exponencial empleada en los algoritmos (3) y (4) garantiza que las cotas tengan distancia de a lo más $\lfloor\!\lfloor m \rfloor\!\rfloor_k$. Dado que en ciclo loop se hace el cálculo de un simplex, la complejidad resulta en $O(\mathcal{S}(n,k)\log_2(n))$ con $n = \lfloor\!\lfloor m \rfloor\!\rfloor_k$. Más en general la complejidad en tiempo de FLOORSIMPLEX(m,k) es

$$O\big(\mathcal{S}(k,n)\log_2(d_{m,k}) + c_{m,k}\big), \tag{2.14}$$

donde $n = \lfloor\!\lfloor m \rfloor\!\rfloor_k$, $d_{m,k}$ es la distancia entre las cotas, $c_{m,k}$ es el tiempo de cálculo de las cotas y $\mathcal{S}(k,n)$ es el tiempo de cómputo de un simplex.

En las siguientes versiones de FLOORSIMPLEX se substituye la búsqueda exponencial de cotas de $\lfloor\!\lfloor m \rfloor\!\rfloor_k$, para usar las cotas provistas por la proposición 2.2.7. En esta proposición se dan dos juegos de cotas, y cada una tiene sus ventajas y desventajas. Para cada uno de las parejas de cotas, la distancia $d_{m,k}$ es:

$$\begin{aligned} \lfloor k\sqrt[k]{m} \rfloor - \lfloor \sqrt[k]{m} - k) \rfloor - 1 &\leq k\sqrt[k]{m} - (\sqrt[k]{m} - k) = (k-1)\sqrt[k]{m} + k \in O(k\sqrt[k]{m}) \\ \lfloor \sqrt[k]{k!m} \rfloor - \lfloor \sqrt[k]{k!m} - k \rfloor - 1 &\leq \sqrt[k]{k!m} - (\sqrt[k]{k!m} - k) = k \in O(k) \end{aligned}$$

El primer juego de cotas muestra que la distancia $d_{m,k}$ no mejora mucho respecto a las cotas obtenidas por la búsqueda exponencial, mientras que el segundo reduce significativamente la

[2] Por la observación 2.2.8, esta complejidad también es $O(\log_2(k\sqrt[k]{m}))$.

distancia para valores de m arbitrariamente grandes. La desventaja de la segunda es que se tiene que calcular $\sqrt[k]{k!m}$, valor el cual puede crecer muy rápido a lo largo que crece k. Este cálculo puede ser incluso más caro pues, el costo $c_{m,k}$ de calcular esta cota sobrepasar la ventaja provista en hacer una búsqueda logarítmica en k. Más aún, el costo en espacio para el cálculo de las cotas $\bar{c}_{m,k}$ puede no ser constante, dependiendo del algoritmo empleado.

En ambos casos, se requiere hacer un cálculo de una raíz k-ésima, la cual puede ser costosa, además de vulnerable a inestabilidad numérica para valores de m grandes (que son sobre los que más nos interesa hacer crecer). Por esta razón nos enfocaremos en las versiones provistas en los algoritmos 3 y 4, las cuales tendrán un desempeño parecido con mejor estabilidad numérica. No obstante, si para algún problema en particular no se requieran valores de k muy grandes y en los cuales el cálculo de las cotas sea barato, las otras versiones pueden tener un mejor desempeño.

Para lenguajes específicos se pueden hacer mediciones de desempeño y desarrollar versiones híbridas que incluyan cláusulas para elegir la versión más rápida para una entrada dada (lo cual dependerá también del algoritmo empleado para el cálculo de la raíz k-ésima).

Algoritmo 5 FloorSimplex(m, k)(Con cotas)

Entrada: $m, k \in \mathbb{N}$ con $k > 0$
Salida: $\lfloor\!\lfloor m \rfloor\!\rfloor_k$.
1: **if** $m = 0$ **then**
2: **return** 0
3: **end if**
4: $a \leftarrow \lfloor \sqrt[k]{m} - k \rfloor$
▷ También se puede $\lfloor \sqrt[k]{k!m} - k \rfloor$
5: $b \leftarrow \lceil k \sqrt[k]{m} \rceil$
▷ También se puede $\lceil \sqrt[k]{k!m} \rceil$
6: **loop** ▷ Búsqueda binaria para índice
7: $d \leftarrow b - a$
8: **if** $d = 1$ **then**
9: **return** a
10: **end if**
11: $c \leftarrow a + \lfloor \frac{d}{2} \rfloor$
12: **if** $m < S_k(c)$ **then**
13: $b \leftarrow c$
14: **else**
15: $a \leftarrow c$
16: **end if**
17: **end loop**

Proposición 2.2.24. *Si la entrada al algoritmo (5) es $m, k \in \mathbb{N}$ con $k > 0$, entonces la salida es $\lfloor\!\lfloor m \rfloor\!\rfloor_k$.*

Demostración. De la proposición 2.2.7 se tiene que los valores de las variables a y b previo a entrar al ciclo loop de las líneas 6-17 cumplen que $a \leq \sqrt[k]{m} - k \leq \lfloor\!\lfloor m \rfloor\!\rfloor_k \leq k\sqrt[k]{m} < b$.

Siendo así, la afirmación 2.2.13 se sigue sosteniendo y la prueba es la misma salvo el caso base que es directo. De aquí el resto de las afirmaciones 2.2.14, 2.2.15 y 2.2.17 del ciclo loop de las líneas 6-17 son completamente iguales. De forma semejante las proposiciones 2.2.18 y 2.2.19 son iguales, teniéndose el resultado. QED

Proposición 2.2.25. *Sean $m, k \in \mathbb{N}$ con $k > 0$. La complejidad en tiempo del algoritmo (5) es $O(k \log_2(d_{m,k}) + c_{m,k})$ y la complejidad en espacio es $O(e_{m,k})$. Donde $d_{m,k}, c_{m,k}, e_{m,k}$ denotan respectivamente la distancia entre cotas, el tiempo de cálculo y espacio necesario para el cálculo de las cotas de las líneas 5 y 6.*

Demostración. El procedimiento es análogo al de la proposición 2.2.20. Aplicando la proposición 2.1.16 en la ecuación (2.14) se tiene complejidad $O(k \log_2(d_{m,k}) + c_{m,k})$. Al igual que en la

afirmación 2.2.20, la complejidad en espacio del algoritmo es constante salvo el espacio $e_{m,k}$ que se use para el cálculo de las cotas. Así la complejidad en espacio es $O(e_{m,k})$. QED

Algoritmo 6 FLOORSIMPLEX(m, k, TS) (Con cotas y tabla precomputada)

Entrada: $m, k \in \mathbb{N}$ tales que $\lVert m \rVert_k < TS.colLen$ y $0 < k < TS.rowLen$.
Salida: $\lVert m \rVert_k$.

```
1: if m = 0 then
2:     return 0
3: end if
4: a ← máx{⌊ᵏ√m − k⌋, 1}
       ▷ O bien máx{⌊ᵏ√(k!m) − k⌋, 1}
5: b ← mín{⌈k ᵏ√m ⌉, TS.colLen − 1}
       ▷ O bien mín{⌈ᵏ√(k!m)⌉, TS.colLen − 1}
6: loop                ▷ Búsqueda binaria para índice
7:     d ← b − a
8:     if d = 1 then
9:         return a
10:    end if
11:    c ← a + ⌊d/2⌋
12:    if m < TS[k, c] then
13:        b ← c
14:    else
15:        a ← c
16:    end if
17: end loop
```

Proposición 2.2.26. *Si la entrada al algoritmo (6) es $m, k, TS \in \mathbb{N}$ con TS suficientemente grande, $\lVert m \rVert_k < TS.colLen - 1$ y $0 < k < TS.rowLen$, entonces termina y la salida es $\lVert m \rVert_k$.*

Demostración. Si $m = 0$, entonces claramente la salida es $\lVert m \rVert_k$. Si $m > 0$ entonces $\lVert m \rVert_k \geq 1$. Por la proposición 2.2.7, se tiene que $\lfloor \sqrt[k]{m} - k \rfloor \leq \lVert m \rVert_k$ (análogamente $\lVert m \rVert_k \leq \lfloor \sqrt[k]{k!m} - k \rfloor$). Siendo así, $a = \text{máx}\{\lfloor \sqrt[k]{m} - k \rfloor, 1\} \leq \lVert m \rVert_k$. De forma muy semejante, dado que

$$\lVert m \rVert_k < TS.colLen \quad \text{se sigue que} \quad \lVert m \rVert_k < \text{mín}\{\lceil k \sqrt[k]{m} \rceil, TS.colLen - 1\} = b,$$

(análogamente $\lVert m \rVert_k < \text{mín}\{\lceil \sqrt[k]{k!m} \rceil, TS.colLen - 1\}$). Siendo así, $a \leq \lVert m \rVert_k < b$ y por tanto $S_k(a) \leq m < S_k(b)$.

De nuevo, la afirmación 2.2.13 y también las afirmaciones 2.2.14, 2.2.15 y 2.2.17 del ciclo loop de las líneas 6-17 son completamente iguales. Finalmente las proposiciones 2.2.18 y 2.2.19 se cumplen. QED

Proposición 2.2.27. *Sean $m, k \in \mathbb{N}$ y TS una tabla suficientemente grande. La complejidad en tiempo del algoritmo (6) con la entrada (m, k, TS) es $O(\log_2(d_{m,k}) + c_{m,k})$ y la complejidad en espacio es $O(e_{m,k})$. Donde $d_{m,k}, c_{m,k}, e_{m,k}$ denotan respectivamente la distancia entre cotas, el tiempo de cálculo y espacio necesario para el cálculo de las cotas de las líneas 5 y 6.*

Demostración. Este procedimiento es análogo a la proposición 2.2.23. Aplicando la proposición 2.1.16 en la ecuación (2.14) se sigue la complejidad en tiempo es $O(\log_2(d_{m,k}) + c_{m,k})$. De igual forma la complejidad en espacio es constante salvo el espacio $e_{m,k}$ usado para las cotas. Quedando así complejidad en espacio $O(e_{m,k})$. QED

Capítulo 3

Enumeración de Composiciones

Siendo nuestro objetivo dar una enumeración de tuplas que sea estructuralmente creciente, generalizando la enumeración de Cantor (ver [12]), se busca enumerar a través de las diagonales. Como se trabajó en la primera generalización para 3-tuplas, para dimensión k, una diagonal es el conjunto de k-tuplas que suman un mismo valor (a saber, una composición en k partes). Como se vio anteriormente, usar la suma de las entradas como medida de tamaño para tuplas no es la única posible (por ejemplo, usando cascarones cúbicos como se menciona en [22]), pero será la que se manejará para el desarrollo de esta enumeración.

Funciones Rank y Unrank para composiciones serán útiles para otras tareas más allá de enumerar tuplas. Por ejemplo, si tenemos tres polinomios $p_1 = \sum_{i=0}^{m} a_i x^i$, $p_2 = \sum_{i=0}^{m} b_i x^i$ y $p_3 = \sum_{i=0}^{m} c_i x^i$ y queremos obtener el polinomio producto $p_1 p_2 p_3 = \sum_{r=1}^{3m} d_r x^r$, entonces $d_r = \sum_{a_i+b_j+c_k=r} a_i b_j c_k$ y por tanto debemos obtener todas las composiciones de r en 3 partes para $r \leq m$ (ejemplo tomado de [18]).

Nuestra tarea en este capítulo es plantear un algoritmo para enumerar exhaustivamente cada diagonal, que comprende la parte más pesada para la enumeración de k-tuplas. Se plantearán los conceptos y propiedades que sustentan la corrección de la enumeración de k-tuplas. Se trabajará con el concepto de k-descomposición, el cual es clave para las enumeraciones y sus propiedades.

3.1. Composiciones de enteros

Encontrar la cantidad de composiciones de un entero n en $k+1$-partes es un problema combinatorio no complicado de resolver. Este problema está relacionado con otros problemas combinatorios, por mencionar algunos:

1. la cantidad de subconjuntos de tamaño k que hay en un conjunto de $n + k$ elementos,
2. la cantidad de multiconjuntos de tamaño k que hay en un conjunto de $n + 1$ elementos[1],
3. la cantidad de caminos de una esquina a su opuesta en una retícula cuadrada de $k \times n$ cuadros (equivalentemente $k + 1 \times n + 1$ vértices)[2],
4. la cantidad de k-tuplas no decrecientes de valores $\leq n$, entre otros.

En [16, 32, 19] se detallan más algunas de estas equivalencias. En [16, 18] se presentan algoritmos iterativos y en [17] hay algoritmos de *ranking* para algunas de las estructuras combinatorias antes mencionadas,

[1] Un multiconjunto es un conjunto que permite repeticiones de elementos. Por ejemplo, $\{1, 1, 2, 3\}$ y $\{2, 2, 3, 3\}$ son multiconjuntos de tamaño 4 del conjunto $\{1, 2, 3\}$.

[2] Los caminos son sobre artistas de la retícula y empiezan en la esquina inferior izquierda. Los caminos sólo pueden hacer avances ascendentes o a la derecha y nunca descendentes ni a la izquierda.

El concepto de composiciones tiene variantes, por ejemplo en [32], solicita que los valores de la composición sean positivos. Hay versiones donde se piden que estos valores se encuentren en un rango. Dado que buscamos hacer enumeraciones las cuales comienzan en 0, tomamos la versión más relajada de esta definición, la cual nos basta para definir una diagonal.

Definición 3.1.1. Sean $n, k \in \mathbb{N}$ con $k \geq 1$.

1. Definimos una *k-composición* o una *composición en k partes* de n como una k-tupla $(x_1, \ldots, x_k)$ de naturales tales que $\sum_{i=1}^{k} x_i = n$
2. Definimos la n-ésima *k-diagonal* como $\mathrm{Diag}_k(n) := \{(x_1, \ldots, x_k) \in \mathbb{N}^k \mid \sum_{i=1}^{k} x_i = n\}$.
3. Definimos el n-ésimo k-gnomon como $\mathrm{Gnom}_k(n) := \{m \in \mathbb{N} \mid S_k(n) \leq m < S_k(n+1)\}$.
4. Definimos el segmento inicial de n naturales como $I_n := \{m \in \mathbb{N} \mid m < n\}$.

Observación 3.1.2. Sean $n, k \in \mathbb{N}$. Las siguientes son sencillas de verificar:

1. $|\mathrm{Gnom}_{k+1}(n)| = S_{k+1}(n+1) - S_{k+1}(n)$, 2. $|I_n| = n$.

Proposición 3.1.3. *Sea $k \in \mathbb{N}$. Para cualquier $n \in \mathbb{N}$ la cantidad de $k+1$-composiciones de n, así como la cantidad de elementos en el n-ésmimo k-gnomon es $S_k(n+1)$. Esto es,*

$$|\mathrm{Diag}_{k+1}(n)| = S_k(n+1) = |\mathrm{Gnom}_{k+1}(n)|.$$

Demostración. Comencemos verificando la primera igualdad por inducción sobre k. Para $k = 0$ se tiene que $\mathrm{Diag}_1(n) = \{n\}$ y por tanto $|\mathrm{Diag}_1(n)| = 1 = S_0(n+1)$ para cualquier $n \in \mathbb{N}$.

Supongamos que para cualquier $n \in \mathbb{N}$ se sigue que $|\mathrm{Diag}_{k+1}(n)| = S_k(n+1)$ y veamos que se cumple que $|\mathrm{Diag}_{k+2}(n)| = S_{k+1}(n+1)$.

Notemos que si tenemos una $k+2$-composición $x_1 + \ldots + x_{k+2} = n$ y dejamos fija la última entrada, se convierte en una $k+1$-composición, esto es, tenemos que $x_1 + \ldots + x_{k+1} = n - x_{k+2}$ es una $k+1$-composición de $n - x_{k+2}$. En este sentido podemos particionar todas las $k+2$-composiciones en clases que coincidan en la última entrada.

Siendo así, consideremos los conjuntos A_i con $i \leq n$ definido como

$$A_i := \{(x_1, \ldots, x_{k+2}) \in \mathrm{Diag}_{k+2}(n) \mid x_{k+2} = i\}.$$

Mostremos que $\{A_i \mid i \leq n\}$ forma una partición de $\mathrm{Diag}_{k+2}(n)$. Por definición es claro que para toda $i \leq n$ se cumple que $A_i \subseteq \mathrm{Diag}_{k+2}(n)$ y por tanto $\bigcup_{i \leq n} A_i \subseteq \mathrm{Diag}_{k+2}(n)$.

Ahora bien, si $(x_1, \ldots, x_{k+2}) \in \mathrm{Diag}_{k+2}(n)$ entonces, dado que $\sum_{j=1}^{k+2} x_j = n$ y $x_j \in \mathbb{N}$ se cumple que $x_{k+2} \leq n$. De esta manera $(x_1, \ldots, x_{k+2}) \in A_{x_{k+2}}$ y por tanto $\bigcup_{i \leq n} A_i = \mathrm{Diag}_{k+2}(n)$. Si fuera el caso de que $(x_1, \ldots, x_{k+2}) \in A_i \cap A_j$ para $i, j \leq n$, entonces $i = x_{k+2} = j$, por lo que $i = j$ y así $\{A_i \mid i \leq n\}$ forma una partición.

Tomemos $i \leq n$, es decir $n - i \geq 0$. Veamos ahora que $|A_i| = |\mathrm{Diag}_{k+1}(n-i)|$ para toda $i \leq n$. Consideremos el encaje φ_i definido como:

$$\begin{aligned} \varphi_i : \mathrm{Diag}_{k+1}(n-i) &\hookrightarrow A_i \\ (x_1, \ldots, x_{k+1}) &\mapsto (x_1, \ldots, x_{k+1}, i) \end{aligned}$$

Observemos que $(x_1, \ldots, x_{k+1}) \in \mathrm{Diag}_{k+1}(n-i)$, y entonces $\sum_{j=1}^{k+1} x_j = n - i$ por lo que

$$\Big(\sum_{j=1}^{k+1} x_j\Big) + i = n, \quad \text{esto es,} (x_1, \ldots, x_{k+1}, i) = \varphi(x_1, \ldots, x_{k+1}) \in A_i$$

mostrándose que φ_i está bien definido.

No es complicado ver que φ_i es suprayectiva ya que para $(x_1, \ldots, x_{k+1}, i) \in A_i$ se tiene que $\varphi_i(x_1, \ldots, x_{k+1}) = (x_1, \ldots, x_{k+1}, i)$. Asimismo, si $\varphi_i(x_1, \ldots, x_{k+1}) = \varphi_i(y_1, \ldots, y_{k+1})$ entonces $(x_1, \ldots, x_{k+1}, i) = (y_1, \ldots, y_{k+1}, i)$. Mostrándose que $x_j = y_j$ para $1 \leq j \leq k+1$ y por tanto $(x_1, \ldots, x_{k+1}) = (y_1, \ldots, y_{k+1})$, de donde se concluye que φ_i es biyectiva.

Con base en lo anterior tenemos que

$$|\,\mathrm{Diag}_{k+2}(n)| = \Big|\bigcup_{i\leq n} A_i\Big| = \sum_{i\leq n} |A_i| = \sum_{i\leq n} |\,\mathrm{Diag}_{k+1}(n-i)| = \sum_{i\leq n} |\,\mathrm{Diag}_{k+1}(i)| =$$

$$\overset{\text{HI}}{=} \sum_{i\leq n} S_k(i+1) = \sum_{i\leq n+1} S_k(i) = S_{k+1}(n+1), \text{ por la proposición 2.1.9.}$$

De la observación 3.1.2 se sigue que $|\,\mathrm{Gnom}_{k+1}(n)| = S_{k+1}(n+1) - S_{k+1}(n) = S_k(n+1)$, esto último debido al corolario 2.1.8. QED

Proposición 3.1.4. *Para toda $k \in \mathbb{Z}^+$ se siguen las siguientes:*

1. $\{\,\mathrm{Diag}_k(n) \mid n \in \mathbb{N}\}$ *forma una partición de* $\mathbb{N}^k$*, y*
2. $\{\,\mathrm{Gnom}_k(n) \mid n \in \mathbb{N}\}$ *forma una partición de* $\mathbb{N}$*.*

Demostración. **(1)** Por un lado, $\mathrm{Diag}_k(n) \subseteq \mathbb{N}^k$ para toda $n \in \mathbb{N}$, de donde $\bigcup_{n\in\mathbb{N}} \mathrm{Diag}_k(n) \subseteq \mathbb{N}^k$. Por otro lado, tomemos $(x_1, \ldots, x_k) \in \mathbb{N}^k$ y sea $n := \sum_{i=1}^k x_i$, entonces $(x_1, \ldots, x_k) \in \mathrm{Diag}_k(n) \subseteq \bigcup_{n\in\mathbb{N}} \mathrm{Diag}_k(n)$. Por lo que $\bigcup_{n\in\mathbb{N}} \mathrm{Diag}_k(n) = \mathbb{N}^k$.

Si $(x_1, \ldots, x_k) \in \mathrm{Diag}_k(n) \cap \mathrm{Diag}_k(m)$ para $n, m \in \mathbb{N}$, entonces $m = \sum_{i=1}^k x_i = n$, lo que muestra lo buscado.

(2) Por definición, $\mathrm{Gnom}_k(n) \subseteq \mathbb{N}$ para toda $n \in \mathbb{N}$, por lo que $\bigcup_{n\in\mathbb{N}} \mathrm{Gnom}_k(n) \subseteq \mathbb{N}$.

Si $m \in \mathbb{N}$, entonces para $n := \llbracket m \rrbracket_k$ se sigue que $S_k(n) \leq m < S_k(n+1)$ y por tanto $m \in \mathrm{Gnom}_k(n)$ de donde se concluye que $\bigcup_{n\in\mathbb{N}} \mathrm{Gnom}_k(n) = \mathbb{N}$.

Si $m \in \mathrm{Gnom}_k(n) \cap \mathrm{Gnom}_k(p)$ entonces $n = \llbracket m \rrbracket_k = p$. Con lo anterior se llega a lo buscado. QED

3.1.1. Representación por barras y estrellas

Para enumerar todas las k-composiciones, damos una representación de éstas que hagan fácil esta labor. La representación por barras y estrellas[3] expondrá como hacer esta enumeración, al relacionar una composición con tuplas ascendentes. Esta representación también es útil para mostrar la relación entre una composición y otras estructuras combinatorias como las mencionadas al principio del capítulo.

Dado que en una composición es fundamental que los valores sumen n, podemos descomponer a este número en unidades (expresadas en estrellas), esto es, verlo como su representación unaria. Al representarlo de esta manera podemos generar una composición sencillamente particionando

[3] El planteamiento de barras y estrellas es de uso común en el razonamiento combinatorio. En [18] en la sección de composiciones se puede encontrar este concepto.

estas unidades en subconjuntos. Cada subconjunto representará un sumando de la composición. Dado que es relevante el orden en que se presentan los sumandos y no importa si hay sumandos repetidos, bastará con alinear las unidades y poner separadores (barras) entre estas unidades. Por ejemplo, la representación en barras y estrellas de la terna ordenada $(2, 1, 3)$ es ★★ | ★ | ★★★. Podemos observar que la tarea de generar todas las composiciones se reduce a seleccionar todas las posibles colocaciones de los separadores.

La cantidad de estrellas determina el número a componer n y la cantidad de barras más uno representa la cantidad de partes k. Si ponemos casillas que almacenen ya sea barras o estrellas, requeriremos $n + k - 1$ casillas describir cualquier k-composición de n. Dos ejemplos de esto se pueden apreciar en la figura 3.1. Si nos enfocamos en repartir las n las estrellas, podemos deducir que la cantidad de k-composiciones de n es $\binom{n+k-1}{n} = \binom{n+k-1}{k-1}$ que es igual $S_{k-1}(n+1)$ por la observación 2.1.5. Esto es otra forma de mostrar la proposición 3.1.3.

$(1,1,0,3)$	$\longrightarrow$	★	|	★	|	|	★	★	★	$\longrightarrow$ $(1,2,2)$
		1	2	3	4	5	6	7	8	
$(2,0,2,1)$	$\longrightarrow$	★	★	|	|	★	★	|	★	$\longrightarrow$ $(2,2,4)$
		1	2	3	4	5	6	7	8	

(a) 4-tuplas (b) Representación barras y estrellas (c) Sepraradores

Figura 3.1: Ejemplos de representaciones de dos 4-tuplas

Si ordenamos un conjunto, y la cantidad de estrellas antes del primer separador lo tomamos como la cantidad de repeticiones del primer elemento del conjunto, la cantidad de estrellas entre el primer y segundo de separador como las repeticiones del segundo y proseguimos sucesivamente, entonces tenemos un multiconjunto de un tamaño deseado del conjunto inicialmente tomado. Por ejemplo para $\{a_1, a_2, a_3, a_4\}$ el diagrama $\underbrace{★★}_{a_1} \mid \underbrace{\quad}_{a_2} \mid \underbrace{★}_{a_3} \mid \underbrace{★★}_{a_4}$ representa el multiconjunto $\{a_1, a_1, a_3, a_4, a_4\}$.

Ahora bien, sabiendo que los separadores determinan completamente una k-composición de n, basta con codificar las colocaciones de los separadores. Una forma adecuada es almacenando en una tupla la cantidad de estrellas anteriores a cada separador. En la figura 3.1c se ven ejemplos de la representación por separadores de las tuplas en 3.1a. Esto muestra la relación entre una composición y una tupla ascendiente, relación que estudiaremos más a fondo.

Definición 3.1.5. Sean $n, k \in \mathbb{N}$ y $(x_1, \ldots, x_k, x_{k+1})$ una $k+1$-composición de n. Decimos que $(a_1, \ldots, a_k) \in \mathbb{N}^k$ es la *representación por separadores* de $(x_1, \ldots, x_{k+1})$ si para cualquier $j \in \{1, \ldots, k\}$ se cumple que $a_j = \sum_{i=1}^{j} x_j$.

Definamos a $\mathrm{Sep}_k(n)$ como el conjunto de todas tuplas $(a_1, \ldots, a_k)$ que son la representación por separadores de alguna $k+1$-composición de n.

Proposición 3.1.6. *Sean $n, k \in \mathbb{N}$ con $k \geq 1$, $(x_1, \ldots, x_k, x_{k+1})$ una $k+1$-composición de n y $(a_1, \ldots, a_k)$ su representación por separadores. Se sigue que $x_1 = a_1$ y para toda $j \in \{1, \ldots, k-1\}$ se cumple que $x_{j+1} = a_{j+1} - a_j$. Finalmente también se cumple que $x_{k+1} = n - a_k$.*

Demostración. Procedamos por inducción sobre k. Para $k = 1$ se tiene que $a_1 = \sum_{i=1}^{1} x_i = x_1$. Dado que $k = 1$ se sigue que para toda $j \in \{1, \ldots, k-1\}$ se cumple que $x_{j+1} = a_{j+1} - a_j$ por vacuidad. Finalmente, como $x_1 + x_2 = n$ entonces $x_2 = n - x_1 = n - a_1$.

Supongamos que para toda n y toda $k+1$-composición se sigue el resultado. Veamos que el resultado se sostiene para $k+1$. Sea $n \in \mathbb{N}$ y tomemos $(a_1, \ldots, a_{k+1})$ una $k+2$-composición de n y mostremos el resultado.

Definamos $n_0 := \sum_{i=1}^{k+1} x_i$. Claramente $(x_1, \ldots, x_{k+1})$ es una $k+1$-composición de n_0. Por definición de representación por separadores, en particular se tiene que para toda $j \in \{1, \ldots, k\}$ se cumple que $x_j = \sum_{i=1}^{j} a_i$, por lo que $(a_1, \ldots, a_k)$ es una representación por separadores de n_0. En este sentido, por hipótesis inductiva se tiene que $x_1 = a_1$ y para toda $j \in \{1, \ldots, k-1\}$ pasa que $x_{j+1} = a_{j+1} - a_j$. Además

$$x_{k+1} = n_0 - a_k = \Big(\sum_{i=1}^{k+1} x_i\Big) - a_k = a_{k+1} - a_k.$$

Con lo anterior también se tiene que

$$n = \sum_{i=1}^{k+2} x_i = \Big(\sum_{i=1}^{k+1} x_i\Big) + x_{k+2} = a_{k+1} + x_{k+2} \quad \text{por lo que} \quad x_{k+2} = n - a_{k+1}.$$

QED

Observación 3.1.7. Notemos que por definición que cada representación por separadores proviene de una composición. Esta representación no es única, por ejemplo $(1, 1)$ es una representación por separadores tanto de la 3-composición de 2 dada por $(1, 1, 0)$ así como de la 3-composición de 3 dada por $(1, 1, 1)$. Sin embargo, si dejamos fijo el entero n la representación sí se vuelve única.

Proposición 3.1.8. *Sean $n, k \in \mathbb{N}$ con $k \geq 1$.*

1. *Si $(a_1, \ldots, a_k)$ y $(b_1, \ldots, b_k)$ son representaciones por separadores de una $k+1$-composición $(x_1, \ldots, x_{k+1})$ entonces $a_j = b_j$ para toda $j \in \{1, \ldots, k\}$.*

2. *Si $(x_1, \ldots, x_{k+1})$ y $(y_1, \ldots, y_{k+1})$ son $k+1$-composiciones de n que cumplen que tienen la misma representación por separadores $(a_1, \ldots, a_k)$, entonces $x_j = y_j$ para toda $j \in \{1, \ldots, k+1\}$.*

Demostración. **(1)** Es claro que por definición se tiene que $a_j = \sum_{i=1}^{j} x_i = b_j$ para toda $j \in \{1, \ldots, k\}$.

(2) Por la proposición 3.1.6 se tiene que $x_1 = a_1 = y_1$ y además para toda $j \in \{1, \ldots, k-1\}$ se tiene que $x_{j+1} = a_{j+1} - a_j = y_{j+1}$. Finalmente $x_{k+1} = n - a_k = y_{k+1}$. QED

Observación 3.1.9. Es importante notar que de la definición 3.1.5 se sigue que para cualquier $k+1$-composición de un natural n existe una representación por separadores de ésta. La proposición anterior (3.1.8) afirma que esta representación es única. Para cada representación por separadores existe una única $k+1$-composición que le corresponda.

Esto establece una biyección $\psi_{k,n} : \mathrm{Diag}_{k+1}(n) \longrightarrow \mathrm{Sep}_k(n)$ entre estos conjuntos de tuplas que a cada composición le asocia su representación por separadores. Los algoritmos Comp2Sep y Sep2Comp computan esta función y su inversa respectivamente.

Por otro lado, podemos observar que toda representación por separadores es una tupla con valores no decrecientes acotada por n. Algo interesante es que toda tupla no decreciente es a su vez un a representación por separadores. Esto es, hay una biyección entre composiciones y

tuplas no decrecientes. El siguiente teorema muestra la equivalencia entre separadores y tuplas no decrecientes.

Teorema 3.1.10. *Sean $n, k \in \mathbb{N}$ y $(a_1, \ldots, a_k) \in \mathbb{N}^k$ con $k \geq 1$. Se tiene que $(a_1, \ldots, a_k)$ es una representación por separadores de una $k+1$-composición de n si y sólo si para toda $1 \leq p \leq q \leq k$ se cumple que $0 \leq a_p \leq a_q \leq n$.*

Demostración. Para la ida sea $(x_1, \ldots, x_{k+1})$ una $k+1$-composición de n tal que $a_j = \sum_{i=1}^{j} x_i$ para toda $j \in \{1, \ldots, k\}$. Si tomamos $1 \leq p \leq q \leq k$, debido a que $x_p \in \mathbb{N}$ es claro que

$$0 \leq \sum_{i=1}^{p} x_i \leq \sum_{i=1}^{q} x_i \leq \sum_{i=1}^{k} x_i = n, \qquad \text{es decir,}$$
$$0 \leq a_p \leq a_q \leq n$$

Para el regreso, tomemos una tupla $(a_1, \ldots, a_k)$ que cumpla que $0 \leq a_p \leq a_q \leq n$ para $1 \leq p \leq q \leq k$. Definamos $x_1 = a_1$, $x_j := a_{j+1} - a_j$ para toda $j \in \{1, \ldots, k\}$ y $x_{k+1} := n - a_k$.

No es difícil ver que $j \in \{1, \ldots, k+1\}$ se sigue que $x_j \geq 0$. Se tiene así una suma telescópica que suma n.

$$\sum_{i=1}^{k+1} x_i = (n - a_k) + (a_k - a_{k-1}) + \cdots + (a_2 - a_1) + a_1 = n.$$

Por lo tanto $(x_1, \ldots, x_{k+1})$ es una $k+1$-composición de n.

QED

3.1.2. Algoritmos de transformación de representaciones

Antes de mostrar como enumerar todas las k-composiciones, planteemos dos algoritmos para transformar una representación en otra y mostremos que son correctos. Estos algoritmos calculan una función biyectiva entre las k-composiciones y su representación por separadores.

Algoritmo 7 COMP2SEP$((x_1, \ldots, x_{k+1}))$

Entrada: Una $k+1$-composición de un valor n $(x_1, \ldots, x_{k+1}) \in \text{Diag}_{k+1}(n)$.
Salida: La representación por separadores de $(x_1, \ldots, x_{k+1})$ y n.

1: $a_1 \leftarrow x_1$
2: **for** $i \leftarrow 1$ **upto** k **do**
3: $\quad a_{i+1} \leftarrow a_i + x_{i+1}$
4: **end for**
5: **return** $(a_1, \ldots, a_k), a_{k+1}$

Proposición 3.1.11. *Si la entrada es una $k+1$-composición $(x_1, \ldots, x_{k+1})$ de un número n, entonces el algoritmo* COMP2SEP*(7) termina y devuelve su correspondiente representación por separadores, así como n.*

Demostración. Observemos primero que la línea 1 es una asignación y el ciclo for de la línea 2-4 solo hace una asignación y su cota jamás es modificada por lo que necesariamente termina. De esta manera el algoritmo termina.

Sea $(a_1, \ldots, a_k), a_{k+1}$ la salida del algoritmo, y mostremos que $(a_1, \ldots, a_k)$ es la representación por separadores de $(x_1, \ldots, x_{k+1})$.

Veamos que para toda $i \in \{1, \ldots, k+1\}$ sucede que la variable $a_i = \sum_{j=1}^{i} x_i$. Procedamos por inducción sobre i. Para $i = 1$ por la línea 1 se sigue que $a_1 = x_1 = \sum_{j=1}^{1} x_j$. Supongamos que el resultado se sigue para i y veamos que se sigue para $i+1$ con $i+1 \leq k+1$. Se puede ver que la variable a_{i+1} solo será modificada en la interación $i+1$ del ciclo for de las líneas 2-4. Siendo así, por nuestra hipótesis inductiva se sigue que $a_i = \sum_{j=1}^{i} x_j$. De la línea 3 es inmediato ver que

$$a_{i+1} = a_i + x_{i+1} = \Big(\sum_{j=1}^{i} x_j\Big) + x_{i+1} = \sum_{j=1}^{i+1} x_j.$$

Finalmente, de lo anterior para $i = k+1$ se sigue que. $a_{k+1} = \sum_{j=1}^{k+1} x_j = n_0$, como se quería mostrar. QED

Proposición 3.1.12. *El algoritmo* Comp2Sep*(7) tiene complejidad en tiempo $O(k)$ y complejidad en espacio $O(k)$.*

Demostración. Para la complejidad en tiempo es fácil notar que la asignación en la línea 1 toma tiempo constante. Más aún, el ciclo for de las líneas 2-4 solo hace una suma y una asignación, por lo que cada iteración toma el mismo tiempo constante. Recordando que la cantidad de iteraciones de este ciclo jamás es alterada siendo un total de k iteraciones, se concluye que la complejidad en tiempo es $O(k)$.

Para medir el espacio empleado por este algoritmo se contemplan las variables a_1 así como todas las variables generadas en el ciclo for de las líneas 2-4. No es complicado ver que en cada iteración se genera una nueva variable, por lo que, al finalizar el algoritmo, se habrán planteado $k+1$ variables nuevas. Esto es, la complejidad en tiempo es $O(k)$. QED

Algoritmo 8 Sep2Comp$((a_1, \ldots, a_k), n)$

Entrada: La representación por separadores $(a_1, \ldots, a_k)$ de una $k+1$-composición de n, así como el valor n.

Salida: La $k+1$-composición de n con representación por separadores $(a_1, \ldots, a_k)$.

1: $a_0 \leftarrow 0$
2: $a_{k+1} \leftarrow n$
3: **for** $i \leftarrow 0$ **upto** k **do**
4: $\quad x_{i+1} \leftarrow a_{i+1} - a_i$
5: **end for**
6: **return** $(x_1, \ldots, x_{k+1})$

Proposición 3.1.13. *Si la entrada es $(a_1, \ldots, a_k), n$ donde $(a_1, \ldots, a_k)$ es la representación por separadores de una k-composición de n, entonces el algoritmo* Sep2Comp*(8) termina y devuelve tal $k+1$-composición de n.*

Demostración. De forma semejante al algoritmo anterior, el ciclo for de la línea 3-5 solo hace una asignación y su cota jamás es modificada por lo que necesariamente termina. Dado que en la línea 1 y 2 se define valores para a_0 y a_{k+1}, el ciclo for no desborda los índices en sus asignaciones.

Sea $(x_1, \ldots, x_{k+1})$ la salida del algoritmo. Se tiene que $x_1 = a_1 - a_0 = a_1 - 0 = a_1$. Más aún, para toda $i \in \{1, \ldots, k\}$ se sigue que $x_{i+1} = a_{i+1} - a_i$. Finalmente, en la k-ésima iteración del ciclo for se asigna a $x_{k+1} = a_{k+1} - a_k = n - a_k$.

En virtud de la proposición 3.1.6, y dado que para cada representación por separadores existe una única $k+1$-composición como se plantea en la proposición 3.1.8 se concluye que

$(x_1, \ldots, x_{k+1})$ es la $k+1$-composición que corresponde con la entrada. Esto muestra que el algoritmo es correcto. QED

Proposición 3.1.14. *La complejidad en tiempo y en espacio del algoritmo* Sep2Comp *(8) son ambas* $O(k)$.

Demostración. Para la complejidad en tiempo veamos que las líneas 1 y 2 toman tiempo constantes por ser meras asignaciones. En seguida, el ciclo for de las líneas 3-4 realiza una resta y una asignación por lo que toma el mismo tiempo constante para cada iteración. Observando que este ciclo hace $k+1$ iteraciones, se concluye que el algoritmo toma tiempo $O(k)$.

Para la complejidad en espacio se contemplan las dos variables auxiliares de las líneas 1 y 2, y además en cada iteración del ciclo de las líneas 3-5 se genera una nueva variable, resultando en $k+3$ variables empleadas en este proceso. Por lo que la complejidad en espacio es $O(k)$. QED

Proposición 3.1.15. *Las funciones definidas mediante los algoritmos* Comp2Sep *y* Sep2Comp *(7 y 8) son una inversa de la otra, y por tanto biyectivas.*

Demostración. Por la proposición 3.1.8, sabemos que por cada $k+1$ composición existe una única representación por separadores y vicecersa. Por la proposición 3.1.13 sabemos que el algoritmo Sep2Comp necesariamente devuelve la $k+1$-composición correspondiente a la entrada. Si a esta salida le aplicamos el algoritmo Comp2Sep, por la proposición 3.1.11 la nueva salida será exactamente su representación por separadores.

Análogamente en el otro sentido, mostrándose que Comp2Sep y Sep2Comp son una inversa de la otra. QED

3.2. k-descomposiciones

Mostremos como descomponer un entero como suma de números simplex. Este concepto es clave en la enumeración de tuplas ya que la función Rank para tuplas se puede reducir a manejar una k-descomposición. Este concepto ha sido planteado en la literatura de diversas maneras y bajo distintos nombres. En [25] llaman a esta descomposición como *sistema numérico combinatorio* ("*combinatorial number system*"), en [22] como *representación k-canónica* ("*k-canonical representation*"), en [26] como *k-ésima representación de Macaulay* ("*k-th Macaulay representation*") y en [27] como *forma de k-cascada* ("*k-cascade form*") por mencionar algunos. Estas definiciones trabajan en esencia el mismo concepto, con ligeras variantes para acoplarse a lo buscado. Particularmente, se trabaja con esta representación mediante coeficientes binomiales y no como números simplex. Muchas de las propiedades de estas representaciones se mencionan, pero no se prueban con detalle. En esta sección demostraremos varias propiedades de una k-descomposición visto desde la perspectiva de un número simplex con el fin de dar una intuición geométrica al problema.

Descomponer un número como suma de simplex es semejante a una división entera. La división de a entre b consiste en encontrar dos valores q (cociente) y r (residuo) tales que $a = bq+r$ donde $r < q$. En este caso buscamos el múltiplo de b más cercano a a. Esto lo podemos hacer con simplex en lugar de múltiplos.

Si dejamos fijo un valor k, notemos que podemos descomponer a todo número m como el n-ésimo número k-simplex más un valor d para alguna n. Para este caso n puede tomar el papel de cociente y d de residuo. Sabemos que hay varias descomposiciones de $a = bq + r$, pero para el caso de la división nos enfocamos en la que maximiza el valor del cociente. De aquí que se solicite que $r < q$.

Como no es difícil de observar hay varias descomposiciones de m como $m = S_k(n) + d$ para k fija. Por ejemplo $11 = S_3(2) + 7 = 4 + 7$ y $11 = S_3(3) + 1 = 10 + 1$. Semejante a la división, nos enfocamos en la que maximice n, por lo que buscamos expresar a m como $m = S_k(n) + d$ donde $n = \lfloor\!\lfloor m \rfloor\!\rfloor_k$. Teniendo esta descomposición, el residuo d será acotado por un simplex de con dimensión menor pero índice sucesor.

Proposición 3.2.1. *Para cualesquiera $n, d \in \mathbb{N}$ se tiene que $\lfloor\!\lfloor S_{k+1}(n) + d \rfloor\!\rfloor_{k+1} = n$ si y sólo si $d < S_k(n+1)$.*

Demostración. Supongamos primero que $\lfloor\!\lfloor S_{k+1}(n)+d \rfloor\!\rfloor_{k+1} = n$. Entonces $S_k(n) \leq S_{k+1}(n)+d < S_{k+1}(n+1)$ y utilizando el corolario 2.1.8 se obtiene $S_{k+1}(n) + d < S_{k+1}(n) + S_k(n+1)$. Esto último implica que $d < S_k(n+1)$.

Por otro lado, si suponemos que $d < S_k(n+1)$, entonces sumando $S_{k+1}(n)$ de ambos lados y aplicando el corolario 2.1.8 tenemos que $S_{k+1}(n) + d < S_{k+1}(n) + S_k(n+1) = S_{k+1}(n+1)$. Finalmente, es claro que $S_{k+1}(n) \leq S_{k+1}(n)+d$ por lo que $S_{k+1}(n) \leq S_{k+1}(n)+d < S_{k+1}(n+1)$, es decir, $\lfloor\!\lfloor S_{k+1}(n) + d \rfloor\!\rfloor_{k+1} = n$. QED

Proposición 3.2.2. *Si $n, m, k \in \mathbb{N}$ y $n = \lfloor\!\lfloor m \rfloor\!\rfloor_{k+1}$ entonces $0 \leq m - S_{k+1}(n) < S_k(n+1)$.*

Demostración. Supongamos que $n = \lfloor\!\lfloor m \rfloor\!\rfloor_{k+1}$. Por definición se sigue que $S_{k+1}(n) \leq m < S_{k+1}(n+1)$. Por el corolario 2.1.8 se tiene que $S_{k+1}(n) \leq m < S_k(n+1) + S_{k+1}(n)$. Restando $S_{k+1}(n)$ a la desigualdad se tiene que $0 \leq m - S_{k+1}(n) < S_k(n+1)$. QED

Corolario 3.2.3. *Para cualesquiera $n, k \in \mathbb{N}$ se cumple que* $\text{Gnom}_{k+1}(n) = \{a \in \mathbb{N} \mid \lfloor\!\lfloor a \rfloor\!\rfloor_{k+1} = n\}$.

Resulta interesante mostrar una similitud entre la descomposición de un número como suma de simplex con el sistema usual de representación numérica por cifras. Si nos enfocamos en la representación decimal de un número, por ejemplo $m = [235]_{10}$, recordaremos que la cifra '2' representa el múltiplo no cero menor a la base de la mayor potencia de 10 que divide a m. En este caso $m = (2 \times 10^2) + r$ con $r < 10^2$. Esta última condición sobre r hace que podamos repetir este procedimiento hasta agotar las potencias de 10 y obtener todas las cifras de m. Esto es, $m = 2 \times 10^2 + 3 \times 10^1 + 5 \times 10^0 = [235]_{10}$. Las potencias de un número se pueden pensar como números politopos correspondientes a la generalización de un número cuadrado a mayores dimensiones (un segmento, un cuadrado, un cubo, un teseracto para dimensiones 1,2,3 y 4 respectivamente).

Esta propiedad numérica (e incluso geométrica) la podemos aplicar sobre m, pero en lugar de utilizar las potencias de alguna base como es usual, podemos usar números simplex.

En el ejemplo anterior, $235 = S_4(7) + d = 210 + 25$ con $d = 25$. Repitiendo para d tenemos que $235 = S_4(7) + S_3(4) + S_2(2) + S_1(2) = 210 + 20 + 3 + 2$. De esta manera podemos codificar en cifras al valor 235 como suma de simplex con la tupla $(2, 2, 4, 7)$ (nótese que las entradas no necesariamente serán de una cifra). Convenimos escribir esta representación con el índice del simplex de menor a mayor dimensión, para que la i-ésima entrada de la tupla sea el índice del i-simplex correspondiente. Esto también hará concordancia con la representación por separadores de una composición como se verá más adelante.

Al igual que un mismo número tiene múltiples representaciones por cifras dependiendo de la base, por ejemplo $[235]_{10} = [353]_8 = [11101011]_2$, cada número tiene múltiples representaciones como suma de simplex dependiendo de la dimensión k máxima convenida. Con este mismo ejemplo tenemos que $235 = S_5(5) + S_4(5) + S_3(5) + S_2(2) + S_1(1) = 126 + 70 + 35 + 3 + 1$ correspondiente a $(1, 2, 5, 5, 5)$ y también la representación previamente obtenida $(2, 2, 4, 7)$.

En la tabla 3.1 podemos apreciar distintas descomposiciones del número 19 y su correspondiente codificación. Se puede apreciar claramente que extraer el simplex que antecede, el residuo puede manejarse como una versión reducida del mismo problema con un simplex de menor dimensión. Se mencionó cuando se definió a los simplex que el gnomon correspondiente de estos números politopo es a su vez un simplex, cualidad que explotamos justamente en este proceso de descomposición.

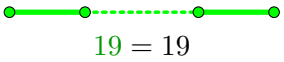
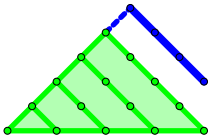
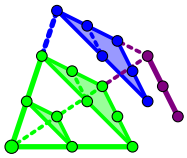
k	Simplex	Completo	k-descomposición
1	$S_1(19) = 19$	 $19 = 19$	(19)
2	$S_2(5) = 15$ $S_1(4) = 4$	 $4 + 15 = 19$	$(4, 5)$
3	$S_3(3) = 10$ $S_2(3) = 6$ $S_1(3) = 3$	 $3 + 6 + 10 = 19$	$(3, 3, 3)$

Tabla 3.1: Representaciones geométricas de k-descomposiciones de 19 para $k = 1, 2, 3$.

En virtud de la proposición 3.2.1 tenemos la condición para el residuo para asegurar tener la descomposición de un número m como suma de simplex. Con la intuición del ejemplo de la tabla 3.1, tiene lugar la siguiente definición.

Definición 3.2.4. Sea $m, k \in \mathbb{N}$ con $k \geq 1$. Definimos la *k-descomposición* de m como una k-tupla $(n_1, \ldots, n_k)$ tal que $m = \sum_{i=1}^{k} S_i(n_i)$ y para cualquier $j \in \{1, \ldots, k-1\}$ se cumpla que $\sum_{i=1}^{j} S_i(n_i) < S_j(n_{j+1} + 1)$.

Definimos $\mathrm{Desc}_k(n)$ como el conjunto k-descomposiciones de un valor $m < S_k(n+1)$.

Proposición 3.2.5. *Para cualesquiera $m, k \in \mathbb{N}$ con $k \geq 1$ existe una k-descomposición de m.*

Demostración. Procedamos por inducción fuerte sobre m. Para $m = 0$, podemos definir la tupla $(0, \ldots, 0)$ la cual claramente cumple que $\sum_{i=1}^{k} S_1(0) = 0$ y además $\sum_{i=1}^{j} S_1(0) = 0 < 1 = S_j(1)$ para cualquier $j \in \{1, \ldots, k-1\}$ por la observación 2.1.5.

Tomemos $m > 0$. Supongamos que para todo valor menor a m se sigue que para toda k con $k \geq 1$ tienen una k-descomposición.

Sea $k \in \mathbb{N}$ con $k \geq 1$, veamos que m tiene una k-descomposición. Si $k = 1$ entonces m mismo es una 1-descomposición de sí mismo. Asumamos que $k \geq 2$. Tomemos $n_k := \llbracket m \rrbracket_k$ y definamos $d := m - S_k(n_k)$. De aquí $m = d + S_k(n_k)$, más aún, $d < m$ pues $m > 0$ por lo que podemos aplicarle la hipótesis inductiva a d.

Dado que $k \geq 2$ entonces $k-1 \geq 1$ por lo que para d debe de existir una $k-1$-descomposición, digamos $(n_1, \ldots, n_{k-1})$. Veamos que $(n_1, \ldots, n_k)$ es una k-descomposición de m. Tenemos que

$$\sum_{i=1}^{k-1} S_i(n_i) = d \text{ por ser } k\text{-descomposición por lo que}$$
$$\sum_{i=1}^{k} S_i(n_i) = \Big(\sum_{i=1}^{k-1} S_i(n_i)\Big) + S_k(n_k) = d + S_k(n_k) = m$$

Tenemos también que para toda $j \in \{1,\ldots,k-1\}$ se cumple que $\sum_{i=1}^{j} S_i(n_i) < S_j(n_{j+1}+1)$. Por la proposición 3.2.1 se cumple que $d < S_k(n_k+1)$. lo que implica que $\sum_{i=1}^{k-1} S_i(n_i) < S_k(n_k+1)$ con lo que hemos mostrado que $(n_1,\ldots,n_k)$ es una k-descomposición de m. QED

Teorema 3.2.6. *Sea $m,k \in \mathbb{N}$ con $k \geq 1$. Si $(n_1,\ldots,n_k)$ es una k-descomposición de alguna m y $d_j = \sum_{i=1}^{j} S_i(n_i)$ para toda $j \in \{1,\ldots,k\}$, entonces $\llbracket d_j \rrbracket_j = n_j$ para toda $j \in \{1,\ldots,k\}$.*

Demostración. Por inducción sobre k. Para $k = 1$ tomemos (n_1) una 1-descomposición de m. Evidentemente $\llbracket m \rrbracket_1 = m = S_1(m) = n_1$.

Supongamos que para cualquier k-descomposición se sigue el resultado y veamos que se cumple para cualquier $k+1$-descomposición. Tomemos $(n_1,\ldots,n_k,n_{k+1})$ una $k+1$-descomposición de un valor m. Observemos que $d_{k+1} = m$ por definición.

Si tomamos $d_k = \sum_{i=1}^{k} S_i(n_i)$ se sigue que $(n_1,\ldots,n_k)$ es una k-descomposición de m_k que coincide en los valores d_i para $i \in \{1,\ldots,k-1\}$. Por hipótesis de inducción se tiene que $\llbracket d_i \rrbracket_i = n_i$ para toda $i \in \{1,\ldots,k\}$.

Mostremos que $\llbracket d_{k+1} \rrbracket_{k+1} = n_{k+1}$. Por definición se sigue

$$m = \sum_{i=1}^{k+1} S_i(n_i) = S_{k+1}(n_{k+1}) + \sum_{i=1}^{k} S_i(n_i) = S_{k+1}(n_{k+1}) + d_k \quad \text{donde } d_k < S_k(n_{k+1}+1).$$

Siendo así, por la proposición 3.2.1 se tiene que

$$\llbracket d_{k+1} \rrbracket_{k+1} = \llbracket m \rrbracket_{k+1} = \llbracket S_{k+1}(n_{k+1}) + d_k \rrbracket_{k+1} = n_{k+1}.$$

QED

Corolario 3.2.7. *Sean $m,k \in \mathbb{N}$ y $(n_1,\ldots,n_k) \in \mathbb{N}^k$ con $k \geq 1$. Si $(n_1,\ldots,n_k)$ es una k-descomposición de m, entonces $\llbracket m \rrbracket_k = n_k$ y por tanto $S_k(n_k) \leq m < S_k(n_k+1)$.*

Demostración. Resultado directo del teorema 3.2.6 para $j = k$. QED

Los teoremas 3.2.8 y 3.2.9 se mencionan brevemente en [26, 27] pero no se prueban. En seguida demostramos estas propiedades.

Teorema 3.2.8. *Sean $m,\hat{m},k \in \mathbb{N}$ con $k \geq 1$, se cumplen las siguientes.*

1. *Existe una única k-descomposición de m.*
2. *Si $(n_1,\ldots,n_k)$ es una k-descomposición tanto de m como de $\hat{m}$ entonces $m = \hat{m}$.*

Demostración. **(1)** La existencia de una k-descomposición es exactamente la proposición 3.2.5. Mostremos la unicidad. Procedamos por inducción sobre k. Para $k = 1$ se tiene que si (n_1) y $(\hat{n}_1)$ son 1-descomposiciones de m entonces $n_1 = S_1(n_1) = m = S_1(\hat{n}_1) = \hat{n}_1$.

Supongamos que se cumple el resultado para k y veamos que se sostiene para $k+1$. Tomemos $m \in \mathbb{N}$ y $(n_1, \ldots, n_{k+1})$ y $(\hat{n}_1, \ldots, \hat{n}_{k+1})$ dos $k+1$-descomposiciones de m. Tenemos que

$$S_{k+1}(n_{k+1}) + \sum_{i=1}^{k} S_i(n_i) = m = S_{k+1}(\hat{n}_{k+1}) + \sum_{i=1}^{k} S_i(\hat{n}_i)$$

y entonces por el teorema 3.2.6 se tiene que

$$n_{k+1} = \left\| S_{k+1}(n_{k+1}) + \sum_{i=1}^{k} S_i(n_i) \right\|_{k+1} = \|m\|_{k+1} = \left\| S_{k+1}(\hat{n}_{k+1}) + \sum_{i=1}^{k} S_i(\hat{n}_i) \right\|_{k+1} = \hat{n}_{k+1}.$$

Debido a que $n_{k+1} = \hat{n}_{k+1}$ tenemos que

$$\sum_{i=1}^{k} S_i(n_i) = m - S_{k+1}(n_{k+1}) = m - S_{k+1}(\hat{n}_{k+1}) = \sum_{i=1}^{k} S_i(\hat{n}_i).$$

De esta manera podemos definir a $m_0 := \sum_{i=1}^{k} S_i(n_i) = \sum_{i=1}^{k} S_i(\hat{n}_i)$ y claramente $(n_1, \ldots, n_k)$ y $(\hat{n}_1, \ldots, \hat{n}_k)$ son k-descomposiciones de m_0. Por hipótesis inductiva se tiene que $n_i = \hat{n}_i$ para toda $i \in \{1, \ldots, k\}$. Así hemos mostrado que $n_i = \hat{n}_i$ para toda $i \in \{1, \ldots, k+1\}$, como se quería probar.

(2) Por definición de k-descomposición se sigue que $m = \sum_{i=1}^{k} S_i(n_i) = \hat{m}$. QED

Teorema 3.2.9. *Sean $n, k \in \mathbb{N}$ con $k \geq 1$ y $(n_1, \ldots, n_k) \in \mathbb{N}^k$. Se tiene que existe $m \in \mathbb{N}$ tal que $(n_1, \ldots, n_k)$ es una k-descomposición y $m < S_k(n+1)$ si y sólo si para toda $1 \leq i \leq j \leq k$ se cumple que $n_i \leq n_j \leq n$.*

Demostración. Procedamos por inducción sobre k.
Caso Base. Veamos la equivalencia para $k = 1$.

$\boxed{\Rightarrow}$ Sea $m < S_{k+1}(n+1)$ y (n_1) una 1-descomposición de m. Tenemos que $m < n+1$, esto es $m \leq n$. También se tiene que $n_1 = S_1(n_1) = m$ por ser una descomposición. Así $n_1 \leq n$.

$\boxed{\Leftarrow}$ Si $1 \leq n_1 \leq n$, entonces definimos $m := S_1(n_1) = n_1$. Claramente (n_1) es una 1-descomposición de m. Más aún, como $n_1 \leq n$, entonces $m = n_1 < n+1 = S_1(n+1)$.

Caso inductivo. Supongamos que el resultado se cumple para k y mostremos la equivalencia para $k+1$.

$\boxed{\Rightarrow}$ Sea $m < S_{k+1}(n+1)$ y $(n_1, \ldots, n_{k+1})$ una $k+1$-descomposición de m.

Por definición de sigue que

$$\sum_{i=1}^{k+1} S_i(n_i) = m \quad \text{y} \quad \sum_{i=1}^{j} S_i(n_i) < S_j(n_{j+1}+1) \text{ para toda } j \in \{1, \ldots, k\}. \tag{3.1}$$

Sea $d := \sum_{i=1}^{k} S_i(n_i)$. Es claro que de (3.1) se sigue que $(n_1, \ldots, n_k)$ es una k-descomposición de d. Incluso $d < S_k(n_{k+1}+1)$ para $j = k$ en (3.1).

Aplicando la hipótesis inductiva con d, n_{k+1} y $(n_1, \ldots, n_{k+1})$, se sigue que $1 \leq n_i \leq n_j \leq n_{k+1}$ para toda $1 \leq i \leq j \leq k$. Basta mostrar que $n_{k+1} \leq n$. Tenemos que

$$m = \sum_{i=1}^{k+1} S_i(n_i) < S_{k+1}(n_{k+1}) \text{ y } m < S_{k+1}(n+1), \text{ entonces } d + S_{k+1}(n_{k+1}) < S_{k+1}(n+1).$$

Por lo tanto $S_{k+1}(n_{k+1}) < S_{k+1}(n+1)$ y de la proposición 2.1.13 se obtiene $n_{k+1} < n+1$, es decir $n_{k+1} \leq n$.

$\boxed{\Leftarrow}$ Sea $(n_1, \ldots, n_{k+1})$ tal que $n_i \leq n_j \leq n$ cuando $1 \leq i \leq j \leq k+1$.

Por hipótesis inductiva, $(n_1, \ldots, n_k)$ es k-descomposición de alguna $d < S_k(n+1)$. Por definición

$$d = \sum_{i=1}^{k} S_i(n_i) \quad \text{y} \quad \sum_{i=1}^{j} S_i(n_i) < S_j(n_{j+1}+1) \text{ para toda } j \in \{1, \ldots, k-1\}. \tag{3.2}$$

Sea $m := d + S_{k+1}(n_{k+1}) = \sum_{i=1}^{k} S_i(n_i) + S_{k+1}(n_{k+1})$. Mostremos que $(n_1, \ldots, n_{k+1})$ es una $k+1$-descomposición de m. Para esto, por la ecuación (3.2), basta mostrar que $d < S_k(n_{k+1}+1)$. También buscamos mostrar que $m < S_{k+1}(n+1)$.

Como $n_i \leq n_{k+1} \leq n$ para $1 \leq i \leq k$ se sigue de la proposición 2.1.13 que $S_i(n_i) \leq S_i(n_{k+1}) \leq S_i(n)$. Sumando sobre i cada una de las desigualdades anteriores y aplicando la proposición 2.1.10 se obtiene

$$d = \sum_{i=1}^{k} S_i(n_i) \leq \sum_{i=1}^{k} S_i(n_{k+1}) < \sum_{i=1}^{k} S_i(n_{k+1}) + 1 = S_k(n_{k+1}+1) \quad \text{y}$$
$$m = \sum_{i=1}^{k+1} S_i(n_k) \leq \sum_{i=1}^{k+1} S_i(n) < \sum_{i=1}^{k+1} S_i(n) + 1 = S_{k+1}(n+1).$$

QED

Definición 3.2.10 (Orden antilexicográfico)**.** Sea $k \in \mathbb{N}$. Definimos el orden antilexicográfico para k-tuplas $<_{\text{AL}}$ recursivamente sobre k como

1. $(a_1) <_{\text{AL}} (b_1)$ si y sólo si $a_1 < b_1$, y
2. $(a_1 \ldots, a_k, a_{k+1}) <_{\text{AL}} (b_1 \ldots, b_k, b_{k+1})$ si y sólo si $a_{k+1} < b_{k+1}$ o bien $a_{k+1} = b_{k+1}$ y $(a_1, \ldots, a_k) <_{\text{AL}} (b_1, \ldots, b_k)$.

Teorema 3.2.11. *Sean $(n_1, \ldots, n_k)$ y $(m_1, \ldots, m_k)$ k-descomposiciones de n y m respectivamente, entonces $n < m$ si y sólo si $(n_1, \ldots, n_k) <_{AL} (m_1, \ldots, m_k)$.*

Demostración. Procedamos por inducción sobre k.
Caso Base. Si $k = 1$ entonces es claro que si (n_1) y (m_1) son respectivamente 1-descomposiciones de n y m, entonces $n = n_1$ y $m = m_1$. Se sigue que $n < m$ si y sólo si $(n_1) <_{\text{AL}} (m_1)$.
Caso Inductivo. Supongamos que el resultado se sigue para k, y veamos que la equivalencia se sostiene para $k+1$. Tomemos $(n_1, \ldots, n_k, n_{k+1})$ y $(m_1, \ldots, m_k, m_{k+1})$ $k+1$-descomposiciones de n y m respectivamente. De la definición se sigue $(n_1, \ldots, n_k)$ y $(m_1, \ldots, m_k)$ son k-descomposiciones de $\sum_{i=1}^{k} S_i(n_i)$ y $\sum_{i=1}^{k} S_i(m_i)$ respectivamente.

Para la ida, tomemos $n < m$. Por el teorema 3.2.6 se tiene $n_{k+1} = \lVert n \rVert_{k+1}$ y $\lVert m \rVert_{k+1}$, y por tanto $n_{k+1} \leq m_{k+1}$. Hay dos casos.

(1) Si $n_{k+1} < m_{k+1}$ entonces $(n_1, \ldots, n_{k+1}) <_{\text{AL}} (m_1, \ldots, m_{k+1})$.

(2) Si $n_{k+1} = m_{k+1}$, entonces $S_{k+1}(n_{k+1}) = S_{k+1}(m_{k+1})$. Dado que

$$n = \sum_{i=1}^{k+1} S_i(n_i) < \sum_{i=1}^{k+1} S_i(m_i) = m, \text{ entonces } \sum_{i=1}^{k} S_i(n_i) < \sum_{i=1}^{k} S_i(m_i).$$

Por hipótesis inductiva

$$(n_1, \ldots, n_k) <_{\text{AL}} (m_1, \ldots, m_k) \quad \text{así} \quad (n_1, \ldots, n_{k+1}) <_{\text{AL}} (m_1, \ldots, m_{k+1}).$$

Para el regreso tomemos $(n_1, \ldots, n_{k+1}) <_{\text{AL}} (m_1, \ldots, m_{k+1})$ y analicemos los dos casos.

(1) Si $n_{k+1} < m_{k+1}$, entonces $\llbracket n \rrbracket_{k+1} < \llbracket m \rrbracket_{k+1}$ y por la proposición 2.2.3 pasa $n < m$.

(2) Si $n_{k+1} = m_{k+1}$ y $(n_1, \ldots, n_k) <_{\text{AL}} (m_1, \ldots, m_k)$, entonces $S_{k+1}(n_{k+1}) = S_{k+1}(m_{k+1})$. Por hipótesis inductiva se sigue

$$\sum_{i=1}^{k} S_i(n_i) < \sum_{i=1}^{k} S_i(m_i) \text{ y sumando se tiene } \sum_{i=1}^{k+1} S_i(n_i) < \sum_{i=1}^{k+1} S_i(m_i), \text{ es decir } n < m.$$

QED

Observación 3.2.12. Del teorema 3.2.8 concluimos que todo valor m tiene una única k-descomposición y viceversa. Considerando también el teorema 3.2.9 sabemos que si $m < S_k(n+1)$ entonces su k-descomposición será una k-tupla que cumple que sus entradas están acotadas por n y además sus valores son crecientes.

La primera de estas propiedades muestra que hay una biyección natural entre los naturales menores a $S_k(n+1)$ y las k-descomposiciones cuyas entradas están acotadas por n. Esto es, hay una función biyectiva $\varphi_{k,n} : \text{Desc}_k(n) \longrightarrow I_{S_k(n+1)}$ que a cada k-descomposición le asigna el natural que descompone como suma de simplex (ver definiciones 3.1.1 y 3.2.4).

De la segunda propiedad se obtiene que toda k-descomposición es a su vez una k-tupla creciente. Como se vio anteriormente toda k-tupla es a su vez una representación por separadores. En la siguiente sección se detallará esta observación.

3.3. Enumeración de composiciones

Como visto en la sección 3.1.2, transformar una composición en su representación por separadores es bastante directo. Aprovecharemos las propiedades de la representación por separadores tanto para dar un ordenamiento a la diagonal, como para asistir en el proceso de Rank y Unrank de las diagonales. Versiones semejantes a los algoritmos planteados en esta sección se pueden encontrar en la sección 2.3.1 de [17].

Ordenemos las representaciones por separadores en el orden antilexicográfico con lo cual tendremos un orden para todas las k-composiciones de un número n. Si bien se puede utilizar el orden lexicográfico habitual, resulta conveniente usar el propuesto pues en virtud exhibirá una relación entre k-descomposiciones y tuplas de separadores. Relacionar elementos de una diagonal con sus k-representaciones es un encaje que respeta el orden antilexicográfico gracias al teorema 3.2.11.

Cabe mencionar que alterar el orden en la enumeración de las diagonales no corrompe nuestra intención de enumerar todas las tuplas de forma estructuralmente creciente pues el tamaño de todas estas tuplas (a saber, la suma de sus entradas) en la diagonal es el mismo.

Mostraremos que cada uno de los valores a_i está en relación con el índice de un i-simplex. Debido a esta cualidad podremos calcular de forma independiente el valor en la enumeración de cada separador, y por ende de cada composición. Esta independencia de cálculos es la que hace que para obtener el valor en la enumeración no dependa de valores previos.

Los algoritmos propuestos en esta sección básicamente explotan ese hecho.

En la tabla 3.2 se tienen ordenados todas las 4-tuplas de $\mathrm{Diag}_4(n)$ con el orden que su representación por separadores le otorga.

Rank	Sep	BarEst	Comp
0	(0,0,0)	||| ★★★	(0,0,0,3)
1	(0,0,1)			★	★★	(0,0,1,2)
2	(0,1,1)		★		★★	(0,1,0,2)
3	(1,1,1)	★ ||| ★★	(1,0,0,2)
4	(0,0,2)			★★	★	(0,0,2,1)
5	(0,1,2)		★	★	★	(0,1,1,1)
6	(1,1,2)	★		★	★	(1,0,1,1)
7	(0,2,2)		★★		★	(0,2,0,1)
8	(1,2,2)	★	★		★	(1,1,0,1)
9	(2,2,2)	★★ ||| ★	(2,0,0,1)

Rank	Sep	BarEst	Comp
10	(0,0,3)	|| ★★★	(0,0,3,0)
11	(0,1,3)		★	★★	(0,1,2,0)
12	(1,1,3)	★ || ★★	(1,0,2,0)
13	(0,2,3)		★★	★	(0,2,1,0)
14	(1,2,3)	★	★	★	(1,1,1,0)
15	(2,2,3)	★★ || ★	(2,0,1,0)
16	(0,3,3)		★★★ || (0,3,0,0)
17	(1,3,3)	★	★★ || (1,2,0,0)
18	(2,3,3)	★★	★ || (2,1,0,0)
19	(3,3,3)	★★★			(3,0,0,0)

Rank	SubRank	SubsubRank	Sep	BarEst	Comp
4	0	0	(0,0,2)			★★	★	(0,0,2,1)
5	1	0	(0,1,2)		★	★	★	(0,1,1,1)
6	2	1	(1,1,2)	★		★	★	(1,0,1,1)
7	3	0	(0,2,2)		★★		★	(0,2,0,1)
8	4	1	(1,2,2)	★	★		★	(1,1,0,1)
9	5	2	(2,2,2)	★★ ||| ★	(2,0,0,1)

(a) Subdiagonal de $\mathrm{Diag}_4(3)$ divida en sub-subdiagonales.

Tabla 3.2: Orden y representaciones de los elementos de $\mathrm{Diag}_4(3)$ divididos en subdiagonales.

Algo interesante es que podemos dividir las diagonales en subdiagonales, las cuales se caracterizan por coincidir la última entrada en su representación por separadores. Por ejemplo en la tabla 3.2 se dividen en 4 subdiagonales determinadas por los valores rank 0;0, 1;3, 4;9, y 10;19. Por la proposición 3.1.3 sabemos que la cantidad de elementos de $\mathrm{Diag}_4(3) = S_3(4) = 20$. Es importante notar que los enumerandos de los elementos en las subdiagonales corresponden con un 3-gnomon y por el corolario 3.2.3 todos tienen el mismo índice de 3-simplex. Esto se puede apreciar en la segunda columna de la tabla 3.3 en cada subdiagonal.

Consecuentemente, podemos tomar una de estas subdiagonales, por ejemplo, la 4:9, y volverla a dividir en subsubdiagonales ahora determinadas por su penúltimo valor en le representación por separadores. Observemos en la tabla 3.2a que enumeramos esta subdiagonal, cada subsubdivisión corresponde con un 2-gnomon y por un argumento similar coinciden en índice del 2-simplex que el anteceden. Podemos observar esto muy claramente en la cuarta columna de la tabla 3.3 para cada subsubdiagonal.

Este procedimiento, lo podríamos hacer una vez más, pero para el caso de 1-simplex se vuelve trivial.

En esencia, acabamos de observar que la representación por separadores es a su vez la 3-descomposición del valor que le corresponde en la enumeración de cada diagonal. Esta cualidad

se preserva para tantas subdivisiones de diagonales con sus enumeraciones correspondientes. Provisto que dado un número podemos transformarlo en su representación como k-descomposición de simplex y viceversa, el procedimiento para la enumeración de composiciones se basará en esto.

Más precisamente, la representación por separadores de una $k+1$-tupla es a su vez una k-tupla de valores crecientes, así como una k-descomposición de un número d.

Rank		SubRank		Subsubrank		Sep	Comp
d_3	$a_3 := \lVert d_3 \rVert_3$	$d_2 := d_3 - S_3(a_3)$	$a_2 := \lVert d_2 \rVert_2$	$d_1 := d_2 - S_2(a_2)$	$a_1 := \lVert d_1 \rVert_1$	(a_1, a_2, a_3)	
0	$\lVert 0 \rVert_3 = 0$	$0 - 0 = 0$	$\lVert 0 \rVert_2 = 0$	$0 - 0 = 0$	$\lVert 0 \rVert_1 = 0$	$(0,0,0)$	$(0,0,0,3)$
1	$\lVert 1 \rVert_3 = 1$	$1 - 1 = 0$	$\lVert 0 \rVert_2 = 0$	$0 - 0 = 0$	$\lVert 0 \rVert_1 = 0$	$(0,0,1)$	$(0,0,1,2)$
2	$\lVert 4 \rVert_3 = 1$	$2 - 1 = 1$	$\lVert 1 \rVert_2 = 1$	$1 - 1 = 0$	$\lVert 0 \rVert_1 = 0$	$(0,1,1)$	$(0,1,0,2)$
3	$\lVert 3 \rVert_3 = 1$	$3 - 1 = 2$	$\lVert 2 \rVert_2 = 1$	$2 - 1 = 1$	$\lVert 1 \rVert_1 = 1$	$(1,1,1)$	$(1,0,0,2)$
4	$\lVert 4 \rVert_3 = 2$	$4 - 4 = 0$	$\lVert 0 \rVert_2 = 0$	$0 - 0 = 0$	$\lVert 0 \rVert_1 = 0$	$(0,0,2)$	$(0,0,2,1)$
5	$\lVert 5 \rVert_3 = 2$	$5 - 4 = 1$	$\lVert 1 \rVert_2 = 1$	$1 - 1 = 0$	$\lVert 0 \rVert_1 = 0$	$(0,1,2)$	$(0,1,1,1)$
6	$\lVert 6 \rVert_3 = 2$	$6 - 4 = 2$	$\lVert 2 \rVert_2 = 1$	$2 - 1 = 1$	$\lVert 1 \rVert_1 = 1$	$(1,1,2)$	$(1,0,1,1)$
7	$\lVert 7 \rVert_3 = 2$	$7 - 4 = 3$	$\lVert 3 \rVert_2 = 2$	$3 - 3 = 0$	$\lVert 0 \rVert_1 = 0$	$(0,2,2)$	$(0,2,0,1)$
8	$\lVert 8 \rVert_3 = 2$	$8 - 4 = 4$	$\lVert 4 \rVert_2 = 2$	$4 - 3 = 1$	$\lVert 1 \rVert_1 = 1$	$(1,2,2)$	$(1,1,0,1)$
9	$\lVert 9 \rVert_3 = 2$	$9 - 4 = 5$	$\lVert 5 \rVert_2 = 2$	$5 - 3 = 2$	$\lVert 2 \rVert_1 = 2$	$(2,2,2)$	$(2,0,0,1)$

Tabla 3.3: UNRANK de los primeros diez elementos de $\mathrm{Diag}_4(3)$. Cada valor subrank (subsubrank) corresponde a su enumeración en la subdiagonal (subsubdiagonal).

La tabla 3.3 refleja de manera más explícita como se transforma el valor d de una diagonal en una k-tupla de seperadores descomponiendo a d como una suma de simplex obteniendo los índices en los valores a_i en cada subdivisión. Esto corresponde con el de UNRANK de $\mathrm{Diag}_4(3)$.

El proceso inverso RANK que consiste en obtener el valor d a partir de la representación por separadores es un tanto más sencillo pues ésta corresponde con una k-descomposición de d la cual ya se tiene como entrada. Siendo así, se reduce el trabajo a reconstruir d como suma de simplex. Este proceso se puede apreciar en la tabla 3.4.

Comp	Sep		Rank
	(a_1, a_2, a_3)	$S_1(a_1) + S_2(a_2) + S_3(a_3)$	d
$(0,0,0,3)$	$(0,0,0)$	$S_1(0) + S_2(0) + S_3(0) = 0 + 0 + 0 =$	0
$(0,0,1,2)$	$(0,0,1)$	$S_1(0) + S_2(0) + S_3(1) = 0 + 0 + 1 =$	1
$(0,1,0,2)$	$(0,1,1)$	$S_1(0) + S_2(1) + S_3(1) = 0 + 1 + 1 =$	2
$(1,0,0,2)$	$(1,1,1)$	$S_1(1) + S_2(1) + S_3(1) = 1 + 1 + 1 =$	3
$(0,0,2,1)$	$(0,0,2)$	$S_1(0) + S_2(0) + S_3(2) = 0 + 0 + 4 =$	4
$(0,1,1,1)$	$(0,1,2)$	$S_1(0) + S_2(1) + S_3(2) = 0 + 1 + 4 =$	5
$(1,0,1,1)$	$(1,1,2)$	$S_1(1) + S_2(1) + S_3(2) = 1 + 1 + 4 =$	6
$(0,2,0,1)$	$(0,2,2)$	$S_1(0) + S_2(2) + S_3(2) = 0 + 3 + 4 =$	7
$(1,1,0,1)$	$(1,2,2)$	$S_1(1) + S_2(2) + S_3(2) = 1 + 3 + 4 =$	8
$(2,0,0,1)$	$(2,2,2)$	$S_1(2) + S_2(2) + S_3(2) = 2 + 3 + 4 =$	9

Tabla 3.4: RANK de los primeros diez elementos de $\mathrm{Diag}_4(3)$.

La piedra angular de este procedimiento es que la representación por separadores (que codifica la composición) es simultáneamente una k-descomposición (que codifica a d), así como una k-tupla de valores ascendientes (dota de orden a las composiciones). Este hecho prácticamente ya lo mostramos por lo que tiene lugar el siguiente teorema.

Teorema 3.3.1. *Sea $n, k \in \mathbb{N}$ con $k \geq 1$ y $(a_1, \ldots, a_k) \in \mathbb{N}^k$. Las siguientes son equivalentes:*

1. *$(a_1, \ldots, a_k)$ es una representación por separadores de una k-composición de n,*

2. $(a_1, \ldots, a_k)$ *cumple que* $a_i \leq a_j \leq n$ *para* $1 \leq i \leq j \leq k$,

3. $(a_1, \ldots, a_k)$ *es una* k*-descomposición de algún número* $d < S_k(n+1)$.

Demostración. La equivalencia entre (1) y (2) se sigue directamente del teorema 3.1.10. Por otro lado, la equivalencia entre (2) y (3) se sigue del teorema 3.2.9. QED

Corolario 3.3.2. *Si* $n, k \in \mathbb{N}$ *con* $k \geq 1$, *entonces* $\mathrm{Sep}_k(n) = \mathrm{Desc}_k(n)$[4].

Demostración. Resultado directo del teorema 3.3.1. QED

3.3.1. Algoritmos Rank y Unrank para composiciones

Como se mencionó en las observación 3.1.9 se tiene una biyección $\psi_{k,n}\,\mathrm{Diag}_{k+1}(n) \longrightarrow \mathrm{Sep}_k(n)$ que a cada $k+1$-composición de n le asigna su correspondiente representación por separadores. Esta función es calculada por el algortimo Comp2Sep y cuya inversa es calculada por el algoritmo Sep2Comp. Por otro lado, la observación 3.2.12 hay una biyección $\varphi_{k,n} : \mathrm{Desc}_k(n) \longrightarrow I_{S_k(n+1)}$ que cada k-descomposición le asigna el natural al cual descompone. Por el corolario 3.3.2 los conjuntos $\mathrm{Desc}_k(n)$ y $\mathrm{Sep}_k(n)$ son iguales, induciéndose una biyección entre $I_{S_k(n+1)}$ y $\mathrm{Diag}_{k+1}(n)$ como se puede apreciar en la figura 3.2

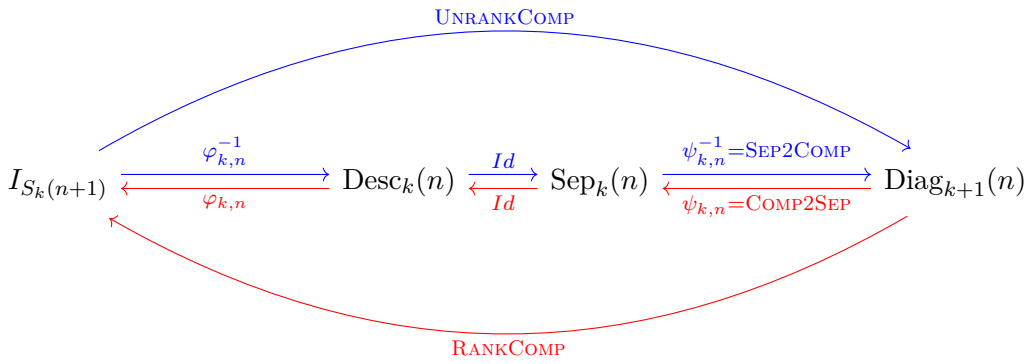


Figura 3.2: Diagrama de funciones biyectivas que describen el proceso de Unrank-Comp y Rank-Comp

Los algoritmos Unrank-Comp y Rank-Comp calculan justamente esta biyección. Los ejemplos mostrados en las tablas 3.3 y 3.4 dan pauta a como los algoritmos Unrank-Comp y Rank-Comp siguen el esquema de la figura 3.2 respectivamente.

[4] ver definiciones 3.1.5 y 3.2.4 respectivamente

Unrank de Composiciones

Algoritmo 9 UNRANK-COMP$(d, n, k+1)$

Entrada: Una terna $d, k+1, n \in \mathbb{N}$ tal que $k \geq 1$ y $d < S_k(n+1)$.
Salida: Una $k+1$-composición de n.
1: **for** $i \leftarrow k$ **downto** 1 **do**
2: $\quad a_i \leftarrow \lfloor\!\lfloor d \rfloor\!\rfloor_i$
3: $\quad d \leftarrow d - S_i(a_i)$
4: **end for**
5: $(x_1, \ldots, x_{k+1}) \leftarrow$ SEP2COMP$((a_1, \ldots, a_k), n)$
6: **return** $(x_1, \ldots, x_{k+1})$

Afirmación 3.3.3. El algoritmo UNRANK-COMP (9) termina.

Demostración. El ciclo for de las líneas 1-4 no altera el contador i, pues ni la llamada a FLOOR-SIMPLEX en la línea 2, ni el cálculo del simplex en la línea 3, alteran este valor. Más aún, las llamadas en 2 y 3 terminan, por lo que el ciclo termina.

Asimismo, la llamada a SEP2COMP en la línea 5 termina por la proposición 3.1.13, concluyéndose lo buscado. QED

Afirmación 3.3.4. Si $d, n, k+1$ es la entrada al algoritmo UNRANK-COMP (9) y $k \geq 1$, entonces los valores a_i con $1 \leq i \leq k$ definidos en el ciclo for de las líneas 1-4 en forman una k-descomposición de d.

Demostración. Distingamos la entrada d con los valores de la variable d definiendo d_i al valor de la variable d cuando termine la iteración cuyo contador se encuentre en el valor i, definamos d_{k+1} como el valor de la entrada. Hagamos hincapié en que el contador comienza en k y finaliza en 1.

Afirmamos que para toda k y para toda $1 \leq i \leq k$ se cumple que $(a_1, \ldots, a_i)$ forma una i-descomposición de d_{i+1}.

Procedamos por inducción sobre k. Si $k = 1$ entonces solo hay una iteración la cual $a_1 = \lfloor\!\lfloor d_2 \rfloor\!\rfloor_1 = d_2$ y así (a_1) es una 1-descomposición de d_2.

Supongamos que el resultado se cumple para k y veamos que se sostiene para $k+1$. Esto es, veamos que para toda $1 \leq i \leq k+1$ se cumple que $(a_1, \ldots, a_i)$ forma una i-descomposición de d_{i+1}.

Nuestra hipótesis de inducción nos garantiza que el resultado se tiene para toda $1 \leq i \leq k$, por lo que basta mostrar que $(a_1, \ldots, a_k, a_{k+1})$ forma una $k+1$-descomposición de d_{k+2}.

Por hipótesis inductiva sabemos que

$$\sum_{i=1}^{k} S_i(a_i) = d_{k+1}, \text{ y además} \tag{3.3}$$

$$\sum_{i=1}^{j} S_i(a_i) < S_j(a_{j+1} + 1) \text{ para toda } j \in \{1, \ldots, k-1\} \tag{3.4}$$

De la línea 3 se tiene que $d_{k+1} = d_{k+2} - S_{k+1}(a_{k+1})$ y por (3.3) se concluye que

$$\sum_{i=1}^{k} S_i(a_i) = d_{k+1} = d_{k+2} - S_{k+1}(a_{k+1}) \text{ por lo tanto}$$

$$d_{k+2} = \Big(\sum_{i=1}^{k} S_i(a_i)\Big) + S_{k+1}(a_{k+1}) = \sum_{i=1}^{k+1} S_i(a_i) \tag{3.5}$$

De la línea 2 sabemos que $a_{k+1} = \lVert d_{k+2} \rVert_{k+1}$, por lo que

$$S_{k+1}(a_{k+1}) \leq d_{k+2} < S_{k+1}(a_{k+1}+1) \text{ sustituyendo con (3.5) se tiene}$$

$$\sum_{i=1}^{k+1} S_i(a_i) < S_{k+1}(a_{k+1}+1) = S_k(a_{k+1}+1) + S_{k+1}(a_{k+1}) \text{ despejando}$$

$$\Big(\sum_{i=1}^{k} S_i(a_i)\Big) + S_{k+1}(a_{k+1}) < S_k(a_{k+1}+1) + S_{k+1}(a_{k+1}) \text{ es decir}$$

$$\sum_{i=1}^{k} S_i(a_i) < S_k(a_{k+1}+1) \tag{3.6}$$

Conjuntando (3.6) y (3.4) se tiene que

$$\sum_{i=1}^{j} S_i(a_i) < S_j(a_{j+1}+1) \text{ para toda } j \in \{1, \ldots, k\},$$

así con (3.5) se muestra la afirmación planteada. Utilizando esta afirmación para $i = k$ se concluye que $(a_1, \ldots, a_k)$ forma una k-descomposición de d_{k+1}, es decir, una k-descomposición del valor de entrada d. Esto muestra lo que queríamos demostrar. QED

Proposición 3.3.5. *Si la entrada al algoritmo* Unrank-Comp *(9) es $d, n, k+1$ y $d < S_k(n+1)$, entonces la salida es la $k+1$-composición cuya representación por separadores corresponde con una k-descomposición de d.*

Demostración. Por la afirmación 3.3.4 sabemos que la tupla $(a_1, \ldots, a_k)$ obtenida del ciclo for de las líneas 1-4 es la k-descomposición correspondiente a d, la cual es única por el teorema 3.2.8. Por el teorema 3.2.9, se cumple que $0 \leq a_i \leq a_j \leq n$ para toda $0 \leq i \leq j \leq k$ por lo que es una representación por separadores de alguna $k+1$-composición.

De la línea 5 se tiene que $(x_1, \ldots, x_{k+1}) = \text{Sep2Comp}((a_1, \ldots, a_k), n)$ y por la proposición 3.1.13 se tiene que $(x_1, \ldots, x_{k+1})$ es la $k+1$-composición de n correspondiente a la representación $(a_1, \ldots, a_k)$ la cual es única por la proposición 3.1.8. QED

Proposición 3.3.6. *Si el cálculo de un número simplex $S_k(n)$ y el índice de un número simplex $\lVert m \rVert_k$ tienen complejidad $\mathcal{S}(k,n)$ y $\mathcal{FS}(m,k)$ respectivamente, entonces la complejidad en tiempo del algoritmo* Unrank-Comp$(d, n, k+1)$ *(9) con $k > 1$ es $O\big(k(\mathcal{FS}(d,k) + \mathcal{S}(k, \lVert d \rVert_k))\big)$.*

Demostración. Definamos d_i para $1 \leq i \leq k$ como en la afirmación 3.3.4 y d_{k+1} corresponde con la entrada d. De esta misma afirmación sabemos que $(a_1, \ldots, a_k)$ forma una k-descomposición de d_{k+1}, por lo que por los teoremas 3.3.1 y 3.2.6 se cumple que $\lVert d_{i+1} \rVert_i \leq \lVert d_{k+1} \rVert_k$ para $1 \leq i \leq k$. Así el índice más grande a calcular es $\lVert d_{k+1} \rVert_k = \lVert d \rVert_k$. Así la línea 2 toma a lo más $\mathcal{FS}(d,k)$ en cada iteración.

Por lo antes mostrado, se tiene que $a_i \leq a_k$ para $1 \leq i \leq k$, por lo que el simplex de mayor índice y dimensión que se calcula en la línea 3 es $a_k = \lVert d_{k+1} \rVert_k = \lVert d \rVert_k$. Siendo así, cada iteración de la línea 3 toma tiempo a lo más $\mathcal{S}(k, \lVert d \rVert_k)$.

El ciclo for de las líneas 1-4 toma tiempo $O\big(k(\mathcal{FS}(d,k) + \mathcal{S}(k, \lVert d \rVert_k))\big)$ pues realiza k iteraciones. En virtud de la proposición 3.1.14, la línea 5 toma tiempo $O(k)$, de donde el cómputo del algoritmo toma tiempo $O\big(k(\mathcal{FS}(d,k) + \mathcal{S}(k, \lVert d \rVert_k)) + k\big) = O\big(k(\mathcal{FS}(d,k) + \mathcal{S}(k, \lVert d \rVert_k))\big)$.

QED

Rank de Composiciones

El siguiente algoritmo recibe como entrada una $k+1$-tupla de naturales, la cual es a su vez una $k+1$-composición de algún número n y devuelve su valor $d < S_k(n+1)$ en la enumeración de la diagonal $\text{Diag}_{k+1}(n)$, así como el valor n. Nótese que el valor buscado es d pero se aprovecha el cómputo para también devolver n, lo cual cobrará más sentido cuando se enumeren k-tuplas.

Algoritmo 10 RANK-COMP$((x_1, \ldots, x_{k+1}))$

Entrada: Una $k+1$-composición de algún valor n.
Salida: Una pareja de números d, n tal que $d < S_k(n+1)$.

1: $(a_1, \ldots, a_k), n \leftarrow$ COMP2SEP$(x_1, \ldots, x_{k+1})$ ▷ Representación por separadores.
2: $d \leftarrow 0$
3: **for** $i \leftarrow 1$ **upto** k **do** ▷ $d = \sum_{i=1}^{k} S_i(a_i)$
4: $\quad d \leftarrow d + S_i(a_i)$
5: **end for**
6: **return** d, n

Proposición 3.3.7. *Si la entrada de* RANK-COMP *(10) es una $k+1$-composición $(x_1, \ldots, x_{k+1})$ algún natural n, entonces el algoritmo termina y devuelve los valores d y n, donde la k-descomposición de d corresponde con la representación por separadores de la entrada y $d < S_k(n+1)$.*

Demostración. Por la proposición 3.1.11 la línea 1 termina y además se tiene que los valores $(a_1, \ldots, a_k), n$ obtenidos corresponden con la representación por separadores de la entrada y el valor del cual la entrada es una $k+1$-composición respectivamente.

El ciclo for de las líneas 3-4 termina ya que su índice jamás es alterado y cada iteración consta de una suma y un cálculo de un simplex.

Por la proposición 3.1.13 $(a_1, \ldots, a_k)$ es la representación por separadores de $(x_1, \ldots, x_{k+1})$ y $n = \sum_{i=1}^{k} x_i$. Por el teorema 3.3.1 ésta última tupla es a su vez k-descomposición de algún valor $d_0 < S_k(n+1)$. Por definición de k-descomposición necesariamente $d_0 = \sum_{i=1}^{k} S_i(a_i)$. Veamos que la salida d coincide con d_0.

Dado que las k-descomposiciones son únicas por el teorema 3.2.8, basta ver que $(a_1, \ldots, a_k)$ descompone a d. Veamos que al final del ciclo for de las líneas 3-5 la variable $d = \sum_{i=1}^{k} S_i(a_i)$ y por tanto $d = d_0$.

Afirmamos que en la j-ésima iteración del ciclo for de las líneas 3-5 el valor $d = \sum_{i=1}^{j} S_i(a_i)$. Procedamos por inducción sobre j.

Para $j = 1$ se tiene que $d = 0 + S_1(a_1) = S_1(a_1)$. Supongamos que el resultado se tiene para la j-ésima iteración y mostremos para la $j+1$-ésima iteración. Previa a esta iteración d almacena el valor $\sum_{i=1}^{j} S_i(a_i)$, por lo que al finalizar tal iteración d almacenará el valor $\sum_{i=1}^{j} S_i(a_i) + S_{j+1}(a_{j+1}) = \sum_{i=1}^{j+1} S_i(a_i)$. Esto muestra que $d = d_0$ y por tanto $d < S_k(n+1)$. QED

Proposición 3.3.8. *Si el cálculo de un número simplex $S_k(n)$ tiene complejidad $\mathcal{S}(k,n)$, entonces la complejidad en tiempo del algoritmo* Rank-Comp$(x_1,\ldots,x_{k+1})$ *(9) con $k \geq 1$ es $O\big(k\mathcal{S}(k,a)\big)$ con $a = \sum_{i=1}^{k} x_i$.*

Demostración. Por la proposición 3.1.12, la línea 1 la complejidad en tiempo y espacio de la línea 1 es $O(k)$.

El ciclo for de las líneas 3-5 toma k iteraciones y en la i-ésima iteración calcula un i-simplex. Debido a esto, la dimensión del simplex más grande que va a calcular es k. En virtud del teorema 3.3.1, $a_i \leq a_k$ para $1 \leq i \leq k$ por lo que el simplex más grande a calcular es $S_k(a_k)$. Por $(a_1,\ldots,a_k)$ ser una representación por separadores, entones $a_k = \sum_{i=1}^{k} x_i$. Siendo así, la línea 4 toma tiempo a lo más $\mathcal{S}(k,a_k)$, por lo que el ciclo toma tiempo $O\big(k\mathcal{S}(k,a_k)\big)$.

Por lo tanto, el algoritmo toma tiempo $O\big(k\mathcal{S}(k,a_k)+k\big) = O\big(k\mathcal{S}(k,a_k)\big)$. QED

El siguiente teorema exhibe como los algoritmos Unrank-Comp y Rank-Comp son uno inverso del otro.

Teorema 3.3.9 (Invertibilidad Rank y Unrank)**.** *Sean $d,k,n \in \mathbb{N}$ y $(x_1,\ldots,x_{k+1}) \in \mathbb{N}^{k+1}$ con $k > 0$, se siguen las siguientes:*

1. *Si $d < S_k(n+1)$, entonces* Rank-Comp$\Big($Unrank-Comp$\big(d,n,k+1\big)\Big) = d,n,$
2. Unrank-Comp$\Big($Rank-Comp$\big((x_1,\ldots,x_{k+1}),k+1\big)\Big) = (x_1,\ldots,x_{k+1}).$

Demostración. **(1)** Sea Unrank-Comp$(d,n,k+1) = (x_1,\ldots,x_{k+1})$. Por la proposición 3.3.5 $(x_1,\ldots,x_{k+1})$ es una $k+1$-composición de n y cuya representación por separadores $(a_1,\ldots,a_k)$ que es a su vez una k-descomposición de d.

Necesariamente Rank-Comp$\big((x_1,\ldots,x_{k+1})\big) = d,n$, por la proposición 3.3.7.

(2) Sea Rank-Comp$\big((x_1,\ldots,x_{k+1})\big) = d,n$, entonces $(x_1,\ldots,x_{k+1})$ es una $k+1$-composición de n y d es el valor cuya k-descomposición es una representación por separadores de la entrada por la proposición 3.3.7. Así Unrank-Comp$\big(d,n,k+1\big)$ necesariamente tiene como salida $(x_1,\ldots,x_{k+1})$ por la proposición 3.3.5. QED

3.3.2. Complejidades para enumeración de composiciones

Proposición 3.3.10. *Si el cálculo de un número simplex y el índice de un número simplex se hacen con los algoritmos* Simplex *(1) y* FloorSimplex *(3), entonces se cumplen las siguientes.*

1. *La complejidad en tiempo del algoritmo* Unrank-Comp$(d,n,k+1)$ *(9) con $k \geq 1$ es*[5] *$O\big(k^2\log_2(\|d\|_k)\big)$ y la complejidad en espacio es $O(k)$.*
2. *La complejidad en tiempo del algoritmo* Rank-Comp$(x_1,\ldots,x_{k+1})$ *(10) con $k \geq 1$ es $O\big(k^2\big)$ y la complejidad en espacio es $O(k)$.*

Demostración. **(1)** Aplicando las proposiciones 2.1.16 y 2.2.20 en la proposición 3.3.6, se obtiene que la complejidad en tiempo es

$$O\big(k(k\log_2(\|d\|_k))\big) = O\big(k^2\log_2(\|d\|_k)\big).$$

[5]Por la observación 2.2.8 también es $O\big(k^2\log_2(k\sqrt[k]{d})\big)$

Para la complejidad en espacio, de nuevo por las proposiciones 2.1.16 y 2.2.20 el ciclo for de las líneas 1-4 toma espacio $O(1)$ y por la proposición 3.1.14 la línea 5 toma espacio $O(k)$. Por lo que el espacio será de $O(k)$.
(2) Aplicando la proposición 2.2.20 en la proposición 3.3.8, la complejidad en tiempo es $O(k^2)$. Por esta misma proposición, el ciclo for de las líneas 3-5, tiene complejidad en espacio $O(1)$, al igual que la línea 2. Por la proposición 3.1.12, la línea 1 toma espacio $O(k)$ y así la complejidad en espacio es $O(k)$. QED

Recordando que podemos usar una tabla de simplex precomputados mediante el algoritmo 2, podemos optimizar el proceso tanto para el cálculo de un simplex como para FloorSimplex. Es inminente saber qué dimensiones debe tener esta tabla para poder hacer estos cómputos sin desbordar la tabla.

Proposición 3.3.11.

1. *Si* Tabla-Simplex$(k+1, \lfloor\!\lfloor d \rfloor\!\rfloor_k + 1) = TS$ *(2), entonces* TS *es suficientemente grande*[6] *para computar* Unrank-Comp$(d, n, k+1)$ *(9).*
2. *Si* Tabla-Simplex$\left(k+1, 1+\sum_{i=1}^{k} x_i\right) = TS$ *(2), entonces* TS *es suficientemente grande para computar* Rank-Comp$\left((x_1, \ldots, x_{k+1})\right)$ *(10).*

Demostración. **(1)** Observemos primero que toda llamada a FloorSimplex se hace en la línea 2. Veamos que toda llamada es menor a $\lfloor\!\lfloor d \rfloor\!\rfloor_k$ con lo cual cualquier índice buscado yace en la tabla TS. Definamos d_i como en la afirmación 3.3.4. Por esta misma afirmación los valores $(a_1, \ldots, a_k)$ forman una k-descomposición de $d_{k+1} = d$.

Del teorema 3.3.1 se sigue que $a_i \leq a_k$ para toda $1 \leq i \leq k$. De la línea 2 se tiene que

$$a_i = \lfloor\!\lfloor d_{i+1} \rfloor\!\rfloor_i, \text{ por lo que } \lfloor\!\lfloor d_{i+1} \rfloor\!\rfloor_i \leq \lfloor\!\lfloor d_{k+1} \rfloor\!\rfloor_k = \lfloor\!\lfloor d \rfloor\!\rfloor_k < \lfloor\!\lfloor d \rfloor\!\rfloor_k + 1. \tag{3.7}$$

Veamos ahora que todo simplex calculado cae en esta misma tabla. De la línea 3 se calcula $S_i(a_i)$ en la iteración cuyo contador vale i. En la iteración con contador igual a k se calcula $S_k(a_k)$ el cual claramente es el simplex de mayor índice y dimensión posible (ambas ocurren simultáneamente pues $a_i \leq a_k$). De la ecuación (3.7) se sigue que

$$a_k = \lfloor\!\lfloor d_{k+1} \rfloor\!\rfloor_k < \lfloor\!\lfloor d \rfloor\!\rfloor_k + 1.$$

(2) Este algoritmo precisa de números simplex, por lo que basta ver que el índice y dimensión del simplex más grande durante el cómputo de este algoritmo no sobrepase a $1 + \sum_{i=1}^{k} x_i$ y $k+1$ respectivamente para garantizar que no desbordan a TS.

Por la proposición 3.1.11, $(a_1, \ldots, a_k)$ es la representación por separadores de $(x_1, \ldots, x_{k+1})$. De la línea 4, se calculan los simplex $S_i(a_i)$ para $1 \leq i \leq k$. Por el teorema 3.3.1 se tiene

$$a_i \leq a_k = \sum_{i=1}^{k} x_i < 1 + \sum_{i=1}^{k} x_i \quad \text{para toda } 1 \leq i \leq k.$$

De lo anterior se sigue inmediatamente lo que se buscaba. QED

Proposición 3.3.12. *Si el cálculo de un número simplex se hace consultando una tabla precomputada (algoritmo 2)* TS *suficientemente grande y el índice de un número simplex se obtiene mediante el algoritmo* FloorSimplex *(Con tabla) (4), entonces se cumplen las siguientes.*

[6] Ver definición 2.2.22.

1. *La complejidad en tiempo del algoritmo* UNRANK-COMP$(d, n, k+1)$ *(9) con* $k \geq 1$ *es*[7] $O\big(k \log_2(\|d\|_k)\big)$ *y la complejidad en espacio es* $O(k)$ *(sin considerar el espacio de* TS*).*

2. *La complejidad en tiempo del algoritmo* RANK-COMP$(x_1, \ldots, x_{k+1})$ *(10) con* $k \geq 1$ *es* $O\big(k\big)$ *y la complejidad en espacio es* $O(k)$ *(sin considerar el espacio de* TS*).*

Demostración. **(1)** Dado que TS es suficientemente grande, los simplex requeridos para el cómputo yacen en TS, por lo que se pueden obtener en tiempo constante. Aplicando esto , y la proposición 2.2.27 en la proposición 3.3.6, se tiene que la complejidad en espacio es $O\big(k \log_2(\|d\|_k)\big)$.

De forma completamente análoga para la complejidad en espacio, las proposiciones 2.1.16 y 2.2.20 el ciclo for de las líneas 1-4 toma espacio $O(1)$ y por la proposición 3.1.14 la línea 5 toma espacio $O(k)$ resultando en $O(k)$.

(2) Como todo número simplex requerido se puede obtener en tiempo constante, por la proposición 3.3.8, se sigue que la complejidad en tiempo es $O(k)$. De la proposición mencionada, el ciclo for de las líneas 3-5, tiene complejidad en espacio $O(1)$, al igual que la línea 2. Por la proposición 3.1.12, la línea 1 toma espacio $O(k)$ resultando en complejidad en espacio $O(k)$. QED

[7] Por la observación 2.2.8, también es $O\big(k \log_2(k \sqrt[k]{d})\big)$

Capítulo 4

Enumeración de k-tuplas

En este capítulo se generalizará los algoritmos Rank-Comp y Unrank-Comp planteados en el capítulo pasado para que enumeren todas las k-tuplas de naturales. Se plantean versiones que resuelven esta tarea basándose en los algoritmos Rank-Comp y Unrank-Comp y se mostrará su corrección para que basado en esto, se planteen versiones equivalentes que ahorran espacio.

4.1. Los algoritmos Rank y Unrank de k-tuplas

Trabajando las funciones Rank y Unrank entre $\mathbb{N}^{k+1}$ y $\mathbb{N}$, particionamos estos conjuntos en la familia de diagonales $\text{Diag}_{k+1}(n)$ y la familia de gnomons $\text{Gnom}_{k+1}(n)$ respectivamente. Tanto $\text{Diag}_{k+1}(n)$ como $\text{Gnom}_{k+1}(n)$ tienen $S_k(n+1)$ elementos (ver proposición 3.1.3).

Naturalmente la idea es definir estas funciones a pedazos donde se tenga un mapeo entre $\text{Diag}_{k+1}(n)$ y $\text{Gnom}_{k+1}(n)$. El proceso de la enumeración interna de una diagonal descrito por los algoritmos Rank-Comp y Unrank-Comp hacen la mayor parte de esta labor al plantear un mapeo entre $I_{S_k(n)}$ y $\text{Diag}_{k+1}(n)$. Mapear el segmento inicial $I_{S_k(n)}$ y $\text{Gnom}_{k+1}(n)$ es muy sencillo pues basta con hacer una traslación que consta de sumar $S_{k+1}(n)$ a los elementos de $I_{S_k(n)}$. Esta traslación la denotamos con la función τ_k como se puede apreciar en la figura 4.1. Es importante recordar que $\text{Gnom}_{k+1}(n)$ y $\text{Diag}_{k+1}(n)$ forman particiones de $\mathbb{N}$ y $\mathbb{N}^{k+1}$ respectivamente, y en cada una de estas se tiene una enumeración individual. La enumeración total consiste en "apilar" estas enumeraciones trasladándolas lo necesario.

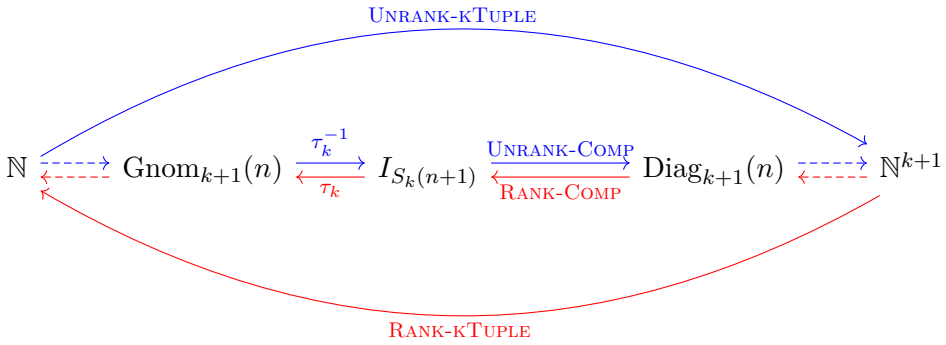


Figura 4.1: Diagrama de funciones biyectivas que describen el proceso de Unrank-kTuple y Rank-kTuple

Algoritmo Unrank-kTuple

El algoritmo inverso de enumeración UNRANK-KTUPLE sigue las mismas ideas. Primero calcula el índice del $k+1$-gnomon al que pertenece el natural de entrada. Luego, la diferencia de este valor con el inicio del gnomon es exactamente el valor de la enumeración interna de una diagonal, por lo que con la llamada a UNRANK-COMP se puede obtener la $k+1$-tupla correspondiente.

Algoritmo 11 UNRANK-KTUPLE$(m, k+1)$

Entrada: Una pareja de números $m, k+1 \in \mathbb{N}$ con $m \in \text{Gnom}_{k+1}(n)$ para alguna n,
Salida: Una $k+1$-composición de n.

1: $n \leftarrow \lfloor\!\lfloor m \rfloor\!\rfloor_{k+1}$ ▷ Índice del $k+1$-gnomon de m.
2: $d \leftarrow m - S_{k+1}(n)$ ▷ Distancia al inicio del gnomon.
3: $(x_1, \ldots, x_{k+1}) \leftarrow$ UNRANK-COMP$(d, n, k+1)$ ▷ $k+1$-tupla en $\text{Diag}_{k+1}(n)$.
4: **return** $(x_1, \ldots, x_{k+1})$

Afirmación 4.1.1. El algoritmo UNRANK-KTUPLE (11) termina.

Demostración. La línea 1 termina pues es un cálculo de FLOORSIMPLEX, y la línea 2 termina pues es un cálculo de un simplex, así como una resta. Por la afirmación 3.3.3, la línea 3 termina, llegándose a que la ejecución termina. QED

Proposición 4.1.2 (Unrank es correcto)**.** *Si la entrada al algoritmo* UNRANK-KTUPLE *(11) es $m, k+1$, con $m, k \in \mathbb{N}$ y $m \in \text{Gnom}_{k+1}(n)$ para alguna $n \in \mathbb{N}$ y la salida es $(x_1, \ldots, x_{k+1})$ entonces existe d tales que:*

1. $\lfloor\!\lfloor m \rfloor\!\rfloor_{k+1} = n$,
2. $m = d + S_{k+1}(n)$,
3. $d < S_k(n+1)$,
4. UNRANK-COMP$(d, n, k+1) = (x_1, \ldots, x_{k+1})$, *y*
5. $(x_1, \ldots, x_{k+1}) \in \text{Diag}_{k+1}(n)$.

Demostración. Sea $m \in \mathbb{N}$. Por la proposición 3.1.4, para $k+1 \in \mathbb{N}$, existe un único $n_0 \in \mathbb{N}$ tal que $m \in \text{Gnom}_{k+1}(n_0)$. Por definición $m = S_{k+1}(n_0) + d_0$ para alguna $d_0 \in \mathbb{N}$. Por la proposición 3.2.1 se sigue que $d_0 < S_k(n+1)$. Por el corolario 3.2.3 necesariamente $\lfloor\!\lfloor m \rfloor\!\rfloor_{k+1} = n_0$.

De la línea 1 se sigue que $n = \lfloor\!\lfloor m \rfloor\!\rfloor_{k+1} = n_0$. De la línea 2 se tiene que $d = m - S_{k+1}(n)$ y dado que $m = d_0 + S_{k+1}(n_0)$ sucede que $m - S_{k+1}(n_0) = d_0$, es decir $m - S_{k+1}(n) = d_0$. Por lo que $d = d_0$.

De la línea 3 se tiene que UNRANK-COMP$(d, n, k+1) = (x_1, \ldots, x_{k+1})$, en otras palabras, UNRANK-COMP$(d_0, n_0, k+1) = (x_1, \ldots, x_{k+1})$. Se sigue que $(x_1, \ldots, x_{k+1}) \in \text{Diag}_{k+1}(n_0)$, gracias a la proposición 3.3.5.

QED

Algoritmo Rank-kTuple

Definimos el algoritmo RANK-KTUPLE que dada una $k+1$-tupla de naturales, calcula la $k+1$-diagonal a la que pertenece, así como el valor en la enumeración interna de esta diagonal (labores que realiza RANK-COMP) para sumarle $S_k(n)$ y obtener el valor en el $k+1$-gnomon correspondiente.

Algoritmo 12 RANK-KTUPLE$((x_1, \ldots, x_{k+1}))$

Entrada: Una $k+1$-composición $(x_1, \ldots, x_{k+1}) \in diag_{k+1}(n)$ de algún valor n.
Salida: Un natural $m \in \text{Gnom}_{k+1}(n)$.
1: $d, n \leftarrow$ RANK-COMP$((x_1, \ldots, x_{k+1}))$ ▷ Distancia d al inicio del n-ésimo $k+1$-gnomon.
2: $m \leftarrow S_{k+1}(n) + d$ ▷ Inicio del $k+1$-gnomon más distancia.
3: **return** m

Afirmación 4.1.3. El algoritmo RANK-KTUPLE (12) termina.

Demostración. Por la proposición 3.3.7, la llamada en la línea 1, termina. La línea 2 es una asignación junto con el cálculo de un simplex y una suma por lo que toda la ejecución termina. QED

Proposición 4.1.4 (Rank es correcto)**.** *Si* $(x_1, \ldots, x_{k+1}) \in \text{Diag}_{k+1}(n)$ *para alguna* $n \in \mathbb{N}$ *es la entrada al algoritmo* RANK-KTUPLE *(12) y la salida es* $m \in \mathbb{N}$*, entonces existe* d *tal que:*

1. RANK-COMP$((x_1, \ldots, x_{k+1})) = d, n$,
2. $m = d + S_{k+1}(n)$,
3. $d < S_k(n+1)$, *y*
4. $m \in \text{Gnom}_{k+1}(n)$.

Demostración. Sea $(x_1, \ldots, x_{k+1}) \in \mathbb{N}^{k+1}$. Por la proposición 3.1.4 existe un único $n_0 \in \mathbb{N}$ tal que $(x_1, \ldots, x_{k+1}) \in \text{Diag}_{k+1}(n_0)$. Es claro por la línea 1 y 2 que la salida es $m = d + S_{k+1}(n)$, donde $d, n =$ RANK-COMP$(x_1, \ldots, x_{k+1})$. Por la proposición 3.3.7 se tiene que $n_0 = n$ y $d < S_{k+1}(n)$. Por la proposición 3.2.1 y el corolario 3.2.3 se tiene que $m \in \text{Gnom}_{k+1}(n)$. QED

Se puede mostrar que la complejidad en espacio de UNRANK-KTUPLE(m, k)(11) es $O(k)$ y la complejidad en tiempo es $O(k^2 \log_2(\lVert m \rVert_k))$ don $k > 1$. Asimismo, si $k > 1$ se puede mostrar que la complejidad en espacio de RANK-KTUPLE$(x_1, \ldots, x_k)$ (12) es $O(k)$ y en tiempo es $O(k^2)$. Esto último provisto que se utilicen los algoritmos (1) y (3) para el manejo de simplex. En la siguiente sección daremos versiones compactas que ocupan espacio constante y toman el mismo tiempo. Será en tales versiones que se hará el análisis de complejidad con detalle para las distintas maneras de manejar simplex.

Teorema 4.1.5 (Inversión de Rank y Unrank)**.** *Sean* $d, k, n \in \mathbb{N}$ *y* $(x_1, \ldots, x_{k+1}) \in \mathbb{N}^{k+1}$ *con* $k > 0$*, se siguen las siguientes:*

1. RANK-KTUPLE$\Big($UNRANK-KTUPLE$(m, k+1)\Big) = m$, *y*
2. UNRANK-KTUPLE$\Big($RANK-KTUPLE$((x_1, \ldots, x_{k+1})), k+1\Big) = (x_1, \ldots, x_{k+1})$.

Demostración. **(1)** Sean $(x_1, \ldots, x_{k+1}) =$ UNRANK-KTUPLE$(m, k+1)$ y $n_0 \in \mathbb{N}$ tal que $m \in \text{Gnom}_{k+1}(n_0)$. De lo anterior y por la proposición 4.1.2 existen $d_0, n \in \mathbb{N}$ para los cuales se siguen las siguientes:

(i) $m = d_0 + S_{k+1}(n)$,
(ii) $d_0 < S_k(n+1)$,
(iii) UNRANK-COMP$(d_0, n, k+1) = (x_1, \ldots, x_{k+1})$, y
(iv) $(x_1, \ldots, x_{k+1}) \in \text{Diag}_{k+1}(n)$.

Sea $m_1 \in \mathbb{N}$ tal que RANK-KTUPLE$(x_1, \ldots, x_{k+1}) = m_1$. Por la proposición 4.1.4 se siguen las siguientes,

(v) $m_1 \in \text{Gnom}_{k+1}(n)$ por (iv), (vii) $d_1 < S_k(n+1)$, y

(vi) $m_1 = d_1 + S_{k+1}(n)$ para alguna d_1, (viii) $\text{Rank-Comp}(x_1, \ldots, x_{k+1}) = d_1, n$.

Por (iii) se tienen que $\text{Unrank-Comp}(d_0, n, k+1) = (x_1, \ldots, x_{k+1})$ por lo tanto

$$\text{Rank-Comp}\Big(\text{Unrank-Comp}(d_0, n, k+1)\Big) = \text{Rank-Comp}(x_1, \ldots, x_{k+1}),$$

pero $\text{Rank-Comp}\Big(\text{Unrank-Comp}(d_0, n, k+1)\Big) = d_0, n$ por el teorema 3.3.9 y, por otro lado, de (ix) se tiene $\text{Rank-Comp}(x_1, \ldots, x_{k+1}) = d_1, n$. De esta manera $d_0 = d_1$ y se sigue que $m = d_0 + S_{k+1}(n) = d_1 + S_{k+1}(n) = m_1$.

(2) Sean $(x_1, \ldots, x_{k+1}) \in \text{Diag}_{k+1}(n)$ para alguna $n \in \mathbb{N}$ y $\text{Rank-kTuple}(x_1, \ldots, x_{k+1}) = m$. Tomemos $\text{Unrank-kTuple}(m, k+1) = (y_1, \ldots, y_{k+1})$.

Procedamos a mostrar que $(x_1, \ldots, x_{k+1}) = (y_1, \ldots, y_{k+1})$. De la proposición 4.1.4, existe n y d para los cuales:

(i) $m = d + S_{k+1}(n)$, (iii) $m \in \text{Gnom}_{k+1}(n)$, y

(ii) $d < S_k(n+1)$, (iv) $\text{Rank-Comp}(x_1, \ldots, x_{k+1}) = d, n$.

Si aplicamos Unrank-Comp a (vi), por el teorema 3.3.9 se tiene que

$$\begin{aligned}(x_1, \ldots, x_{k+1}) &= \text{Unrank-Comp}\Big(\text{Rank-Comp}(x_1, \ldots, x_{k+1}), k+1\Big) \\ &= \text{Unrank-Comp}(d, n, k+1). \end{aligned} \tag{4.1}$$

Por otro lado, si $\text{Unrank-kTuple}(m, k+1) = (y_1, \ldots, y_{k+1})$, entonces por la proposición 4.1.2 se tiene que $\text{Unrank-Comp}(d, n, k+1) = (y_1, \ldots, y_{k+1})$ por (iii). De la ecuación (4.1) se concluye que $(x_1, \ldots, x_{k+1}) = (y_1, \ldots, y_{k+1})$. QED

4.2. Versiones compactas para la enumeración de k-tuplas

Los siguientes algoritmos Rank-kTuple (12) y Unrank-kTuple (11) son bastante claros, pero el proceso las llamadas internas a Rank-Comp (10), Sep2Comp (8), Unrank-Comp (9) y Comp2Sep (7) utilizan memoria $O(k)$ para cálculos que sólo se requieren hacer una vez. Reutilizando la memoria, se puede hacer una versión compacta que utilice memoria constante para hacer estos mismos cálculos.

Algoritmo Unrank-kTuple compacto

Algoritmo 13 UNRANK-KTUPLE$(m, k+1)$(compacto)

Entrada: Una pareja de números $m, k+1 \in \mathbb{N}$ con $m \in \text{Gnom}_{k+1}(n)$ para alguna n,
Salida: Una $k+1$-composición de n.

1: $n \leftarrow \lfloor\!\lfloor m \rfloor\!\rfloor_{k+1}$
2: $d \leftarrow m - S_{k+1}(n)$
3: **for** $i = k$ **downto** 1 **do**
4: $\quad a \leftarrow \lfloor\!\lfloor d \rfloor\!\rfloor_i$
5: $\quad x_{i+1} \leftarrow n - a$
6: $\quad n \leftarrow a$
7: $\quad d \leftarrow d - S_i(a)$
8: **end for**
9: $x_1 \leftarrow a$
10: **return** $(x_1, \ldots, x_{k+1})$

Afirmación 4.2.1. El algoritmo UNRANK-KTUPLE (13) termina.

Demostración. Las líneas 1 y 2 son cálculos de un simplex y de FLOORSIMPLEX por lo que terminan. El ciclo for de las líneas 3-8 contiene puras asignaciones junto con un cálculo de FLOORSIMPLEX y un simplex. Durante este ciclo no se hacen llamadas a subrutinas que no terminen ni se modifica el contador de ninguna manera. Siendo así, este ciclo termina tras $k+1$ iteraciones y por tanto el algoritmo termina. QED

Procedamos a mostrar que el algoritmo (13) tiene la misma salida que su homónimo (11). Es claro que el ciclo for de las líneas 3-8 realiza k iteraciones y es importante hacer hincapié en que el contador desciende. Sea n_{k+1} y d_{k+1} el valor de las variables n y d antes de comenzar este ciclo. De forma semejante sean n_i, d_i, a_i los valores de n, d y a respectivamente al finalizar la iteración donde el contador vale i.

Afirmación 4.2.2. Tomemos a_i, n_i, d_i definidas como previamente mencionado. Para toda $1 \leq i \leq k$ se sigue que $(a_1, \ldots, a_i)$ forma una i-descomposición de d_{i+1}.

Demostración. Procedamos por inducción sobre k. Si $k = 1$ entonces solo hay una iteración de donde $a_1 = \lfloor\!\lfloor d_2 \rfloor\!\rfloor_1 = d_2$ por la línea 4 y así por (a_1) es una 1-descomposición de d_2.

Supongamos que el resultado se tiene para k y mostremos que se sostiene para $k+1$. Es decir, veamos que para toda $1 \leq i \leq k+1$ se sigue que $(a_1, \ldots, a_i)$ forma una i-descomposición de d_{i+1}.

Por la hipótesis inductiva se tiene que para toda $1 \leq i \leq k$ el resultado se cumple, por lo que basta mostrar que $(a_1, \ldots, a_k, a_{k+1})$ forma una $k+1$-descomposición de d_{k+2}.

De la hipótesis inductiva se tiene que

$$\sum_{i=1}^{k} S_i(a_i) = d_{k+1}, \text{ y además} \tag{4.2}$$

$$\sum_{i=1}^{j} S_i(a_i) < S_j(a_{j+1} + 1) \text{ para toda } j \in \{1, \ldots, k-1\} \tag{4.3}$$

De la línea 7 se tiene que $d_{k+1} = d_{k+2} - S_{k+1}(a_{k+1})$ y por (4.2) se concluye que

$$\sum_{i=1}^{k} S_i(a_i) = d_{k+1} = d_{k+2} - S_{k+1}(a_{k+1}) \text{ por lo tanto}$$

$$d_{k+2} = \Big(\sum_{i=1}^{k} S_i(a_i)\Big) + S_{k+1}(a_{k+1}) = \sum_{i=1}^{k+1} S_i(a_i) \tag{4.4}$$

De la línea 4 sabemos que $a_{k+1} = \lfloor\!\lfloor d_{k+2} \rfloor\!\rfloor_{k+1}$, por lo que

$$S_{k+1}(a_{k+1}) \leq d_{k+2} < S_{k+1}(a_{k+1}+1) \text{ sustituyendo con (4.2) se tiene}$$

$$\sum_{i=1}^{k+1} S_i(a_i) < S_{k+1}(a_{k+1}+1) = S_k(a_{k+1}+1) + S_{k+1}(a_{k+1}) \text{ despejando}$$

$$\Big(\sum_{i=1}^{k} S_i(a_i)\Big) + S_{k+1}(a_{k+1}) < S_k(a_{k+1}+1) + S_{k+1}(a_{k+1}) \text{ es decir}$$

$$\sum_{i=1}^{k} S_i(a_i) < S_k(a_{k+1}+1) \tag{4.5}$$

Conjuntando (4.5) y (4.3) se tiene que $\sum_{i=1}^{j} S_i(a_i) < S_j(a_{j+1}+1)$ para toda $j \in \{1,\ldots,k\}$, así con (4.4) llegamos a lo que queríamos demostrar. QED

Afirmación 4.2.3. Si tomamos a_i, d_i y n_i como previamente mencionado y $(x_1,\ldots,x_{k+1})$ la salida de Rank-kTuple (13), entonces:

1. $d_{k+1} < S_{k+1}(n_{k+1}+1)$,
2. Sep2Comp$((a_1,\ldots,a_k), n_{k+1}) = (x_1,\ldots,x_{k+1})$, y
3. Unrank-Comp$(d_{k+1}, n_{k+1}, k+1) = (x_1,\ldots,x_{k+1})$.

Demostración. Recordemos que x_i, n_i, a_i se definen a través de la ejecución de Unrank-kTuple (13).

(1) De las líneas 1 y 2 se tiene que $n_{k+1} = \lfloor\!\lfloor m \rfloor\!\rfloor_{k+1}$ y $d_{k+1} = m - S_{k+1}(n_{k+1})$. De la proposición 3.2.2 se sigue que $d_{k+1} < S_{k+1}(n_{k+1}+1)$.

(2) De las líneas 5 y 6 para toda $1 \leq i \leq k$ se sigue que $x_{i+1} = n_{i+1} - a_i$ y $n_i = a_i$ respectivamente.

Por la afirmación 4.2.2 sabemos que $(a_1,\ldots,a_k)$ es una k-descomposición de d_{k+1}. Aplicando esto último y el inciso (1) en el teorema 3.3.1 se tiene que $(a_1,\ldots,a_k)$ es una representación por separadores de una $k+1$-composición de n_{k+1}.

Por otro lado, de la línea 5 se sigue que $x_{k+1} = n_{k+1} - a_k$, y también $x_{i+1} = n_{i+1} - a_i = a_{i+1} - a_i$ para $1 \leq i \leq k-1$. De la línea 9 se tiene $x_1 = a_1$. Por la proposición 3.1.6 necesariamente $(x_1,\ldots,x_{k+1})$ es una $k+1$-composición de n_{k+1} cuya representación por separadores es $(a_1,\ldots,a_k)$.

Aplicando lo anterior a la proposición 3.1.13 se sigue que Sep2Comp$((a_1,\ldots,a_k), n_{k+1}) = (x_1,\ldots,x_{k+1})$.

(3) Del inciso (2) se tiene que $(a_1,\ldots,a_k)$ es una k-descomposición de d_{k+1} y además es la representación por separadores de $(x_1,\ldots,x_{k+1})$. Dado que estas representaciones son únicas (ver 3.2.8 y 3.1.8), por la proposición 3.3.5 se sigue que Unrank-Comp$(d_{k+1}, n_{k+1}, k+1) = (x_1,\ldots,x_{k+1})$. QED

Proposición 4.2.4. *Si la entrada para los algoritmos* UNRANK-KTUPLE *(11) y (13) es $m, k+1$, entonces coinciden en salida.*

Demostración. Tomemos d_i, n_i los valores previamente definidos para la versión compacta del algoritmo UNRANK-KTUPLE (13). Denotemos a los valores de las variables d y n en el algoritmo (11) como $\hat{d}$ y $\hat{n}$ respectivamente. Si estos algoritmos tienen como entrada $m, k+1$, de las líneas 1 y 2 en (13) se tiene que $n_{k+1} = \lVert m \rVert_{k+1}$ y $d_{k+1} = m - S_{k+1}(n_{k+1})$. De igual forma en las líneas 1 y 2 en (11) se obtiene $\hat{n} = \lVert m \rVert_{k+1}$ y $\hat{d} = m - S_{k+1}(\hat{n})$, de donde se sigue que $\hat{n} = n_{k+1}$ y $\hat{d} = d_{k+1}$.

Si la salida de la versión compacta es $(x_1, \ldots, x_{k+1})$, siguiendo con la ejecución de (11) llegamos a que se llama a UNRANK-COMP$(\hat{d}, \hat{n}, k+1)$, pero esto último es equivalente a llamar UNRANK-COMP$(d_{k+1}, n_{k+1}, k+1)$.

Por la afirmación 4.2.3 se tiene que la salida de esta última llamada debe ser $(x_1, \ldots, x_{k+1})$, entonces la salida de (11) debe de ser la misma que de (13), llegándose a lo que se quería mostrar. QED

Algoritmo Rank-kTuple compacto

Algoritmo 14 RANK-KTUPLE$((x_1, \ldots, x_{k+1}))$(compacto)

Entrada: Una $k+1$-composición $(x_1, \ldots, x_{k+1}) \in \text{Diag}_{k+1}(n)$ de algún valor n.
Salida: Un natural $m \in \text{Gnom}_{k+1}(n)$.

```
1: a ← 0
2: m ← 0
3: for i = 1 upto k + 1 do
4:     a ← x_i + a
5:     m ← S_i(a) + m
6: end for
7: return m
```

Veamos que esta versión de RANK-KTUPLE y la versión en el algoritmo homónimo (12) coinciden en salidas para entradas iguales.

Afirmación 4.2.5. El algoritmo RANK-KTUPLE (14) termina.

Demostración. Basta mostrar que el ciclo for de las líneas 3-6 termina. Esto es fácil de ver pues ya que en las líneas 4 y 5 solo se hacen sumas y un cálculo de un simplex y jamás se altera el índice i de este ciclo ni su cota superior. QED

Sean a_0, m_0 los valores de las variables a y m antes de iniciar el ciclio for de las líneas 3-6 y sean a_i, m_i sus valores tras la i-ésima iteración de este ciclo. Se cumplen las siguientes afirmaciones.

Afirmación 4.2.6. Sea $(x_1, \ldots, x_{k+1})$ la entrada del algoritmo RANK-KTUPLE (14). Para toda $i \in \mathbb{N}$, si $i > 0$ se cumplen las siguientes:

1. $a_i = \sum_{j=1}^{i} x_j$, y
2. $m_i = \sum_{j=1}^{i} S_j(a_j)$.

Demostración. Demostremos la conjunción de estos incisos por inducción sobre i.

Caso Base. Para $i = 0$ se siguen ambas afirmaciones por vacuidad. Para $i = 1$ se tiene que por la línea 4 $a_1 = x_1 + a_0 = x_1 + 0 = x_1$. Por la línea 5 se sigue que $m_1 = S_1 a_1 + m_0 = S_1 a_1 + 0 = S_1 a_1$.

Caso Inductivo. Supongamos que los incisos (1) y (2) se siguen para la iteración i y procedamos que se siguen para la iteración $i+1$.

En efecto, de la línea 4 se tiene que $a_{i+1} = x_{i+1} + a_i = x_{i+1} + \sum_{j=1}^{i} a_j = \sum_{j=1}^{i+1} a_j$. De la línea 5 se tiene que $m_{i+1} = S_{i+1}(a_{i+1}) + m_i = S_{i+1}(a_{i+1}) + \sum_{j=1}^{i} S_j(a_j) = \sum_{j=1}^{i+1} S_j(a_j)$ QED

Proposición 4.2.7. *Tomemos los valores a_i y m_i como previamente definidos. Sea $(x_1, \ldots, x_{k+1})$ la entrada al algoritmo* Rank-kTuple *(14). Se cumplen las siguientes.*

1. *$(a_1, \ldots, a_k)$ es una representación por separadores de $(x_1, \ldots, x_{k+1})$,*
2. *Si* Rank-Comp$(x_1, \ldots, x_{k+1}) = d, n$, *entonces $a_{k+1} = n$ y $d = \sum_{i=1}^{k} S_i(a_i)$.*
3. *Si la entrada a los algoritmos homónimos* Rank-kTuple *(12) y (14) es la misma, entonces coinciden en salida, esto es $m_{k+1} = d + S_{k+1}(n)$.*

Demostración. **(1)** De la afirmación 4.2.6 se tiene que $a_i = \sum_{j=1}^{i} x_j$ para $1 \leq i \leq k$, por lo que por definición se tiene el resultado.

(2) Sea Rank-Comp$(x_1, \ldots, x_{k+1}) = d, n$. De la proposición 3.3.7 se tiene que $n = \sum_{i=1}^{k+1} x_i = a_{k+1}$, esta última igualdad por la afirmación 4.2.6.

Por el inciso (1), al ser $(a_1, \ldots, a_k)$ una representación por separadores de la entrada, entonces $d = \sum_{i=1}^{k} S_i(a_i)$ por la proposición 3.3.7.

(3) Observemos que la salida en el algoritmo (14) es m_{k+1} pues el ciclo for de las líneas 3-6 termina en $k+1$ iteraciones.

Basta mostrar que $m_{k+1} = d + S_{k+1}(n)$ que corresponde con la salida del algoritmo (12). Por la afirmación 4.2.6 y en inciso (2) se sigue que

$$m_{k+1} = \sum_{i=1}^{k+1} S_i(a_i) = \sum_{i=1}^{k} S_i(a_i) + S_{k+1}(a_{k+1}) = d + S_{k+1}(n).$$

QED

Corolario 4.2.8 (Rank y Unrank son correctos e inversos)**.** *Para los algoritmos* Rank-kTuple *(14) y* Unrank-kTuple *(13) en sus versiones compactas, se sostienen las proposiciones 4.1.4 y 4.1.2 respectivamente. Adicionalmente, uno es inverso del otro, esto es, se sigue cumpliendo el teorema 4.1.5.*

Demostración. De las proposiciones 4.2.7 y 4.2.4 estos algoritmos tienen las mismas salidas que sus homónimos (14) y (13), los cuales satisfacen las proposiciones 4.1.4 y 4.1.2 y el teorema 3.3.9. QED

4.3. Propiedades y complejidades de enumeración de k-tuplas

Teorema 4.3.1. *Son equivalentes las siguientes:*

1. Rank-kTuple$((x_1, \ldots, x_{k+1})) = m$,
2. $\left(\sum_{i=1}^{1} x_i, \ldots, \sum_{i=1}^{k+1} x_i\right)$ *es una $k+1$-descomposición de m, y*
3. Unrank-kTuple$(m, k+1) = (x_1, \ldots, x_{k+1})$.

Demostración. **(1)⇒(2)**. Supongamos que RANK-kTUPLE$((x_1,\ldots,x_{k+1}))=m$.

Definamos $a_j:=\sum_{i=1}^{j}x_i$ para toda $j\in\{1,\ldots,k+1\}$, es decir, $(a_1,\ldots,a_k)$ es una representación por separadores de $(x_1,\ldots,x_{k+1})$. Sea $n\in\mathbb{N}$ tal que $(x_1,\ldots,x_{k+1})\in \mathrm{Diag}_{k+1}(n)$, entonces $n=a_{k+1}$.

De la proposición 4.1.4 existe $d\in\mathbb{N}$ tal que se cumplen las siguientes:

(i) RANK-COMP$((x_1,\ldots,x_{k+1}))=d,n$,
(ii) $m=d+S_{k+1}(n)$,
(iii) $d<S_k(n+1)$, y
(iv) $m\in \mathrm{Gnom}_{k+1}(n)$.

Aplicando la proposición 3.3.7 al inciso (i), sabemos que $(a_1,\ldots,a_k)$ es una k-descomposición de d.

Procedamos a mostrar que $(a_1,\ldots,a_{k+1})$ es una $k+1$-descomposición de m. Esto es, mostremos que:

(v) $m=\sum_{i=1}^{k+1}S_i(a_i)$ y (vi) $\sum_{i=1}^{j}S_i(a_i)<S_j(a_{j+1}+1)$ para toda $j\in\{1,\ldots,k\}$.

Para mostrar (v) basta analizar el inciso (ii). Como $(a_1,\ldots,a_k)$ es una k-descomposición de d, entonces $d=\sum_{i=1}^{k}S_i(a_i)$. De aquí obtenemos

$$m=d+S_{k+1}(n)=\Big(\sum_{i=1}^{k}S_i(a_i)\Big)+S_{k+1}(n)=\Big(\sum_{i=1}^{k}S_i(a_i)\Big)+S_{k+1}(a_{k+1})=\sum_{i=1}^{k+1}S_i(a_i)$$

De nuevo, al $(a_1,\ldots,a_k)$ ser una k-descomposición de d tenemos que $\sum_{i=1}^{j}S_i(a_i)<S_j(a_{j+1}+1)$ para toda $j\in\{1,\ldots,k-1\}$. Por el inciso (iii), se tiene que

$$d<S_k(n+1),\ \text{es decir}\ \sum_{i=1}^{k}S_i(a_i)<S_k(a_{k+1}+1).$$

De esto último se sigue (vi).

(2)⇒(3). Sean $(x_1,\ldots,x_{k+1})\in\mathbb{N}^{k+1}$ y $(a_1,\ldots,a_{k+1})$ definido como antes de forma que sea una $k+1$-descomposición de m. Sea UNRANK-kTUPLE$(m,k+1)=(y_1,\ldots,y_{k+1})$. Procedamos a mostrar que $x_i=y_i$ para toda $i\in\{1,\ldots,k+1\}$.

Por la proposición 4.1.2 existen n y d tales que se siguen las siguientes:

(vii) $\llbracket m\rrbracket_{k+1}=n$,
(viii) $m=d+S_{k+1}(n)$,
(ix) $d<S_k(n+1)$,
(x) UNRANK-COMP$(d,n,k+1)=(y_1,\ldots,y_{k+1})$, y
(xi) $(y_1,\ldots,y_{k+1})\in \mathrm{Diag}_{k+1}(n)$.

Por el teorema 3.2.6 se sigue que $\llbracket m\rrbracket_{k+1}=a_{k+1}$, conjuntando con el inciso (vii) se sigue que

$$n=\llbracket m\rrbracket_{k+1}=a_{k+1}=\sum_{i=1}^{k+1}x_i\quad\text{y por tanto}\quad(x_1,\ldots,x_{k+1})\in\mathrm{Diag}_{k+1}(n).$$

Nuestra hipótesis nos afirma que $(a_1,\ldots,a_{k+1})$ es una $k+1$-descomposición de m, por lo que $m=\sum_{i=1}^{k+1}S_i(a_i)$ y, además $\sum_{i=1}^{j}S_i(a_i)<S_j(a_{j+1}+1)$ para toda $j\in\{1,\ldots,k\}$.

Por el inciso (viii) se tiene que

$$d+S_{k+1}(n)=m=\sum_{i=1}^{k+1}S_i(a_i)=\Big(\sum_{i=1}^{k}S_i(a_i)\Big)+S_{k+1}(a_{k+1})\quad\text{por lo que}\quad d=\sum_{i=1}^{k}S_i(a_i).$$

Lo anterior muestra que $(a_1, \ldots, a_k)$ forma una k-descomposición de d. Aplicando la proposición 3.3.5 a los incisos (ix), (x) y (xi), se concluye que $(a_1, \ldots, a_k)$ es la representación por separadores de $(y_1, \ldots, y_{k+1})$. Es claro que por hipótesis $(a_1, \ldots, a_k)$ es también la representación por separadores de $(x_1, \ldots, x_{k+1})$ y por la proposición 3.1.8 se tiene que $(x_1, \ldots, x_{k+1}) = (y_1, \ldots, y_{k+1})$, como se quería mostrar.

(1)⇔(3). Se sigue directamente del teorema 4.1.5. QED

Teorema 4.3.2 (Enumeración estructuralmente creciente)**.** *Para toda $k \in \mathbb{Z}^+$, $n, m \in \mathbb{N}$, $(x_1, \ldots, x_k), (y_1, \ldots, y_k) \in \mathbb{N}^k$ suceden los siguientes.*

1. *Si* Unrank-kTuple$(m, k) = (x_1, \ldots, x_k)$*, entonces* $\sum_{i=1}^{k} x_i = \lVert m \rVert_k$.
2. *Si* $n < m$, Unrank-kTuple$(n, k) = (x_1, \ldots, x_k)$ *y* Unrank-kTuple$(m, k) = (y_1, \ldots, y_k)$, *entonces* $\sum_{i=1}^{k} x_i \leq \sum_{i=1}^{k} y_i$.
3. *Si* Rank-kTuple$(y_1, \ldots, y_k) = n$ Rank-kTuple$(x_1, \ldots, x_k) = m$, *y* $\sum_{i=1}^{k} y_i < \sum_{i=1}^{k} x_i$, *entonces* $n < m$.

Demostración. **(1)** Por el teorema 4.3.1 sabemos que $\left(\sum_{i=1}^{1} x_i, \ldots, \sum_{i=1}^{k} x_i\right)$ forma una k-descomposición de m. Así por el teorema 3.2.6 se sigue que $\lVert m \rVert_k = \sum_{i=1}^{k} x_i$.

(2) Si $n < m$ entonces por la proposición 2.2.3 $\lVert n \rVert_k \leq \lVert m \rVert_k$. Por el inciso (1) se sigue que $\sum_{i=1}^{k} x_i \leq \sum_{i=1}^{k} y_i$.

(3) Si $\sum_{i=1}^{k} y_i < \sum_{i=1}^{k} x_i$, por el inciso (1) se tiene que $\lVert n \rVert_k < \lVert m \rVert_k$. Por la proposición 2.2.3 se sigue que $n < m$. QED

Proposición 4.3.3. *Si el cálculo de un número simplex $S_k(n)$ y el índice de un número simplex $\lVert m \rVert_k$ tienen complejidad $\mathcal{S}(k, n)$ y $\mathcal{FS}(m, k)$ respectivamente, entonces se cumplen las siguientes.*

1. *La complejidad en tiempo de* Unrank-kTuple$(m, k+1)$ *(compacto)(13) es*
$$O\Big((k+1)(\mathcal{FS}(m, k+1) + \mathcal{S}(k+1, \lVert m \rVert_{k+1}))\Big).$$
2. *Si $n = \sum_{i=1}^{k+1} x_i$, entonces la complejidad en tiempo de* Rank-kTuple$((x_1, \ldots, x_{k+1}))$ *(compacto)(14) es*
$$O\Big((k+1)\mathcal{S}(k+1, n)\Big).$$

Demostración. **(1)** Tomemos como entrada $(m, k+1)$ y veamos que la complejidad es $O((k+1)(\mathcal{FS}(m, k+1) + \mathcal{S}(k+1, \lVert m \rVert_{k+1})))$. Definamos a_i, d_i, n_i como en la afirmación 4.2.2.

Claramente la línea 1 toma tiempo $\mathcal{FS}(m, k+1)$ y la línea 2 toma tiempo $\mathcal{S}(k+1, \lVert m \rVert_{k+1}))$. En cada iteración del ciclo for de las líneas 3-8 se calcula $\lVert d_i \rVert_i$ y el simplex $S_i(a_i) = S_i(\lVert d_i \rVert_i)$ (línea 4).

Del teorema 4.3.1, en conjunto con el teorema 3.2.6 se sigue que $(\lVert d_1 \rVert_1, \ldots, \lVert d_k \rVert_k, \lVert m \rVert_{k+1})$ forma una $k+1$-descomposición de m. Así por el teorema 3.3.1 sucede que $\lVert d_i \rVert_i \leq \lVert m \rVert_{k+1}$ para $1 \leq i \leq k$.

Así cada iteración de la línea 4 en el ciclo for de las líneas 3-8 toma tiempo a lo más $\mathcal{FS}(m, k+1)$. De forma semejante, la línea 7 toma a lo más $\mathcal{S}(k+1, \lVert m \rVert_{k+1})$ quedando que cada iteración toma tiempo a lo más $\mathcal{FS}(m, k+1) + \mathcal{S}(k+1, \lVert m \rVert_{k+1})$. Siendo que este ciclo hace k iteraciones,

este ciclo toma tiempo $O\big(k(\mathcal{FS}(m,k+1)+\mathcal{S}(k+1,\lVert m\rVert_{k+1}))\big)$. El tiempo de este algoritmo es entonces

$$O\Big(\mathcal{FS}(m,k+1)+\mathcal{S}(k+1,\lVert m\rVert_{k+1})+k(\mathcal{FS}(m,k+1)+\mathcal{S}(k+1,\lVert m\rVert_{k+1}))\Big)=$$
$$O\Big((k+1)(\mathcal{FS}(m,k+1)+\mathcal{S}(k+1,\lVert m\rVert_{k+1}))\Big).$$

(2) Tomemos la entrada como $(x_1,\ldots,x_{k+1})$ y veamos que la complejidad es $O\big((k+1)\mathcal{S}(k+1,a)\big)$, con $a=\sum_{i=1}^{k+1}x_i$.

Las líneas 1 y 2 toman tiempo constante. Si tomamos a_i como se definió en la proposición 4.2.7, de esta misma se obtiene que $(a_1,\ldots,a_k)$ es una representación por separadores con $a_{k+1}=\sum_{i=1}^{k+1}x_i$. Por el teorema 3.3.1 $a_i\leq a_{k+1}$ para $1\leq i\leq k+1$. Así cada iteración del ciclo for de las líneas 3-6 toma a lo más $\mathcal{S}(k+1,a_{k+1})$. Siendo que son $k+1$ iteraciones, entonces, el tiempo de este ciclo y claramente también del algoritmo es $O\big(k\mathcal{S}(k+1,a_{k+1})\big)$, como se quería mostrar. QED

Proposición 4.3.4. *Si el cálculo de un número simplex y el índice de un número simplex se hacen con los algoritmos* Simplex *(1) y* FloorSimplex *(3), entonces se cumplen las siguientes.*

1. *La complejidad en tiempo del algoritmo* Unrank-kTuple$(m,k+1)$ *(compacto) (13) es* $O\big((k+1)^2\log_2(\lVert m\rVert_{k+1})\big)$ *y la complejidad en espacio es* $O(1)$ *(sin considerar la salida).*

2. *La complejidad en tiempo del algoritmo* Rank-kTuple$(x_1,\ldots,x_{k+1})$*(compacto) (14) es* $O\big((k+1)^2\big)$ *y la complejidad en espacio es* $O(1)$.

Demostración. **(1)** Aplicando las proposiciones 2.1.16 y 2.2.27 en la proposición 4.3.3, se obtiene que la complejidad en tiempo es $O\big((k+1)^2\log_2(\lVert m\rVert_{k+1})\big)$.

Para la complejidad en espacio, basta notar que el cálculo de un simplex con Simplex y de FloorSimplex toma espacio constante. Sin considerar la salida, el algoritmo 13 sólo usa las variables i,n,m,a,d por lo que toma espacio $O(1)$.

(2) Aplicando la proposición 2.1.16 en 4.3.3 se sigue que la complejidad en tiempo es $O\big((k+1)^2\big)$. El cálculo de un simplex toma espacio constante y el algoritmo 14 sólo usa las variables i,a,m por lo que su espacio es $O(1)$. QED

Proposición 4.3.5.

1. *Si* Tabla-Simplex$\big(k+2,\lVert m\rVert_{k+1}+2\big)=TS$ *(2), entonces* TS *es suficientemente grande*[1] *para computar* Unrank-kTuple$(m,k+1)$ *(13).*

2. *Si* Tabla-Simplex$\Big(k+2,2+\sum_{i=1}^{k+1}x_i\Big)=TS$ *(2), entonces* TS *es suficientemente grande para computar* Rank-kTuple$\big((x_1,\ldots,x_{k+1})\big)$ *(14).*

Demostración. **(1)** Definamos a_i, n_i y d_i como en la afirmación 4.2.2. Es fácil ver que la única llamada a FloorSimplex es en la línea 4 y en la línea 1. Más precisamente, se hacen las llamadas $\lVert d_{i+1}\rVert_i$ para $1\leq i\leq k$ y $\lVert m\rVert_{k+1}$. Veamos que todos estos índices están acotados por $\lVert m\rVert_{k+1}+1$. Claramente $\lVert m\rVert_{k+1}<\lVert m\rVert_{k+1}+1$.

Se puede ver que $a_i=\lVert d_{i+1}\rVert_i$ para $1\leq i\leq k$, por lo que basta mostrar que $a_i<\lVert m\rVert_{k+1}+1$ para $1\leq i\leq k$. Por la afirmación 4.2.2, se tiene que $(a_1,\ldots,a_k)$ es una k-descomposición. Por el teorema 3.3.1 $a_i\leq a_k$ para $1\leq i\leq k$. También por este teorema se tiene que $(a_1,\ldots,a_k)$

[1] Ver definición 2.2.22.

es la representación por separadores de la salida $(x_1, \ldots, x_{k+1})$ por lo que $a_i = \sum_{j=1}^{i} x_j$. Por el teorema 4.3.2 se tiene $\sum_{i=1}^{k+1} x_i = \lfloor\!\lfloor m \rfloor\!\rfloor_{k+1}$ y por el teorema 4.3.1

$$\left(\sum_{i=1}^{1} x_i, \ldots, \sum_{i=1}^{k} x_i, \sum_{i=1}^{k+1} x_i\right) = \left(a_1, \ldots, a_k, \lfloor\!\lfloor m \rfloor\!\rfloor_{k+1}\right) \text{ es una } k+1\text{-descomposición de } m. \quad (4.6)$$

Aplicando el teorema 3.3.1 en (4.6) se concluye que $a_i \leq a_k \leq \lfloor\!\lfloor m \rfloor\!\rfloor_{k+1} < \lfloor\!\lfloor m \rfloor\!\rfloor_{k+1} + 1$, que es lo que se quería mostrar.

Ahora veamos que los simplex calculados tienen índices y dimensión menor a $\lfloor\!\lfloor m \rfloor\!\rfloor_{k+1}$ y $k+2$ respectivamente. Es fácil observar que los únicos simplex llamados son en las líneas 2 y 4. Más precisamente, se calculan $S_{k+1}(n_{k+1}) = S_{k+1}(\lfloor\!\lfloor m \rfloor\!\rfloor_{k+1})$. y $S_i(a_i)$ para $1 \leq i \leq k$. Para el primer cómputo se sigue lo buscado. Como se dedujo de (4.6) se tiene que $a_i < \lfloor\!\lfloor m \rfloor\!\rfloor_{k+1}$ Por lo que se tiene lo buscado.

(2) Dado que en este algoritmo sólo se llaman simplex, basta mostrar que los índices y dimensiones de estos simplex están acotados por $2 + \sum_{i=1}^{k+1} x_i$ y $k+1$ respectivamente. En la línea 5 se hacen todas las llamadas a simplex que son $S_i(a_i)$ para $1 \leq i \leq k+1$. Por la afirmación 4.2.6 $a_i = \sum_{j=1}^{i} x_j$ de donde se sigue claramente que

$$a_i \leq a_{k+1} = \sum_{i=1}^{k+1} x_i < 1 + \sum_{i=1}^{k+1} x_i \quad \text{para toda } 1 \leq i \leq k+1.$$

Esto último muestra lo buscado. QED

Proposición 4.3.6. *Si el cálculo de un número simplex se hace consultando una tabla precomputada (2) TS suficientemente grande y el índice de un número simplex se obtiene con* FloorSimplex *(Con tabla) (4), entonces se cumplen las siguientes.*

1. *La complejidad en tiempo del algoritmo* Unrank-kTuple$(m, k+1)$ *(13)(compacto) es* $O((k+1)\log_2(\lfloor\!\lfloor m \rfloor\!\rfloor_{k+1}))$ *y la complejidad en espacio es* $O(1)$ *(sin considerar la tabla ni la salida).*

2. *La complejidad en tiempo del algoritmo* Rank-kTuple$((x_1, \ldots, x_{k+1}))$ *(14)(compacto) es* $O((k+1))$ *y la complejidad en espacio es* $O(1)$ *(sin considerar la tabla).*

Demostración. **(1)** Aplicando la proposición 2.2.23 en la proposición 4.3.3 y recordando que calcular un simplex con la tabla TS toma tiempo constante se obtiene que la complejidad en tiempo del algoritmo 13 es $O((k+1)\log_2(\lfloor\!\lfloor m \rfloor\!\rfloor_{k+1}))$.

Para la complejidad en espacio, el algoritmo 4 tiene complejidad $O(1)$ sin considerar la entrada TS. Sin considerar la salida, el algoritmo 13 sólo usa las variables i, n, m, a, d por lo que toma espacio $O(1)$.

(2) Dado que calcular un simplex en la tabla TS toma tiempo constante, de la proposición 2.2.23 se sigue que la complejidad en tiempo del algoritmo 14 es $O((k+1))$.

Sin contemplar la tabla TS se tiene que el algoritmo 14 sólo usa las variables i, a, m por lo que su espacio es $O(1)$. QED

4.4. Algoritmos Rank y Unrank de tuplas multidimensionales

Los algoritmos 14 y 13 establecen funciones biyectivas una inversa de la otra entre $\mathbb{N}$ y $\mathbb{N}^{k+1}$. Sin embargo, estas funciones precisan tener fija la dimensión de tuplas con las que trabaja. Los

siguientes algoritmos plantean una enumeración entre el conjunto de todas las tuplas $\bigcup_{k\in\mathbb{N}} \mathbb{N}^{k+1}$ y $\mathbb{N}$. El proceso Unrank-Tuple es de lo más natural, al transformar un valor w en una pareja (m, k), para tomar la primera entrada m en el valor de la enumeración de la tupla buscada y usar k para obtener la dimensión, y usar estos valores para aplicar Unrank. Dado que k puede ser 0 y no planteamos una enumeración con tuplas vacías (es decir, de dimensión 0), solventamos esto tomando $k + 1$ como la dimensión de búsqueda. Definamos por π^{-1} la función que toma w y la transforma en (m, k) con Unrank-kTuple y devuelve $(m, k + 1)$. Si volvemos a aplicar Unrank-kTuple con estos valores obtenemos una $k + 1$-tupla. El proceso para hacer Rank es lo recíproco, al tomar una tupla $(x_1, \ldots, x_{k+1})$, transformarla en un natural m con Rank-kTuple y mediante la función π, conjuntar estos valores en una pareja (m, k) para volver a aplicar Unrank-kTuple.

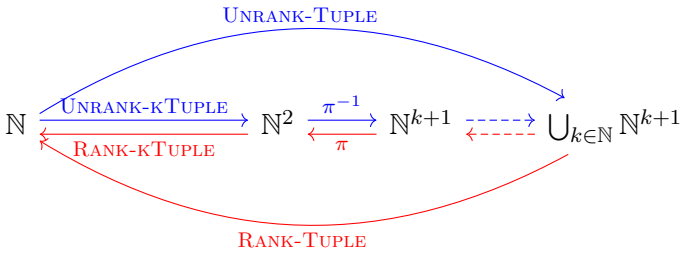


Figura 4.2: Diagrama de funciones biyectivas que describen el proceso de Unrank-Tuple y Rank-Tuple

Si denotamos a las funciones Unrank-kTuple, Rank-kTuple, Unrank-Tuple y Rank-Tuple como UK, RK, UT y RT respectivamente, podemos definir UT y RT como:

$$UT[w] := UK\big[UK[w, 2] + (0, 1)\big] \tag{4.7}$$

$$RT[(x_1 \ldots, x_k)] := RK\Big[\Big(RK\big[(x_1, \ldots, x_k)\big]\,,\, k - 1\Big)\Big] \tag{4.8}$$

Los siguientes algoritmos son bastante directos para el proceso antes descrito. Cabe mencionar, que se manejan los casos triviales aparte para simplificar el proceso. Como es de esperarse con las funciones Rank y Unrank, una es inversa de la otra. La prueba de esto yace en que Unrank-kTuple y Rank-kTuple son inversas (ver teorema 4.3.1).

Algoritmo 15 Unrank-Tuple(w)

Entrada: Un natural w.
Salida: Una k-tupla para alguna k.

```
(m, k) ← Unrank-kTuple(w, 2)      ▷ m es valor en la enumeración y k + 1 la dimensión
if k = 0 then                      ▷ Si la dimensión es 1 regresa una 1-tupla.
    return (m)
end if
return Unrank-kTuple(m, k + 1)     ▷ Regresa la k + 1-tupla correspondiente a m.
```

Algoritmo 16 RANK-TUPLE$((x_1, \ldots, x_{k+1}))$

Entrada: Una $k+1$-composición de n para alguna $k+1$ y alguna n.
Salida: Un natural w.
1: **if** $k = 0$ **then**
2: **return** RANK-KTUPLE$((x_1, 0))$
3: **end if**
4: $m \leftarrow$ RANK-KTUPLE$((x_1, \ldots, x_{k+1}))$
5: **return** RANK-KTUPLE$((m, k))$

Proposición 4.4.1 (Rank y Unrank son inversas)**.** *Para cualesquiera $w, k \in \mathbb{N}$ y $(x_1, \ldots, x_{k+1}) \in \mathbb{N}^{k+1}$ se cumplen las siguientes.*

1. RANK-TUPLE(UNRANK-TUPLE(w)) $= w$.
2. UNRANK-TUPLE(RANK-TUPLE$((x_1, \ldots, x_{k+1}))) = (x_1, \ldots, x_{k+1})$.

Demostración. **(1)** Consideremos la salida de UNRANK-TUPLE(w) una $k+1$-tupla, tenemos dos casos.

Si $k = 0$, sea UNRANK-TUPLE$(w) = (x_1)$ y mostremos que RANK-TUPLE$((x_1)) = w$. Dado que la salida de UNRANK-TUPLE es una 1-tupla, entonces necesariamente se entró en el condicional if de las líneas 2-4 del algoritmo (15). De esta manera se cumple que

$$\text{UNRANK-KTUPLE}(w, 2) = (x_1, 0) \quad \text{y así} \quad \text{RANK-KTUPLE}((x_1, 0)) = w \quad \text{por 4.1.5.} \tag{4.9}$$

Computando en el algoritmo (16) con (x_1), al $k = 0$ se entra en el condicional if de las líneas 1-3 y por la línea 2 se sigue que RANK-TUPLE$((x_1))$ = RANK-KTUPLE$((x_1, 0)) = w$ por la ecuación (4.9).

Si $k > 0$, entonces sea UNRANK-TUPLE$(w) = (x_1, \ldots, x_{k+1})$. En (15), al ser $k > 0$ tiene que ser el caso que

$$\begin{aligned} &\text{UNRANK-KTUPLE}(w, 2) = (m, k) \quad \text{y UNRANK-KTUPLE}(m, k+1) = (x_1, \ldots, x_{k+1}), \text{ y así} \\ &\text{RANK-KTUPLE}((m, k+1)) = w \quad \text{y RANK-KTUPLE}((x_1, \ldots, x_{k+1})) = m \text{ por 4.1.5.} \end{aligned} \tag{4.10}$$

Siguiendo el cómputo en (16) se sigue de las líneas 4 y 5 que RANK-TUPLE$((x_1, \ldots, x_k))$ = RANK-KTUPLE$((m, k)) = w$ por (4.10).

(2) Tomemos una $k+1$-tupla y computemos RANK-TUPLE para obtener un valor w. Tenemos dos casos.

Si $k = 0$, digamos que RANK-TUPLE$((x_1)) = w$, veamos que UNRANK-TUPLE$(w) = (x_1)$. Como $k = 0$, se sigue de la línea 2 en el algoritmo (16) que

$$\begin{aligned} \text{RANK-TUPLE}((x_1)) = &\text{RANK-KTUPLE}((x_1, 0)) = w \quad \text{y así por el teorema 4.1.5} \\ &\text{UNRANK-KTUPLE}(w, 2) = (x_1, 0) \end{aligned} \tag{4.11}$$

Computando el algoritmo (15), se entra en el condicional if de la líneas 2-4. De la ecuación (4.11) se sigue que UNRANK-TUPLE$(w) = (x_1)$.

Si $k > 0$, digamos que RANK-TUPLE$((x_1, \ldots, x_{k+1})) = w$. Al ser el caso que $k > 0$ en (16) se obtiene que

$$w = \text{RANK-kTUPLE}((m, k)) \text{ de donde } m = \text{RANK-kTUPLE}((x_1, \ldots, x_{k+1})) \text{ y por 4.1.5,}$$
$$\text{UNRANK-kTUPLE}(w, 2) = (m, k) \text{ y UNRANK-kTUPLE}(m, k+1) = (x_1, \ldots, x_{k+1}) \quad (4.12)$$

Computando el algoritmo (15), al $k > 0$ de las líneas 1 y 5 se obtiene que UNRANK-TUPLE$(w) =$ UNRANK-kTUPLE$(m, k+1) = (x_1, \ldots, x_{k+1})$, que es lo que se quería mostrar. QED

Proposición 4.4.2. *Si el cálculo de un número simplex $S_k(n)$ y el índice de un número simplex $\llbracket m \rrbracket_k$ tienen complejidad $\mathcal{S}(k, n)$ y $\mathcal{FS}(m, k)$ respectivamente, entonces se cumplen las siguientes.*

1. *La complejidad en tiempo de* UNRANK-TUPLE(w) *(15) es*

$$O\Big(\llbracket w \rrbracket_2 \Big[\mathcal{FS}(w, \llbracket w \rrbracket_2 + 1) + \mathcal{S}(\llbracket w \rrbracket_2 + 1, \llbracket w \rrbracket_2)\Big]\Big).$$

2. *La complejidad en tiempo de* RANK-TUPLE$((x_1, \ldots, x_{k+1}))$ *(16) es*

$$O\Big((k+1)\mathcal{S}(k+1, a)\Big),$$

donde $n := \sum_{i=1}^{k+1} x_i$ y $a := S_{k+1}(n+1) + k$.

Demostración. **(1)** En la línea 1 se computa $(m, k) =$ UNRANK-kTUPLE$(w, 2)$, lo cual por la proposición 4.3.3 toma tiempo

$$2\Big[\mathcal{FS}(w, 2) + \mathcal{S}(2, \llbracket w \rrbracket_2)\Big]. \quad (4.13)$$

Si fuera el caso de que $k = 0$, se entra en el condicional de las líneas 2-4, las cuales claramente toman tiempo constante. Si $k > 0$ se ejecuta la línea 5. Esta línea computa UNRANK-kTUPLE$(m, k+1)$, y por la proposición 4.3.3, toma tiempo

$$(k+1)\Big[\mathcal{FS}(m, k+1) + \mathcal{S}(k+1, \llbracket m \rrbracket_{k+1})\Big]. \quad (4.14)$$

Sumando las ecuaciones (4.13) y (4.14) se obtiene el tiempo de ejecución del algoritmo cuando $k > 0$. Es sencillo constatar que este tiempo acota superiormente al tiempo cuando $k = 0$, de donde podemos plantear la complejidad en tiempo como

$$O\Big(2\Big[\mathcal{FS}(w, 2) + \mathcal{S}(2, \llbracket w \rrbracket_2)\Big] + (k+1)\Big[\mathcal{FS}(m, k+1) + \mathcal{S}(k+1, \llbracket m \rrbracket_{k+1})\Big]\Big). \quad (4.15)$$

Procedamos a simplificar esta expresión. Por las proposiciones 4.1.2 y 4.2.4 se sigue que $(m, k) \in \text{Diag}_2(\llbracket w \rrbracket_2)$, por lo que $m+k = \llbracket w \rrbracket_2$ y así $m \leq \llbracket w \rrbracket_2$ y $k \leq \llbracket w \rrbracket_2$. Así mismo, al ser $m \leq \llbracket w \rrbracket_2$, por la proposición 2.2.3 sucede que $\llbracket m \rrbracket_{k+1} \leq m \leq \llbracket w \rrbracket_2 \leq \llbracket w \rrbracket_1 = w$, entonces $m \leq w$.

Dado que $\mathcal{FS}$ es una función de complejidad en tiempo, el tiempo aumenta cuando ambas entradas aumentan simultáneamente. Aplicando las desigualdades anteriores para w suficientemente grande obtenemos que

$$\mathcal{FS}(m, k+1) \leq \mathcal{FS}(w, \llbracket w \rrbracket_2 + 1) \qquad \text{y} \qquad \mathcal{FS}(w, 2) \leq \mathcal{FS}(w, \llbracket w \rrbracket_2 + 1). \quad (4.16)$$

De forma semejante, $\mathcal{S}$ aumenta su tiempo cuando ambas entradas aumentan simultáneamente, por lo que se sigue que

$$\mathcal{S}(k+1, \lVert m \rVert_{k+1}) \leq \mathcal{S}(\lVert w \rVert_2 + 1, \lVert w \rVert_2) \qquad \text{y} \qquad \mathcal{S}(2, \lVert w \rVert_2) \leq \mathcal{S}(\lVert w \rVert_2 + 1, \lVert w \rVert_2). \tag{4.17}$$

Utilizando las ecuaciones (4.16) y (4.17) y $k \leq \lVert w \rVert_2$, es fácil mostrar que la suma de las ecuaciones (4.13) y (4.14) es menor o igual a

$$2(\lVert w \rVert_2 + 1)\Big[\mathcal{FS}(w, \lVert w \rVert_2 + 1) + \mathcal{S}(\lVert w \rVert_2 + 1, \lVert w \rVert_2)\Big] \tag{4.18}$$

Utilizando la afirmación (4.18) en la ecuación (4.15), concluimos que el tiempo de ejecución es

$$O\bigg((\lVert w \rVert_2 + 1)\Big[\mathcal{FS}(w, \lVert w \rVert_2 + 1) + \mathcal{S}(\lVert w \rVert_2 + 1, \lVert w \rVert_2)\Big]\bigg)$$

(2) Tomemos como entrada $(x_1, \ldots, x_{k+1})$. Tenemos dos casos. Si $k = 0$, entonces la entrada es (x_1) y se entra en el condicional de la línea 1, por lo que se ejecuta la línea 2 y se finaliza la ejecución. Por la proposición 4.3.3, la línea 2 toma tiempo $O(2\mathcal{S}(2, n))$.

Si $k > 0$ entonces claramente se ejecutan las líneas 4 y 5, las cuales por la proposición 4.3.3 toman tiempo $(k+1)\mathcal{S}(k+1, n)$ y $2\mathcal{S}(2, m+k)$ respectivamente, de donde m es la salida de Rank-kTuple$((x_1, \ldots, x_{k+1}))$. Por las proposiciones 4.2.7 y 4.1.4 se sigue que $m \in \text{Gnom}_{k+1}(n)$, por lo que $m < S_{k+1}(n+1)$ donde $n = \sum_{i=1}^{k+1} x_i$.

De aquí, debido a que $\mathcal{S}$ es una función de complejidad, aumenta cuando las entradas aumentan, por lo que $\mathcal{S}(2, m+k) \leq \mathcal{S}(2, a)$, donde $a := S_{k+1}(n+1) + k$. De esta forma la ejecución cuando $k > 0$, toma tiempo $(k+1)\mathcal{S}(k+1, n) + 2\mathcal{S}(2, a)$.

Ahora, es claro que $n \leq S_{k+1}(n+1) + k = a$ por lo que $\mathcal{S}(2, n) \leq \mathcal{S}(2, a)$. Siendo así, sin importar si $k = 0$ o $k > 0$ la ejecución del algoritmo toma tiempo a lo más

$$O\Big(2\mathcal{S}(2, a) + (k+1)\mathcal{S}(k+1, n)\Big). \tag{4.19}$$

Para simplificar la expresión, tomamos k suficientemente grande, y se sigue que

$$\mathcal{S}(2, a) \leq \mathcal{S}(k+1, a) \quad \text{y} \quad \mathcal{S}(k+1, n) \leq \mathcal{S}(k+1, a). \tag{4.20}$$

Utilizando las desigualdades (4.19) en (4.20) llegamos a que la complejidad en tiempo es

$$O\Big(2(k+1)\mathcal{S}(k+1, a)\Big) = O\Big((k+1)\mathcal{S}(k+1, a)\Big).$$

QED

Proposición 4.4.3. *Si el cálculo de un número simplex y el índice de un número simplex se hacen con los algoritmos* Simplex *(1) y* FloorSimplex *(3), entonces se cumplen las siguientes.*

1. *La complejidad en tiempo del algoritmo* Unrank-Tuple(w) *(15) es* $O(\lVert w \rVert_2^2 \log_2(\lVert w \rVert_2))$ *y la complejidad en espacio es* $O(1)$ *(sin considerar la salida).*

2. *La complejidad en tiempo del algoritmo* Rank-Tuple$((x_1, \ldots, x_{k+1}))$ *(16) es* $O((k+1)^2)$ *y la complejidad en espacio es* $O(1)$.

Demostración. **(1)** Aplicando las proposiciones 2.2.20 y 2.1.16 en la proposición 4.4.2 se obtiene que el tiempo es

$$O\Big((\lVert w\rVert_2+1)\Big[(\lVert w\rVert_2+1)\log_2(w)+\lVert w\rVert_2\Big]\Big)\subseteq O\Big(\lVert w\rVert_2^2\log_2(w)\Big).$$

Por estas mismas proposiciones, el espacio usado por las líneas 1 y 5 es constante. Más aún, es claro ver que solo se usan las variables k y m por lo que el espacio es $O(1)$.

(2) Aplicando la proposición 2.1.16 en la proposición 4.4.2 se sigue que la complejidad en tiempo es $O((k+1)^2)$. Por la proposición 4.3.4 el espacio usado en las líneas 2,4 y 5 es constante. Considerando que sólo se usan las variables k y m, se sigue que el algoritmo usa espacio constante. QED

Proposición 4.4.4.

1. *Si* Tabla-Simplex$(\lVert w\rVert_2+2,\lVert w\rVert_2+2)=TS$ *(2), entonces* TS *es suficientemente grande*[2] *para computar* Unrank-Tuple(w) *(15).*
2. *Si* Tabla-Simplex$\Big(k+2,S_{k+1}(n+1)+k\Big)=TS$ *(2) con* $n:=\sum_{i=1}^{k+1}x_i$*, entonces* TS *es suficientemente grande para computar* Rank-kTuple$((x_1,\ldots,x_{k+1}))$ *(16).*

Demostración. **(1)** Si $w=0$, el resultado es directo por lo que tomemos $w\geq 1$. Para ejecutar la línea 1, por la proposición 4.3.5 se requiere una tabla de tamaño $3\times(\lVert w\rVert_2+2)$. Dado que $w\geq 1$, entonces $\lVert w\rVert_2+2\geq 3$ por lo que con una tabla de tamaño $(\lVert w\rVert_2+2)\times(\lVert w\rVert_2+2)$ basta para hacer este cómputo.

Sea $(m,k)=$ Unrank-kTuple$(w,2)$, por las proposiciones 4.1.2 y 4.2.4 es claro que que $(m,k)\in\mathrm{Diag}_2\left(\lVert w\rVert_2\right)$, por lo que

$$m\leq\lVert w\rVert_2\quad\text{ y }\quad k\leq\lVert w\rVert_2 \tag{4.21}$$

Si $k=0$ el resultado se sigue, y si $k>0$ se computa la línea 5 por lo que se calcula Unrank-Tuple$(m,k+1)$. De nuevo, por la proposición 4.3.5 se necesita una tabla de tamaño $(k+2)\times(\lVert m\rVert_{k+1}+2)$. De las desigualdades (4.21) se sigue que con una tabla de tamaño $(\lVert w\rVert_2+2)\times(\lVert w\rVert_2+2)$ se logra hacer el cómputo.

(2) Tomemos la entrada $(x_1,\ldots,x_{k+1})$ y sea $n=\sum_{i=1}^{k+1}x_i$. Si $k=0$ entonces se computa Rank-kTuple$(x_1,0)$, que por la proposición 4.3.5 requiere de una tabla de tamaño $2\times(2+x_1)$ por lo que claramente se sigue el resultado. Si $k>0$, se ejecuta la línea 4, la cual requiere una tabla de tamaño $(k+2)\times(2+n)$ por la proposición 4.3.5. Es claro que $n\leq S_{k+1}(n)$, por lo que con una tabla de tamaño $(k+2)\times(S_{k+1}(n+1)+k+2)$ basta para hacer el cómputo de la línea 4.

Sea $m=$ Rank-kTuple$((x_1,\ldots,x_{k+1}))$, por las proposiciones 4.1.4 y 4.2.7 se sigue que $m\in\mathrm{Diag}_{k+1}(n)$, por lo que $m<S_{k+1}(n)$. Ejecutando la línea 5, por la proposición 4.3.5, sabemos que esta línea se puede computar con una tabla de tamaño $3\times(m+k+2)$. De aquí es claro que con una tabla de tamaño $(k+2)\times(S_{k+1}(n+1)+k+2)$ se logra hacer el cómputo del algoritmo. QED

Proposición 4.4.5. *Si el cálculo de un número simplex se hace consultando una tabla pre-computada (2)* TS *suficientemente grande y el índice de un número simplex se obtiene con* FloorSimplex *(Con tabla) (4), entonces se cumplen las siguientes.*

[2] Ver definición 2.2.22.

1. *La complejidad en tiempo del algoritmo* Unrank-Tuple(w) *(9) es* $O(\lVert w \rVert_2 \log_2(\lVert w \rVert_2))$ *y la complejidad en espacio es* $O(1)$ *(sin considerar la tabla ni la salida).*

2. *La complejidad en tiempo del algoritmo* Rank-kTuple$((x_1, \ldots, x_{k+1}))$ *(10) es* $O((k+1))$ *y la complejidad en espacio es* $O(1)$ *(sin considerar la tabla).*

Demostración. **(1)** Recordando que obtener un un número simplex en una tabla toma tiempo constante, aplicando la proposición 2.2.23 en la proposición 4.4.2 se sigue que la complejidad en tiempo es

$$O\Big(\lVert w \rVert_2 [\log_2(w) + 1]\Big) \subseteq O\Big(\lVert w \rVert_2 \log_2(w)\Big).$$

De forma muy semejante, la prueba de que toma espacio constante es consecuencia de la proposición 4.3.4.

(2) De la proposición 4.4.2 se sigue que la complejidad en tiempo es $O(k+1)$. La prueba de que usa espacio constante es consecuencia directa de la proposición 4.3.4. QED

4.5. Pruebas de escritorio

En esta sección mostraremos algunos resultados de tiempos de ejecución de los algoritmos Simplex, FloorSimplex,Rank-kTuple,Unrank-kTuple, Rank-Tuple y Unrank-Tuple, en todas sus versiones. Las implementaciones de estos algoritmos se hicieron en *Python 3* en las libretas *Jupyter Rank_Unrank_Tuplas.ipynb*[3] y *Plot_RU.ipynb* [4]. Las pruebas se corrieron en el ambiente de *Google Colab*, el cual ofrece 12GB de RAM, dos procesadores *Intel(R) Xeon(R)* 2.20GHz y poco más de 107GB de ROM.

En general, para cada variante de los algoritmos se tomaron muestras de 1000 entradas repartidas en en un espacio fijo, donde cada muestra se ejecutó una cantidad determinada de veces y se reporta el tiempo promedio de estas ejecuciones. Cada algoritmo tiene al menos dos variantes, una sencilla (*plain*) y una versión con tabla (*table*). La tabla se calcula como en el algoritmo 4 sobre un arreglo de *numpy* [33] y cuyas entradas están en el tipo *int* innato a *Python*. Este tipo de dato tiene la ventaja que maneja enteros de tamaño arbitrario, lo cual es deseable ya que los números simplex crecen considerablemente rápido. Esto da cabida a errores de desbordamiento si se usan enteros de 2^{16}, 2^{32} o bien 2^{64} bits. En la tabla 4.1 se tienen los índices máximos n en los que $S_k(n) < 2^i$ para $i \in \{16, 32, 64\}$ con $k \leq 30$ y se puede observar que para dimensiones un tanto grandes, el índice máximo es relativamente pequeño.

Dim	2^{16}	2^{32}	2^{64}	Dim	2^{16}	2^{32}	2^{64}	Dim	2^{16}	2^{32}	2^{64}
1	$2^{16}-1$	$2^{32}-1$	$2^{64}-1$	11	8	32	272	21	5	15	62
2	361	92681	6074000999	12	8	28	207	22	5	15	57
3	72	2952	4801278	13	7	25	165	23	5	14	54
4	33	565	145053	14	7	23	137	24	5	14	51
5	21	218	18576	15	7	21	116	25	5	13	48
6	16	118	4863	16	6	20	101	26	5	13	46
7	13	77	1907	17	6	18	89	27	5	13	44
8	11	56	960	18	6	17	80	28	5	12	42
9	10	44	569	19	6	17	73	29	5	12	40
10	9	37	377	20	6	16	67	30	5	12	39

Tabla 4.1: Máximos índices de k-simplex menores a las cotas 2^{16}, 2^{32}, 2^{64} respectivamente

[3] `https://colab.research.google.com/drive/1s5_ydQWwm2OcDOeZ7VgXoTOgNNnjp9UF?usp=sharing`
[4] `https://colab.research.google.com/drive/1XZe2LSLXPS23qXb5WzVsn8BG-l7SRuRT?usp=sharing`

Simplex

Se implementó el cálculo de números simplex con dos variantes. La primera variante calcula su valor utilizando la librería de *scipy.spetial.comb* [34] cuya implementación es muy semejante al algoritmo (1).

La segunda versión se calcula con una tabla precomputada de números simplex. En la figura 4.3 se muestran los tiempos que tomó calcular un simplex con su versión sencilla y con tabla para k-simplex con dimensión $k \leq 10$ e índice $n \leq 10^7$. Cada cálculo se repitió 1000 veces y se tiene el promedio.

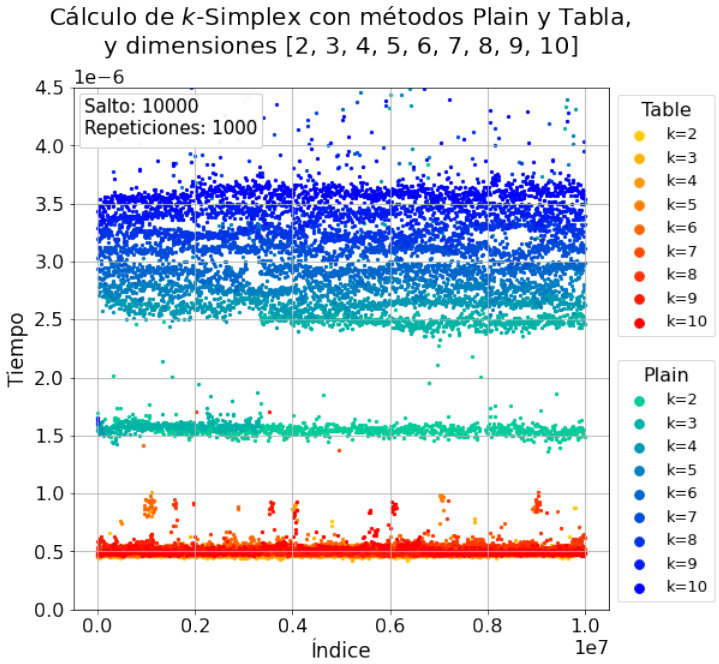


Figura 4.3: Promedio de 1000 tiempos de ejecución en muestra de 1000 índices en los primeros 10^7 valores para dimensiones $k \leq 10$

Se puede apreciar que, para cada dimensión en sus dos versiones, el tiempo de ejecución tomado es oscila alrededor de una constante. Más aún, esta constante crece a lo largo que la dimensión aumenta para la versión sencilla siguiendo la complejidad provista en la proposición 2.1.16. Para la versión con tabla el tiempo es también constante, pero sin distinguir la dimensión, lo cual era de esperarse al ser una consulta a una tabla precomputada.

Finalmente, los tiempos provistos por la versión de tabla son claramente menores a la versión sencilla, mostrándose las ventajas de tiempo.

FloorSimplex

Para este algoritmo se plantearon siete versiones distintas. La version *exact* calcula el índice del simplex que antecede con las soluciones exactas provistas en la proposición 2.2.6 para dimensiones menores a 5. Las restantes seis versiones son variantes de una búsqueda binaria, en donde cada versión es una combinación entre la forma de hacer la búsqueda de cotas para el índice y el método con el que se calculan los simplex.

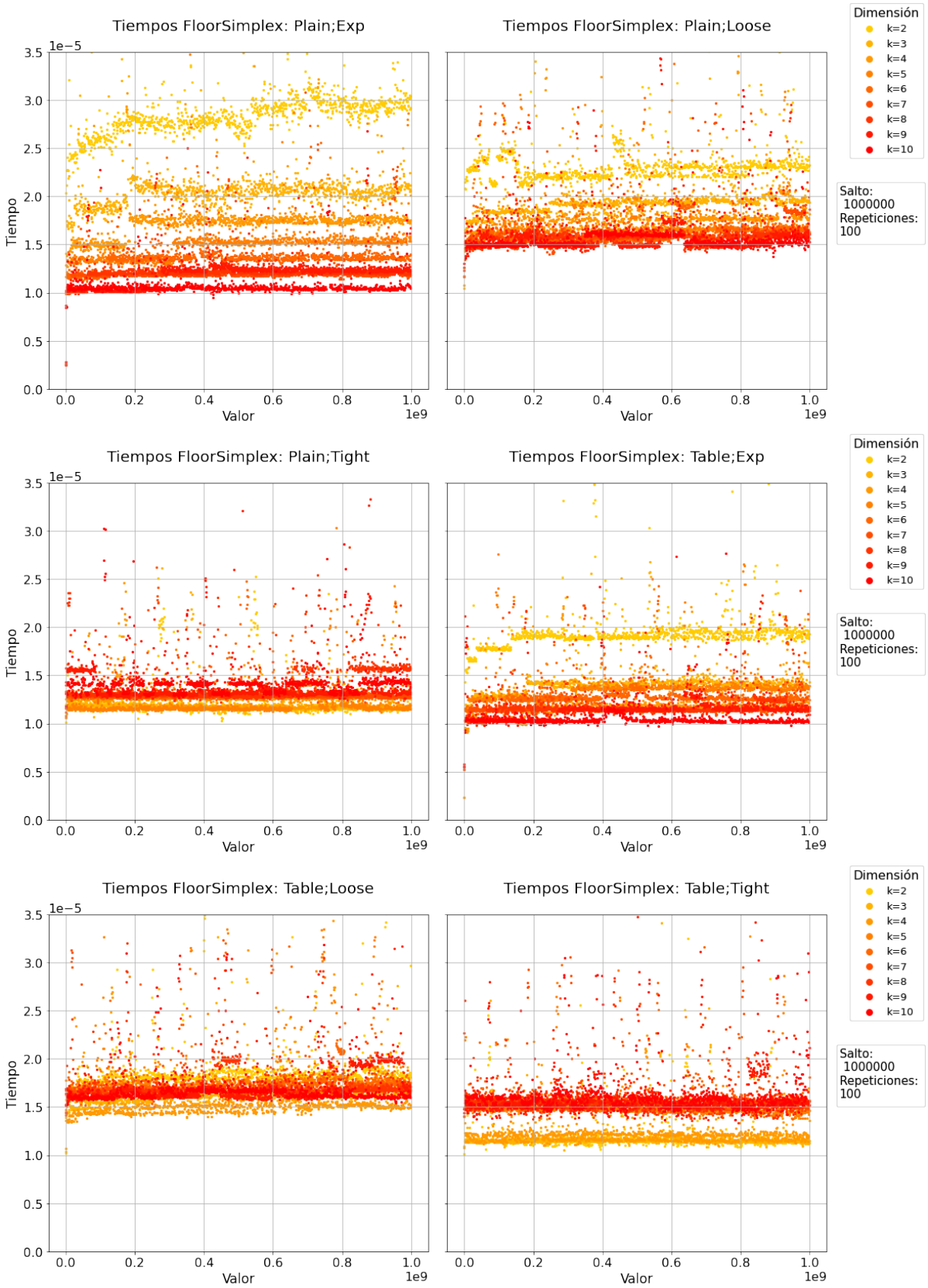


Figura 4.4: Gráficas de FloorSimplex comparando sus las dimensiones para las distintas versiones

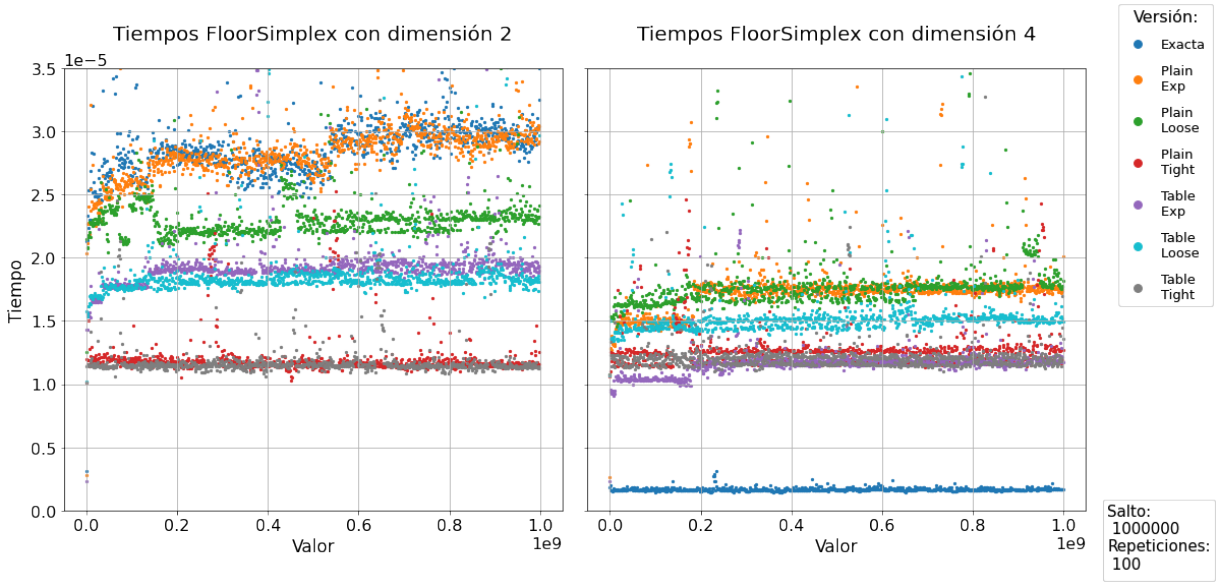
Para la búsqueda de las cotas para el índice del simplex, la versión *exp* hace una búsqueda exponencial. Las otras formas de encontrar las cotas, utilizan la proposición 2.2.7, donde *loose*(relajadas) representa si se usan las cotas $\sqrt[k]{m} - k, k\sqrt[k]{m}$ y *tight*(estrechas) si se usan $\sqrt[k]{k!m} - k, \sqrt[k]{k!m}$. De forma semejante, el método *plain* denota si se calcula un simplex de manera sencilla y *table* si se usa una tabla. La versión *plain;exp* es la versión correspondiente al algoritmo 3 la versión *table;exp* corresponde con el algoritmo 4, las versiones *plain;loose* y *plain;tight* con 5 y finalmente *table;loose* y *table; tight* con 6.

Se tomaron 1000 valores enteros repartidos homogéneamente en el intervalo $[1, 10^9]$ y se calculó el simplex que les antecede 100 veces y se reporta el tiempo promedio. Esto para cada versión en dimensiones 2-10 (2-4 para exacta).

Lo primero que podemos notar en las gráficas de la figura 4.4 es el bajo crecimiento de los tiempos para valores grandes. Este bajo crecimiento es un buen indicativo de la rapidez del algoritmo. Más aún, para dimensiones pequeñas se puede apreciar un crecimiento semejante al logarítmico, el cual es más evidente en la versión *plain:exp* y *table:exp* y que concuerda con los resultados obtenidos en las proposiciones 2.2.20 y 2.2.23.

Las versiones con búsqueda exponencial tienen una reducción en tiempo mientras la dimensión aumenta. Esto puede ser consecuencia de que $\|m\|_k$ es menor a lo largo que k aumenta, resultando en que se halle el índice en menos pasos de la búsqueda exponencial de cotas. Este decrecimiento en tiempo mientras las dimensiones aumentan no se mantiene en las versiones *loose* y *tight* lo que se puede deber a que el cálculo de una raíz k-ésima agrega demoras en el cómputo del índice. Estas demoras pueden hacerse más significativas a lo largo que la dimensión aumenta además de introducir potenciales errores de desbordamiento dependiendo del tipo de dato en el que se almacenen los valores. En otras palabras, los resultados que se pueden apreciar en la figura 4.4 pueden ser un indicativo de que computar FloorSimplex con las cotas no escale bien a dimensiones mayores, mientras que usar una búsqueda exponencial puede escalar mejor debido al ahorro de tiempo y la estabilidad numérica.

Si comparamos las distintas versiones dejando fijas las dimensiones, como se muestra en las gráficas de la figura 4.5, se puede ver que en las dimensiones 7 y 10, las versiones con búsqueda exponencial con y sin tabla, son más rápidas que cualquiera de las versiones con uso de cotas (*loose* y *tight*). Sólo en dimensión 2 es donde la versión *plain;exp* es más tardada.



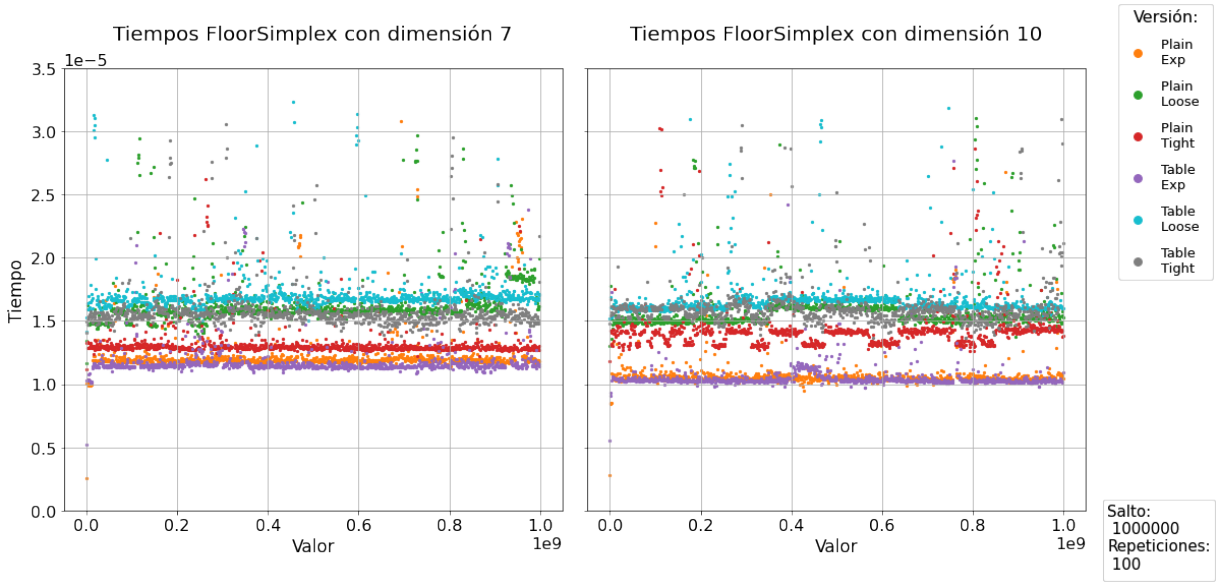


Figura 4.5: Gráficas de FloorSimplex comparando sus distintas versiones para dimensiones $k = 2, 4, 7, 10$

Para esta última versión, el descenso en tiempo a lo largo que aumenta la dimensión es tanto que para las dimensiones 7 y 10, el tiempo de computar FloorSimplex es virtualmente el mismo para las versiones con y sin tabla. Esto último puede deberse a que para hacer que el índice del simplex a un valor sea grande en una dimensión grande (índices grandes aumentan la cantidad de pasos en la búsqueda exponencial), este valor debe de ser de una magnitud enorme.

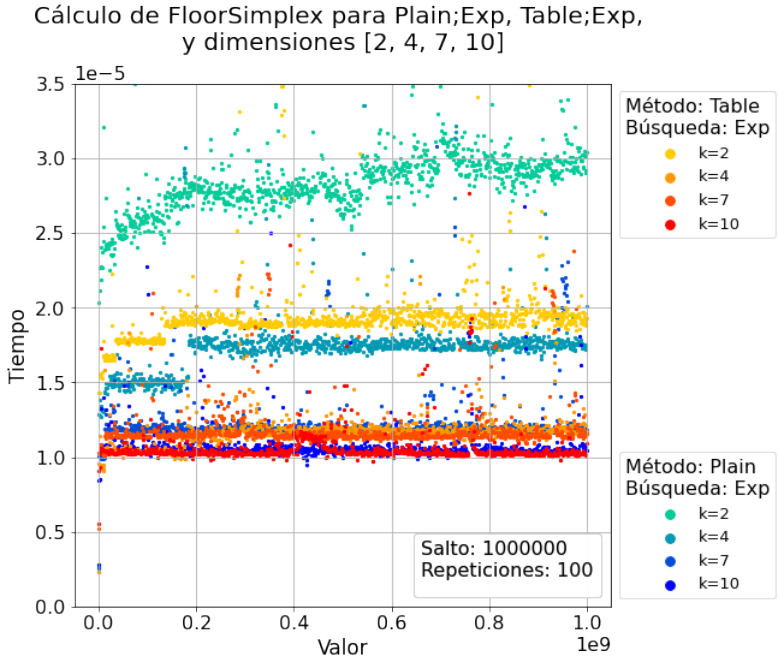


Figura 4.6: Gráfica para comparar las versiones *plain;exp* y *table;exp* para dimensiones 2,4,7 y 10.

El fenómeno antes descrito, puede observarse cuando conjuntamos los tiempos para las versiones con búsqueda exponencial con y sin tabla en la gráfica de la figura 4.6. Para las dimensiones 2 y 4, hay una diferencia perceptible entre usando y no usando tabla, donde la versión con tabla toma menos que la de sin tabla a como apuntaría la intuición. Esta diferencia es imperceptible para dimensiones 7 y 10.

En conclusión, dado que los tiempos de la versión con búsqueda exponencial fueron mejores para dimensiones más grandes, junto con el hecho de que estas versiones sólo requieren trabajar con enteros y no con decimales fue la motivación de seleccionar estas versiones para usarse en la implementación de Rank-kTuple, Unrank-kTuple, Rank-Tuple y Unrank-Tuple.

Rank-kTuple y Unrank-kTuple

Se implementaron los algoritmos 14 y 13 con dos versiones. La versión sencilla (*plain*) calcula cada simplex que se requiera y calcula FloorSimplex en su versión sencilla con búsqueda exponencial. La versión con tabla (*table*) calcula todo simplex por medio de una tabla precomputada y calcula FloorSimplex en su versión con tabla y búsqueda exponencial.

Para Unrank-kTuple, se midió el tiempo de cómputo del algoritmo en una muestra de 1000 valores repartidos homogéneamente en el intervalo $[1, 10^9]$, repitiéndose 100 cada cómputo y reportando el promedio. Esto para las versiones con tabla y sencilla para cada dimensión de 2-10. Con la una tupla obtenida de cada una de estas muestras en el intervalo mencionado, se midió el tiempo de cómputo al aplicarle Rank-kTuple, un total de 100 cálculos por muestra y se reportó el promedio de estas muestras. De forma semejante, esto se hizo para las versiones sencilla y con tabla para las dimensiones 2-10.

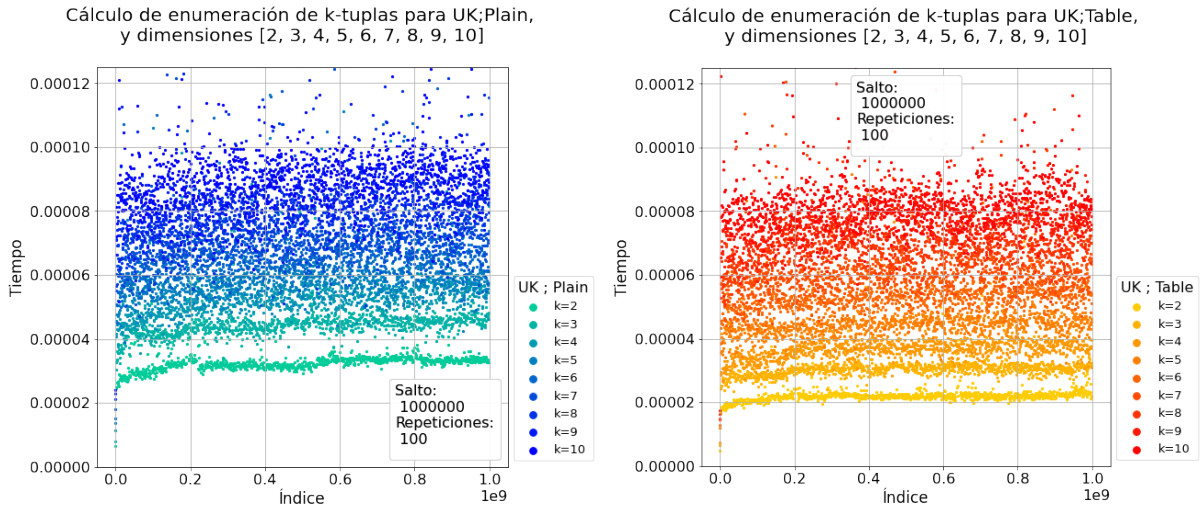


Figura 4.7: Gráficas de tiempos para Unrank-kTuple comparando dimensiones de 2-10 en su versión sencilla y con tabla.

En las gráficas de la figura 4.7, los tiempos para cada dimensión asemejan curvas parecidas a logaritmos a distintas alturas tanto para la versión con tabla como sencilla. Estas curvas son más evidentes para dimensiones pequeñas. Más aún, la altura de estas curvas está en el orden de las dimensiones con la menor dimensión correspondiente a el menor tiempo. Estas observaciones son concordes a lo esperado de acuerdo con las proposiciones 4.3.4 y 4.3.6.

Algo interesante es que, para dimensiones grandes, la diferencia de tiempos entra la versión sencilla y con tabla es menos pronunciada que para dimensiones pequeñas. Esto puede deberse al fenómeno descrito en la subsección anterior al ser que los tiempos de cómputo de FloorSimplex en su versión sencilla son parecidos a la versión con tabla. Dado que se hace una llamada a FloorSimplex para cada dimensión desde la provista hasta 1, la diferencia de velocidad en las versiones con dimensiones pequeñas se ve amortiguada por las velocidades semejantes en dimensiones grandes.

En cambio, dado que en Rank-kTuple no se llama a FloorSimplex, no hay un amortiguamiento para dimensiones grandes, y por ende se expresan las ventajas de usar una tabla precomputada con una aceleración al usar la versión con tabla.

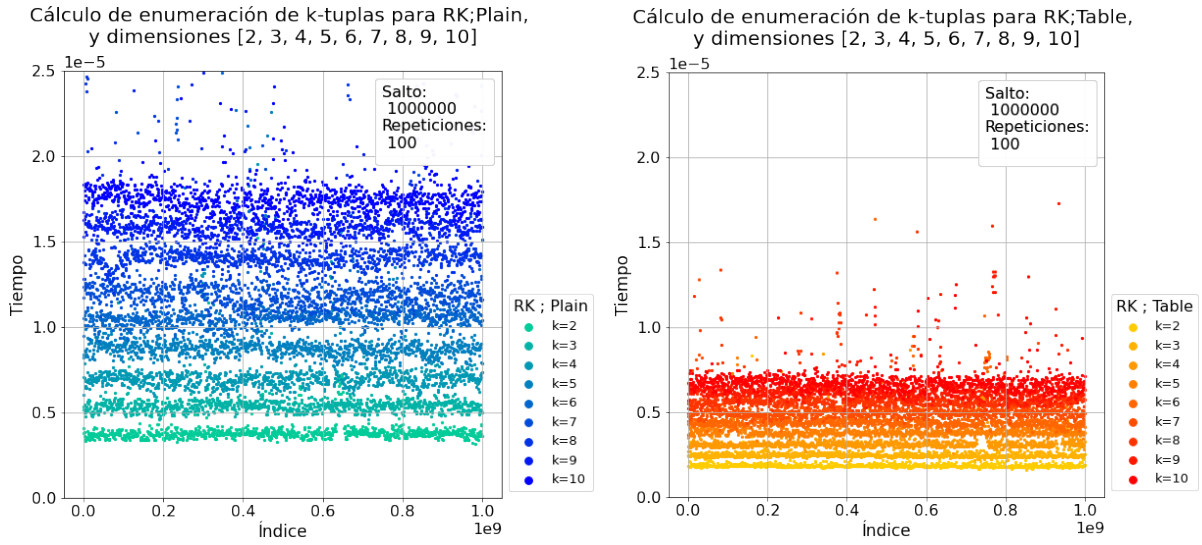


Figura 4.8: Gráficas de tiempos para Rank-Tuple comparando dimensiones de 2-10 en su versión sencilla y con tabla.

En las gráficas de la figura 4.8, se puede apreciar una clara diferencia en los tiempos de cómputo entre las versiones con tabla y sencilla como era de esperarse. Los tiempos para cada dimensión se asemejan a líneas a distintas alturas, donde la versión con sencilla ocupa aproximadamente el doble que la versión con tabla. Esto es consistente con las complejidades enunciadas en las proposiciones 4.3.4 y 4.3.6.

Rank-Tuple y Unrank-Tuple

De forma análoga las pruebas de escritorio anteriores, se implementaron los algoritmos Unrank-Tuple y Rank-Tuple con las versiones sencillas y con tabla. Se tomó una muestra de 1000 valores repartidas homogéneamente en el intervalo $[1, 10^7]$ y se corrió Unrank-Tuple 100 en cada valor, almacenando el promedio de tiempos para cada versión. Para la tupla correspondiente a cada valor, se computó Rank-Tuple 100 veces y se almacenó el promedio con cada versión.

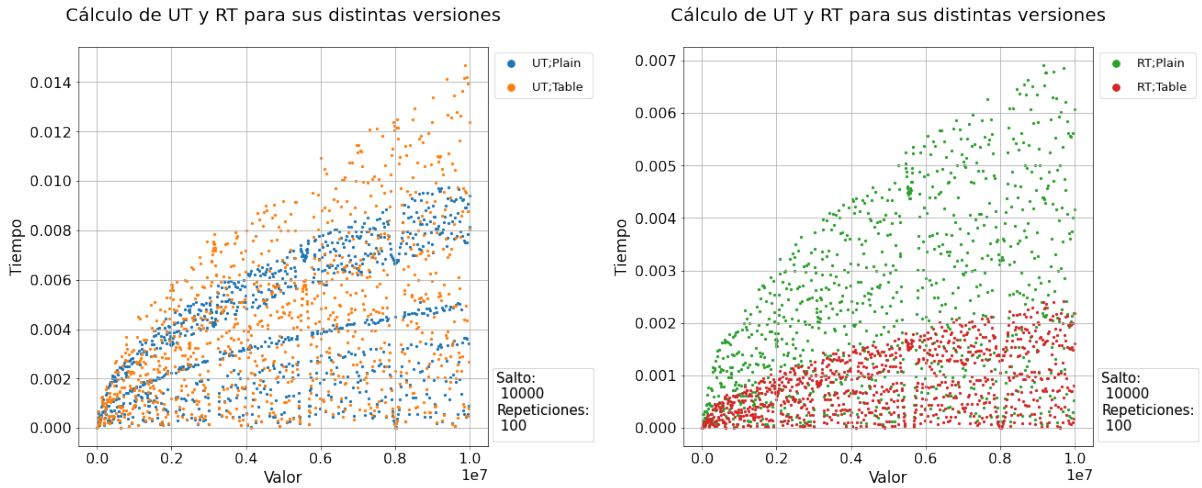


Figura 4.9: Gráficas de tiempos para UNRANK-TUPLE y RANK-TUPLE en su versión sencilla y con tabla.

En las gráficas de la figura 4.9 se observan diferencias tangibles entre la versión con tabla y la versión sencilla, y como dicta la intuición, la versión con tabla es más rápida.

Los crecimientos son consistentes con los enunciados en las proposiciones 4.4.3 y 4.4.5 al describir curvas que pueden ser acotadas por una raíz o un logaritmo por un factor.

Conclusiones

En este trabajo se resolvió la tarea de plantear una pareja de algoritmos Rank y Unrank que plantean una biyección entre $\mathbb{N}$ y $\mathbb{N}^k$ expandiendo a la propuesta por Skolem en [12]. Usando estos algoritmos se generalizó a una biyección entre $\mathbb{N}$ y $\bigcup_{k\in\mathbb{Z}^+}\mathbb{N}^k$. Para desarrollar estos algoritmos se requirió manejar números simplex y de aquí que se dedicara un capítulo de este trabajo a estudiar sus propiedades básicas y algoritmos para su cálculo. En la tabla 4.2a se enuncian los algoritmos que se plantearon. Con ayuda de estos algoritmos se plantearon los algoritmos de la tabla 4.2b

Algoritmo	Abreviatura
Simplex	Sim
Tabla-Simplex	TS
FloorSimplex	FS
FloorSimplex(con tabla)	$FSTS$
FloorSimplex(con cotas)	$FSCot$
FloorSimplex(con tabla y cotas)	$FSTSCot$

(a)

Algoritmo	Abreviatura
Rank-Comp	RC
Unrank-Comp	UC
Rank-kTuple	RK
Unrank-kTuple	UK
Rank-Tuple	RT
Unrank-Tuple	UT

(b)

Tabla 4.2: Abreviaturas para algoritmos

En el capítulo 2 se mostró la corrección de los algoritmos de 4.2a así como su complejidad en tiempo y espacio. En la tabla 4.3 se resumen las complejidades en tiempo y espacio obtenidas así como las dimensiones suficientes que debe tener la tabla para computar correctamente el algoritmo (según sea el caso).

#	**Alg**	**Ent**	**Tiempo**	**Aux**	ref	**DimTab**
1	Sim	$[k,n]$	k	1	2.1.16	
2	TS	$[k,n]$	kn	kn	2.1.18	
3	FS	$[m,k]$	$k\log_2(\lfloor\!\lfloor m \rfloor\!\rfloor_k)$	1	2.2.20	
4	$FSTS$	$[m,k,TS]$	$\log_2(\lfloor\!\lfloor m \rfloor\!\rfloor_k)$	1	2.2.27	$(k+1)\times(\lfloor\!\lfloor m \rfloor\!\rfloor_k+1)$
5	$FSCot$	$[m,k]$	$k\log_2(d_{m,k})+c_{m,k}$	$e_{m,k}$	2.2.25	
6	$FSTSCot$	$[m,k,TS]$	$\log_2(d_{m,k})+c_{m,k}$	$e_{m,k}$	2.2.27	$e_{m,k}+(k+1)\times(\lfloor\!\lfloor m \rfloor\!\rfloor_k+1)$

Tabla 4.3: Se definen $d_{m,k}$:= la distancia entre las cotas, $c_{m,k}$:= el tiempo que toma calcular las cotas, $e_{m,k}$:= el espacio que toma calcular las cotas.

Los algoritmos Rank-kTuple y Unrank-kTuple son los que más nos interesaban plantear y mostrar que son inversos y por tanto una biyección entre $\mathbb{N}$ y $\mathbb{N}^k$ (ver teorema 4.3.1). La

propiedad referente a una enumeración estructuralmente creciente se mostró en el teorema 4.3.2. Para todos los algoritmos de la tabla 4.2b se mostró su corrección y complejidad en tiempo (ver tabla 4.4a)y espacio (ver tabla 4.4b).

		#	Alg	Ent	General	Ref	Sencilla	Ref	Tabla	Ref
Tiempo	Rank	10	*RC*	$[\bar{x}]$	$k_0\mathcal{S}(k_0, a)$	3.3.8	k_0^2	3.3.10	k_0	3.3.12
		14	*RK*	$[\bar{x}]$	$k\mathcal{S}(k, n)$	4.3.3	k^2	4.3.4	k	4.3.6
		16	*RT*	$[\bar{x}]$	$2\mathcal{S}(2, S_k(n+1)) + k\mathcal{S}(k, n)$	4.4.2	k^2	4.4.3	k	4.4.5
	Unrank	9	*UC*	$[d, n, k]$	$k_0(\mathcal{FS}(d, k_0) + \mathcal{S}(k_0, \lVert d \rVert_{k_0}))$	3.3.6	$k_0^2 \log_2(\lVert d \rVert_{k_0})$	3.3.10	$k_0 \log_2(\lVert d \rVert_{k_0})$	3.3.12
		13	*UK*	$[m, k]$	$k(\mathcal{FS}(m, k) + \mathcal{S}(k, \lVert m \rVert_k))$	4.3.3	$k^2 \log_2(\lVert m \rVert_k)$	4.3.4	$k \log_2(\lVert m \rVert_k)$	4.3.6
		15	*UT*	$[w]$	$\lVert w \rVert_2(\mathcal{FS}(w, \lVert w \rVert_2 + 1) + \mathcal{S}(\lVert w \rVert_2 + 1, \lVert w \rVert_2))$	4.4.2	$\lVert w \rVert_2^2 \log_2(\lVert w \rVert_2)$	4.4.3	$\lVert w \rVert_2 \log_2(\lVert w \rVert_2)$	4.4.5

(a) Tabla de Tiempo

		#	Alg	Ent	DimTab	Ref	Auxiliar	Ref
Espacio	Rank	10	*RC*	$[\bar{x}]$	$k \times (n+1)$	3.3.11	k_0	3.3.10
		14	*RK*	$[\bar{x}]$	$(k+1) \times (n+1)$	4.3.5	1	4.3.4
		16	*RT*	$[\bar{x}]$	$(k+1) \times (S_k(n+1) + k + 1)$	4.4.4	1	4.4.3
	Unrank	9	*UC*	$[d, n, k]$	$k \times (\lVert d \rVert_{k_0} + 1)$	3.3.11	k_0	3.3.10
		13	*UK*	$[m, k]$	$(k+1) \times (\lVert m \rVert_k + 2)$	4.3.5	k	4.3.4
		15	*UT*	$[w]$	$(\lVert w \rVert_2 + 2) \times (\lVert w \rVert_2 + 2)$	4.4.4	1	4.4.3

(b) Tabla de Espacio

Tabla 4.4: Tablas de tiempo y espacio. Se definen las siguientes: $\bar{x} := (x_1, \ldots, x_k)$, $k_0 := k-1$,$n := \sum_{i=1}^{k} x_i$, $a := \sum_{i=1}^{k-1} x_i$. $\mathcal{FS}(m, k)$ es el tiempo que se requiere para computar $\lVert m \rVert_k$ y $\mathcal{S}(k, n)$ el tiempo para computar $S_k(n)$.

En la sección 4.5 se dio evidencia de la rapidez de los algoritmos propuestos y se obtuvieron resultados de tiempo consistentes con los esperados de acuerdo a la tabla 4.4a. Como era de esperarse, ejecutar los algoritmos con una tabla de simplex precomputada ofrece cómputos más rápidos en la mayoría de los casos pero a costa de algo de memoria.

Trabajo a futuro

Como se menciona en [22] existen varias funciones de emparejamiento entre $\mathbb{N}$ y $\mathbb{N}^2$. Cada una se puede tomar como punto de partida para como se hizo con la función de Cantor definida en las ecuaciones (1.3) y (1.4). Por ejemplo, en lugar de recorrer exhaustivamente cada diagonal se puede hacer un recorrido exhaustivo sobre cascarones cúbicos. Generalizar esta función de emparejamiento a k-dimensiones puede dar pie a versiones alternativas de Unrank-kTuple y Rank-kTuple (y por consiguiente de Rank-Tuple y Unrank-Tuple). La propiedad deseable de una enumeración estructuralmente creciente puede preservarse si definimos el tamaño de una k-tupla como el máximo de sus entradas en lugar de la suma de las mismas. Esta noción alternativa es también amena a la intuición por lo que en un futuro se pudiera desarrollar tales algoritmos alternativos.

Otra posible aplicación de los algoritmos desarrollados puede ser para la definición de un compresor de información que no necesariamente sean programas, sino datos estructurados que se requieran colapsar a un sólo número de manera reversible.

Apéndice A

Más pruebas de corrección y complejidad

A.1. Algoritmos de cálculo de simplex

Algoritmo simplex

En esta sección demostraremos la corrección del algoritmo SIMPLEX (1) así como su complejidad en tiempo y espacio.

Para mostrar la corrección de este algoritmo tenemos que analizar el comportamiento de la variable A durante la ejecución del algoritmo SIMPLEX 1 particularmente durante el ciclo for de las líneas 5-8. Sean $A_i^{(1)}$ y $A_i^{(2)}$ los valores de la variable A en la iteración del ciclo for tras ejecutarse las líneas 6 y 7 respectivamente en la iteración cuyo contador tiene el valor i. Así mismo, tomemos $A_{-1}^{(2)}$ al valor de A antes de entrar al ciclo.

Afirmación A.1.1. Sea (n, k) la entrada para el algoritmo SIMPLEX(1), se cumplen las siguientes.

1. Si $k > 0$, entonces el ciclo for de las líneas 5-8 termina en exactamente k iteraciones.
2. El algoritmo SIMPLEX termina.

Demostración. **(1)** Sea $k > 0$, es sencillo observar que el ciclo for sólo modifica la variable A con un par de operaciones enteras y jamás se altera el contador i ni la cota $k - 1$. Siendo así, este ciclo toma exactamente k iteraciones (pues se comienza con $i = 0$).

(2) Si $n = 0$ o $n = 1$ el algoritmo termina claramente. Si $n > 1$ y $k = 0$, no se entra en el ciclo for y por la línea 9 el algoritmo termina. Si $n > 1$ y $k > 0$, por el inciso (1) el ciclo for termina, y por la línea 9 se termina la ejecución del algoritmo. QED

Afirmación A.1.2. Si la entrada al algoritmo SIMPLEX(1) es (n, k) con $n > 1$ y $k > 0$, entonces para toda $i \leq k - 1$ se cumplen las siguientes.

1. $A_i^{(1)} = A_i^{(2)} \cdot (n + i)$.
2. $A_i^{(2)} = S_{i+1}(n)$.

Demostración. Sean $n > 1$ y $k > 0$. Observemos que de la línea 4 se sigue que $A_{-1}^{(2)} = 1$. Mostremos la conjunción de estos incisos por inducción sobre i.

(Caso base). Veamos para $i = 0$ que se cumplen los incisos (1) y (2).

(1) De la línea 6 se tiene que $A_0^{(1)} = A_{-1}^{(2)} \cdot n = 1 \cdot n = S_0(n) \cdot n$, pues $n > 1$.

(2) De la línea 7 se sigue que $A_0^{(2)} = A_0^{(1)} // 1 = A_0^{(1)}$. Por (1) se cumple que $A_0^{(2)} = S_0(n) \cdot n = n = S_1(n)$.

(Caso Inductivo). Supongamos que el resultado se sostiene para i y mostremos los incisos para $i + 1$. Sea $i + 1 \leq k$.

(1) De la línea 6 se sigue que $A_{i+1}^{(1)} = A_i^{(2)} \cdot (n + i + 1) = S_{i+1}(n) \cdot (n + i + 1)$.

(2) De la línea 7 se tiene que $A_{i+1}^{(2)} = A_{i+1}^{(1)} // (i + 2)$. Por el inciso (1) se tiene que

$$A_{i+1}^{(2)} = S_{i+1}(n) \cdot (n + i + 1) // (i + 2), \quad \text{esto es} \quad A_{i+1}^{(2)} = \frac{n^{\overline{i+2}}}{(i+1)!} /\!/ (i + 2).$$

Afirmamos que $(i + 2) \left| \frac{n^{\overline{i+1}}}{(i+1)!} \right.$ pues por el lema 2.1.7 se tiene que $S_{i+2}(n) = \frac{n^{\overline{i+2}}}{(i+2)!} \in \mathbb{N}$. Debido a esta última afirmación se sigue que

$$\frac{n^{\overline{i+2}}}{(i+1)!} /\!/ (i + 2) = \frac{n^{\overline{i+2}}}{(i+1)!(i+2)} = \frac{n^{\overline{i+2}}}{(i+2)!} = S_{i+2}(n) \quad \text{por tanto} \quad A_{i+1} = S_{i+2}(n).$$

QED

Proposición 2.1.15. Si la entrada al algoritmo SIMPLEX (3) es (n, k), la salida es el número $S_k(n)$.

Demostración de la proposición 2.1.15. Mostremos por los siguientes casos en la entrada n, k. Si $n = 0$ y $k \in \mathbb{N}$ o bien $n = 1$ y $k \in \mathbb{N}$, entonces se satisface el condicional if de las líneas 1-3 y las salidas son $n = 0 = S_k(0) = S_k(n)$ y $n = 1 = S_k(1) = S_k(n)$ respectivamente.

Si $n > 1$ y $k = 0$ no se entra en el ciclo for de las líneas 5-8 y por las líneas 4 y 9 la salida es $1 = S_0(n)$ pues $n > 1$.

Si $n > 1$ y $k > 0$ se entra en el ciclo for de las líneas 5-8, y por la afirmación A.1.2 y la afirmación A.1.1 el valor en la variable A al finalizar es ciclo es $S_k(n)$, y así la salida es $S_k(n)$. QED

Proposición 2.1.16. La complejidad en tiempo del algoritmo SIMPLEX(n, k) (1) es $O(k)$ y la complejidad en espacio es $O(1)$.

Demostración de la proposición 2.1.16. Si $n = 0$ o bien $n = 1$ y $k \in \mathbb{N}$ se satisface el condicional if de las líneas 1-3, por lo que se toma tiempo constante. Si $n > 1$ y $k = 0$, no se entra en el ciclo for de las líneas 5-8, por lo que se termina en un tiempo constante.

Si $n > 1$ y $k > 0$, no se entra en el condicional de las líneas 1-3 y sí en el ciclo de las líneas 5-8. Cada iteración de este ciclo toma el mismo tiempo constante, pues es sólo se calculan una multiplicación y una división entera, así como un par de asignaciones. Por la afirmación A.1.1, el ciclo termina en k iteraciones por lo que toda la ejecución de este ciclo toma tiempo $O(k)$. Siendo así, la ejecución del algoritmo toma tiempo $O(k)$.

La complejidad en espacio es $O(1)$ pues sólo se emplean las variables i y A. QED

Algoritmo tabla simplex

En esta sección se mostrará la corrección del algoritmo TablaSimplex 2 y se mostrará su complejidad en tiempo y espacio. Particularmente se demostrarán las proposiciones 2.1.17 y 2.1.18.

Afirmación A.1.3. Si $k, n \in \mathbb{Z}^+$, el algoritmo TablaSimplex(k, n) 2 termina.

Demostración. La inicialización en la línea 1 claramente termina. El ciclo for de las líneas 2-4 claramente termina pues el contador j no se altera dentro del ciclo ni su cota. Por las mismas razones los ciclos for de las líneas 6-8 y 5-9 terminan. Así el algoritmo termina. QED

Proposición 2.1.17. Sean $k, n \in \mathbb{Z}^+$ y TablaSimplex$(k, n) = TS$. Si $a < k$ y $b < n$ entonces $TS[a, b] = S_a(b)$.

Demostración de la proposición 2.1.17. Primero observemos que para $b = 0$ el ciclo for de las líneas 2-4 no sobreescribe el valor en $TS[a, 0]$ pues sólo sobrescribe valores $TS[0, j]$ para $1 \leq j \leq n-1$. Así mismo en los ciclos anidados for de las líneas 5-9 sólo sobrescriben valores de la forma $TS[i+1, j+1]$ para $0 \leq i \leq k-2$ y $0 \leq j \leq n-2$. De esta manera, por la línea 1 para toda $a < k$ se cumple que $TS[a, 0] = 0 = S_a(0)$.

Procedamos por inducción sobre a. Para $a = 0$ veamos que $TS[0, b] = S_0(b)$. Si $b = 0$ por la afirmación anterior $TS[0, 0] = 0 = S_0(0)$. Para $b+1 < n$ con $b \in \mathbb{N}$ se cumple que en la iteración del ciclo for de las líneas 2-4 donde $j = b+1$ se asigna $TS[0, b+1] = 1 = S_0(b+1)$ (línea 3). Claramente esta entrada no se vuelve a sobrescribir por lo que se mantiene el resultado.

Supongamos que para toda $b < n$ se cumple que $TS[a, b] = S_a(b)$ si $a < k$, y veamos que si $a+1 < k$ entonces $TS[a+1, b] = S_{a+1}(b)$. Sea $a+1 < k$, y procedamos por inducción sobre b. Para $b = 0$ por la afirmación inicial se tiene que $TS[a+1, 0] = 0 = S_{a+1}(0)$. Supongamos que $TS[a+1, b] = S_{a+1}(b)$.

Para la iteración de los ciclos for de las líneas 5-9 donde los contadores toman el valor $i = 1$ y $j = b$ se cumple por la línea 7 que

$$TS[a+1, b+1] = TS[a, b+1] + TS[i+1, b] \overset{\text{HI}}{=} S_a(b+1) + S_{a+1}(b) = S_{a+1}(b+1) \text{ por el corolario 2.1.8.}$$

No es difícil ver que la entrada $TS[a+1, b+1]$ no se vuelve a asignar, por lo que llegamos a lo que queríamos mostrar. QED

Proposición 2.1.18. Si $k, n \in \mathbb{Z}^+$, el algoritmo TablaSimplex(k, n) 2 tiene complejidad en tiempo y en espacio $O(kn)$ (contando la salida).

Demostración de la proposición 2.1.18. Notemos que la salida es un arreglo de enteros TS de $k \times n$, y sólo se usan variables i, j por lo que la complejidad en espacio es $O(kn)$.

El ciclo for de las líneas 2-4 solo hace una asignación por cada una de las $n-2$ iteraciones, por lo que toma tiempo $O(n-2)$. El ciclo for de las líneas 6-8 hace dos consultas al arreglo TS, una suma y una asignación, por lo que cada iteración toma tiempo constante. Dado que realiza $n-2$ iteraciones, este ciclo toma tiempo $O(n-2)$. El ciclo for de las líneas 5-9 ejecuta el ciclo previamente mencionado, por lo que cada iteración toma tiempo $O(n-2)$. Siendo que realiza $k-2$ iteraciones, este ciclo toma tiempo $O\big((n-2)(k-2)\big) = O(kn)$. QED

A.2. Versiones compactas de enumeración de composiciones

Se presentan las versiones compactas para los algoritmos Unrank-Comp(9) y Rank-Comp(10), versiones que prescinden de usar los algoritmos Sep2Comp y Comp2Sep, reduciendo el uso de memoria a memoria constante. Estas versiones preservan todas las propiedades de sus versiones son muy semejantes a los algoritmos Unrank-kTuple(13) y Rank-kTuple(14) respectivamente. Más aún, la complejidad en tiempo y espacio de estas versiones es análoga a las pruebas para estos algoritmos por lo que sólo enunciaremos las propiedades sin probarlas.

Algoritmo 17 Unrank-Comp$(d, n, k+1)$(compacto)

Entrada: Una terna de números $d, n, k+1 \in \mathbb{N}$ con $d < S_k(n+1)$.
Salida: Una $k+1$-composición de n.

1: **for** $i = k$ **downto** 1 **do**
2: $\quad a \leftarrow \llbracket d \rrbracket_i$
3: $\quad x_{i+1} \leftarrow n - a$
4: $\quad n \leftarrow a$
5: $\quad d \leftarrow d - S_i(a)$
6: **end for**
7: $x_1 \leftarrow a$
8: **return** $(x_1, \ldots, x_{k+1})$

Algoritmo 18 Rank-Comp$((x_1, \ldots, x_{k+1}))$(compacto)

Entrada: Una $k+1$-composición $(x_1, \ldots, x_{k+1}) \in \mathrm{Diag}_{k+1}(n)$ de algún valor n.
Salida: Un natural $d < S_k(n+1)$.

1: $a \leftarrow 0$
2: $d \leftarrow 0$
3: **for** $i = 1$ **upto** k **do**
4: $\quad a \leftarrow x_i + a$
5: $\quad d \leftarrow S_i(a) + d$
6: **end for**
7: **return** $d, a + x_{k+1}$

Proposición A.2.1. *Se cumplen las siguientes para los algoritmos compactos de enumeración de composiciones.*

1. *Si la entrada a los algoritmos homónimos* Unrank-Comp *(9) y (17) es la misma, entonces coinciden en la salida.*

2. *Si la entrada a los algoritmos homónimos* Rank-Comp *(10) y (18) es la misma, entonces coinciden en la salida.*

3. *La proposición 3.3.11 se sigue sosteniendo para las versiones compactas. Es decir, con tablas de tamaño* $k+1 \times \llbracket d \rrbracket_k + 1$ *y* $k+1, 1+\sum_{i=1}^{k} x_i$ *basta para computar* Unrank-Comp$(d, n, k+1)$ *y* Rank-Comp$((x_1, \ldots, x_{k+1}))$ *respectivamente.*

4. *El teorema 3.3.9 se sigue sosteniendo, es decir,* Unrank-Comp *y* Rank-Comp *son inversos uno del otro.*

Proposición A.2.2. *Las complejidades en tiempo y espacio de* Unrank-Comp$(d, n, k+1)$ *(17) y* Rank-Comp$(x_1, \ldots, x_{k+1})$ *(18) dependiendo del método para calcular un número simplex y el índice de simplex están dadas en las siguientes.*

1. *La complejidad en espacio para* Unrank-Comp *es* $O(1)$ *y en tiempo son las siguientes.*
   a) *En general la complejidad en tiempo es* $O(k(\mathcal{FS}(d,k) + \mathcal{S}(k, \lfloor\!\lfloor d \rfloor\!\rfloor_k)))$.
   b) *Usando* Simplex*(1) y* FloorSimplex*(3)* $O(k^2 \log_2(\lfloor\!\lfloor d \rfloor\!\rfloor_k))$.
   c) *Usando una tabla suficientemente grande y* FloorSimplex*(4)* $O(k \log_2(\lfloor\!\lfloor d \rfloor\!\rfloor_k))$.
2. *La complejidad en espacio para* Rank-Comp *es* $O(1)$ *y en tiempo son las siguientes.*
   a) *Si* $a = \sum_{i=1}^{k} x_i$ *la complejidad en tiempo en general es* $O(k\mathcal{S}(k, a))$.
   b) *Usando* Simplex*(1)* $O(k^2)$.
   c) *Usando una tabla se tiene que la complejidad es* $O(k)$.

Índice de Algoritmos

1. SIMPLEX(n, k) 25
2. TABLA-SIMPLEX(k, n) 26
3. FLOORSIMPLEX(m, k) 34
4. FLOORSIMPLEX(m, k, TS)(Con tabla precomputada) 41
5. FLOORSIMPLEX(m, k)(Con cotas) 43
6. FLOORSIMPLEX(m, k, TS) (Con cotas y tabla precomputada) 44
7. COMP2SEP$((x_1, \ldots, x_{k+1}))$ 50
8. SEP2COMP$((a_1, \ldots, a_k), n)$ 51
9. UNRANK-COMP$(d, n, k + 1)$ 62
10. RANK-COMP$((x_1, \ldots, x_{k+1}))$ 64
11. UNRANK-KTUPLE$(m, k + 1)$ 69
12. RANK-KTUPLE$((x_1, \ldots, x_{k+1}))$ 70
13. UNRANK-KTUPLE$(m, k + 1)$(compacto) 72
14. RANK-KTUPLE$((x_1, \ldots, x_{k+1}))$(compacto) 74
15. UNRANK-TUPLE(w) 80
16. RANK-TUPLE$((x_1, \ldots, x_{k+1}))$ 81
17. UNRANK-COMP$(d, n, k + 1)$(compacto) 98
18. RANK-COMP$((x_1, \ldots, x_{k+1}))$(compacto) 98

Referencias

[1] M. V. Lemus Yáñez, *Aproximación a la distribución universal por medio de simulaciones exhaustivas*. Proyecto de investigación para el Doctorado en Ciencias de la computación, Universidad Nacional Autónoma de México, 2019.

[2] A. N. Kolmogorov, "Three approaches to the quantitative definition of information," *International Journal of Computer Mathematics*, vol. 2, pp. 157–168, Jan. 1968. Publisher: Taylor & Francis.

[3] G. J. Chaitin, "On the Length of Programs for Computing Finite Binary Sequences," *Journal of the ACM*, vol. 13, pp. 547–569, oct 1966.

[4] R. J. Solomonoff, "A preliminary report on a general theory of inductive inference," in *Zator Tech. Bull.*, Citeseer, 1960.

[5] R. J. Solomonoff, "The kolmogorov lecture the universal distribution and machine learning," *The Computer Journal*, vol. 46, no. 6, pp. 598–601, 2003. Publisher: Oxford University Press.

[6] R. J. Solomonoff, "A formal theory of inductive inference. Part I," *Information and control*, vol. 7, no. 1, pp. 1–22, 1964. Publisher: Elsevier.

[7] M. Li and P. Vitányi, *An introduction to Kolmogorov complexity and its applications*, vol. 3. Springer, 2008.

[8] J.-P. Delahaye and H. Zenil, "Numerical evaluation of algorithmic complexity for short strings: A glance into the innermost structure of randomness," *Applied Mathematics and Computation*, vol. 219, no. 1, pp. 63–77, 2012. Publisher: Elsevier.

[9] P. Vitányi, "How incomputable is Kolmogorov complexity?," *Entropy*, vol. 22, no. 4, p. 408, 2020. Publisher: Multidisciplinary Digital Publishing Institute.

[10] G. Winskel, *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[11] E. Acuña Yeomans, *Enumeración de programas*. Proyecto de investigación para la maestría en Ciencias de la Computación, Universidad Nacional Autónoma de México, 2021.

[12] T. Skolem, "Über die Zurückführbarkeit einiger durch Rekursionen definierter Relationen auf "Arithmetische" (1937)," in *Selected Works in Logic* (T. Skolem, ed.), Oslo: Universitetsforlaget, 1970. Ver [35] para resumen al inglés.

[13] F. Soler-Toscano, H. Zenil, J.-P. Delahaye, and N. Gauvrit, "Calculating Kolmogorov Complexity from the Output Frequency Distributions of Small Turing Machines," *PLoS ONE*, vol. 9, May 2014.

[14] "AUTOMACOIN." `www.automacoin.com.html`.

[15] C. Sajedi and M. Lind, "Automacoin, a cryptocurrency for Algorithmic Information Dynamics." `https://www.automata2020.com/videos--material.html`, Aug. 2020.

[16] D. E. Knuth, *The art of computer programming, volume 4A: combinatorial algorithms, part 1*. Pearson Education India, 2011.

[17] D. L. Kreher and D. R. Stinson, *Combinatorial Algorithms: Generation, Enumeration, and Search*, vol. 7. CRC Press, 1998.

[18] A. Nijenhuis and H. S. Wilf, *Combinatorial algorithms: for computers and calculators*. Elsevier, 2014.

[19] F. Ruskey, *Combinatorial generation*, vol. 11. Preliminary working draft. University of Victoria, Victoria, BC, Canada, 2003.

[20] G. Cantor, "Ein Beitrag zur Mannigfaltigkeitslehre.," *Journal für die reine und angewandte Mathematik*, vol. 84, pp. 242–258, 1877.

[21] E. Deza and M. Deza, *Figurate numbers*. World Scientific, 2012.

[22] M. P. Szudzik, "The Rosenberg-Strong Pairing Function," *arXiv preprint arXiv:1706.04129*, 2017.

[23] A. L. Rosenberg, "Allocating Storage for Extendible Arrays," *Journal of the ACM*, vol. 21, pp. 652–670, Oct. 1974.

[24] A. L. Rosenberg, "Efficient pairing functions — and why you should care," *International Journal of Foundations of Computer Science*, vol. 14, pp. 3–17, Feb. 2003. Publisher: World Scientific Publishing Co.

[25] D. E. Knuth, "The Art of Computer Programming, vol 1: Fundamental," *algorithms*, p. 187, 1968.

[26] M. Green, "Restrictions of linear series to hyperplanes, and some results of Macaulay and Gotzmann," in *Algebraic curves and projective geometry*, pp. 76–86, Springer, 1989.

[27] P. Frankl, "A new short proof for the Kruskal-Katona theorem," *Discrete Mathematics*, vol. 48, no. 2-3, pp. 327–329, 1984. Publisher: Elsevier.

[28] S. S. Skiena, *The algorithm design manual*. Springer International Publishing, 2020.

[29] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[30] J. Kleinberg and E. Tardos, *Algorithm design*. Pearson Education India, 2006.

[31] A. B. Mojica, H. R. Mejía, and C. R. Orta, *Álgebra superior*. UNAM, Facultad de Ciencias, 2006.

[32] D. R. Mazur, *Combinatorics: A Guided Tour*. American Mathematical Soc., Feb. 2020.

[33] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, 2020.

[34] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, l. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[35] B. Rosser, "Th. skolem. über die zurückführbarkeit einiger dutch rekursionen definierter relationen auf 'arithmetische.'acta lilterarum ac scientiarum regiae universitatis hungaricae francisco-iosephinae, sectio scientiarum mathematicarum, vol. 8 (1936–1937), pp. 73–88.," *The Journal of Symbolic Logic*, vol. 2, no. 2, pp. 85–86, 1937.