Teórica 1: Algoritmos - Complejidad computacional
Técnicas de diseño de algoritmos (1era parte)

Programa de la materia

Algoritmos:

- Definición de algoritmo. Máquina RAM. Complejidad. Algoritmos de tiempo polinomial y no polinomial. Límite inferior.
- Técnicas de diseño de algoritmos: *divide and conquer*, *backtracking*, algoritmos golosos, programación dinámica.
- Algoritmos aproximados y algoritmos heurísticos.

Grafos:

- Definiciones básicas. Adyacencia, grado de un nodo, isomorfismos, caminos, conexión, etc.
- Grafos eulerianos y hamiltonianos.
- Grafos bipartitos.
- Árboles: caracterización, árboles orientados, árbol generador.
- Planaridad. Coloreo. Número cromático.
- Matching, conjunto independiente. Recubrimiento de aristas y vértices.

Algoritmos en grafos y aplicaciones:

- Representación de un grafo en la computadora: matrices de incidencia y adyacencia, listas.
- Algoritmos de búsqueda en grafos: BFS, DFS.
- Mínimo árbol generador, algoritmos de Prim y Kruskal.
- Algoritmos para encontrar el camino mínimo en un grafo: Dijkstra, Ford, Floyd, Dantzig.
- Algoritmos para determinar si un grafo es planar. Algoritmos para coloreo de grafos.
- Algoritmos para encontrar el flujo máximo en una red: Ford y Fulkerson.
- Matching: algoritmos para correspondencias máximas en grafos bipartitos. Otras aplicaciones.

Complejidad computacional:

- Problemas tratables e intratables. Problemas de decisión. P y NP. Máquinas de Türing determinísticas vs no determinísticas. Problemas NP-completos. Relación entre P y NP.
- Problemas de grafos NP-completos: coloreo de grafos, grafos hamiltonianos, recubrimiento mínimo de las aristas, corte máximo, etc.

Bibliografía

Según orden de utilización en la materia:

1. G. Brassard and P. Bratley, *Fundamental of Algorithmics*, Prentice-Hall, 1996.
2. T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*, The MIT Press, McGraw-Hill, 2001.
3. F. Harary, *Graph theory*, Addison-Wesley, 1969.
4. J. Gross and J. Yellen, *Graph theory and its applications*, CRC Press, 1999.
5. R. Ahuja, T. Magnanti and J. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, 1993.
6. M. Garey and D. Johnson, *Computers and intractability: a guide to the theory of NP- Completeness*, W. Freeman and Co., 1979.

1. Algoritmos

Un ***algoritmo*** es un procedimiento para resolver un problema, descripto por una secuencia finita de pasos, que termina en un tiempo finito. Debe estar formulado en términos de pasos sencillos que sean:

- precisos: se debe indicar el orden de ejecución de cada paso,
- bien definidos: en toda ejecución del algoritmo se debe obtener el mismo resultado,
- finitos: el algoritmo tiene que tener un número determinado de pasos.

Los métodos enseñados en la primaria para multiplicar o dividir números son ejemplos de algoritmos, también el algoritmo de Euclides para calcular el máximo común divisor entre dos números.

Un algoritmo siempre debe brindar una respuesta. Puede ser que la respuesta sea que no hay respuesta. También debe estar garantizado que el algoritmo termina. La descripción debe ser clara y precisa, no dejar lugar a la utilización de la intuición o creatividad. Una receta de cocina es un algoritmo si no dice, por ejemplo, "salar a gusto".

Muchas veces, cuando tenemos que resolver un problema contamos con varios algoritmos para hacerlo. Obviamente, quisieramos elegir el *mejor* de ellos. Pero, ¿cuándo un algoritmo es mejor que otro? Algunas propiedades que nos gustaría que tenga el algoritmo pueden ser: que sea fácil de programar, que su ejecución sea rápida, que no requiera de mucho espacio de memoria. Entonces, ¿cómo medir la eficiencia de un algoritmo? Si sólo vamos a ejecutar el algoritmo sobre pocas instancias pequeñas, la decisión de cuál algoritmo elegir seguramente no sea muy importante y preferiremos el más fácil de programar. Sin embargo, si tenemos que resolver muchas instancias de medida considerable o si el problema es *difícil* debemos elegir el algoritmo que utilizaremos muy cuidadosamente. Abordaremos luego este tema en detalle.

Otra pregunta que intentaremos responder a lo largo del curso es cuándo consideramos que un ***problema*** está bien resuelto. Veremos que algunos problemas son *fáciles* de resolver mientras que otros son *difíciles*. Pero, ¿cómo se mide la dificultad de un problema? Esto lo discutiremos hacia el final del curso.

2. Pseudocódigo

Si queremos describir un algoritmo en lenguaje natural podemos generar confusión. Una forma de escribir un algoritmo para intentar evitar imprecisiones es mediante ***pseudocódigo***, semejantes a los usados para escribir los programas informáticos.

El pseudocódigo es una descripción informal de alto nivel de un algoritmo. En la materia no vamos a definir un pseudocódigo específico, vamos a usar el sentido común (frases cortas descriptivas que expliquen tareas específicas, sangría para mostrar bloques de instrucciones). Lo importante es transmitir el procedimiento de forma clara y precisa.

En el siguiente ejemplo mostramos el pseudocódigo de un algoritmo para calcular el entero máximo de un arreglo. Sin necesidad de habernos puesto de acuerdo en la sintaxis utilizada, creemos que todos entendemos de forma clara el algoritmo que describe.

Ejemplo 1. *Encontrar el máximo de un arreglo de enteros:*

```
maximo(A,n)
    entrada: arreglo de enteros A no vacío
    salida: el elemento máximo de A

    max ← A[0]
    para i = 1 hasta dim(A) − 1 hacer
        si A[i] > max entonces
            max ← A[i]
        fin si
    fin para
    retornar max
```

3. Análisis de algoritmos

Como mencionamos, cuando tenemos más de un algoritmo para resolver un problema y queremos elegir el *mejor* entre ellos, hay diferentes criterios que podemos analizar para medir la eficiencia de los mismos, según el contexto de aplicación. Los recursos de mayor interés suelen ser el tiempo de cómputo y el espacio de almacenamiento requerido, con el primero generalmente más crítico. En la materia, en general, analizaremos los algoritmos utilizando como medida de eficiencia su tiempo de ejecución.
Para realizar esta elección podemos basarnos en dos enfoques:

El análisis empírico (o a posteriori) consiste en implementar los algoritmos disponibles en una máquina determinada utilizando un lenguaje determinado, luego ejecutarlos sobre un conjunto de instancias representativas y comparar sus tiempos de ejecución.

Las principales desventajas de este enfoque son:

- pérdida de tiempo y esfuerzo de programador: ya que debemos programar todos los algoritmos que deseamos evaluar.
- pérdida de tiempo de cómputo: ya que debemos ejecutar todos los programas sobre todas las instancias de prueba
- conjunto de instancias acotado: sólo vamos a poder ejecutar los algoritmos sobre un conjunto limitado de instancias relativamente pequeñas y las conclusiones alcanzadas pueden no ser verdaderas para instancias de mayor tamaño, que son las críticas.

Además este análisis dependerá de la habilidad del programador, del lenguaje de implementación, del compilador, del sistema operativo, entre otros factores que afectan al tiempo de cómputo requerido por un programa.

El análisis teórico (o a priori) consiste en determinar matemáticamente la cantidad de tiempo que llevará su ejecución como una función de la *medida de la instancia* considerada, independizándonos de la máquina sobre la cuál es implementado el algoritmo y del lenguaje para hacerlo. Para esto necesitamos definir:

- un modelo de cómputo
- un lenguaje sobre este modelo
- tamaño de la instancia
- instancias relevantes

3.1. Modelo de cómputo: Máquina RAM

En el análisis de un algoritmo, nos queremos independizar de la plataforma y lenguaje particular que utilicemos al implementarlo. Entonces, lo primero que haremos, es definir un modelo de cómputo formal conveniente sobre el cual haremos el análisis. La ***máquina de acceso random (RAM por Random Access Machine)*** fue introducida por Cook y Reckhow en 1973 [1] para estudiar la complejidad algorítmica de programas escritos en computadoras basadas en registros. Modela, adecuadamente para este propósito, computadoras en las que la memoria es suficiente y donde los datos involucrados en los cálculos entran en una palabra. Esta memoria está dividida en celdas, que se suelen llamar registros, que se pueden acceder (leer o escribir) de forma directa. Esta es la principal característica de una RAM y lo que le da su nombre (acceso random).

Un programa en una RAM es una sucesión finita de instrucciones. El proceso comienza ejecutando la primera instrucción y las instrucciones son ejecutadas secuencialmente (respetando las instrucciones de control de flujo).
Para reflejar computadoras reales, el conjunto de instrucciones de una RAM es el esperable en cualquier computadora que normalmente utilizamos: operaciones aritméticas, instrucciones de movimiento de datos e instrucciones de control de flujo. Consideramos a una operación elemental si su tiempo de ejecución puede ser acotado por una constante dependiente sólo de la implementación particular utilizada (la máquina, el lenguaje de programación). Esta constante no depende de la medida de los parámetros de la instancia considerada.

En una Máquina RAM, asumimos que toda instrucción (operaciones aritméticas, instrucciones de movimiento de datos e instrucciones de control de flujo) es una operación elemental con un **tiempo de ejecución** asociado y definimos el tiempo de ejecución de un programa A como:

$t_A(I)$ = suma de los tiempos de ejecución de las instrucciones realizadas por el programa A con la *instancia* I.

Como queremos tener una estimación de cómo crece el tiempo de ejecución de un algoritmo cuando el tamaño de las instancias de entrada crece, lo importante en el análisis es la cantidad de operaciones elementales ejecutadas, y no el tiempo exacto requerido por cada una de ellas.

Por ejemplo, supongamos que al analizar un programa A determinamos que para resolver una instancia I de un tamaño determinado, necesitamos realizar s sumas, m multiplicaciones y a asignaciones. También supongamos que una suma nunca requiere más de t_s microsegundos, una multiplicación nunca más que t_m microsegundos y una asignación nunca más que t_a microsegundos, donde t_s, t_m y t_a son constantes. Suma, multiplicación y asignación son operaciones elementales y el tiempo total t requerido por nuestro programa está acotado por:

$$t_A(I) \leq st_s + mt_m + at_a \leq max(t_s, t_m, t_a) * (s + m + a)$$

Entonces $t_A(I)$ está acotado por una constante multiplicada por el número de operaciones elementales realizadas al ejecutar el programa sobre I. Como el tiempo exacto requerido por cada operación elemental no es importante,

simplificamos asumiendo que toda operación elemental puede ser ejecutada en una unidad de tiempo.

Pasando a lo práctico... El modelo de máquina RAM brinda un marco adecuado para estudiar la complejidad algorítmica, pero no resulta conveniente para describir algoritmos. Para esto seguiremos utilizando pseudocódigo. En la especificación de un algoritmo mediante pseudocódigo (o en la implementación), una instrucción puede requerir la ejecución de varias operaciones elementales y algunas operaciones matemáticas podrían ser muy complejas para ser consideradas elementales. ¿Qué sucede por ejemplo con x^y?

Además, en una máquina RAM asumimos que la suma y la multiplicación son operaciones elementales. Pero esto no es del todo cierto si pasamos a las computadoras reales, ya que el tiempo necesario para ejecutarlas incrementa con la medida de los operandos. Sin embargo, en la práctica, a veces es posible hacer esta asunción, porque los operandos envueltos en las instancias que esperamos encontrar son de una medida razonable. Si analizamos el siguiente pseudocódigo para sumar los elementos de un arreglo de enteros:

```
suma(A)
    entrada: arreglo de enteros A
    salida: suma de los elemento de A

    resu ← 0
    para i = 0 hasta dim(A) − 1 hacer
        resu ← resu + A[i]
    fin para
    retornar resu
```

el valor de *resu* se mantiene en una medida *razonable* para todas las instancias que esperamos encontrarnos en la práctica. En teoría, sin embargo, el algoritmo debería poder ser aplicado a todas las instancias posibles y ninguna máquina real puede ejecutar estas sumas en tiempo constante si el valor de los elementos o la dimensión del arreglo es elegida lo suficientemente grande. Entonces, las suposiciones válidas al analizar un algoritmo dependen del dominio de aplicación esperado.

En cambio, en el siguiente pseudocódigo para calcular $n!$ la situación es muy distinta.

```
factorial(n)
    entrada: n ∈ ℕ
    salida: n!

    resu ← 1
    para i = 2 hasta n hacer
        resu ← resu ∗ i
    fin para
    retornar resu
```

Ya para valores relativamente chicos de n, no es para nada realista considerar que la operación $resu \leftarrow resu * i$ puede ser realizada en una unidad de tiempo. Esto deriva en dos modelos de cómputo:

Modelo uniforme: Se asume que cada operación básica tiene un tiempo de ejecución constante. Por lo tanto este enfoque consiste en determinar el número total de operaciones básicas ejecutadas por el algoritmo. Es

apropiado cuando los operandos de las operaciones entran en una palabra. Este método es sencillo, pero puede generar serias anomalías para valores arbitrariamente grandes de operandos.

Modelo logarítmico: El tiempo de ejecución de cada operación es una función (que dependerá del algotimo utilizado para resolverla) del tamaño (cantidad de bits) de los operandos (una suma es proporcional al tamaño del mayor operando, una multiplicación *ingenua* es proporcional al producto de los tamaños). Es apropiado cuando los operandos de las operaciones intermedias pueden crecer arbitrariamente. Por ejemplo, en el cálculo del factorial de un número los operandos de los cálculos que realiza el algoritmo cambian de orden de magnitud con respecto al valor de la entrada.

3.2. Tamaño de la instancia

El *tamaño* de una instancia formalmente corresponde al número de bits necesarios para representar la instancia en una computadora, usando algún esquema de codificación preciso y razonable.

Es decir, dada una instancia I, se define $|I|$ como el número de símbolos de un alfabeto finito necesarios para codificar I.

Por lo tanto el tamaño de una instancia depende del *alfabeto* y de la *base* utilizada en la representación. Por ejemplo, para almacenar $n \in \mathbb{N}$, se necesitan $L(n) = \lfloor \log_2(n) \rfloor + 1$ dígitos binarios. Mientras que para almacenar una lista de m enteros, se necesitan $L(m) + mL(N)$ dígitos binarios, donde N es el valor máximo de la lista (notar que es una cota que se puede mejorar).

Sin embargo, para hacer el análisis más claro, muchas veces seremos menos rigurosos que esto, y dependiendo del problema que estemos analizando, usaremos como tamaño un entero que, de alguna forma, mida el número de componentes de una instancia. Por ejemplo, si la instancia es un conjunto de enteros, el tamaño de la entrada será la cantidad de enteros, ignorando que cada uno de ellos requiera más de un bit para ser almacenado en la computadora.

Esta simplificación alcanza para reflejar de forma adecuada el espacio de instancias que esperamos encontrar en la práctica, donde podemos considerar que los valores de las componentes de las instancias se mantienen dentro de valores razonables, mientras que la cantidad de estas componentes crece de forma más significativa y es lo que influye en la performance de los algoritmos. En general, para problemas sobre arreglos, matrices o grafos, utilizaremos este enfoque.

Cuando trabajamos con algoritmos que sólo reciben un número fijo de números como parámetros, o cuando la performace del algoritmo depende estrechamente del parámetro, esta simplificación hace que carezca de sentido el análisis, y entonces debemos alejarnos de esta regla. En estos casos, mediremos el tamaño de la instancia como la cantidad de bits necesarios para almacenar los parámetros. Por ejemplo, para cálculo del factorial es más apropiado utilizar como tamaño de la entrada la cantidad de bits necesarios para representar la instancia de entrada en notación binaria.

Otra vez, el criterio más adecuado dependerá del algoritmo y su contexto de uso.

3.3. Análisis promedio y peor caso

Ya discutimos sobre el modelo de cómputo y el tamaño de la entrada. Dicho de una forma informal, la ***complejidad*** de un algoritmo es una función que *representa* el tiempo de ejecución en función del tamaño de la entrada. Pero el tiempo requerido por un algoritmo puede variar considerablemente entre dos instancias del mismo tamaño. Por

ejemplo, en el siguiente algoritmo de búsqueda,

```
esta?(A,elem)
    entrada: arreglo de enteros A no vacío, entero elem
    salida: Verdadero si elem se encuentra en A, Falso en caso contrario

    i ← 0
    mientras i < dim(A) y elem ≠ A[i] hacer
        i ← i + 1
    fin mientras
    si i < dim(A) entonces
        retornar Verdadero
    sino
        retornar Falso
```

¿qué sucede si el elemento buscado se encuentra en la primera posición del arreglo? ¿Y si en cambio *elem* no está en A? En cuanto a tiempo de ejecución, el primer escenario (que $elem = A[0]$) es lo mejor que nos puede pasar, mientras que el segundo es lo peor.

Esto abre dos criterios posibles para el análisis (en realidad tres si agregamos *mejor caso*):

Complejidad en el peor caso: Para cada tamaño de instancia, consideramos aquellas para las cuales el algoritmo requiere la mayor cantidad de tiempo. Definiremos la complejidad de un algoritmo A para las instancias de tamaño n, $T_A(n)$, como:

$$T_A(n) = \text{máx}_{I:|I|=n}\, t_A(I).$$

Complejidad en el caso promedio: Por otro lado, si un algoritmo será utilizado sobre un conjunto de instancias determinado, tal vez es importante saber el tiempo de ejecución promedio sobre el subconjunto de instancias de medida n. Generalmente el análisis del tiempo de ejecución promedio es más dificíl que cuando se considera el peor caso. Por otro lado, para estudiar el comportamiento promedio se debe conocer la distribución de las instancias que se resolverán y en muchas aplicaciones esto no es posible.

Si no se aclara lo contrario, en la materia siempre consideraremos el análisis del peor caso. Cuando no haya confusión a qué algoritmo nos estamos referiendo vamos a obviar el subíndice A, utilizando $T(n)$ en lugar de $T_A(n)$.

3.4. Notación asintótica: $\mathcal{O}$

Esta notación es llamada ***asintótica*** porque expresa el comportamiento de la función en el límite, esto es, para valores suficientemente grandes de los parámetros. Aunque las conclusiones basadas en la notación asintótica pueden fallar cuando los parámetros toman valores *pequeños*, un algoritmo mejor asintóticamente, casi siempre tiene performance muy superior ya en instancias de medida moderada. Esto hace importante su estudio.

Formalmente, dadas dos funciones $T, g : \mathbb{N} \to \mathbb{R}$, decimos que:

- $T(n)$ es $\mathcal{O}(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

  $$T(n) \leq c\, g(n) \text{ para todo } n \geq n_0.$$

  Es decir, T no crece más rápido que g, $T \preceq g$.

- $T(n)$ es $\Omega(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

  $$T(n) \geq c\, g(n) \text{ para todo } n \geq n_0.$$

  Lo que significa que T crece al menos tan rápido como g, $T \succeq g$.

- $T(n)$ es $\Theta(g(n))$ si

  $$T \text{ es } O(g(n)) \text{ y } T \text{ es } \Omega(g(n)).$$

  T crece al mismo ritmo que g, $T \approx g$.

Obs: $T(n)$ es $\mathcal{O}(g(n))$ si y sólo si $g(n)$ es $\Omega(T(n))$.

Veamos un ejemplo:

Ejemplo 2. $2n^2 + 10n$ *es* $\mathcal{O}(n^2)$*, porque tomando* $n_0 = 0$ *y* $c = 12$*, tenemos que*

$$2n^2 + 10n \leq 12n^2.$$

También podríamos decir que $2n^2 + 10n$ es $\mathcal{O}(n^3)$, cosa que es cierta. Sin embargo, nos interesa calcular de la mejor manera posible el orden de una función, es decir de la forma más ajustada posible.

Entonces, si una implementación de un algoritmo requiere en el peor caso $2n^2 + 10n$ microsegundos para resolver una instancia de tamaño n, podemos simplificar diciendo que el algoritmo es de orden de n^2, es decir, $\mathcal{O}(n^2)$. El uso de microsegundos es totalmente irrelevante, ya que sólo necesitamos cambiar la constante para acotar el tiempo por años o nanosegundos.

Ejemplo 3. 3^n *no es* $\mathcal{O}(2^n)$.

Vamos a demostrarlo por el absurdo. Supongamos que sí, es decir que 3^n *es* $\mathcal{O}(2^n)$*. Entonces, por definición, existirían* $c \in \mathbb{R}_+$ *y* $n_0 \in \mathbb{N}$ *tales que* $3^n \leq c\, 2^n$ *para todo* $n \geq n_0$*.*

Por lo tanto, $(\frac{3}{2})^n \leq c$ *para todo* $n \geq n_0$*. Esto genera un absurdo porque* c *debe ser una constante y no es posible que una constante siempre sea mayor que* $(\frac{3}{2})^n$ *cuando* n *crece.*

Ejemplo 4. *Si* $a, b \in \mathbb{R}_+$*, entonces* $(\log_a(n))$ *es* $\Theta(\log_b(n))$*. Es decir, todas las funciones logarítmicas crecen de igual forma sin importar la base.*

Como $\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$*, la constante* $\frac{1}{\log_b(a)}$ *sirve tanto para ver que* $(\log_a(n))$ *es* $\mathcal{O}(\log_b(n))$ *como para* $(\log_a(n))$ *es* $\Omega(\log_b(n))$*.*

Las definiciones de $\mathcal{O}$, Ω y Θ piden que existan las constantes $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$, pero no son relevantes sus valores específicos (que además no son únicos), lo relevante es que existan. Sabemos, por definición, que $T(n)$ es $\mathcal{O}(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tal que $\frac{T(n)}{g(n)} \leq c$. Esto sugiere relacionar la definición de $\mathcal{O}$ con el cálculo de límites.

Si $\lim_{n\to\infty} \frac{T(n)}{g(n)} = a$ con $0 \leq a < \infty$, significa que $|\frac{T(n)}{g(n)} - a| < \epsilon$ para algún $\epsilon > 0$. Entonces, $\frac{T(n)}{g(n)} < \epsilon + a$, donde $\epsilon + a$ es una constante, lo que es equivalente a decir que $T(n)$ es $\mathcal{O}(g(n))$.

De forma similar podemos analizar Ω y Θ, llegando a las siguientes propiedades.

- $T(n)$ es $\mathcal{O}(g(n))$ si y sólo si $\lim_{n\to\infty} \frac{T(n)}{g(n)} \in [0, \infty)$.
- $T(n)$ es $\Omega(g(n))$ si y sólo si $\lim_{n\to\infty} \frac{T(n)}{g(n)} \in (0, \infty]$.
- $T(n)$ es $\Theta(g(n))$ si y sólo si $\lim_{n\to\infty} \frac{T(n)}{g(n)} \in (0, \infty)$.

Esta equivalencia muchas veces puede ser muy útil para simplificar las demostraciones, como en los siguientes ejemplos. El primer caso, demuestra que cualquier función exponencial es *peor* que cualquier función polinomial.

Ejemplo 5. *Si $k, d \in \mathbb{N}$ entonces k^n no es $\mathcal{O}(n^d)$, con $k \geq 2$.*

Por la propiedad anterior, sabemos que k^n es $\mathcal{O}(n^d)$ si y sólo si $\lim_{n\to\infty} \frac{k^n}{n^d} \in [0, \infty)$.

Aplicando l'Hôpital, podemos ver que $\lim_{n\to\infty} \frac{k^n}{n^d} = \infty$. Por lo tanto, k^n no es $\mathcal{O}(n^d)$

Y ahora, mostraremos que la función logarítmica es *mejor* que la función lineal (ya vimos que no importa la base), es decir $ln(n) \prec O(n)$.

Ejemplo 6. *$ln(n)$ es $\mathcal{O}(n)$ y n no es $\mathcal{O}(ln(n))$.*

Nuevamente, operando y aplicando l'Hôpial, podemos ver que $\lim_{n\to\infty} \frac{\ln(n)}{n} = 0$, que es lo mismo que $\lim_{n\to\infty} \frac{n}{\ln(n)} = \infty$.

Lo que implica que $ln(n)$ es $\mathcal{O}(n)$ ($ln(n)$ no crece más rápido que n) y que n no es $\mathcal{O}(ln(n))$ ($ln(n)$ no acota superiormente el crecimiento de n).

Si un algoritmo es $\mathcal{O}(n)$, se dice *lineal*. En el caso que sea $\mathcal{O}(n^2)$, se dice *cuadrático*. Similarmente, un algoritmo es *logarítmico*, *cúbico*, *polinomial* o *exponencial* si son $\mathcal{O}(\log n)$, $\mathcal{O}(n^3)$, $\mathcal{O}(n^k)$ o $\mathcal{O}(d^n)$ respectivamente, donde k y d son constantes apropiadas.

Comúnmente son ignoradas las constantes multiplicativas (las c) y se asume que son todas del mismo orden de magnitud. Por eso, se suele decir que un algoritmo lineal es más rápido que uno cuadrático, sin considerar que esto puede no ser cierto para algunos casos. En algunos casos puede ser necesario ser más cuidadoso en el análisis.

Por ejemplo, consideremos dos algoritmos para resolver el mismo problema. Uno de ellos necesita n^2 días y el otro n^3 segundos para resolver una instancia de tamaño n. Desde un punto de vista teórico, el primero es *asintóticamente* mejor que el segundo, es decir, su performance es mejor sobre todas las instancias suficientemente grandes. Sin embargo, desde un punto de vista práctico, seguramente prefiramos el algoritmo cúbico. Esto sucede porque aunque el algoritmo cuadrático es asintóticamente mejor, su constante multiplicativa es muy grande para ser ignorada cuando se consideran instancias de tamaño razonable. Si bien la diferencia de magnitud de las constantes en este ejemplo fue muy grosera, la intención es remarcar que para instancias de tamaño relativamente pequeño puede que el análisis asintótico no sea adecuado.

Ejemplo 7. *Las complejidades de algunos algoritmos conocidos son:*

- *Búsqueda secuencial:* $\mathcal{O}(n)$.
- *Búsqueda binaria:* $\mathcal{O}(\log(n))$.
- *Ordenar un arreglo (bubblesort):* $\mathcal{O}(n^2)$.
- *Ordenar un arreglo (quicksort):* $\mathcal{O}(n^2)$ *en el peor caso (!).*

- *Ordenar un arreglo (heapsort):* $\mathcal{O}(n \log(n))$.

Es interesante notar que $\mathcal{O}(n \log(n))$ es la complejidad **óptima** para algoritmos de ordenamiento basados en comparaciones.

3.5. Algoritmos eficientes vs no eficientes

Para comparar algunas posibles complejidades, en la tabla 1 mostramos los tiempos insumidos por algoritmos con esas complejidades cuando son ejecutados sobre instancias de distintos tamaños por la misma máquina (suponiendo 0.001 miliseg por operación).

	10	20	30	40	50	60
$\log(n)$	0.000001 seg	0.000001 seg	0.000001 seg	0.000002 seg	0.000002 seg	0.000002 seg
n	0.00001 seg	0.00002 seg	0.00003 seg	0.00004 seg	0.00005 seg	0.00006 seg
$n \log(n)$	0.00001 seg	0.000026 seg	0.000044 seg	0.000064 seg	0.000085 seg	0.0001 seg
n^2	0.0001 seg	0.0004 seg	0.0009 seg	0.0016 seg	0.0025 seg	0.0036 seg
n^3	0.001 seg	0.008 seg	0.027 seg	0.064 seg	0.125 seg	0.216 seg
n^5	0.1 seg	3.2 seg	24.3 seg	1.7 min	5.2 min	13.0 min
2^n	0.001 seg	1.0 seg	17.9 min	12.7 días	35.6 años	366 siglos
3^n	0.59 seg	58 min	6.5 años	3855 siglos	2 E+8 siglos	1.3 E+13 siglos
$n!$	3.63 seg	771 siglos	8.4 E+16 siglos	2.5 E+32 siglos	9.6 E+48 siglos	2.6 E+66 siglos

Cuadro 1: Complejidad vs tamaño de la entrada

Los datos de la tabla 1 corresponden a una máquina muy vieja (datos del libro de Garey y Johnson de 1979). ¿Qué pasa si tenemos una máquina 1000 veces más rápida? ¿Un millón de veces más rápida? ¿Cuál sería el tamaño del problema que podemos resolver en una hora comparado con el problema que podemos resolver ahora? En la tabla 2 comparamos el tamaño de las instancias que podríamos resolver en 1 hora con una máquina 1000 veces más rápida para distintas complejidades algorítmicas.

	actual	1000 veces mas rápida
$\log n$	N1	$N1^{1000}$
n	N2	1000 N2
n^2	N3	31.6 N3
n^3	N4	10 N4
n^5	N5	3.98 N5
2^n	N6	N6 + 9.97
3^n	N7	N7 + 6.29

Cuadro 2: Comparación de tamaño de instancias resueltas en 1 hora

Obviamente, el tamaño de las instancias incrementa, pero para las complejidades exponenciales, 2^n y 3^n, el incremento es muy pequeño. Entonces, ¿cuándo un algoritmo es bueno o eficiente?

Esto sugiere el siguiente criterio:

POLINOMIAL = "bueno"

EXPONENCIAL = "malo"

3.6. Problemas bien resueltos

Pero, ¿siempre será posible contar con algoritmos polinomiales para resolver un determinado problema? Existen problemas para los cuales no se conocen algoritmos polinomiales para resolverlos, aunque tampoco se ha probado que estos no existan. Sobre esto hablaremos hacia el final del curso. Por ahora nos conformamos sólo con la idea de que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para resolverlo.

4. Resumiendo

- Vamos a describir nuestro algoritmos mediante pseudocódigo.
- Usaremos el análisis teórico. En el labo lo validarán con el análisis empírico.
- En general, vamos a utilizar el modelo uniforme.
- Cuando los operandos de las operaciones intermedias crezcan *mucho*, para acercarnos a modelar la realidad, usaremos el modelo logarítmico.
- En el caso de arreglos, matrices, grafos, como tamaño de entrada generalmente usaremos la cantidad de componentes de la instancia.
- En el caso de algoritmos que reciben como parámetros sólo una cantidad fija de números, como tamaño de entrada usaremos la cantidad de bits necesarios para su representación.
- En general, calcularemos la complejidad de un algoritmo para el peor caso.
- Consideraremos que los algoritmos polinomiales son satisfactorios (cuanto menor sea el grado, mejor), mientras que los algoritmos supra-polinomiales son no satisfactorios.

5. Más ejemplos

Ejemplo 8. *Cálculo del promedio de un arreglo:*

```
promedio(A)
    entrada: arreglo de reales A de tamaño ≥ 1
    salida: promedio de los elementos de A

    suma ← 0                                          O(1)
    para i = 0 hasta dim(A) − 1 hacer                 dim(A) veces
        suma ← suma + A[i]                            O(1)
    fin para
    retornar suma/dim(A)                              O(1)
```

Si llamamos $n = dim(A)$, la entrada es $\mathcal{O}(n)$ y el algoritmo es lineal en función del tamaño de la entrada. Notar que estamos utilizando el modelo uniforme, porque asumimos que los valores de suma entran en una palabra.

Ejemplo 9. *Algoritmo de búsqueda:*

```
esta?(A,elem)
    entrada: arreglo de enteros A no vacío
    salida: Verdadero si el entero elem se encuentra en A, Falso en caso contrario

    i ← 0                                                    O(1)
    mientras i < dim(A) y elem ≠ A[i] hacer                  a lo sumo dim(A) veces
        i ← i + 1                                            O(1)
    fin mientras
    si i < dim(A) entonces                                   O(1)
        retornar Verdadero                                   O(1)
    sino
        retornar Falso                                       O(1)
```

Este algoritmo es lineal en función del tamaño de la entrada.

Ejemplo 10. *Invertir los dígitos de un entero:*

```
invertir_entero(n)
    entrada: entero n ≥ 0
    salida: número formado por los dígitos de n invertidos

    resu ← 0                                                 O(1)
    aux ← n                                                  O(log2(n))
    mientras aux ≠ 0 hacer                                   log2(n) veces
        resu ← 10 * resu + aux mod 10                        O(log2(n))
        aux ← aux div 10                                     O(log10(n))
    fin mientras
    retornar resu                                            O(log2(n))
```

El tamaño de la entrada es $t = \log_2 n$. El algoritmo es orden $\mathcal{O}(1) + 2 * \mathcal{O}(log_2(n)) + \log_{10} n * 2 * \mathcal{O}(log_2(n))$, que es $\mathcal{O}(\log_{10})(n) * \log_2(n)) = \mathcal{O}(\log_2^2(n)) = \mathcal{O}(t^2)$. El algoritmo es cuadrático en el tamaño de la entrada.

Ejemplo 11. *Sean m y n dos enteros positivos. El máximo común divisor (mcd) de m y n es el entero más grande que divide a ambos. El algoritmo obvio para calcular el (mcd) se desprende de la definición:*

```
mcd(m,n)
    entrada: m, n ∈ ℕ+
    salida: mcd de m y n

    i ←min(m, n)+1                                  O(???)
    repetir                                         a lo sumo mín(m, n) veces
        i ← i − 1                                   O(1)
    hasta m mod i = 0 y n mod i = 0                 O(1)
    retornar i                                      O(1)
```

Para poder calcular la complejidad del algoritmo anterior, necesitamos conocer la complejidad de la función min.

```
min(m,n)
    entrada: m, n ∈ ℕ+
    salida: mcd de m y n

    si m < n entonces                               O(1)
        retornar m                                  O(1)
    sino
        retornar n                                  O(1)
```

El tamaño de la entrada es $t = O(log_2(\text{máx}\{m,n\}))$. Como la función min es $\mathcal{O}(1)$, el orden de mcd es $2 * O(1) + \text{mín}\{m,n\} * 2 * O(1)$, que es $\mathcal{O}(\text{mín}\{m,n\})$. Si m y n son del mismo orden, el algoritmo es $\mathcal{O}(2^t)$, exponencial en el tamaño de la entrada.

Ejemplo 12. *Algoritmo de ordenamiento (selection sort):*

```
selection(A)
    entrada: A arreglo de reales
    salida: A ordenado

    para i = 0 hasta dim(A) − 2 hacer               dim(A) − 1 veces
        minj ← i                                    O(1)
        minval ← A[i]                               O(1)
        para j = i + 1 hasta dim(A) − 1 hacer       a lo sumo dim(A) veces
            si A[j] < minval entonces               O(1)
                minj ← j                            O(1)
                minval ← A[j]                       O(1)
            fin si
        fin para
        A[minj] ← A[i]                              O(1)
        A[i] ← minval                               O(1)
    retornar A                                      O(dim(A))
```

Si $dim(A) = n$, el algoritmo es orden $n * (4 * O(1) + n * 3 * O(1) + 1)$, que es $\mathcal{O}(n^2)$, es decir, cuadrático en función

del tamaño de la entrada.

Ejemplo 13. *Primalidad: Dado un número $n \in \mathbb{N}$, ¿n es primo?*

```
esPrimo(n)
    entrada: n ∈ ℕ
    salida: Verdadero si n es primo, Falso en caso contrario

    i ← 2                                          O(1)
    mientras i ≤ √n hacer                          √n veces - O(1)
        si n mod i = 0 hacer                       O(1)
            retornar Falso                         O(1)
        fin si
    fin mientras
    retornar Verdadero                             O(1)
```

El tamaño de la entrada es $t = \log_2(n)$. Bajo el modelo de costo uniforme suponemos que todas las operaciones son $\mathcal{O}(1)$. Por lo tanto la cantidad de operaciones totales es $\mathcal{O}(\sqrt{n}) = O(\sqrt{(2^t)}) = O(\sqrt{2}^t)$. Notar que se conocen algoritmos polinomiales para saber si un entero es primo.

Ejemplo 14. *Cálculo iterativo del n-ésimo número de Fibonacci, $Fib(n)$:*

```
Fibonacci(n)
    entrada: entero n ≥ 1
    salida: n-ésimo número de Fibonacci

    i ← 1                                          O(1)
    j ← 0                                          O(1)
    para k = 0 hasta n − 1 hacer                   n veces
        j ← i + j                                  O(??)
        i ← j − i                                  O(??)
        k ← k + 1                                  O(1)
    fin para
    retornar j                                     O(log2(Fib(n))) = O(??)
```

Como hacían en Algo1, pueden demostrar que el invariante del ciclo asevera que j_k es $Fib(k)$ y i_k es $Fib(k-1)$, donde j_k e i_k son los valores de las variables j e i en la iteración k-ésima. Moivre encontró una fórmula cerrada para calcular los número de Fibonacci: $Fib(k) = \frac{(1+\sqrt{5})^k - (1-\sqrt{5})^k}{2^k\sqrt{5}}$. Por lo tanto, en la iteración k-ésima, los operandos de $j \leftarrow i + j$ y $i \leftarrow j - i$ son $\mathcal{O}(2^k)$, requiriendo $\mathcal{O}(log_2(2^k)) = O(k)$ bits para almacenarlos.

Esto muestra que los operandos de las operaciones intermedias crecen exponencialmente, por lo que es necesario utilizar el modelo logarítmico para realizar un análisis realista. Entonces, el tiempo requerido por las operaciones es proporcional al tamaño de los operandos (cantidad de bits que ocupan), que en este caso es $\mathcal{O}(k)$ en la iteración k-ésima.

Como $k \leq n$, podemos suponer que el valor de los operandos son $\mathcal{O}(2^n)$ y por lo tanto su tamaño $\mathcal{O}(n)$. Esto implica que las operaciones $j \leftarrow i + j$ y $i \leftarrow j - i$ son $\mathcal{O}(n)$, resultando el algoritmo $\mathcal{O}(n^2)$. Como el tamaño de la entrada

es $t = log_2(n)$, el algoritmo es $\mathcal{O}(4^t)$, exponencial en el tamaño de la entrada.

Ejemplo 15. *Cálculo iterativo de $n!$:*

```
factorial(n)
    entrada: n ∈ Z_{≥0}
    salida: n!

    resu ← 1                                  O(1)
    para k = 2 hasta n hacer                  n − 1 veces
        resu ← resu * k                       O(??)
    fin para
    retornar resu                             O(log2(n!)) = O(n * log2(n))
```

Al ingresar a la k-ésima iteración del ciclo, el valor de la variable resu es $(k-1)!$. Esto muestra que los operandos de las operaciones intermedias crecen factorialmente, por lo que es necesario utilizar el modelo logarítmico para realizar un análisis realista, donde el tiempo requerido por las operaciones es proporcional al tamaño de los operandos (cantidad de bits que ocupan).

En la iteración k-ésima, el operando resu de $resu \leftarrow resu * k$ requiere $log_2((k-1)!)$ bits para almacenarlos. Podemos acotar esto, $log_2((k-1)!) \leq log_2(k!) \leq log_2(k^k) = k * log_2(k)$. El otro operando es k, que requiere $log_2(k)$ bits.

Entonces, la operación $resu \leftarrow resu * k$ de la iteración k-ésima es $\mathcal{O}(k * log_2(k) log_2(k))$ suponiendo un algoritmo de multiplicación ingenuo, y como $k \leq n$ esto es $\mathcal{O}(n * log_2^2(n))$. Como esta operación está dentro de un ciclo que se ejecuta $n-1$ veces, la complejidad del algoritmo es $\mathcal{O}(n^2 log_2^2(n))$.

Como el tamaño de la entrada es $t = log_2(n)$, el algoritmo es $\mathcal{O}(t^2 4^t)$, exponencial en el tamaño de la entrada.

6. Técnicas de diseño de algoritmos - 1era parte

Algunas técnicas de diseño de algoritmos son:

- Fuerza bruta
- Búsqueda con retroceso (*backtracking*)
- Recursividad
- Dividir y conquistar (*divide and conquer*)
- Programación dinámica
- Algoritmos golosos
- Heurísticas y algoritmos aproximados

Ya en el Taller de Álgebra y Algoritmos y Estrucutras de Datos I y II trabajaron ampliamente con algoritmos recursivos y algoritmos de dividi y conquistar. Ahora nos vamos a dedicar a las otras técnias.

6.1. Fuerza bruta

Muchos problemas pueden resolverse buscando una solución de forma fácil, pero, a la vez, generalmente ineficiente. El método consiste en analizar ***todas*** las posibilidades.

Estos algoritmos son fáciles de inventar e implementar y siempre ***funcionan***. Pero generalmente son muy ineficientes.

Veamos un ejemplo.

Ejemplo 16. ***Problema de las n reinas****: Queremos hallar todas las formas posibles de colocar n reinas en un tablero de ajedrez de $n \times n$ casillas, de forma que ninguna reina amenace a otra. Es decir, no puede haber dos reinas en la misma fila, columna o diagonal.*

Una solución inmediata es aplicar fuerza bruta, es decir hallar ***todas*** *las formas posibles de colocar n reinas en un tablero de $n \times n$ y luego seleccionar las que satisfagan las restricciones del problema. En cada configuración tenemos que elegir las n posiciones donde ubicar a las reinas de los n^2 posibles casilleros. Entonces, el número de configuraciones que analizará el algoritmo es:*

$$\binom{n^2}{n} = \frac{n^2!}{(n^2-n)!n!}$$

Pero fácilmente podemos ver que la mayoría de las configuraciones que analizaríamos no cumplen las restricciones del problema y que trabajamos de más.

6.2. Backtracking

Esta técnica recorre sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema buscando aquellas que cumplen las propiedades deseadas. A estas configuraciones las llamaremos ***soluciones válidas***. Puede utilizarse para resolver problemas de factibilidad (donde se quiere encontrar cualquier solución válida o todas ellas) o problemas de optimización (donde de todas las configuraciones válidas se quiere la ***mejor***).

Ya vimos el método de fuerza bruta, pero, excepto para instancias muy pequeñas, su costo computacional es prohibitivo. ***Backtracking*** es una modificación de esa técnica que aprovecha propiedades del problema para evitar analizar todas las configuraciones. Obviamente, para que el algoritmo sea correcto debemos estar seguros de no dejar de examinar configuraciones que estamos buscando.

La idea básica es tratar de extender una solución parcial del problema hasta, eventualmete, llegar a obtener una solución completa, que podría ser válida o no. Habitualmente, se utiliza un ***vector*** $a = (a_1, a_2, \ldots, a_n)$ para representar una solución candidata, cada a_i pertenece a un dominio/conjunto finito A_i. El espacio de soluciones es el producto cartesiano $A_1 \times \ldots \times A_n$.

El procedimiento construye este vector elemento a elemento: primero considera un posible valor para a_1; después, uno para a_2, y así sucesivamente. Cuando una solución parcial no puede ser extendida, se retrocede (se eliminan valores de los a_i en orden inverso). Este retroceso se detiene al encontrar una configuración que permita avanzar de forma diferente. Este retroceso es lo que le da el nombre al método: backtracking.

En síntesis, en cada paso se extienden las soluciones parciales $a = (a_1, a_2, \ldots, a_k)$, $k < n$, agregando un elemento más, $a_{k+1} \in S_{k+1} \subseteq A_{k+1}$, al final del vector a. Las nuevas soluciones parciales son sucesoras de la anterior. Si S_{k+1} (conjunto de soluciones sucesoras) es vacío, esa rama no se continua explorando.

Se puede pensar este espacio de búsqueda como un árbol dirigido donde cada vértice representa una solución parcial y un vértice x es hijo de y si la solución parcial x se puede extender desde la solución parcial y. La raíz del árbol se corresponde con el vector vacío (la solución parcial vacía). Los vértices del primer nivel del árbol serán las soluciones parciales que ya tienen defenidas el primer elemento. Los de segundo nivel las que tiene los dos primeros y así siguiendo.

Si el vértice x corresponde a la solución parcial $a = (a_1, a_2, \ldots, a_k)$, por cada valor posible que puede tomar a_{k+1} se ramifica el árbol, generando tantos hijos de x como posibilidades haya para a_{k+1}. Las soluciones completas (cuandos todos los a_i tienen valor) corresponden a las hojas del árbol.

El proceso de backtracking recorre este árbol en profundidad. Cuando podemos deducir que una solución parcial no nos llevará a una solución válida, no es necesario seguir explorando esa rama del árbol de búsqueda (se ***poda*** el árbol) y se retrocede hasta encontrar un vértice con un hijo válido por donde seguir la exploración. Esta poda puede ser por:

- Factibilidad: ninguna extensión de la solución parcial derivará en una solución válida del problema.
- Optimalidad (en problemas de optimización): ninguna extensión de la solución parcial derivará en una solución óptima del problema.

De esta poda depende el éxito del método aplicado a un problema. Para poder aplicar la poda por factibilidad, la representación de las soluciones debe cumplir que, si una solución parcial $a = (a_1, a_2, \ldots, a_k)$ no cumple las propiedades deseadas, tampoco lo hará cualquier extensión posible de ella (propiedad dominó).

Podemos estar interesados en encontrar todas las soluciones válidas de nuestro problema, o nos puede alcanzar con sólo encontrar una. Los esquemas generales son los siguientes:

Backtracking: Esquema General - Todas las soluciones

```
BT(a,k)
entrada: a = (a_1, ..., a_k) solución parcial
salida: se procesan todas las soluciones válidas

si k == n + 1 entonces
      procesar(a)
      retornar
sino
      para cada a' ∈ Sucesores(a, k)
            BT(a', k + 1)
      fin para
fin si
retornar
```

Backtracking: Esquema General - Una solución

```
BT(a,k)
    entrada: a = (a_1,...,a_k) solución parcial
    salida: sol = (a_1,...,a_k,...,a_n) solución válida

    si k == n+1 entonces
        sol ← a
        encontro ← true
    sino
        para cada a' ∈ Sucesores(a,k)
            BT(a',k+1)
            si encontro entonces
                retornar
            fin si
        fin para
    fin si
    retornar
```

Donde *sol* variable global que guarda la solución y *encontro* variable booleana global que indica si ya se encontró una solución (inicialmente está en **false**).

Para demostrar la correctitud de un algoritmo de backtracking, debemos demostrar que se enumeran todas las posibles configuraciones válidas. Es decir, que las ramificaciones y podas son correctas.

Para el cálculo de la complejidad debemos acotar la cantidad de vértices que tendrá el árbol y considerar el costo de procesar cada uno.

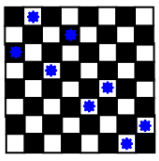
Ejemplo 17. *Problema de las 8 reinas: Volvamos a este problema, que es el ejemplo estándar para introducir esta técnica. Recordemos que el objetivo es ubicar 8 reinas en el tablero de ajedrez (8×8) sin que ninguna "amenace" a otra. Una reina amenaza a otra si ambas se encuentran en la misma fila, en la misma columna o en la misma diagonal.*

Veamos cuántas posibles configuraciones tenemos. Hay que elegir las 8 casillas donde ubicaremos nuestras reinas de entre las 64 casillas que tiene el tablero. Esto es:

$$\binom{64}{8} = 442616536$$

Un algoritmo de fuerza bruta, generaría todas estas posibilidades y luego analizaría cada una para ver si cumple las restricciones del problema.

Pero sabemos que, en las soluciones que nos interesan, cada fila debe tener exactamente una reina. Entonces, una solución puede estar representada por $(a_1, \ldots, a_8)$, con $a_i \in \{1, \ldots, 8\}$ indicando la columna de la reina que está en la fila i. Una solución parcial puede estar representada por $(a_1, \ldots, a_k)$, $k \leq 8$.

La configuración  *está representada por* $(2, 4, 1, 3, 6, 5, 8, 7)$.

Tenemos ahora $8^8 = 16777216$ *combinaciones.*

Podemos mejorar este cálculo considerando que una columna debe tener exactamente una reina. Es decir, todos los a_i*, para* $i = 1, \ldots, 8$*, deben ser distintos. Ahora redujimos a* $8! = 40320$ *combinaciones. No todas estas configuraciones serán solución de nuestro problema, ya que falta verificar que no haya reinas que se amenacen por estar en la misma diagonal.*

Cumplimos la propiedad dominó: Si una solución parcial $(a_1, \ldots, a_k)$*,* $k \leq 8$*, tiene reinas que se amenazan, entonces toda extensión de ella seguro tendrá reinas que se amenazan. Por lo tanto es correcto podar una solución parcial que tiene reinas que se amenacen.*

Dada una solución parcial $(a_1, \ldots, a_k)$*,* $k \leq 8$ *(sin reinas que se amenacen), construiremos el conjunto de sus vértices hijos,* `Sucesores(`a, k`)`*, asegurando que siga sin haber reinas que se amenacen (en lugar de construir todos y luego podar). Entonces la nueva reina, la* $(k+1)$*-ésima (correspondiente a la fila* $k+1$*), no podrá estar ubicada en ninguna de las columnas ni en ninguna de las diagonales donde están ubicadas las* k *anteriores.*

$$\texttt{Sucesores}(a, k) = \{(a, a_k) : a_k \in \{1, \ldots, 8\}, |a_k - a_j| \notin \{0, k-j\} \; \forall \; j \in \{1, \ldots, k-1\}\}.$$

Que no haya dos reinas en la misma fila, está implícito en la representación, en el número de coordenada. Esto es, la coordenada 1 de a*, o sea* a_1*, va a tener la columna de la reina que está en la primera fila. La coordenada* 2 *de* a*, o sea* a_2*, va a tener la columna de la reina que está en la segunda fila. De forma general, la coordenada* k *de* a*, o sea* a_k*, va a tener la columna de la reina que está en la* k*-ésima fila. Esto implícitamente está prohibiendo que haya dos reinas en la misma fila (porque ya la representación hace que no se pueda).*

Que no haya dos reinas en la misma columna, lo estamos exigiendo con $a_k - a_j \notin \{0\}$*, es decir* $a_k \neq a_j$*, para* $j = 1, \ldots, k-1$*.*
Que no haya dos reinas en la misma diagonal, lo estamos asegurando al pedir que $|a_k - a_j| \notin \{k-j\}$*, es decir* $|a_k - a_j| \neq k - j$*, para* $j = 1, \ldots, k-1$*. Esto sería que la diferencia entre las filas de dos reinas* $(k-j)$*, sea diferente a la diferencia entre sus columnas* $(|a_k - a_j|)$*. Notar que, que se cumpla* $k - j = |a_k - a_j|$ *es lo mismo que decir que las reinas de las posiciones* (j, a_j) *y* (k, a_k) *están en la misma diagonal.*

Ahora ya estamos en condición de implementar un algoritmo para resolver el problema.

Ejemplo 18. ***Suma de subconjuntos:*** *Dado un conjunto de naturales* $C = \{c_1, \ldots, c_n\}$ *(sin valores repetidos) y* $k \in \mathbb{N}$*, queremos encontrar un subconjunto de* C *(o todos los subconjuntos) cuyos elementos sumen* k*.*

Si resolvemos este problema mediante un algoritmo de fuerza bruta, generaríamos los 2^n *posibles subconjuntos y luego verificaríamos si alguno suma* k*.*

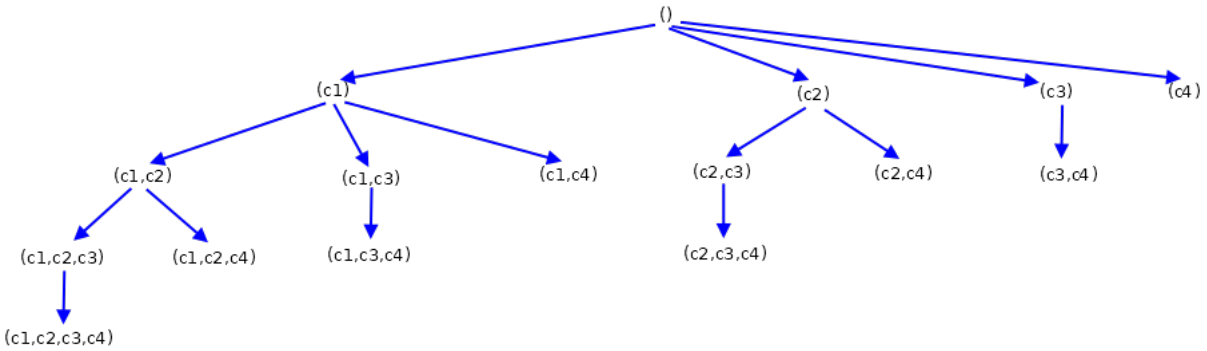
Una posible representación de una solución es $a = (a_1, \ldots, a_n)$*, con* $a_i \in \{V, F\}$ *indicando para cada elemento del conjunto* C *si pertenece o no a la solución.*

Una representación alternativa, que parece más prometedora, es $a = (a_1, \ldots, a_r)$*, con* $r \leq n$*,* $a_i \in C$ *y* $a_i \neq a_j$ *para* $0 \leq i < j \leq r$*. Es decir, el vector* a *contiene los elementos del subconjunto de* C *que representa. En este caso, todos los vértices de árbol corresponden a posibles soluciones (no necesariamente válidas), no sólo las hojas. Hay que adaptar a esto el esquema general que presentamos previamente.*

Como las soluciones son conjuntos, no nos interesa el orden en que los elementos fueron agregados a la solución. Entonces (c_4, c_5, c_2) *y* (c_2, c_4, c_5) *representarían la misma solución, el subconjunto* $\{c_2, c_4, c_5\}$*. Y cualquier permutación de* (c_4, c_5, c_2) *también. Obviamente queremos evitar esto, no queremos generar más de un vector que represente la misma solución (ni parcial ni total), porque estaríamos agrandando aún más el árbol de búsqueda. Veamos cómo*

evitar esto.

Los vértices de nivel 1 del árbol de búsqueda se corresponderán a las n soluciones de subconjuntos de un elemento, (c_i), para $i = 1, \ldots, n$. En el segundo nivel, no quisiera generar los vectores (c_i, c_j) y (c_j, c_i), sino sólo uno de ellos porque representan la misma solución. Para esto, para la solución parcial (c_i) podemos sólo crear los hijos (c_i, c_j) con $j > i$. Así evitaríamos una de las dos posibilidades. De forma general, cuando extendemos una solución, podemos pedir que el nuevo elemento tenga índice mayor que el último elemento de a. Por ejemplo, si $n = 4$, el árbol de búsqueda es:



En el subárbol encabezado por (c_1) estarán todas las permutaciones que incluyan a c_1. En el encabezado por (c_2), todas las que incluyan a c_2 pero no a c_1. En el encabezado por (c_2, c_4), todas las permutaciones que contengan a c_2 y c_4 pero no a c_1 ni a c_3.

Para cada vértice del árbol podemos ir guardando la suma de los elementos que están en la solución. Si encontramos un vértice que sume k, la solución correspondiente será una solución válida. Si la suma excede k, esa rama se puede podar, ya que los elementos de C son positivos.

Entonces podemos definir los sucesores de una solución como:

$$\texttt{Sucesores}(a, k) = \{(a, a_k) : a_k \in \{c_{s+1}, \ldots, c_n\},\ si\ a_{k-1} = c_s\ y\ \textstyle\sum_{i=1}^{k} a_i \leq k\}.$$

*Cumplimos la propiedad dominó: Si una solución parcial $(a_1, \ldots, a_r)$ suma más que k, entonces toda extensión de ella seguro también sumará más que k. Es decir, las soluciones no se **arreglan** al extenderlas. Esto no sería cierto si hubiera elementos negativos en C.*

Y podemos mejorar esto. Si ordenamos los elementos de C en orden creciente y al extender una solución nos encontramos que cuando a_k toma un posible valor c_j $(c_j \in \{s+1, \ldots, n\})$ sucede que $\sum_{i=1}^{k} a_i \geq k$, podemos estar seguros que para ningún $j' > j$ obtendremos una solución válida. Por lo cual no es necesario verificar esto para esos valores.

7. Bibliografía recomendada

- Capítulos 1, 2, 3, 6, 7, 8 y 9.6 de G. Brassard and P. Bratley, *Fundamental of Algorithmics*, Prentice-Hall, 1996.
- Secciones I.1, I.2, I.4, IV.15 IV.16 de T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*, The MIT Press, McGraw-Hill, 2001.

Referencias

[1] S. A. Cook and R. A. Reckhow. Time bounded random access machines. 7(4):354–375, Aug. 1973.