

Algoritmos y Estructuras de Datos III

Tomás Agustín Hernández



Complejidad Computacional (repaso)

Problema

Descripción de los datos de entrada y la respuesta a proporcionar para cada uno de los datos de entrada.

Instancia de un Problema

Es un juego válido de datos de entrada.

Máquina RAM

Supongamos una Máquina RAM.

- La memoria está dada por una sucesión de celdas numeradas. Cada **celda** puede almacenar un valor de **b bits**.
- Supondremos habitualmente que esos **b bits** de cada celda están fijos, y suponemos que todos los datos que maneja el algoritmo se pueden almacenar con **b bits**. **Ej.:** Lo que quiere decir esto es que suponemos que todas las celdas son de 8 bits, y los datos que maneja el algoritmo también son de 8 bits.
- Se tiene un programa imperativo que NO está almacenado en memoria que está compuesto por asignaciones y las estructuras de control habituales.
- Las asignaciones acceden a las celdas de memoria y realizan las operaciones estándar sobre los tipos de datos primitivos habituales.

Cada una de las instrucciones que se ejecuten tienen un tiempo de ejecución asociado

- El acceso a cualquier celda de memoria, tanto lectura como escritura es $O(1)$.
- Las asignaciones y el manejo de las estructuras de control se realiza en $O(1)$.
- Las operaciones entre valores lógicos son $O(1)$.

Las operaciones entre enteros/reales dependen de b

- Las sumas y restas son $O(b)$.
- Las multiplicaciones y divisiones son $O(b \log b)$

Nota: Si b está fijo, entonces las operaciones entre enteros/reales es $O(1)$.

Tiempo de Ejecución de un Algoritmo

Sea A un algoritmo, su tiempo de ejecución es: $T_A(I)$ donde esto indica que es la suma de los tiempos de ejecución del algoritmo en una instancia dada I.

$|I|$: Cantidad de bits necesarios para almacenar los datos de entrada de I.

Nota: Si **b está fijo** y la entrada ocupa n celdas de memoria entonces $|I| = bn = O(n)$

Complejidad de un Algoritmo

Sea A un algoritmo, su complejidad es: $f_A(n) = \max_{I: |I|=n} T_A(I)$ donde esto indica que la complejidad de un algoritmo A dado un n cualquiera, es el que de mayor tiempo de ejecución tiene en una instancia dada I.

Cotas

Cota Superior (O): $f(n) \in O(g(n)) \iff \exists c \in \mathbb{R} > 0, n_0 \in \mathbb{N} \text{ tal que } \forall n \geq n_0 : f(n) \leq c * g(n)$

Cota Inferior (Ω): $f(n) \in \Omega(g(n)) \iff \exists c \in \mathbb{R} > 0, n_0 \in \mathbb{N} \text{ tal que } \forall n \geq n_0 : f(n) \geq c * g(n)$

Cota Ajustada (θ): $f(n) \in \theta(g(n)) \iff f(n) \in O(g(n)) \text{ y } f(n) \in \Omega(g(n))$. Es decir, $\theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

Tipos de Funciones

- $O(n)$: lineal
- $O(n^2)$: cuadrático
- $O(n^3)$: cúbico
- $O(n^k)$ $k \in \mathbb{N}$: polinomial. Ej.: $O(n^4)$, $O(n^5)$
- $O(\log n)$: logarítmico.
- $O(d^n)$ $d \in \mathbb{R}_{>1}$: exponencial. Ej.: $O(2^n)$, $O(4^n)$

Algoritmos Satisfactorios y No Satisfactorios

Un Algoritmo Satisfactorio es un algoritmo que tiene un costo menor a otro.
Los algoritmos polinomiales se consideran satisfactorios (cuanto menor sea el grado, mejor).
Los algoritmos supra-polinomiales se consideran no satisfactorios.

Problema de Optimización

Sea $x \in S$, un problema de optimización consiste en encontrar la mejor solución dentro de un conjunto:

- $z^* = \max f(x)$
- $z^* = \min f(x)$

Función Objetivo: Es una función de la forma $f : S \Rightarrow \mathbb{R}$

- El conjunto S es la **región factible**.
- Los elementos $x \in S$ se llaman **soluciones factibles**.
- El valor $z^x \in \mathbb{R}$ es el **valor óptimo** del problema, y cualquier solución factible $x^* \in S / f(x^*) = z^x$ se llama un **óptimo** del problema

Algoritmos de Fuerza Bruta

También llamado búsqueda exhaustiva o generate and test. Genera todas las soluciones factibles y se queda con la mejor (la que cumpla las restricciones que necesitamos). Suele ser fácil de implementar y es un algoritmo exacto: si hay solución, siempre la encuentra.
Lo malo es su complejidad (suele ser exponencial)

Ej.: Imaginemos que tenemos un tablero de ajedrez y tenemos que buscar las soluciones en las cuales ninguna dama amenace a otra. Una solución por fuerza bruta sería buscar todas las soluciones que existen, y de ahí agarrar las que me sirvan.

Ej.: Imaginemos que tenemos que resolver un Sudoku, si usáramos un casillero de 9x9 y quisiéramos aplicar un algoritmo de fuerza bruta, es decir, primero buscar todas las posibles permutaciones 1966270504755529.... posibilidades y de todas estas posibilidades ver cual es solución. Esto es muy tedioso y lento, hay una mejor opción que la fuerza bruta, y es el Backtracking.

Backtracking

Es una técnica (de fuerza bruta pero mas eficiente) que consiste en una exploración ordenada del espacio de soluciones por medio de la extensión de soluciones parciales (**cuando noto que una solución parcial no me sirve, puedo descartarla e ir a la siguiente sin necesidad de perder más tiempo en esa o sus hijos.**).

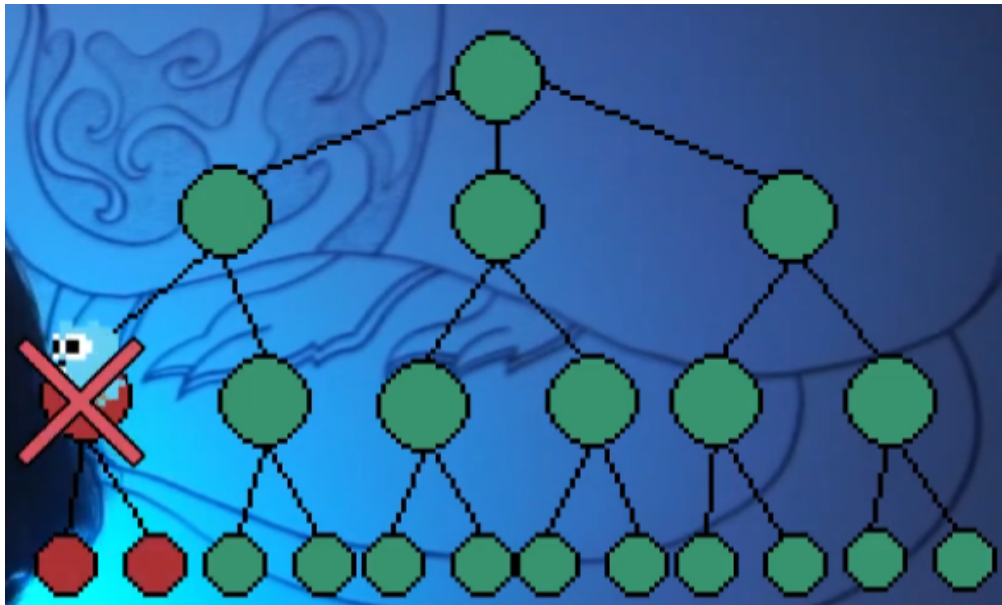
Habitualmente se utiliza un vector $a = (a_1, a_2, \dots, a_n)$ para representar una solución candidata, donde cada $a_i \in A_i$ donde A_i es un conjunto ordenado y finito.

El espacio de soluciones final consiste en $A_1 \times A_2 \times \dots \times A_n$.

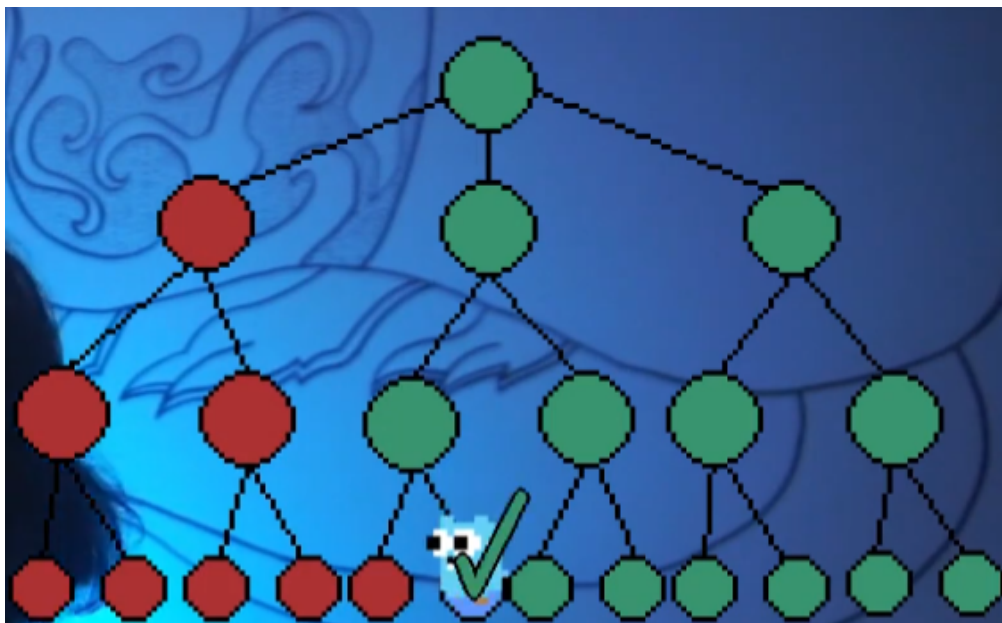
En cada paso se extienden las soluciones parciales $a = (a_1, a_2, \dots, a_k)$ con $k < n$ agregando un elemento más que es $a_{k+1} \in S_{k+1} \subseteq A_{k+1}$ al final del vector a . Esto quiere decir que las nuevas soluciones parciales son sucesores directos de la anterior. Si llegamos a un S_{k+1} que es vacío, retrocedemos a la solución parcial $(a_1, a_2, \dots, a_{k-1})$.

Esto último de retroceder podemos verlo como un árbol de decisiones, los cuales si un nodo ya de por sí no me sirve, sus hijos tampoco y vuelvo al nodo anterior para seguir evaluando los demás hijos.

Importante: Se puede pensar al vector a como un árbol, donde los elementos que hay son vértices. Por lo tanto, cada vértice y tendría vértices hijos x sí y solo sí se puede llegar a x yendo desde y .



Comenzamos evaluando desde la raíz, el nodo de la izquierda cumple hasta ahora nuestra posible solución, nos movemos a ese nodo y luego, evaluando su nodo de la izquierda vemos que se rompe, es decir, alguna restricción que pusimos no se cumple, por lo tanto no tiene sentido seguir explorando las demás soluciones.



Vemos que efectivamente, habiendo vuelto a la raíz, ahora sí encontramos un camino mejor que el camino del nodo de la izquierda. Por lo tanto, podemos seguir evaluando hacia abajo.

Podas por factibilidad: Evito explorar nodos no factibles.

Podas por optimalidad: Evito explorar nodos subóptimos.

Branch and bound: Uso la solución más óptima para comparar y ver si exploro eso no.

Todas las soluciones:

```

1 | algoritmo BI(a,k)
2 |   si a es solución entonces
3 |     procesar(a)
4 |     retornar
5 |   sino
6 |     para cada a' en Sucesores(a,k)
7 |       BI(a', k + 1)
8 |     fin para
9 |   fin si
10 |   retornar

```

Una solución:

```

1 | algoritmo BT(a,k)
2 |   si a es solución entonces
3 |     sol ← a
4 |     encontro ← true
5 |   sino
6 |     para cada a' en Sucesores(a,k)
7 |       -----BT(a', k + 1)
8 |         si encontro entonces
9 |           retornar
10 |        fin si
11 |     fin para
12 |   fin si
13 |   retornar

```

Ej.: Resolver un sudoku se resuelve en forma muy eficiente con un algoritmo de backtracking.

Casos de Uso de Backtracking

- Problemas de Decisión.
- Problemas de Optimización: Encontrar la mejor solución.
- Problemas de Enumeración: Todas las soluciones factibles a un problema.

Programación Dinámica

Consiste en reutilizar valores previamente calculados para ahorrar tiempo. **¿Cuál es el costo?**, el costo es que acá **usamos más espacio de la memoria**.

Nota: Si queremos sacar ventaja de la memorización **tenemos que acceder al valor guardado en la estructura de datos en $O(1)$ o en un menor tiempo que lo que costaría calcularlo** otra vez.

Aunque no lo parezca, guardar información que vamos a terminar reutilizando ahorra mucho tiempo.

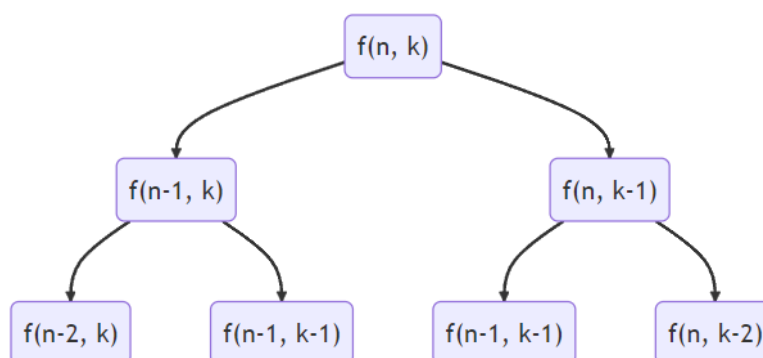
Normalmente, la programación dinámica se utiliza cuando de antemano ya sabemos que vamos a tener que terminar reutilizando valores que ya previamente calculamos para esos parámetros de entrada. A esto se le llama **superposición de estados** y ocurre cuando un árbol de llamadas recursivas resuelve el mismo problema para mismos parámetros de entrada.

Veamos ahora dos formas de encarar un problema de programación dinámica

Enfoque Top-Down

Se implementa recursivamente. La idea es ir guardando el resultado de cada llamada recursiva en una estructura de datos (**memorización**). Si llegase a suceder que en una nueva llamada vemos que esa llamada ya ocurrió previamente con los mismos parámetros podemos devolver el valor previamente calculado almacenado en la estructura de datos.

Una forma de visualizarlo es con el árbol de llamados. Si tenemos una función $f(n, k)$ definida como $f(n, k) = f(n - 1, k) + f(n, k - 1)$ con $k < n$ podríamos representar el pensamiento top-down de la siguiente manera



Claramente podemos observar como $f(n - 1, k - 1)$ se repite de ambos lados del árbol. Si esto fuese costoso de calcular, estaríamos perdiendo tiempo en algo que podríamos haber guardado.

El nombre de Top-Down viene del lado que partimos del problema más grande y terminamos llegando a un caso base. Véase [anexo](#) para ver qué tanto mejora el tiempo de ejecución

Enfoque Bottom-Up

Generalmente es iterativo, pero no siempre. Empieza resolviendo las partes más pequeñas y fundamentales del problema. Cada vez que resuelve un problema, almacena su solución en una estructura de datos auxiliar.

Divide & Conquer

Esta técnica consiste en dividir **un problema en subproblemas del mismo tipo** que en el original, resolver los subproblemas y luego combinar las soluciones.

Importante: Cuando hablamos de subproblemas, hablamos de subproblemas necesariamente más chicos que el problema original; Deben ser todos del mismo tipo y no podemos resolver diferentes subproblemas.

Ej.: Si tengo una pared enorme, la puedo pintar por partes. Si a la primera la pinté de arriba hacia abajo la segunda la voy a pintar igual y exactamente con los mismos movimientos. Al final de todo, veré que la pared quedó perfectamente pintada. El ejemplo representa claramente un problema de Divide & Conquer pues

- Al ser un problema grande, divido la tarea de pintar la pared enorme en partes iguales de una manera más sencilla y llevadera.
- Las paredes las pinto a todas de arriba hacia abajo y con exactamente los mismos movimientos (subproblemas del mismo tipo).
- Como sé que la porción de pared anterior quedó perfectamente pintada y la siguiente también lo estará, entonces terminaré pintando la pared entera.

Importante: Divide & Conquer debería tener una complejidad parecida a resolver el problema original. Si resolver el problema original nos cuesta $O(n)$ y con esta técnica $O(n^3)$ algo anda mal porque el tiempo de resolución debería ser igual (o menor) separando y resolviendo los subproblemas.

Teorema Maestro

Nos permite calcular la complejidad temporal de un algoritmo que involucra recursión.

$$T(n) = \begin{cases} a * T(n/c) + f(n) & \text{si } n > 1 \\ 1 & \text{si } n = 1 \end{cases}$$

Donde:

- a: cantidad de subproblemas en los que el problema se divide.
- n/c: es el tamaño de cada subproblema.
- c: cantidad de problemas.
- T(n/c): Es el tiempo de ejecución del algoritmo para un problema de tamaño n/c.
- f(n): Costo de dividir el problema y combinar los resultados de los subproblemas.

Casos del Teorema Maestro

- Si $f(n) = O(n^{\log_c a - \epsilon})$ con $\epsilon > 0$, entonces $T(n) = \Theta(n^{\log_c a})$
- Si $f(n) = \Theta(n^{\log_c a})$ entonces $T(n) = \Theta(n^{\log_c a} \log n)$
- Si $f(n) = \Omega(n^{\log_c a + \epsilon})$ para $\epsilon > 0$, y si $af(\frac{n}{c}) < kf(n)$ para $k < 1$ y para n suficientemente grandes, entonces $T(n) = \Theta(f(n))$

Qué no puede pasar en el Teorema Maestro

- f(n) no puede ser negativo porque es el tiempo que tarda el algoritmo. Siempre es positivo.
- a es una constante positiva pues el número de subproblemas una vez que arranca la recursión es fija.

Heurísticas

Es un procedimiento computacional que intenta obtener soluciones de buena calidad para un problema y que su comportamiento sea preciso.

Decimos que A es un algoritmo ϵ -aproximado ($\epsilon \geq 0$) para un problema si

$$\left| \frac{X_A - X_{OPT}}{X_{OPT}} \right| \leq \epsilon$$

Algoritmos Golosos (Algoritmos Greedy)

Es una técnica que consiste en elegir en cada paso la mejor opción sin considerar lo que pueda pasar después. Saber cual es la mejor opción depende de qué nos pidan como prioritario.

¿A qué me refiero como prioritario?

- Si te pido que de una lista de 100 películas, tomes la mayor cantidad de películas para que yo vea en 8hs: La idea sería que tomes las películas con menor duración (porque a menor duración, mas películas vas a elegir)
- Si te pido que vayas a la panadería y traigas 4kg de pan: No se puede resolver con greedy, porque no te especifico cuál sería el pan ideal.

Veamos algunos ejemplos más

- Tengo monedas de 1, 5, 10 y 25 centavos. Debo dar 0,69 de vuelto con la **menor cantidad de monedas**.
 - IDEA: La mejor elección en este caso que tengo que devolver menor cantidad de monedas, me conviene devolver más monedas de la moneda con mayor valor.
 - Toma 1 moneda de 25 pues sumando lo que tengo sucede que $0,25 < 0,69$.
 - Toma 1 moneda de 25 pues sumando lo que tengo sucede que $0,50 < 0,69$.
 - Toma 1 moneda de 25 $0,75 < 0,69$ es falso, por lo que sigo teniendo 0,50.
 - Toma 1 moneda de 10 pues sumando lo que tengo sucede que $0,60 < 0,69$.
 - Toma 1 moneda de 10 $0,70 < 0,69$ es falso, por lo que sigo teniendo 0,60.
 - Toma 1 moneda de 5 pues sumando lo que tengo sucede que $0,65 < 0,69$.
 - Toma 1 moneda de 5 $0,70 < 0,69$ es falso, por lo que sigo teniendo 0,65.
 - Toma 1 moneda de 1 (4 veces) pues sumando lo que tengo sucede que $0,69 < 0,69$
- Un servidor tiene n clientes para atender, los puede atender en cualquier orden. El objetivo es determinar en que orden se deben atender los clientes para **minimizar la suma de los tiempos de espera** de los clientes.
 - Me pongo en situación: Supongamos que tenemos 1000 clientes. Tenemos 2 representantes, 2 de los clientes tienen problemas que se resuelven en 4hs. Los demás, tienen problemas pero se resuelven en poco tiempo. IDEA: Ordeno los clientes por tiempo de demora de sus trámites y voy resolviendo los más cortos. Esto producirá que los clientes se vayan contentos porque se los atiende rápido (pues sabemos que a la larga, la suma de los tiempos de los trámites de los clientes con trámites cortos, es igual o menor a atender a solo los 2 clientes).
 - Mal camino: Si priorizáramos a los clientes que tienen trámites más largos tendríamos 998 clientes enojados.
 - Buen camino: Si priorizáramos a los 998 clientes aunque los otros dos hayan llegado antes, seguramente ellos estén enojados pero es mejor tener 998 felices que solo 2.

Esto demuestra que los Algoritmos Golosos nos dan solución para **minimizar el tiempo total de espera en un sistema atendiendo por menor tiempo de atención** y también **minimizar la cantidad de monedas que devolvemos, pero seguramente tendremos más monedas de mayor valor**.

Anexo

Programación Dinámica en C++ (Top-Down)

Importante: Notar que estoy usando long long porque los números que puede tomar fibonacci son tan grandes, que evito correr riesgos con int.

Importante: Notar que estoy enviando el vector por referencia. De lo contrario, se generaría uno nuevo en cada llamada.

```

1 Con memorización
2 #include <iostream>
3 #include <vector>
4 #include <chrono>
5 using namespace std::chrono;
6
7 long long fibonacci(long long n, std::vector<long long>& memo)
8 {
9     if (memo[n] != -1)
10    {
11        return memo[n];
12    }
13
14    if (n == 0)
15    {
16        return 0;
17    }
18    else if (n == 1)
19    {
20        return 1;
21    }
22
23    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
24    return memo[n];
25 }
26
27 int main()
28 {
29     auto start = high_resolution_clock::now();
30     int n = 0;
31     std::cout << "Ingrese un numero para calcular fibonacci" << std::endl;
32     std::cin >> n;
33     std::vector<long long> memo(n + 1, -1);
34     long long fibo = fibonacci(n, memo);
35     std::cout << "Fibonacci-de-" << n << "-es-" << fibo << std::endl;
36
37     // chrono
38     auto stop = high_resolution_clock::now();
39     auto duration = duration_cast<microseconds>(stop - start);
40     std::cout << "El tiempo de ejecucion es:-" << duration.count() << "-milisegundos" << std::endl;
41
42     return 0;
43 }
44
45 Fibon 50 (12586269025): 2887439 milisegundos.

```

```

1 Sin memorización
2 #include <iostream>
3 #include <vector>
4 #include <chrono>
5 using namespace std::chrono;
6
7 long long fibonacci(int n)
8 {
9
10    if (n == 0)
11    {
12        return 0;
13    }
14    else if (n == 1)
15    {
16        return 1;
17    }

```



```

18     return fibonacci(n - 1) + fibonacci(n - 2);
19 }
20
21
22 int main()
23 {
24     auto start = high_resolution_clock::now();
25     int n = 0;
26     std::cout << "Ingrese un numero para calcular fibonacci" << std::endl;
27     std::cin >> n;
28     long long fibo = fibonacci(n);
29     std::cout << "Fibonacci de " << n << " es " << fibo << std::endl;
30
31     // chrono
32     auto stop = high_resolution_clock::now();
33     auto duration = duration_cast<microseconds>(stop - start);
34     std::cout << "El tiempo de ejecucion es: " << duration.count() << " milisegundos" << std::endl;
35
36     return 0;
37 }
38 Fibo 50: ? milisegundos. No termina, se cuelga.

```