

ADA 24A

Análisis y Diseño de Algoritmos

Notas de Clase

INGENIERÍA EN SISTEMAS Y COMPUTACIÓN, UNIVERSIDAD DE CALDAS, MANIZALES

GITHUB.COM/OVERCV

Estas notas de clase han sido realizadas bajo instrucción de la Dr. Luz Guerrero dentro un total de 16 semanas, desde febrero 05 hasta junio 21 del 2024.

Primera publicación, Agosto 2024



Índice general

1	Notación asintótica	7
1.1	Crecimiento en funciones	7
1.1.1	Notación Tilde	7
1.1.2	Big O	7
1.1.3	Big Ω	8
1.1.4	Big Θ	8
1.1.5	Little o	9
1.1.6	Little ω	9
1.1.7	Propiedades generales	10
1.1.8	A dos variables	11
1.2	Funciones comunes	11
1.2.1	Monótonamente incremental	11
1.2.2	Monótonamente decremental	12
1.2.3	Estrictamente incremental	12
1.2.4	Estrictamente decremental	12
1.2.5	Aritmética modular	12
1.2.6	Polinomios	13
1.2.7	Logaritmos	13
1.2.8	Factoriales	13
1.2.9	Iteración funcional	14
1.2.10	Logaritmo estrella	14
2	Análisis algorítmico	15
2.1	Introducción	15
2.1.1	Sequential Search	16

2.2	Probabilidad	17
2.2.1	Eventos deterministas	17
2.3	Principio de correctitud	17
2.3.1	Insertion Sort	18
2.3.2	Correctitud	18
2.3.3	Función de eficiencia	18
2.4	Límites	19
2.5	Series	20
2.5.1	Propiedades	20
2.5.2	Series comunes	20
2.5.3	Productorias	22
2.5.4	Límites	23
2.5.5	Criterios de convergencia	23
3	Divide y vencerás	25
3.1	Ordenamiento	25
3.1.1	Merge Sort	25
3.1.2	Quick sort	26
3.1.3	Heaps	28
3.1.4	Heap Sort	31
3.1.5	Priority Queues	31
4	Algoritmos voraces	33
4.1	Introducción	33
5	Backtracking	35
5.0.1	Las 08 reinas	36
5.0.2	Las N reinas	36
5.0.3	Suma de subconjuntos	36
5.0.4	El viajero comerciante	36
5.0.5	Coloreo de un grafo (m – colorability)	36
5.0.6	Laberintos	36
6	Branch&Bound	39
6.0.1	Problema de la mochila 0-1	42
6.0.2	Asignación de tareas	42
7	Programación dinámica	43
7.1	Introducción	43
7.1.1	Reglas	43
7.1.2	Sucesión de fibonacci	43
7.1.3	Problema del cambio	44
7.1.4	Assembly line scheduling	48

7.1.5	Subtleties	48
7.1.6	Optimal matrix chain product	48
7.1.7	Longest common subsequence <i>LCS</i>	49
7.1.8	Optimal binary search trees	49

8 Algoritmos Heurísticos 51

8.1	Búsquedas	51
8.1.1	Búsqueda de coste uniforme	51
8.1.2	Best First Search (Greedy Best First)	52
8.1.3	A* (A Star)	53
8.1.4	Búsqueda Óptima	53
8.1.5	Búsqueda limitada por profundidad	53
8.2	Teoría de juegos	54
8.2.1	Algoritmo Min-Max	54
8.2.2	Algoritmo MINIMAX	54
8.2.3	Poda $\alpha - \beta$	55

1. Notación asintótica

Son comprendidas como familias, nos preguntamos si una función particular que pertenece a una familia $X(f(n))$ también pertenece a otra familia $Y(g(n))$. Es así que generamos comparaciones entre funciones.

1.1 Crecimiento en funciones

Funciones no negativas

Una $f(x)$ es asintóticamente no negativa si $\exists n_0 \in \mathbb{N}$ $f(n) \geq 0$ y se cumpla $(n \geq n_0) \wedge f(n) \geq 0$
Apreciable con la función $f(x) = a e^{-bx} + cx e^{-dx}$.

Por definición trabajamos funciones asintóticamente positivas ($t \geq 0$).

1.1.1 Notación Tilde

$$f(n) \sim g(n) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1 \implies f(n) \in O(g(n))$$

Donde converger a 1 no necesariamente implica sean la misma función.

1.1.2 Big O

Definition 1.1.1

$$O(g(n)) = \{f : \mathbb{N} \rightarrow \mathbb{R}^* \mid (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(0 \leq f(n) \leq c g(n))\}$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c$$

$$f(n) = O(g(n)) \equiv f(n) \in O(g(n))$$



Dicción en 'Big Oh':

'Está f acotada superiormente por g ', 'Es g cota superior a f '

Si dado $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \infty \Rightarrow \{\infty : g \text{ crece} > f, k : g \text{ crece} \approx f, \}$

Si el conjunto de datos está limitado, una función n^3 puede ser mejor a una n^2 .

■ **Example 1.1** Demostración que $an + b = O(n)$

Ha de satisfacerse que $an + b \leq cn$

$$an + b \leq cn$$

$$a + \frac{b}{n} \leq c$$

Es así que tenemos el valor de nuestra constante, podemos apreciar cómo se empezará a cumplir cuando determinemos un valor para n renombrado a n_0 , desde ese punto se cumplirá la notación. ■

1.1.3 Big Ω

Definition 1.1.2

$$\Omega(g(n)) = \{f : \mathbb{N} \rightarrow \mathbb{R}^* | (\exists c \in \mathbb{R}^+) (\exists n_0 \in \mathbb{N}) (\forall n \geq n_0) (0 \leq cg(n) \leq f(n))\}$$

$$c \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$



Dicción en 'Big Omega':

'Está f acotada asintóticamente por debajo por g' ', 'Es g una cota inferior asintótica para f' '.

1.1.4 Big Θ

Definition 1.1.3

$$\Theta(g(n)) = \{f : \mathbb{N} \rightarrow \mathbb{R}^* | (\exists c_1, c_2 \in \mathbb{R}^+) (\exists n_0 \in \mathbb{N}) (\forall n \geq n_0) (c_1g(n) \leq f(n) \leq c_2g(n))\}$$

$$c_1 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c_2$$



Dicción en 'Big Theta':

'Está f acotada estrechamente por g' ', 'Es g una cota estrecha para f' '.

Cada miembro de $\Theta(g(n))$ es asintóticamente no negativo así como la función misma (*si no* $\Theta(g(n)) = \emptyset$)

Theorem 1.1.1 $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

Propiedades

Un algoritmo α clasifica sí y solo si tiene

- Peor tiempo de ejecución es $O(f(n))$.
- Mejor tiempo de ejecución es $\Omega(f(n))$.

- **Example 1.2** Demostración que $0,5n^2 - 3n = \Theta(n^2)$
Ha de satisfacerse que $c_1 n^2 \leq 0,5n^2 - 3n \leq c_2 n^2$

$$c_1 n^2 \leq 0,5n^2 - 3n \leq c_2 n^2$$

$$c_1 \leq 0,5 - 3/n \leq c_2$$

La inecuación derecha se mantiene para $n \geq 1$ si tomamos $c_2 \geq 0,5$. La inecuación izquierda con $n \geq 7$ y escogiendo $c_1 < 1/14$. ■

1.1.5 Little o Definition 1.1.4

$$o(g(n)) = \{f : \mathbb{N} \rightarrow \mathbb{R}^* | (\forall c \in \mathbb{R}^+) (\exists n_0 \in \mathbb{N}) (\forall n \geq n_0) (0 \leq f(n) < cg(n))\}$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0; \quad \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \infty$$

Son las funciones $o(g(n))$ que crecen más lento que g .



Dicción en 'Little oh':

' f es asintóticamente más pequeña a g '. ' g es cota *estrictamente débil* superior a f '

Demostración. Es $n = o(n^2)$

Ha de satisfacerse que:

$$\lim_{n \rightarrow \infty} \frac{n}{n^2} = \lim_{n \rightarrow \infty} \frac{1}{n} = 0$$



Demostración. Es $n \neq o(3n)$

Ha de satisfacerse que:

$$\lim_{n \rightarrow \infty} \frac{n}{3n} = \lim_{n \rightarrow \infty} \frac{1}{3} = \frac{1}{3} \neq 0$$



1.1.6 Little ω Definition 1.1.5

$$\omega(g(n)) = \{f : \mathbb{N} \rightarrow \mathbb{R}^* | (\forall c \in \mathbb{R}^+) (\exists n_0 \in \mathbb{N}) (\forall n \geq n_0) (cg(n) < f(n))\}$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty; \quad \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$$

Las funciones $\omega(g(n))$ crecen más rápido que g .



Dicción en 'Little omega':

' f es asintóticamente más grande a g '. ' g es cota débil inferior a f '.

Theorem 1.1.2 — Análogo entre números \mathbb{R} y Notación asintótica.

Notación asintótica Número real

$f(n) \in O(g(n))$	$f \leq g$
$f(n) \in \Omega(g(n))$	$f \geq g$
$f(n) \in \Theta(g(n))$	$f = g$
$f(n) \in o(g(n))$	$f < g$
$f(n) \in \omega(g(n))$	$f > g$

No se mantiene la tricotomía.



Notación Little; No son asintóticamente estrechas.

1.1.7 Propiedades generales

Transitividad

$$(f(n) \in \Delta[g(n)] \wedge g(n) \in \Delta[h(n)]) \implies (f(n) \in \Delta[h(n)])$$

$$\forall \Delta = O, \Omega, \Theta$$

Reflexividad

$$f(n) \in \Delta[f(n)]$$

$$\forall \Delta = O, \Omega, \Theta$$

Simetría

$$f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$$

$$\forall \Theta$$

Anti simetría

$$\forall f(n) \notin \Theta(g(n))$$

$$f(n) \in \Delta[g(n)] \implies g(n) \notin \Delta[f(n)]$$

$$\forall \Delta = O, \Omega$$

Simetría transpuesta

$$f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$$

$$f(n) \in o(g(n)) \iff g(n) \in \omega(f(n))$$

$$\forall \Delta = O, \Omega$$

Theorem 1.1.3 — Órdenes de relación.

Órden	Relexiva	Simétrica	Anti simétrica	Transitiva
$f \leq g \iff f(n) \in O(g(n))$	Sí		Sí	Sí
$f \geq g \iff f(n) \in \Omega(g(n))$	Sí		Sí	Sí
$f = g \iff f(n) \in \Theta(g(n))$	Sí	Sí		Sí

Con esto se pueden comprender las relaciones como

- Entre $o \wedge O$

$$f(n) \in o(g(n)) \implies f(n) \in O(g(n))$$

- Entre $\omega \wedge \Omega$

$$g(n) \in \omega(f(n)) \implies g(n) \in \Omega(f(n))$$

Siempre que $o(f(n)) \cap \omega(f(n)) = \emptyset$.

- **Example 1.3** Realizemos estos ejercicios

1.1.8 A dos variables**Definition 1.1.6**

$$\begin{aligned} O(g(m, n)) = \{f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}^* \mid \\ (\exists c \in \mathbb{R}^+)(\exists m_0, n_0 \in \mathbb{N})(\forall n \geq n_0)(\forall m \geq m_0)(f(m, n) \leq cg(m, n))\} \end{aligned}$$

- **Example 1.4 — Ejercicio.** Del 1, 2 demostrar si V o F.

1. Para cualquier función f se tiene que $f \in O(f)$. [V].
2. $O(f) = O(g) \iff f \in O(g) \wedge g \in O(f)$. [V].
3. Para cuáles notaciones es válido afirmar; $f \in \Delta[g] \wedge g \in \Delta[h] \implies f \in \Delta[h]$
4. Si $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k$ dependiendo de los valores en k .
 - a) Si $k = 0 \wedge k < 0 \implies O(f) = O(g)$
 - b) Si $k \neq 0 \wedge k < 0 \implies O(f) \in O(g)$

1.2 Funciones comunes**1.2.1 Monótonamente incremental****Definition 1.2.1**

$$\forall m \leq n \implies f(m) \leq f(n)$$

Definition 1.2.2 — Función suelo. El $\lfloor x \rfloor = \max(\mathbb{Z}) \leq x$

Definition 1.2.3 — Función techo. El $\lceil x \rceil = \min(\mathbb{Z}) \geq x$

$$\begin{aligned} \forall x \in \mathbb{R}, \quad x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1 \\ \forall x \in \mathbb{N}, \quad (\lfloor x \rfloor = x = \lceil x \rceil) \wedge (\lfloor x/2 \rfloor + \lceil x/2 \rceil = x) \end{aligned}$$

Theorem 1.2.1

$$\forall x \in \mathbb{R} \wedge \mathbb{Z} : a, b > 0$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor$$

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil$$

$$\lfloor a/b \rfloor \leq (a + (b - 1))/b$$

$$\lceil a/b \rceil \leq (a - (b - 1))/b$$

Cabe mencionar estas funciones son monótonamente incrementales.

Definition 1.2.4 — Función exponencial. Si $\forall n : a \geq 0 \implies a^n$ es monótonamente incremental.

Si $\forall (a, b) \in \mathbb{R}$, $a > 1$ y

$$\lim_{n \rightarrow \infty} \frac{n^d}{a^n} = 0 \implies n^d = o(a^n)$$

1.2.2 Monótonamente decremental**Definition 1.2.5**

$$\forall m \leq n \implies f(m) \geq f(n)$$

1.2.3 Estrictamente incremental**Definition 1.2.6**

$$\forall m < n \implies f(m) < f(n)$$

1.2.4 Estrictamente decremental**Definition 1.2.7**

$$\forall m < n \implies f(m) > f(n)$$

1.2.5 Aritmética modular**Theorem 1.2.2**

$$\forall \mathbb{Z}a \wedge \exists \mathbb{Z}^+n$$

$$a \pmod n = a - n \lfloor a/n \rfloor$$

El residuo del cociente $\frac{a}{n}$

Theorem 1.2.3 — Equivalencias. Si $a \pmod n = b \pmod n \implies a \equiv b \pmod n$ dictamos a es congruente o equivalente con $b \pmod n$. Dan el mismo residuo al dividirse por n . Lo será si y sólo si n es divisor de $b - a$

1.2.6 Polinomios

Son funciones de la forma;

$$p(n) = \sum_{i=0}^d a_i n^i. \quad d > 0$$

Con coeficientes $a_0, a_1, \dots, a_d \wedge a_d \neq 0$

Theorem 1.2.4

- Es $p(n)$ asintóticamente positivo sí y sólo si $a_d > 0$.
- Sea $p(n)$ de grado d asintóticamente positivo entonces $p(n) = \Theta(n^d)$.
- $\forall a \in \mathbb{R}, \quad a \geq 0, \quad n^a$ es monótonamente incremental.
- $\forall a \in \mathbb{R}, \quad a \leq 0, \quad n^a$ es monótonamente decremental.
- $f(n)$ es polinómicamente acotada si $f(n) = O(n^d)$ para algúna constante d .

1.2.7 Logaritmos

Definition 1.2.8 — Notaciones.

$$\lg n \equiv \log_2 n$$

$$\ln n \equiv \log_e n$$

$$\lg^k n \equiv (\lg n)^k$$

$$\lg \lg n \equiv \lg(\lg n)$$

Sólo aplican sobre el siguiente término.

$$\lg n + k \equiv (\lg n) + k$$

Es incremental para $b > 1 \wedge n > 0$

Theorem 1.2.5 — Identidades. Para los reales $(a, b, c > 0) \wedge n$ tenemos las siguientes identidades

1. $a = b^{\log_b a}$
2. $\log_c(ab) = \log_c a + \log_c b$
3. $\log_b a^n = n \log_b a$
4. $\log_b a = \frac{\log_c a}{\log_c b}$
5. $\log_b a = 1 / \log_a b$
6. $a^{\log_b c} = c^{\log_b a}$

1.2.8 Factoriales

Definition 1.2.9

Sin recursión:

$$n! = \begin{cases} n = 0 \implies 1 \\ n > 0 \implies \prod_{i=1}^n i \end{cases}$$

Con recursión:

$$n! = \begin{cases} n = 0 \implies 1 \\ n > 0 \implies n(n-1)! \end{cases}$$

Fact 1.2.6 Cota débil superior:

$$n! \leq n^n$$

Re-expresiones del factorial:

$$(n+1)! = n!(n+1)$$

$$n! =$$

1.2.9 Iteración funcional

Dada $f(n)$, la i -th iteración funcional es:

$$f^{(i)} : \begin{cases} i = 0 \implies I \\ i > 0 \implies f(f^{(i-1)}) \end{cases}$$

Sea I la función identidad, con n particular:

$$f^{(i)}(n) : \begin{cases} i = 0 \implies n \\ i > 0 \implies f(f^{(i-1)}(n)) \end{cases}$$

1.2.10 Logaritmo estrella

Definition 1.2.10 $\lg^* n = \min\{i \geq 0 \mid \lg^i n \leq 1\}$

Elevando k veces:

$$\lg^* 2^{2^{\dots^2}} = k$$

Fact 1.2.7 — Crecimiento.

$$\lg^* 2^0 = 0$$

$$\lg^* 2^1 = 1$$

$$\lg^* 2^2 = 2$$

$$\lg^* 2^4 = 3$$

$$\lg^* 2^{16} = 4$$

$$\lg^* 2^{256} = 5$$

⋮

Su crecimiento es ínfimo.

2. Análisis algorítmico

Estamos acostumbrados a manejar estructuras:

- De control: if, else if, else, anidaciones.
 - De flujo: for, while, repeat.
 - De secuencia: Donde las instrucciones (accion i) se ejecutan sucesivamente sin omisiones (accion i requiere accion $i - 1$).

2.1 Introducción

Un algoritmo es el conjunto de procedimientos o funciones o bloques con estructuras de control, deben ser precisos y dar término. Una función *func* realizan una tarea y devuelven un valor como resultado o salida. Un procedimiento *proc* es un fragmento de código que no retorna un dato sino que trabaja mutando los datos del encabezado. Los parámetros de un algoritmo son 03:

1. Forma: Entrada, Salida o Mixta.
 2. Tipo: La estructura o tipo de dato.
 3. Nombre: Clave cual referencia la variable.

■ Example 2.1 — Costeo algorítmico.

Dado el procedimiento P_0 en cada línea debemos costear: Número de operaciones elementales, Número de ejecuciones.

```
1| proc P0(E int n):
2|     x = 100
3|     z = x + 5
4|     for (i = 1 to n) do
5|         for (j = 1 to n) do
6|             x = x + 1
```

1. Llamado sin coste asociado.
 2. 1×1
 3. 2×1
 4. $3 \times \sum_{i=1}^{n+1} 1$
 5. $3 \times \sum_{i=1}^n \sum_{j=1}^{n+1} 1$
 6. $2 \times \sum_{i=1}^n \sum_{j=1}^n 1$

En este escenario al usarse un for se iterará siempre la misma cantidad, con ello estamos en un caso invariante o promedio.

Nótese como siempre el número de ejecuciones en un ciclo se sigue por: $cabeza = cuerpo + 1$. Eventualmente resolviendo las sumatorias obtenemos la función de eficiencia para el Coste en Complejidad Computacional Temporal ($CC.T(n)$):

$$T(n) = 1 \times 1 + 2 \times 1 + 3 \times (n + 1) + 3 \times n \times (n + 1) + 2 \times n \times n$$

$$T(n) = an^2 + bn + c \in \Theta(n^2)$$

Si realizamos el costeo en términos de la Complejidad Computacional Espacial ($CC.S(n)$) notamos como sólo se realizan operaciones elementales, todas con un coste constante (c).

$$S(n) = c \in \Theta(1)$$

■

La jerarquía de operaciones para cálculos computacionales es:

1. Brackets.
2. Powers.
3. Products, Ratios.
4. Relational operations.

Realizado sobre una función:

2.1.1 Sequential Search

■ Example 2.2 — Análisis búsqueda secuencial.

Deben definirse:

Precondiciones: Lo que entra; Un arreglo de números enteros y un número entero.

Poscondiciones: Lo obtenido y cómo se obtiene, además de excepciones; Índice del elemento (número entero) ingresado en el arreglo, si no existe retorna -1 .

```

1| func seq_search(
|   E list[int] A[n], int x, S int p
| ):
2|   int i = 1
3|   repeat:
4|     if A(i) == x:
5|       return i
6|     i++
7|   until i > n
8|   return -1

```

El mejor escenario o T_{best} es que exista el elemento y sea el primero en la lista.

1. Sin coste.
2. 1×1
3. Sin coste.
4. 1×1
5. 1×1
6. 0×1
7. 0×1
8. 0×1

$$T_{best}(n) = 1 \times 1 + 1 \times 1 + 1 \times 1$$

$$T_{best}(n) = c \in O(1)$$

No obstante aún faltan escenarios.

```

1| func seq_search(
|   E list[int] A[n], int x, S int p
| ):
2|   int i = 1
3|   repeat:
4|     if A(i) == x:
5|       return i
6|     i++
7|   until i > n
8|   return -1

```

El peor escenario o T_{worst} es que no exista el elemento.

1. Sin coste.
2. 1×1
3. $0 \times n$
4. $1 \times n$
5. $0 \times n$
6. $1 \times n$
7. $1 \times n$
8. 1×1

$$T_{worst}(n) = 1 \times 1 + 1 \times n + 1 \times 1 \times n + 1 \times 1$$

$$T_{worst}(n) = an + b \in O(n)$$

■

2.2 Probabilidad

2.2.1 Eventos deterministas

Definition 2.2.1 — Suceso elemental. Recoge información de todo el experimento

■ **Example 2.3 — Dados.**

Lanzar un dado genera 1 resultado o suceso elemental; Tras 10 resultados generamos un prorratoeo, una razón.

■

Definition 2.2.2 — Sucesos. Seguro: Donde la probabilidad del evento es 1, $P(S) = 1$.

Imposible: Donde la probabilidad del evento es 0, $P(S) = 0$.

■

Probabilidad condicional

Definida como $P(S|T) = \frac{P(S \cap T)}{P(T)}$

■ **Example 2.4 — Dados.**

Tras lanzar 02 dados se tiene que $(S_1 : D_1 \text{ cae } 1)$, para $(S_2 : D_2 \text{ cae } 6)$ y el $(S_3 : D_1 + D_2 \leq 4)$. Se busca hallar la $P(S_1|S_3)$.

Encontramos que $P(S_1) = 6/36 = 1/6$, la $P(S_3) = 6/36 = 1/6$ y que la $P(S_1 \cap S_3) = 3/36 = 1/12$.

$$P(S_1|S_3) = \frac{3/36}{6/36} = 1/2$$

■

2.3 Principio de correctitud

Un algoritmo es correcto si para cada posible entrada se termina con la salida correcta (*hace lo que afirma hacer*), hay 02 pruebas.

- Correctitud parcial: Si el algoritmo no hace *halting* entonces se produce un resultado correcto (*PIM*).
- Terminación: Si da término en finitud de pasos, independientemente a la entrada.

2.3.1 Insertion Sort

Input: Arreglo $A = [a_1, \dots, a_n]$.

Process: Indexado lineal e iterandos contiguos, ante $A_j > A_i$ activa subiteración; sobre-escribe valores $A_{j+1} = A_j$ para ser decreciente izquierda. Finalmente escribe x en penúltima j-iteración.

```
def insertion_sort(A: list[int | float]) -> list[int | float]:
    ''' Numeric list sorted by insertion '''
    for i in range(1, len(A)):
        x: float = A[i]
        j: int = i-1
        while A[j] > x and j >= 0:
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = x
    return A
```

2.3.2 Correctitud

Definición 2.3.1 — Lazo invariante. Sentencia cual prueba la correctitud algorítmica, aplicada sobre variables. A demostrar:

- Inicialización: Ciento previo a iteración primera.
- Mantenimiento: Si es cierto previo a iteración, permanece cierto previo a próxima.
- Terminación: Retorna una propiedad cual muestra la correctitud algorítmica (elemento probatorio).

■ **Example 2.5 — En Insertion sort.** Tomando A en orden.

- Inicialización: El caso de A tamaño 1 es trivial.
- Mantenimiento: Cada elemento es menor a su siguiente.
- Terminación: En $i = n + 1$ permanecen los elementos originales de A **ordenados**.

2.3.3 Función de eficiencia

Contar la función más ejecutada (*permite saber el funcionamiento algorítmico*), es la posible ejecución más anidada.

■ **Example 2.6 — Iteraciones múltiples.** Un ejemplo práctico.

```
def triple_for(n: int) -> int:
    x: int = 0
    for i in range(n):
        for j in range(n):
            for k in range(n):
                x += 1
    return x
```

Con ello se buscará el comportamiento asintótico más representativo para el código mencionado. Para trabajar ciclos es necesaria la representación mediante sumatorias, desde la más interna a la más externa, donde tras resolver el más interno se pasa al más externo

hasta dejarse como una función de tendencia.

Supóngase $n = 3$ entonces lo primero es contarse cada ejecución en k : Haciendo las iteraciones:

$$triple_for \left\{ \begin{array}{l} i = 1 : \quad j = (1, 2), \quad k = (1, 2) \\ i = 2 : \quad j = (1, 2, 3), \quad k = (1, 2)(1, 2, 3) \\ i = 3 : \quad j = (1, 2, 3, 4), \quad k = (1, 2)(1, 2, 3)(1, 2, 3, 4) \end{array} \right.$$

- R** Tenemos en total contando únicamente el número de ejecuciones en k , representable por la fórmula:

$$\begin{aligned} f(n) &= \sum_{i=1}^n i^2 + i - 2 \\ &= \sum_{i=1}^n i^2 + \sum_{i=1}^n i - \sum_{i=1}^n 2 \\ &= \frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} - 2n \end{aligned}$$

Es apreciable que tras realizar la distribución de términos n se obtiene $\frac{1}{3}n^3 + \dots - \frac{4}{3}n$ como el de mayor grado con otros términos de menor grado. Es así que podemos decir eventualmente tenemos un comportamiento asintótico de n^3 , expresado **en la mejor forma** como $\Theta(n^3)$. ■

2.4 Límites

Tenemos como unos casos especiales que aplican a los límites:

$$\lim_{n \rightarrow \infty} \left(\frac{1}{n} \right) = 0$$

$$\lim_{n \rightarrow \infty} \left(\frac{n}{n+1} \right) = 1$$

$$\lim_{n \rightarrow \infty} (\sqrt[n]{a}) = 0; \quad a > 0$$

$$\lim_{n \rightarrow \infty} (\sqrt[n]{n}) = 1$$

$$\lim_{n \rightarrow \infty} \left(\frac{\ln[n]}{x^n} \right) = 0; \quad (a > 0) \wedge (b > 0)$$

$$\lim_{n \rightarrow \infty} \left(1 + \frac{1}{n} \right)^n = e$$

$$\lim_{n \rightarrow \infty} \left(\frac{\sin(x)}{x} \right) = 1$$

$$\lim_{n \rightarrow \infty} ATan(x) = \frac{\pi}{2}$$

$$\lim_{n \rightarrow \infty} ASec(x) = \frac{\pi}{2}$$

$$\lim_{x \rightarrow \infty} f(x) = L \implies \lim_{n \rightarrow \infty} f(n) = L$$

2.5 Series

Generalización sobre la noción de suma sobre una función $f(n)$ acotada superior e inferiormente.

2.5.1 Propiedades

Theorem 2.5.1 — Adiciones y sustracciones.

$$\sum_{i=0}^n [f(i) \pm g(i) \pm \dots \pm z(i)] = \sum_{i=0}^n f(i) \pm \sum_{i=0}^n g(i) \pm \dots \pm \sum_{i=0}^n z(i)$$

Theorem 2.5.2 — Constantes.

$$\sum_{i=0}^n c \cdot f(i) = c \cdot \sum_{i=0}^n f(i)$$

Theorem 2.5.3 — Cambio de índice.

$$\sum_{i=0}^n f(i) = \sum_{i=k}^{n+k} f(i-k)$$

Theorem 2.5.4 — Índice a 0.

$$\sum_{i=a}^b f(i) = \sum_{i=0}^{b-a} f(i+a)$$

Theorem 2.5.5 — Partición.

$$\sum_{i=0}^n f(i) = \sum_{i=0}^c f(i) + \sum_{i=c+1}^n f(i)$$

Theorem 2.5.6 — Dobles.

$$\sum_{i=0}^p \sum_{j=0}^q f(i,j) \equiv \sum_{j=0}^q \sum_{i=0}^p f(i,j)$$

2.5.2 Series comunes

Definition 2.5.1 — Aritmética. Forma general

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Fact 2.5.7 — Orden. $O(n^2)$

Definition 2.5.2 — Geométrica. Forma general

$$\sum_{i=0}^n ar^i = \frac{a(1 - r^{n+1})}{1 - r} \quad r \neq 1$$

a: Primer término. *r:* Razón común.

Fact 2.5.8 — Orden. $|r| > 1 : O(r^n); \quad |r| < 1 : O(1)$

Definition 2.5.3 — Base 2. Forma general

$$\sum_{i=0}^n ar^i = \frac{a(1 - r^{n+1})}{1 - r}; \quad r \neq 1$$

a: Primer término. *r:* Razón común.

Fact 2.5.9 — Orden. $|r| > 1 : O(r^n); \quad |r| < 1 : O(1)$

Definition 2.5.4 — Potencias. Forma general

$$\sum_{i=1}^n i^k$$

Solución indefinida;



Generalización alterna $k = 1$:

$$\begin{aligned} \sum_{i=0}^n ai + b &= \frac{(an + 2b)(n + 1)}{2} \\ k = 2 \implies \frac{2n^3 + 3n^2 + n}{6}; \quad k = 3 \implies \frac{n^4 + 2n^3 + n^2}{4}; \\ k = 4 \implies \frac{6n^5 + 15n^4 + 10n^3 - n}{30}; \quad k = 5 \implies \frac{2n^6 + 6n^5 + 5n^4 - n^2}{12}; \\ k = 6 \implies \frac{6n^7 + 21n^6 + 21n^5 - 7n^3 + n}{42}; \\ k = 7 \implies \frac{3n^8 + 12n^7 + 14n^6 - 7n^4 + 2n^2}{24}; \\ k = 8 \implies \frac{10n^9 + 45n^8 + 60n^7 - 42n^5 + 20n^3 - 3n}{90}; \\ k = 9 \implies \frac{2n^{10} + 10n^9 + 15n^8 - 14n^6 + 10n^4 - 3n^2}{20}; \\ k = 10 \implies \frac{6n^{11} + 33n^{10} + 55n^9 - 66n^7 + 66n^5 - 33n^3 + 5n}{66} \dots \end{aligned}$$

Fact 2.5.10 — Orden. $O(n^{k+1})$

Definition 2.5.5 — Armónica. Forma general

$$\sum_{i=1}^n \frac{1}{i}$$

Solución indefinida;

$$\ln n + \gamma. \quad \gamma : \text{Euler-Mascheroni} = - \int_0^\infty e^{-x} \ln x \, dx$$

Fact 2.5.11 — Orden. $O(\ln n)$

Definition 2.5.6 — Logarítmica. Forma general

$$\sum_{i=1}^n \log i \approx n \log n - n$$

Fact 2.5.12 — Orden. $O(n \log n)$

2.5.3 Productorias

Definition 2.5.7 — Factorial.

$$\prod_{i=1}^n i = n!$$

Definition 2.5.8 — Constante.

$$\prod_{i=1}^n k = k^n$$

Definition 2.5.9 — Generalizada.

$$\prod_i^n a_{i+1} = k^n \prod_{i=1}^n i$$

Definition 2.5.10 — Escalar.

$$\prod_{i=1}^n k i = k^n \prod_{i=1}^n i$$

Propiedades telescopicas

Definition 2.5.11 — Generalizada.

$$\prod_i^n a_{i+1} = k^n \prod_{i=1}^n i$$

Correlación

Definition 2.5.12

$$\lg \prod_{i=1}^n a_i = \sum_{i=1}^n \lg a_i$$

$$\prod_{i=1}^n a_i = 2^{\sum_{i=1}^n \lg a_i}$$

2.5.4 Límites

Es fundamental conocer los límites de cualquier función matemática para establecer relaciones próximamente en notaciones asintóticas. Recordemos sus propiedades:

Constantes	$\lim_{x \rightarrow c} k = k$
Identidad	$\lim_{x \rightarrow c} x = c$
Escalar	$\lim_{x \rightarrow c} kf(x) = k \lim_{x \rightarrow c} f(x)$
Exponente	$\lim_{x \rightarrow c} x^p = c^p; \quad r \geq 0$
Adición	$\lim_{x \rightarrow c} [f(x) + g(x)] = \lim_{x \rightarrow c} f(x) + \lim_{x \rightarrow c} g(x)$
Substracción	$\lim_{x \rightarrow c} [f(x) - g(x)] = \lim_{x \rightarrow c} f(x) - \lim_{x \rightarrow c} g(x)$
Producto	$\lim_{x \rightarrow c} [f(x) \times g(x)] = \lim_{x \rightarrow c} f(x) \times \lim_{x \rightarrow c} g(x)$
Razón	$\lim_{x \rightarrow c} [f(x) \div g(x)] = \lim_{x \rightarrow c} f(x) \div \lim_{x \rightarrow c} g(x); \quad \lim_{x \rightarrow c} g(x) \neq 0$
Potencia	$\lim_{x \rightarrow c} f(x)^{g(x)} = \lim_{x \rightarrow c} f(x)^{\lim_{x \rightarrow c} g(x)}; \quad f(x) > 0$
Logaritmo	$\lim_{x \rightarrow c} [\log f(x)] = \log [\lim_{x \rightarrow c} f(x)]$
Radical	$\lim_{x \rightarrow c} [\sqrt[n]{f(x)}] = \sqrt[n]{\lim_{x \rightarrow c} f(x)}$

Supóngase que se tiene una sucesión tal que tras hallar su n -ésimo término $\lim_{n \rightarrow \infty} = L$. Los casos especiales son los siguientes:

$$\begin{aligned} \lim_{n \rightarrow \infty} \left(\frac{1}{n} \right) &= 0 \\ \lim_{n \rightarrow \infty} \left(\frac{n}{n+1} \right) &= 1 \\ \lim_{n \rightarrow \infty} (\sqrt[n]{a}) &= 0; \quad a > 0 \\ \lim_{n \rightarrow \infty} (\sqrt[n]{n}) &= 1; \quad n > 0 \\ \lim_{n \rightarrow \infty} \left(\frac{\ln[n]}{x^n} \right) &= 0; \quad a, b > 0 \\ \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n} \right)^n &= e; \\ \lim_{n \rightarrow \infty} \left(\frac{\sin(n)}{n} \right) &= 1 \\ \lim_{n \rightarrow \infty} (ATan(x)) &= \lim_{n \rightarrow \infty} (ASec(x)) = \frac{\pi}{2} \end{aligned}$$

2.5.5 Criterios de convergencia

Es fundamental conocer dada una serie su valor finito convergente.

Criterio suficiente de divergencia

- Si $\lim_{n \rightarrow \infty} a_n$ no existe, o si $\lim_{n \rightarrow \infty} a_n \neq 0$ entonces la Serie diverge.
- Si el $\lim_{n \rightarrow \infty} a_n = 0$ entonces **NO** se concluye nada.
- Si $\sum a_n$ sabemos converge entonces $\lim_{n \rightarrow \infty} (a_n = 0)$.



3. Divide y vencerás

3.1 Ordenamiento

3.1.1 Merge Sort

Input: Arreglo $A = [a_0, a_1, \dots, a_n]$

Definition 3.1.1 — Paradigma. Resolución DyV:

- **Dividir:** Problema \rightarrow Sub-problemas $\in type(P_0)$.
- **Conquistar:** \forall Sub-problema (recursiva). $size(\text{sub-problema}) \rightarrow 0$.
- **Combinar:** Solución $=$ Sub-problema $+ \dots +$ Sub-problema.

■ **Example 3.1 — En Merge sort.** 1. **Dividir:** $A \rightarrow A[n/2] \wedge A[n/2]$. ■

Un paso fundamental a realizar es combinar 2 listas ordenadas en sólo una.

```
def merge(L: list[int], R: list[int]) -> list[int]:
    merged = []
    i: int = 0
    j: int = 0

    while i < len(L) and j < len(R):
        if L[i] <= R[j]:
            merged.append(L[i])
            i += 1
        else:
            merged.append(R[j])
            j += 1

    merged.extend(L[i:])
    merged.extend(R[j:])
    return merged
```

La función *merge(...)* tiene Complejidad computacional temporal (*CC.T*) $T(n) = n$, así como su Complejidad espacial $S(n) = n$ (*Isn't an in-place algorithm*).

El corazón del algoritmo

```
def merge_sort(A: list[int]) -> list[int]:
    if len(A) == 1:
        return A

    q = len(A) // 2
    L = merge_sort(A[:q])
    R = merge_sort(A[q:])
    return merge(L, R)
```

Si la longitud del arreglo A es 1 elemento entonces debemos retornar el mismo puesto es caso base. Partimos recursivamente el arreglo en 2, tomando su parte derecha e izquierda para mediante retorno (*backtracking*) combinarlos.

Análisis de eficiencia

Se realizan 02 llamadas dentro el método tal que se realiza una partición del tamaño de la entrada $T(n) = 2T(n/2)$, además, en la función *merge(...)* como se mencionó se hace un consumo lineal de tiempo n . Podemos concluir que su función de eficiencia es:

$$T(n) = 2T(n/2) + n$$

Para calcular su CC podemos hacer uso de un árbol de recursión, notamos se generan $\lg n$ niveles y en ellos, siempre habrán n elementos (*surgidos de la partición del arreglo*). Podemos regir entonces cómo el algoritmo *merge_sort* maneja una CC.T

$$\text{merge_sort} : T(n) \in \Theta(n \lg n)$$

3.1.2 Quick sort

En *quick_sort(...)* el principio es escoger un pivote (*preferiblemente random*) sobre el que, si tiene elementos a su izquierda de tamaño inferior o igual y elementos a su derecha de tamaño superior o igual, entonces este pivote está ordenado. Este proceso recursivamente subdivide en 2 listas hasta llegar al caso base (*0 a 1 elemento*) sobre el que se reconstruye la solución.

```
def quick_sort(A: list[int | float]) -> list[int | float]:
    if len(A) <= 1:
        return A
    pivot: int | float = A[0]
    less: list[int | float] = [x for x in A[1:] if x <= pivot]
    greater: list[int | float] = [y for y in A[1:] if y > pivot]
    return quick_sort(less) + [pivot] + quick_sort(greater)
```

Suponiendo el mejor escenario *quick_sort* tomará el pivote que corresponda el elemento mediano al arreglo, es así que se puede realizar un Árbol de recursión que tome una altura $\log n$, por cada nivel se tendrán n elementos por lo que así como en *merge_sort* su función

de eficiencia será $T(n) = n \lg n \in O(n \lg n)$.

No obstante en su peor escenario la partición se realiza al inicio de arreglo, con ello se generaría un árbol (*línea*) cual en cada nivel del Árbol de derivación tendrá $n, n-1, n-2, \dots, 1$ elementos, con ello su función de eficiencia se denotaría como

$$T(n) = \frac{n(n+1)}{2} \in O(n^2)$$

Así mismo su análisis de CC.S puede determinarse que haciendo uso de una pila, en el mejor escenario usará un tamaño de $\lg n$ y en el peor uno de n . (*El tamaño de la pila depende del tamaño del árbol*).

El caso medio tenemos: (El análisis es después que se ejecuta partición)

$$T(n)_1 = T(0) + T(n-1) + Cn$$

$$T(n)_2 = T(1) + T(n-2) + Cn$$

$$T(n)_3 = T(2) + T(n-3) + Cn$$

⋮

$$T(n)_{n/2} = T(n/2) + T(n/2) + Cn$$

⋮

$$T(n)_{n-2} = T(n-3) + T(2) + Cn$$

$$T(n)_{n-1} = T(n-2) + T(1) + Cn$$

$$T(n)_n = T(n-1) + T(0) + Cn$$

Si se supone que tenemos n casos según la posición del pivote.

Entonces podemos formular la siguiente función de eficiencia

$$T_{avg}(n) = \sum_{I \in D} T(I) \cdot R(I)$$

Como notamos es muy distinto a los algoritmos iterativos (acá se miran la cantidad de pasos). Entonces tenemos que calculando la función de eficiencia para todas las posibles entradas de datos camos a tener una ecuación distinta, sumamos todas las diferentes entradas de datos, por lo que:

$$T_A(n) = \left(\frac{1}{n}\right) \sum_{i=1}^n [T(i-1) + T(n-i)] + Cn$$

Tenemos que $\frac{1}{n}$ es la probabilidad de la entrada de datos.

La idea es resolver la ecuación, quedando como:

$$T_A(n) = \frac{1}{n} (2T(0) + 2T(1) + \dots + 2T(n-2) + 2T(n-1)) + Cn$$

$$T_A(n) = \frac{1}{n} (2T(0) + 2T(1) + \dots + 2T(n-2) + 2T(n-1)) + Cn$$

Es lineal no homogénea. En Quicksort los el mejor

Iniciamos que no sabemos nada, luego nos paramos en el pivote tras partición, no es que digamos "primera posición", es que dependiendo de cómo hagamos las cosas es que tras elegir lo mejor es que elija el pivote tras ubicarse quede en la mitad y con ello 2 particiones de tamaño $n/2$ (una cosa es decir el pivote está en a , otra es que el pivote queda en b), pero, lo peor es que quede en los extremos, donde el pivote genera un árbol desbalanceado

En el mejor escenario de un arreglo $arr = [1, 2, 3, 4, 5, 6, 7]$ es el pivote 4 sería lo mejor que podría pasarme Pero si escogemos a 1 entonces generamos 2 particiones, una con 0 elementos y otra con $n-1$

Entonces quien es el pivote? No lo sabemos, no sabemos qué tenemos, nos da lo mismo que sea cualquiera (pero lo mejor), pero el mejor es que toque 4 (ubicadito en la mitad). Si tenemos $[4, 2, 3, 1, 5, 6, 7]$

3.1.3 Heaps

Deben de aclararse la peor escenario, esto son siguientes definiciones para comprender un Heap (*montículo*):

Definition 3.1.2 — Árbol binario pleno.

Un árbol es pleno si dada una altura h tiene una cantidad de $2^{h+1} - 1$ nodos. Cada nivel no tiene espacio para inserción de nodos.

Definition 3.1.3 — Árbol binario completo.

1. Si el arreglo representativo no tiene elementos nulos entre elementos existentes.
2. Si el último nivel tiene sus elementos contiguos de izquierda a derecha (*considerados árboles binarios casi completos*).

Siempre mínimamente la altura será $h = \lg n$ puesto ha de llenar todo el nivel para pasar al otro.

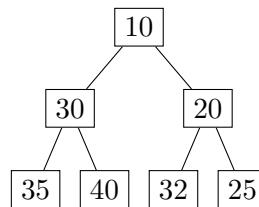
Entonces podemos definir un Heap como un árbol binario completo. Un heap cumple las siguientes propiedades:

- El elemento (*nodo*) de posición i tiene como nodos hijos a *left* y *right*.
- Si un nodo está en un índice i :
 - Su hijo *left* está en $2i$.
 - Su hijo *right* está en $2i + 1$.
 - Su padre *root* está en $\lfloor i/2 \rfloor$.

Existen 02 tipos de Heaps; **Max Heap** y **Min Heap**.

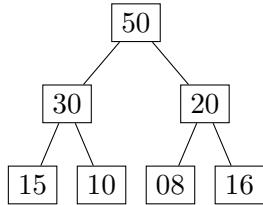
Min Heap

Téngase el arreglo $A_m = [10, 30, 20, 35, 40, 32, 25]$, el nodo posición i es menor o igual a sus hijos; $(A_m[i] \leq A_m[2i]) \vee (A_m[i] \leq A_m[2i + 1])$. Representable como



Max Heap

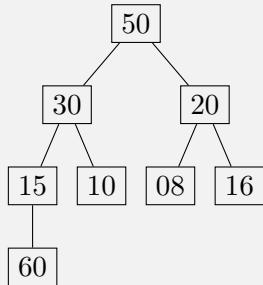
Téngase el arreglo $A_M = [50, 30, 20, 15, 10, 8, 16]$ se aprecia cómo cada padre tiene un valor superior o igual a sus hijos; $(A_M[i] \geq A_M[2i]) \vee (A_M[i] \geq A_M[2i + 1])$. Representable como



Theorem 3.1.1 — Añadir $\text{add}(x)$.

Implica dar al elemento x la posición última del arreglo y con ello tener el padre en posición $\lfloor i/2 \rfloor$. Adicionalmente para cumplir la propiedad **Max Heap** debe ordenarse.

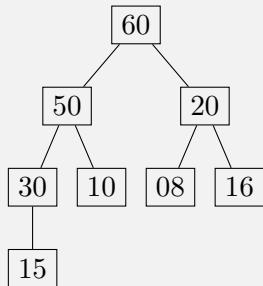
Supóngase buscamos $A.\text{add}(60)$, generaría un $A = [50, 30, 20, 15, 10, 8, 16, 60]$, si 60 está en posición 8, su i parente es 4 con $A[4] = 15$.



$$[i = 1 : 50, i = 2 : 30, i = 3 : 20, i = 4 : 15, i = 4 : 10, i = 6 : 8, i = 7 : 16, i = 8 : 60]$$

Notamos no cumple la propiedad Max Heap, por lo que debemos realizar una serie de cambios entre elementos para volverlo Max Heap.

Con ello realizamos el cambio $(60 \rightarrow 15) \rightarrow (60 \rightarrow 30) \rightarrow (60 \rightarrow 50)$, obteniendo



obtenemos $A = [60, 50, 20, 30, 10, 8, 16, 15]$. Este procedimiento puede costarnos entre $O(1)$ a $O(\lg n)$.

Theorem 3.1.2 — Optimizado a Heap.

Existen 02 procesos para optimizar una lista a Heap, mediante **Create Heap** o la **Heapify**. La diferencia crucial está en la magnitud de complejidad, mientras Create Heap tiene CC.T $O(n \lg n)$ (n : elementos, $\lg n$: Asumir se mueven a la raíz), mediante Heapify tiene CC.T $O(n)$ puesto escaneará cada elemento y si no cumple la propiedad realizará los cambios necesarios.

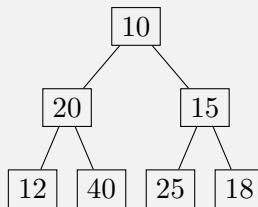
*Demuestra*o. Si queremos optimizar (Max/Min Heap) la lista $A = [10, 20, 15, 30, 40]$ en $B = [-, -, -, -, -]$ siguiendo el siguiente esquema (representable con árbol):

1. $B = [10, -, -, -, -]$
2. $B = [10, 20, -, -, -] \rightarrow [20, 10, -, -, -]$
3. $B = [20, 10, 15, -, -]$
4. $B = [20, 10, 15, 30, -] \rightarrow [30, 20, 15, 10, -]$
5. $B = [30, 20, 15, 10, 40] \rightarrow [40, 30, 15, 10, 20]$

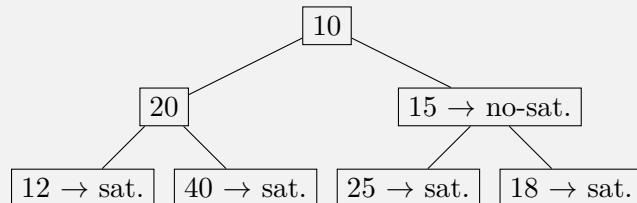
■

Ahora, si realizamos mediante **Heapify** tendremos el siguiente esquema:

*Demuestra*o. Téngase $A = [10, 20, 15, 12, 40, 25, 18]$ representable como heap

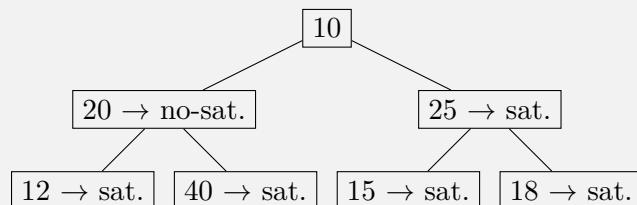


Para el análisis se hace desde el último a primer i , ahora no va de abajo hacia arriba el cambio, si no de arriba a abajo. Verificamos cuáles cumplen la propiedad de Max Heap.

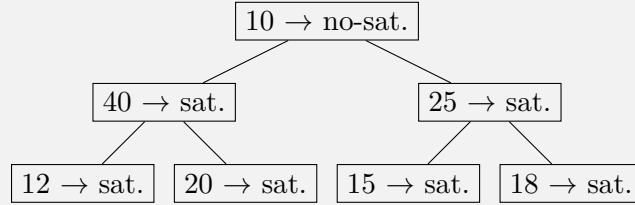


Claramente las hojas por invariante de inicialización cumplen con la propiedad Max Heap, al llegar a $A[3] = 15$ ya no satisface la propiedad, hemos de ordenar.

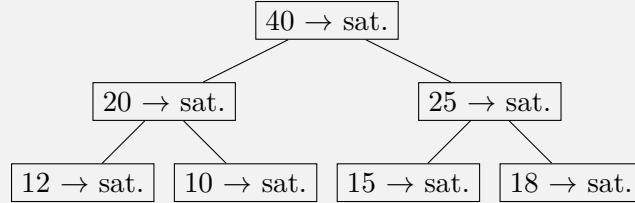
Entre sus hijos es $(A[6] > A[7]) = (25 > 18)$, y es $(A[6] \geq A[3]) = (25 > 15)$, queda



Determinamos $A[5] < A[4]$ por lo que hacemos cambio de $A[2] \wedge A[5]$, tal que



Finalmente notamos como $A[1] < A[2]$ y luego $(A[1] \rightarrow A[2]) < A[5]$ para tener



Con $A = [40, 20, 15, 12, 10, 25, 18]$

■

3.1.4 Heap Sort

Consiste en la eliminación total y almacenado de un Max Heap completo. Tras $A.drop()$ claramente se elimina el máximo elemento del Heap y se almacena iterativamente en la última posición (disponible) separada del arreglo cual representa el árbol.

Demostración. Si tenemos $A = [40, 30, 15, 10, 20]$ podemos realizar la siguiente trazabilidad:

1. $A.drop() : A = [20, 30, 15, 10, -] \rightarrow [30, 20, 15, 10; 40]$.
2. $A.drop() : A = [10, 20, 15, -; 40] \rightarrow [20, 10, 15; 30, 40]$.
3. $A.drop() : A = [15, 10, -; 30, 40] \rightarrow [15, 10; 20, 30, 40]$.
4. $A.drop() : A = [10, -; 20, 30, 40] \rightarrow [10; 15, 20, 30, 40]$.

Notamos la realización óptima del algoritmo **Heap Sort** se da mediante Heapify y su $drop()$ del árbol, obteniendo $A = [10, 15, 20, 30, 40]$. ■

3.1.5 Priority Queues

Son manejados mediante heaps Min o Max donde hay 02 tipos de prioridades; Directamente proporcional o inversamente proporcional al valor del elemento determina el orden para ser removido.



4. Algoritmos voraces

4.1 Introducción

La entrada son los candidatos, se trabaja por etapas (y se hace el mismo proceso), escoge candidatos y si sirve lo lleva al conjunto solución, es como seguir la ramita de un árbol.

Metodología simple, aplicada especialmente sobre optimización (*máximos y mínimos*). Se obtiene un subconjunto sln que satisface restricciones asociadas al problema de n entradas. Si sln satisface las restricciones decimos es 'prometedora'; Una sln prometedora que optimiza la función objetivo (FO) z es una sln 'óptima'.

Son una forma ágil de obtener una solución algorítmica.

Consiste en identificar en la entrada/s de datos quién puede servir para resolver el problema.

Definition 4.1.1 — Elementos. Contiene las etapas:

1. Conjunto de candidatos $\rightarrow n$ entradas.
2. Función de selección \rightarrow Candidatos idóneos.
3. Función de validación (*factibilidad*) \rightarrow Subconjunto prometedor (*Permite adición de candidatos*).
4. Una z a evaluar y optimizar.
5. Una función comprobante \rightarrow subconjunto es solución (óptima o no).

Digamos que se compra un terreno porque se ve muy vacano pero así si esté muy bien ubicado, el mejor costo, el mejor clima pero sobre arenas movedizas, la factibilidad es nula (aplicada sobre candidatos).

Suponiendo que es factible el candidato miramos la función objetivo para evaluar.

La manera como vemos el problema es determinante a su solución.

Trabaja sin pensar en el futuro, por etapas escoge el óptimo local suponiendo será global al problema.

Previo adicionar candidatos, comprueba sea prometedor añadirlo, si no, se descarta para siempre y validará el subconjunto prometedor.

```

algoritmo_avido(entrada: set = {x0..xn}) -> set[type(x)]
    # Declaracion #
    x: elemento
    solucion: set = {}
    encontrada: bool = False

    # Inicializacion #
    while not sea_vacio(entrada) and not encontrada:
        x = seleccionar_candidato(entrada)
        if es_prometedor(x, solucion)
            incluir(x, solucion)
            if es_solucion(solucion):
                encontrada = True

```

Lo importante de un algoritmo ávido **no es diseñarlo**, es **demonstrar** siempre consigue la solución óptima al problema en todos los casos o bien, un contraejemplo mostrando los casos cuales falle.

■ **Example 4.1 — Problema del cambio. Planteamiento:**

Definition 4.1.2 Un sistema monetario está formado por monedas con valores V_0, V_1, \dots, V_n .

¿Cómo descomponer cualquier cantidad M usando el menor número posible de monedas?

Se debe suponer que

1. Existe una moneda de valor unitario. Cada moneda mínimamente duplica su valor sucesivamente. Son ilimitadas las monedas de cada valor.
2. El sistema está compuesto por monedas con valores p^0, p^1, \dots, p^n ; $p > 1, n > 0$.

■

■ **Example 4.2 — Problema del salto de caballo. Planteamiento:**

Definition 4.1.3 Dado un tablero $n \times n$ de ajedrez y una casilla inicial se busca decidir si puede un caballo recorrer todas las casillas sin duplicaciones. No se requiere formar un ciclo.

1. Hallar las casillas solución.
2. Determinar el patrón general de solución a todo recorrido.

■

5. Backtracking

Han de tomarse una serie de decisiones pero no se dispone suficiente información para elegir, cada decisión conlleva a un nuevo subconjunto de soluciones y las secuencias de decisiones (1 o más) puede solventar nuestro problema.

Theorem 5.0.1 — Caracterización. Un algoritmo es resoluble vía Backtracking (BT) si.

1. Solución expresable como n-tupla (x_0, x_1, \dots, x_N) , cada x_i es seleccionado de un conjunto finito S_i .
 2. Es formulable en búsqueda de una tupla que optimice un criterio $P(x_0, x_1, \dots, x_n)$
- Podrían recorrerse todas las hojas del árbol de derivación pero esto sería ineficiente. El Backtracking mejora este proceso.

Cuando se asegura un nodo no alcanza la solución se poda la rama volviendo hacia atrás (Backtracking).

Difiere de algoritmos voráces una solución prometedora nunca se descarta, en BT elegir una solución no la hace irrevocable.

Los problemas no generan subproblemas independientes, no es aplicable la técnica DyV.

Definition 5.0.1 — Solución. $(x_0, x_1, \dots, x_N), x_i \in S_i$

Definition 5.0.2 — Espacio de soluciones. $S_i = \prod |S_i|$

Definition 5.0.3 — Solución parcial. $(x_0, x_1, \dots, x_k, ?, \dots, ?), x_i \in S_i$ Vector solución sin componentes completamente definidos.

Definition 5.0.4 — Función de poda/acotación. Permite identificar cuándo una solución parcial no conduce a una solución del problema.

Definition 5.0.5 — Restricciones asociadas. Hay 02 tipos

- **Explícitas:** Restringen x_i (*definen* $\prod S_i$).

■ **Example 5.1**

$$x_i \geq 0 \implies S_i \in \{R^+\}$$

$$x_i = [0, 1] \implies S_i \in \{0, 1\}$$

$$l_i \leq x_i \leq u_i \implies S_i \in \{a : l_i \leq a \leq u_i\}$$

■

- **Implicitas:** Determinan las tuplas que satisfacen el criterio $P(x_0, x_1, \dots, x_n)$ e indican si una solución parcial puede llevar a una global.

5.0.1 Las 08 reinas

Han de ubicarse 08 reinas en un tablero tal que no hayan ataques

5.0.2 Las N reinas

Es la generalización del problema de las 08 reinas. Dado un tablero $N \times N$ han de ubicarse N reinas tal que no hayan ataques. El S_i son las $N!$ permutaciones de la N-Tupla $(0, 1, \dots, N)$.

5.0.3 Suma de subconjuntos

Dados $n + 1$ números $w_i \in \mathbb{N}$ y un número M se buscan **todos** los subconjuntos de números w_i cuya suma sea M .

5.0.4 El viajero comerciante

Algoritmo vía Bactracking que determine todos los ciclos hamiltonianos de un grafo conexo $G = (V, E)$ con n vértices.

Definition 5.0.6 — Ciclo Hamiltoniano.

- Camino que recorre los n vértices de G , visita una vez cada V finalizando en el de partida.
- Vector solución (x_0, x_1, \dots, x_n) con x_i como el i-ésimo V visitado del ciclo.
- Sólo requiere determinar el conjunto posible de $V \in x_k$ tras elegidos x_0, x_1, \dots, x_{k-1} .

5.0.5 Coloreo de un grafo (m – colorability)

Sea un grafo G y $m \in \mathbb{Z}^+$. Busca determinarse si los nodos de G pueden colorearse de forma no hayan dos vértices adyacentes y tengan el mismo color, esto usando m colores.

5.0.6 Laberintos

Buscar la salida sabiendo su representación como grafo (*cada cruce requiere una decisión que conlleva a otros cruces/nodos*).

Theorem 5.0.2 — Generación de estados. Concebido un árbol de estados al problema, es resoluble con generación sistemática de sus estados y determinando *estados solución* y finalmente *estados respuesta*.

- **Nodo vivo:** Estado generado sin hijos. Puede ser ramificado.
- **Nodo muerto:** Estado generado, fue podado o generó todos sus hijos. Puede haber llegado a una solución o no genera soluciones factibles o mejores que la mejor actual.
- **Nodo E:** Nodo vivo con descendientes en generación (*Expansion node*).

El proceso consta de;

- Comenzar con un nodo raíz, desprenderá otros.
- Durante la generación se mantiene una estructura (*lista*) de vivos.
- Se eliminan los nodos mediante una función de acotación sin generar nodos hijos.

En función de cómo se explora el árbol existen 02 formas de generar los estados de un problema.

- **Backtracking:**

En profundidad, usa una pila (LIFO).

- **Branch&Bound:**

En anchura y suele ser iterativa.

Theorem 5.0.3 — Diferencias.

Bactracking:

- Nodo en curso: Tras generar su hijo será este el nuevo nodo en curso.
- Nodos vivos: Sólo son los que estén en el camino a la raíz del nodo en curso.

B&B:

- Nodo en curso: Genera todos los hijos del nodo en curso antes de decidir cuál va a ser el siguiente en curso (*estrategias LIFO, FIFO, PQ*).
- Nodos vivos: Pueden haber más nodos vivos.

Theorem 5.0.4 — Diferencias.

Bactracking:

- Nodo en curso: Tras generar su hijo será este el nuevo nodo en curso.
- Nodos vivos: Sólo son los que estén en el camino a la raíz del nodo en curso.

B&B:

- Nodo en curso: Genera todos los hijos del nodo en curso antes de decidir cuál va a ser el siguiente en curso (*estrategias LIFO, FIFO, PQ*).
- Nodos vivos: Pueden haber más nodos vivos.

Backtracking:

Generación en profundidad, en un curso R tras generarse de un Nodo-E un nodo C , se vuelve este un nodo-E. R será un Nodo-E cuando el subárbol C termine (sea explorado).

Branch&Bound:

Exploración en anchura a las soluciones, un nodo se mantiene como Nodo-E hasta se convierta en muerto. Las funciones de acotación detienen la exploración, con ello es fundamental tengan un diseño adecuado.

Usa una estructura que almacena los nodos vivos, puede ser usando la ley *FIFO* con colas o Priority Queues (*PQ*) al explorar el más prometedor, quizás pueda usarse pilas (estrategia *LIFO*).

¿Implementación?

Definición 5.0.7 — Implementación. Supóngase han de encontrarse todos los nodos de respuesta (*1 o más*).

El camino desde la raíz hasta un nodo es (x_0, x_1, \dots, x_i) .

El conjunto $T(x_0, x_1, \dots, x_{k-i})$ tiene todos los posibles valores x_k tales que $(x_0, x_1, \dots, x_{k-1}, x_k)$ es un camino válido hasta un estado del problema. Los posibles valores de x_k una vez escogido $(x_0, x_1, \dots, x_{k-1})$.

Las funciones de acotación B_k :

- Es $B_k(x_0, x_1, \dots, x_k)$ falso para un camino (x_0, x_1, \dots, x_k) si el camino no puede extenderse para alcanzar un nodo respuesta; Nos indica si (x_0, x_1, \dots, x_k) satisface las restricciones implícitas del problema.
- Los candidatos para la posición k del vector solución $X = (x_0, x_1, \dots, x_n)$ son aquellos valores generados por $T(x_0, x_1, \dots, x_{k-i})$ que satisfacen B_k .



6. Branch&Bound

El método de Ramificación y Poda : *Branch&Bound = B&P* tiene el siguiente esquema general:

Theorem 6.0.1 — Esquema general.

Selección: Selecciona el nodo vivo que será ramificado (dependerá de la estrategia).

Ramificación: se generan los hijos del nodo seleccionado (sólo tuplas prometedoras).

Cotas: Se calcula en cada nodo una cota del posible mejor valor alcanzable al mismo.

Poda: Se podan los nodos generados en la etapa anterior que no conducen a una solución mejor que la mejor conocida hasta ahora.

Theorem 6.0.2 — Gestión de nodos vivos.

Los nodos no podados forman parte del conjunto de vivos.

El nodo en curso se selecciona en función de la estrategia elegida.

El algoritmo finaliza cuando:

- Se agota el conjunto de nodos vivos (solución óptima).
- Se encuentra una solución que satisface un umbral de calidad.

Es efectivo B&B/RyP si posee una función de coste adecuada, es decir:

- Poda lo máximo posible.
- Su cálculo es eficiente (*low CC*).

Para podar es necesaria una solución (*cota general*) del problema.

Se puede utilizar un algoritmo voraz para tener una primera solución con la que empezar a comparar.

El esquema de la técnica se define con:

Definition 6.0.1 — generar_sln_vacia(). Una tupla vacía.

Definition 6.0.2 — sln_inicial(). Construye una solución vorazmente.

- Definition 6.0.3 — es_sln().** Indica si una tupla es una solución.
- Definition 6.0.4 — cota_inferior().** Estima una cota de la mejor solución posible ramificando el nodo (no tiene por qué ser factible).
- Definition 6.0.5 — complexiones().** Calcula el conjunto de posibles sucesores dada una tupla (*componentes disponibles*).
- Definition 6.0.6 — es_factible().** Determina si una tupla es factible (puede conducir a una solución factible).

```
def RyP():
    sln = generar_sln_vacia() # Tupla vacia, nodo del arbol de exploracion #
    sln_final = sln_inicial() # algoritmo voraz o equivalente #
    cota_superior = coste(sln_final)
    lst = cola_vacia() # Pila, Cola o Priority queue #
    encolar(sln, lst)
    while not es_cola_vacia(lst)
        sln = primero(lst)
        desencolar(lst)
        if es_solucion(sln)
            if coste(sln) < cota_superior # Para minimizacion #
                sln_final = sln
                cota_superior = coste(sln)
        else
            si cota_inferior(sln) < cota_superior
                for hijo in complexiones(sln)
                    if es_factible(hijo) and (cota_inferior(hijo) < cota_superior)
                        encolar(hijo, lst)
    return solucionFinal
```

■ **Example 6.1 — Problema de la mochila.** Vamos a trabajar sobre el problema de la mochila entera (0-1) con 1 repetición por elemento (objeto).

Acá la representación será un vector si toma o no el nodo. Otro será el beneficio máximo o valor ganado de la mochila y el otro es el peso generado en la mochila.

Tenemos la cota inferior, valor estimado y cota superior, el proceso de ramificación, expandir el árbol no se da necesariamente primero en profundidad, es cualquier nodo. Hay varias estructuras de datos, una en el backtracking es la lista de nodos vivos, son los nodos pendientes a explorar, cuando termino esta lista debe quedar vacía, también es fundamental saber contra qué vamos a podar, por lo que vamos a tener una cota global (el manejo de las est. datos y las cotas depende del tipo de problema de optimización que vayamos a manejar, una cosa es maximizar o minimizar), la forma de definir la lista de nodos vivos es fundamental porque nos dirá cual es el primero a revisar, fundamental en la estrategia, hay varios tipos de listas; Pilas, Colas y Colas de prioridades (digamos a un proceso darle más tiempo de atención o atenderlo con mayor frecuencia).

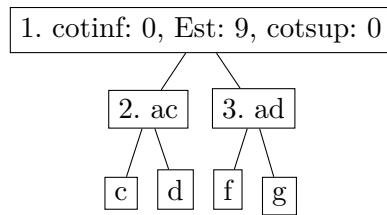
En el backtracking todo nodo sabe quién es su papa (por esto cuando vuelve se vuelve al padre, por eso se puede volver) (Diferencia con back).

Una cola de prioridades sale el que tenga la mayor prioridad es el que sale (en un problema de maximización el de mayor beneficio, de minimización el mayor coste), si todos tienen el mismo y se maneja la estructura LIFO, sale el primer elemento de la lista.

Por ejemplo podemos manejar una pila pero cuando hayan procesos con misma prioridad vamos a manejar una cola.

Cuando manejamos la cola la manejamos como cola de prioridades (primero que entra primero que sale, pero los empates se manejan tipo FIFO o tipo LIFO), si tengo una cola de prioridades y varios procesos con misma prioridad dependerá de que salen según mi manejo de prioridades.

Entonces, para el ejemplo hagamos con cola de prioridades y si hay empate entonces tipo FIFO.



1. La cota inferior, qué es? Cuando vimos algoritmos voráces (Prim, Dijkstra, etc..). La cota inferior es vacío. El estimado es aplicar también una estrategia voraz sobre los datos, hacemos una relación beneficio peso para saber el que más aporte. Entonces la estrategia voraz repetida varias veces dió 9, porque primero hizo $(2/1, 3/2, 4/3)$ según la relación valor/coste fue posible tomar $2+3+4=9$.

Lo más complicado de esta estrategia es que necesitas tener muchísimos datos, objetos, entonces, una forma de definir una cota superior rápida es sumar todos los valores de la mochila es sumar todos los valores, pero como el tope es tan alto no puede podar con facilidad, mientras que si toma una más real habrán procesos de poda. Hasta que no haya nacido el hijo no hay cota superior, entonces para la cota superior podemos hacer un caso de la mochila continua, por ejemplo como quedamos en 9 con peso 6 podríamos completar el peso! Podríamos meter algo que nos de el mejor valor proporcionalmente (el 1/6 de un objeto, qué porcentaje de X objeto tomo para completar el 1 que me falta (6 de 7).) (La otra forma es que como buscamos cota superior es irse por el mayor valor, es 5, coge 4, da 9 y el peso queda 7, fin. O tomar las otras combinaciones).

Las cotas determinan el éxito o fracaso de la técnica. Ahora sí, vamos con la cota global, esta en un problema de MAX será la cota inferior, eso significa que acá todo cambia, pero esta cota global cambia cada que encuentre una cota inferior más grande, es reemplazada por esta.

Vamos a manejarlo de forma binaria. El listado de hijos va:

$LH = [[0, 7, 9], []]$

El 9 sale de $4+5$ que definimos antes del prima.

Ahora miramos que la cota ■

6.0.1 Problema de la mochila 0-1

Dados n objetos y una mochila, cada objeto i tiene un peso $w_i > 0$ y un valor $v_i > 0$. La mochila puede llevar un peso que no sobrepase W .

Se desea llenar la mochila maximizando el valor de los objetos transportados, estos objetos deben ser \mathbb{Z} .

Formalmente se puede denotar como

$$\max \sum_{i=1}^n x_i v_i; \quad s.a : \sum_{i=1}^n x_i w_i \leq W$$

6.0.2 Asignación de tareas

Dadas n tareas y n personas, asignar a cada persona una tarea minimizando el coste de la asignación total. Se tiene una matriz de tarifas que determina el coste de asignar a cada persona una tarea. Si el agente $1 < i < n$ se le asigna la tarea $1 < j < n$ el coste será c_{ij} .



7. Programación dinámica

7.1 Introducción

En Programación dinámica (PD) las ideas principales que el problema debe cumplir son; Una sub-estructura óptima y sub-problemas superpuestos o hasta memoización.

7.1.1 Reglas

El potencial de la PD está en la recursividad.

Definition 7.1.1 — Serie de pasos.

1. Caracterización estructura óptima del problema (si es optimización).

$$SO \rightarrow SSO + SSO + \dots + SSO$$

Determinamos el principio de optimalidad (PO) (Como un problema se resuelve en base a subproblemas, el óptimo del problema grande lo resuelve los óptimos del problema pequeño).

2. Definir el problema de forma recursiva (Modelo recursivo).
3. Calcular la S.O. según el enfoque Bottom-Up (BU) u Top-Down (TD w/memoization).
4. Almacenar la solución en la sub-estructura óptima (*Tabla*).
5. Construir el algoritmo de solución basado en la estructura.

Tenemos series de decisiones d_0, d_1, \dots, d_n con d decisiones por cada decisión, tal que con Fuerza bruta (FB) obtenemos d^n decsisiones, una explosión combinatoria.

7.1.2 Sucesión de fibonacci

■ Example 7.1 — Sobre Fibonacci. Defínase como $F(n) = F(n - 1) + F(n - 2)$.

1. No es un problema de optimización (*no hay fibonacci's mejores o peores, no miramos (PO)*).
2. Está ya definido $F(n)$ recursivamente.

3. Calcularemos la solución con BU: Empezamos por los más pequeños $[F(0), F(1)]$ (*lo que conocemos*). Posteriormente se tomarán los más grandes.
4. Almacenamos la solución según las dimensiones del problema. Tenemos una (entrada|variable|parámetro|dimensión), así que un vector almacenará las soluciones.

$$\text{Vector : } v = [0, 1, \dots, n]$$

Para $v[0]$ almacenamos $F(0)$; $v[0] = F(0) = 0$.

Luego $v[1]$ almacenamos $F(1)$; $v[1] = F(1) = 1$.

Aplicando BU para $v[2]$ almacenamos $F(2)$; $v[2] \equiv F(2) = F(0) + F(1) \equiv v[0] + v[1]$.

Entonces $v[3] \equiv F(3) = F(2) + F(1) \equiv v[2] + v[1]$.

... hasta el n -ésimo término.

a <code>def fib_pd_bu(n: int) -> int:</code>	a) Llamado sin coste asociado.
b <code>tabla: list[int] = [0 for _ in</code>	b) $c_1 \times N$
<code>range(n + 1)]</code>	c) $c_2 \times 1$
c <code>tabla[0], tabla[1] = 0, 1</code>	d) $c_3 \times (N - 2)$
	e) $c_4 \times (N - 3)$
d <code>for i in range(2, n + 1):</code>	f) $c_5 \times 1$ (no se cuenta)
e <code>tabla[i] = tabla[i - 1] +</code>	
<code>tabla[i - 2]</code>	
f <code>return tabla[n]</code>	

Complejidad temporal (CC.T):

Análisis de eficiencia.

$$T(N) = c_1 N + c_2 + c_3(N - 2) + c_4(N - 3)$$

$$T(N) = aN + b$$

$$T(N) \in \Theta(N)$$

Complejidad espacial (CC.S):

Usando un vector tenemos Complejidad computacional (CC) de tamaño N .

$$S(N) \in \Theta(N)$$

■

Se ha realizado un ejercicio cuya resolución tomó el enfoque BU, queda de tarea el enfoque TD.

7.1.3 Problema del cambio

Consideremos el siguiente problema de cambio con las siguientes denominaciones:

$$M = 10$$

$$D = \{5, 2, 1, 10, 12\}$$

Nuestro objetivo es representar la cantidad M usando las denominaciones disponibles en D . Podemos abordar este problema utilizando el **principio de optimalidad**, que podemos ver como una especie de "ingeniería inversa" para comprender cómo se construye una solución. Suponemos que el problema puede dividirse en subproblemas más pequeños e independientes.

Los componentes clave del modelo recursivo son dos: la cantidad M y las denominaciones D . Así, el problema del cambio se puede expresar como:

$$\text{cambio}(M, D)$$

El objetivo es plantear cualquier cantidad que sea menor o igual a M , utilizando las denominaciones en D . El modelo busca construir soluciones iterativamente hasta alcanzar el valor deseado, utilizando variables que se actualizan continuamente. Representamos la relación de cambio con:

$$\text{cambio}(i, j) \quad \{0 \leq j \leq M\}, \{\}$$

Se sugiere construir una tabla que crezca en horizontal (con pocas filas, pero muchas columnas), donde j representará las cantidades y i las denominaciones. Al igual que en cualquier algoritmo de **programación dinámica (PD)**, el modelo parte de un problema elemental resuelto. En este caso, cuando $M = 0$, la solución es trivial: no necesitamos ninguna denominación. Así, nuestro problema base es $j = 0$ y, para cada denominación, expresamos los casos base y las situaciones imposibles.

Por ejemplo, si $j = 10$ y solo tenemos una denominación de 15, este sería un caso absurdo. Si encontramos este tipo de situaciones en un problema de minimización, asignamos un valor de $+\infty$, ya que cualquier solución sería mejor que esta.

En el caso de que tengamos más denominaciones (por ejemplo, $i > 1$) y el valor se pase de la cantidad a representar, aplicamos la relación recursiva:

$$\text{cambio}(i - 1, j) \implies (i > 1 \wedge j < D_i)$$

Si el valor de la denominación es menor o igual a la cantidad restante, procedemos con la siguiente denominación hasta completar la solución.

Ejemplo

Consideremos ahora el caso en que $M = 5$ y las denominaciones son $D = [1, 3, 5]$. Inicialmente, la tabla se ve así:

$T1$	0	1	2	3	4	5
$D_1 = 1$	0	0	0	0	0	0
$D_2 = 3$	0	0	0	0	0	0
$D_3 = 5$	0	0	0	0	0	0

Empezamos por el caso base y comenzamos a llenar la tabla con las condiciones del problema. Al aplicar la primera denominación $D_1 = 1$, la tabla queda de la siguiente manera:

$T1$	0	1	2	3	4	5
$D_1 = 1$	0	1	1	1	1	1
$D_2 = 3$	0	0	0	0	0	0
$D_3 = 5$	0	0	0	0	0	0

Continuamos aplicando las denominaciones siguientes hasta llenar completamente la tabla:

$T2$	1	2	3	4	5
$D_1 = 1$	1	1	1	1	1
$D_2 = 3$	0	0	1	1	1
$D_3 = 5$	0	0	0	0	1

Una vez completadas las dos tablas (una para los óptimos y otra para los caminos), podemos reconstruir la solución siguiendo la tabla de caminos ($T2$). La respuesta final se encuentra en la última celda de la tabla, pero la solución puede estar en cualquier posición, dependiendo de cómo se ha llenado la tabla. Siempre debemos seguir el rastro de cómo se llegó a la respuesta, verificando qué denominaciones se usaron en cada paso.

Al seguir esta metodología, podemos resolver eficientemente el problema del cambio utilizando programación dinámica, garantizando que encontramos la mejor combinación de denominaciones para representar M .

Ahora implementemos el algoritmo iterativo que llena las tablas de óptimos y caminos poco a poco:

```
def cambio(M: int, D: list[int]) -> list[int]:
    # Inicializamos las tablas
    tabla_opt = [[float('inf')]] * (M + 1) for _ in range(len(D) + 1)]

    # Caso base: cuando el valor es 0, no necesitamos monedas
    for i in range(len(D) + 1):
        tabla_opt[i][0] = 0

    # Llenamos las tablas iterativamente
    for i in range(1, len(D) + 1):
        for j in range(1, M + 1):
            if j >= D[i - 1]:
                tabla_opt[i][j] = min(tabla_opt[i - 1][j], 1 + tabla_opt[i][j - D[i - 1]])
            else:
                tabla_opt[i][j] = tabla_opt[i - 1][j]

    return tabla_opt
```

Este algoritmo llena la tabla de óptimos iterativamente. La tabla de caminos se llena al mismo tiempo. Si se usó una denominación, la tabla de caminos marcará 1, de lo contrario

será 0. En el último caso, cuando una denominación es mayor que el valor actual, no se toma y la tabla marca 0. Si es un caso donde la denominación puede ser tomada, se marca 1.

Algoritmo recursivo

Ahora implementemos la versión recursiva. Necesitaremos la tabla que ya tenemos, y en un algoritmo envolvente la llenaremos con valores iniciales, por ejemplo ‘None’ o números negativos, para indicar que aún no hemos resuelto ese subproblema.

La diferencia clave entre un algoritmo recursivo con **memoization** y uno recursivo simple es que, en el primero, verificamos si el subproblema ya ha sido resuelto previamente. Si ya lo resolvimos, simplemente devolvemos el resultado almacenado en la tabla. Si no lo hemos resuelto, procedemos a calcularlo.

```

def cambio_rec(M: int, D: list[int], tabla: list[list[int]]) -> int:
    # Verificamos si el subproblema ya fue resuelto
    if tabla[len(D)][M] != -1:
        return tabla[len(D)][M]

    # Caso base: cuando M es 0, no necesitamos monedas
    if M == 0:
        tabla[len(D)][M] = 0
        return 0

    # Inicializamos el valor mínimo
    min_val = float('inf')

    # Probamos todas las denominaciones
    for i in range(len(D)):
        if M >= D[i]:
            sub_res = cambio_rec(M - D[i], D, tabla)
            if sub_res != float('inf'):
                min_val = min(min_val, 1 + sub_res)

    # Guardamos el resultado en la tabla antes de retornar
    tabla[len(D)][M] = min_val
    return min_val

# Envolvemos la función para inicializar la tabla
def cambio_memo(M: int, D: list[int]) -> int:
    # Inicializamos la tabla con valores no calculados
    tabla = [[-1 for _ in range(M + 1)] for _ in range(len(D) + 1)]
    return cambio_rec(M, D, tabla)

```

Este algoritmo recursivo utiliza **memoization** para optimizar el cálculo, almacenando los resultados intermedios en la tabla ‘tabla’. De esta manera, evitamos recalcular los mismos subproblemas, reduciendo significativamente el tiempo de ejecución en comparación con la versión recursiva simple.

Explicación:

- **Caso base:** Si $M = 0$, no necesitamos monedas, así que devolvemos 0.
- **Memoization:** Si ya resolvimos el subproblema para una cantidad M , devolvemos el resultado almacenado en la tabla.

- **Recursividad:** Probamos todas las denominaciones y seleccionamos la que minimiza el número de monedas.

Este enfoque es fundamental para resolver el problema de manera eficiente, especialmente cuando trabajamos con grandes valores de M y varias denominaciones.

7.1.4 Assembly line scheduling

Una compañía produce en 02 líneas de montaje. Cada línea tiene n estaciones enumeradas con $j = 1, 2, \dots, n$. Denotamos la j -th estación en la línea i como S_{ij} , la línea 1 desempeña la misma que la j -th estación en la línea 2. Denotamos el tiempo requerido en la estación S_{ij} por a_{ij} . El tiempo para transferir un producto fuera de la línea de montaje i tras haber ido por la estación S_{ij} es t_{ij} . También hay un tiempo de entrada e_i para entrar a la línea i y uno de salida x_i por salir la línea i de montaje.

7.1.5 Subtleties

Una *Sutiliza* considera un grafo dirigido $G = (V, E)$ y vértices $u, v \in V$.

Para un **caminó corto sin peso** encontrar el camino de $u \rightarrow v$ con menos aristas. Tal camino debe ser simple, puesto remover un ciclo del camino produce uno con menos aristas.

7.1.6 Optimal matrix chain product

Dada una cadena de n matrices (A_1, A_2, \dots, A_n) , donde $i = 1, 2, \dots, n$. La matriz A_i tiene dimensiones $p_{i-1} \times p_i$. Parentizar el producto de A_1, A_2, \dots, A_n de tal manera que se minimice el número de multiplicaciones escalares.

Fact 7.1.1 El número de parentizaciones alternativas para una secuencia de n matrices se denota por $P(n)$.

Fact 7.1.2 $n \geq 2$ la división ocurre entre la matriz k -ésima y la $(k+1)$ donde $k = 1, 2, 3, \dots, n-1$

Fact 7.1.3 1 si $n = 1$

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k); \quad n \geq 2$$

Theorem 7.1.4 — Fórmula general. Sean d las dimensiones de las matrices.

$$m[i, j] = \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + d_{i-1}d_kd_j\}$$

Donde debemos hallar las posibles combinaciones en la relación de índices para la sub matriz generada.

■ **Example 7.2** Si tenemos que multiplicar las matrices de tamaño $m \times n$ por $n \times$ Principio de optimidad. El problema más grande se tiene que formar con los óptimos de los chiquitos.

■

Fact 7.1.5 Este problema grande que era $A \times B \times C$ se puede volver uno más chiquito.

Entonces para montar el modelo se tiene que puede fluctuar el tamaño inicial y el final, de forma que:

Lo primero que debemos expresar es el caso base. No requiere que se opere, no debemos hacer operaciones

Casos: Cuando no tiene una matriz (requiere o operaciones para llegar a una sola matriz)

$$0 \implies i = j$$

Imaginar que obviamente en el proceso debemos tener las dimensiones ($3 \times 4, 4 \times 2, 2 \times 3, 3 \times 5$) y suponer tenemos 4 matrices, por lo que como arreglo se vería como $[3, 4, 4, 2, 2, 3, 3, 5]$ pero para ahorrar tenemos $P = [3, 4, 2, 3, 5]$ y P será nuestra entrada de datos. Ahora, por gajes del oficio vamos a volver la primera columna la 0 (normalmente es la 1).

Entonces ahora vamos a volver el problema grande en un problema más chiquito $(A_1)A_2A_3A_4$ donde tenemos que A_1 es un problema, que sería desde la matriz 1 hasta la 1 y luego desde la 2 hasta la 4 y al final, júntelo. Pero si lo que tenemos es $(A_1A_2)A_3A_4$.

EL caso es entonces, multiplica desde la matriz i hasta la k y de la k+1 en adelante, entonces, a.

Los iteradores i, j no representan la cantidad de los objetos porque el orden de ideas es saber resolver para $n - 1$ o $n - 2$ y así se ponen a fluctuar entre valores, los subproblemas. El caso base es cuando tengo el problema tan delimitado que sólo tengo una matriz, si tengo una matriz requiere 0 operaciones, cuando $i \equiv j$

Enfoque bottom-up

Algoritmo top-down

7.1.7 Longest common subsequence LCS

Una cadena de ADN consiste de moléculas llamadas bases, hay cuatro; A: Adenina, G: Guanina, C: Citosina. Denotemos el conjunto $B = \{A, C, G, T\}$.

Dados el ADN de 02 individuos $S_1 = ACCGGTCGAGTGCAGCGGAAGGCCGCGAA$ y $S_2 = GTCTCGGAATGCCGTTGCTCTGTAAA$ se busca comparar cuán 'similar' son las cadenas (*medición de cuán similares son*).

La *similaridad* se define en varias formas, por ejemplo:

- Dos hebras son similares si uno es subcadena del otro.
- Dos hebras son similares si son ínfimos los cambios necesarios en uno para llegar al otro.

La segunda definición se puede formalizar como

Definition 7.1.2 — Formalmente. Una subsecuencia dada una secuencia es sólo la secuencia dada con 0 o más elementos dejados.

Dada una secuencia $X = [x_1, x_2, \dots, x_m]$ Otra secuencia $Z = [z_1, z_2, \dots, z_k]$ es subsecuencia de X si existe una secuencia estrictamente creciente. Son índices $i = [i_1, i_2, \dots, i_k]$ los índices de X tal que para todo $j = 1, 2, \dots, k$ tenemos $x_{i_j} = z_j$.

■ Example 7.3

Si tenemos $X = [A, B, C, B, D, A, B]$ entonces $Z = [B, C, D, B]$ es una subsecuencia de X con índices $[2, 3, 5, 7]$. ■

7.1.8 Optimal binary search trees

Suponga se diseñan un programa para traducir texto del inglés al español. Para cada aparición de cada palabra en inglés en el texto, debe buscarse su equivalente en español.

Una forma de realizar estas operaciones de búsqueda es construir un árbol de búsqueda binario con n palabras en inglés como claves y equivalentes en español como datos satelitales. Debido que buscaremos en el árbol cada palabra individual del texto, queremos que el tiempo total dedicado a la búsqueda sea lo más bajo posible.

Definition 7.1.3 — Formalmente.

Se nos da una secuencia $K = [k_1, k_2, \dots, k_n]$ de n claves distintas en orden, deseamos construir un árbol de búsqueda binario a partir de estas claves. Para cada clave k_i , tenemos una probabilidad p_i que se realice una búsqueda de k_i .

Algunas búsquedas pueden ser para valores que no están en K , por lo que también tenemos $n + 1$ *claves ficticias* d_0, d_1, \dots, d_n que representan valores que no están en K . En particular d_0 representa todos los valores menores que k_1 , d_n representa todos los valores entre k_i y k_{i+1} .

Para cada clave ficticia d_i , tenemos una probabilidad q_i que una búsqueda corresponda a d_i .

Theorem 7.1.6 — Fórmula general. Se puede resolver dinámicamente mediante la expresión

$$c[i, j] = \min\{c[i, k - 1] + c[k, j]\} + w(i, j)$$

Dónde w es el coste asociado a la frecuencia f de uso dado el valor en como clave y el índice como i, j .

$$w(i, j) = \sum_i^j f(i)$$

Ignorar índice 0, tomar desde 1.



8. Algoritmos Heurísticos

Acá se habla de todos los algoritmos aproximados.

8.1 Búsquedas

Recordemos el backtracking revisa el espacio de estados, primero viaja por profundidad (DSF), mientras que en B&B se hace según el nodo objetivo que queramos expandir (por lo que iremos en profundidad, anchura o prioridad). ¿Qué estructura de datos usamos en una búsqueda por anchura, quiero que me de el comportamiento? La respuesta es una cola.

Definition 8.1.1 — Heurísticas. Las técnicas heurísticas (NP-Compleitud) son problemas que son intratables con costes temporales sobre los polinomiales, hiperexponentiales, un buen ingeniero que sepa analizar los problemas y abalizar algoritmos debe entender cuando un problema es de orden NP-Completo, porque al entenderlo se da cuenta que no tiene sentido que trate buscar una solución buscando todas las posibilidades, lo mejor que puede hacer es buscar una solución aproximada al óptimo. Hay muchas formas de hacerlo, es ridículo hallar solución al problema, pero una que se aproxime a lo buena.

Las técnicas heurísticas son basadas en funciones de suposición, vi X entonces supongo Y, es suponer base al conocimiento previo que se tiene del problema

8.1.1 Búsqueda de coste uniforme

La búsqueda de coste uniforme se basa en encontrar la ruta de menor coste acumulado entre los nodos de un grafo. Tomemos como ejemplo un viaje de Arad a Bucarest. Partimos de Arad con tres opciones: Timisoara (a 118), Sibiu (a 140) y Zerind (a 75). En este algoritmo, siempre se expande el nodo con el menor coste acumulado.

Inicialmente, se elige Zerind debido a su bajo coste (75). Desde Zerind, la siguiente ciudad es Oradea, con un coste adicional de 71, lo que da un acumulado de 146. En paralelo, no expandimos Sibiu en este momento porque tiene un coste acumulado mayor (151), y procedemos con Timisoara, que tiene un coste de 118.

Así, se continúa expandiendo los nodos según el coste acumulado más bajo, hasta que se encuentre la ruta óptima. Un aspecto clave en este algoritmo es evitar ciclos revisitando nodos ya explorados.

8.1.2 Best First Search (Greedy Best First)

Entonces, ahora vamos a complejizar metiendo una heurística de distancia en línea recta. Acá nos interesa la distancia entre cualquier ciudad hacia bucarest, entonces nos preguntaremos todo el tiempo quién está más cerquita a bucarest. Tenemos las distancias de:

<i>Origen</i>	<i>Destino</i>	<i>Distancia</i>
<i>Arad</i>	<i>Bucarest</i>	366
<i>Craiova</i>	:	242
<i>Drobeta</i>	:	161
<i>Eforia</i>	:	176
<i>Fagaras</i>	:	77
<i>Giurgiu</i>	<i>Bucarest</i>	156
<i>Hlisova</i>	:	226
<i>Iasi</i>	:	116
<i>Lugov</i>	:	24
<i>Mehadia</i>	<i>Bucarest</i>	241
<i>Neamt</i>	:	234
<i>Oradea</i>	:	380
<i>Pitesti</i>	:	100
<i>Rimnicov</i>	<i>Bucarest</i>	153
<i>Sibiu</i>	:	253
<i>Timisoara</i>	:	329
<i>Urziceni</i>	:	80
<i>Vaslui</i>	:	199
<i>Zerind</i>	<i>Bucarest</i>	374

Estas son las distancias en linea recta desde cada ciudad a bucarest, empieza esta BFS (Best First Search) opera siempre con unas funciones de coste, esta función $f(n)$ podemos trabajarla a partir del coste real $g(n)$ que es como hicimos antes, pero ahora en nuestra búsqueda ávara se basará en la heurística de la tabla llámese $h(n)$. Entonces nótese como cada nodo partiendo a Sibiu, coste de 393, el menos costoso luego es hasta Fagaras con un coste total de 470.

■ **Example 8.1 — Explicación completa.** Entonces la siguiente que tomamos es Sibiu con 253,

miramos las siguientes de sibiu que son Fagaras con distancia 176, luego Rimnizu es 193 y finalmente a Oradea son 380 y Arud son 366, el más pequeño es Fagaras por lo que lo tomamos y nuevamente expandimos sus siguientes para finalmente llegar a Bucarest, ese es el camino con la búsqueda Greedy que nos da primero el mejor. ■

Entonces la respuesta real son las aristas con $140 + 99 + 211 = 450$

8.1.3 A* (A Star)

El algoritmo A* es una combinación de búsqueda de coste uniforme y búsqueda heurística. Funciona utilizando una función de evaluación $f(n) = g(n) + h(n)$, donde:

- $g(n)$ es el coste real del camino desde el nodo inicial hasta el nodo actual n .
- $h(n)$ es una estimación heurística del coste desde n hasta el objetivo (generalmente una distancia aproximada, como la distancia euclíadiana o Manhattan).

El objetivo de A* es minimizar $f(n)$ en cada paso, expandiendo el nodo con el menor valor de $f(n)$. A* es completo y óptimo cuando la heurística $h(n)$ es admisible, es decir, cuando no sobreestima el coste real al objetivo.

Este algoritmo es ampliamente utilizado en problemas de optimización de caminos, como la planificación de rutas, videojuegos y aplicaciones de inteligencia artificial.

8.1.4 Búsqueda Óptima

La búsqueda óptima se refiere a un tipo de algoritmo que garantiza encontrar la mejor solución posible (de menor coste) para un problema dado. Para que un algoritmo de búsqueda sea considerado óptimo, debe satisfacer dos condiciones:

1. **Completitud:** El algoritmo debe ser capaz de encontrar una solución si existe una.
2. **Optimalidad:** La solución encontrada debe ser la mejor posible en términos de coste (o la menos costosa entre las soluciones posibles).

Ejemplos de algoritmos de búsqueda óptima incluyen el propio A*, cuando se utiliza una heurística admisible, y la búsqueda de coste uniforme, que expande los nodos en función del coste acumulado más bajo. Estos algoritmos son fundamentales cuando se busca minimizar el coste en problemas de rutas o en escenarios donde las soluciones tienen diferentes valores de calidad.

8.1.5 Búsqueda limitada por profundidad

La búsqueda limitada por profundidad se utiliza para explorar solo hasta cierta profundidad en un árbol de decisiones, aplicando heurísticas para evitar recorrer caminos innecesarios.

Tomemos como ejemplo el juego del tres en raya. Inicialmente, tenemos una matriz de 3×3 . Utilizamos una heurística que cuenta cuántas líneas ganadoras potenciales tiene cada jugador. Supongamos que el estado actual del tablero es:

	<i>o</i>	<i>o</i>
<i>x</i>	<i>x</i>	
	<i>x</i>	<i>o</i>

Como este escenario lleva a un empate, la heurística nos sugiere no continuar explorando esta línea. En lugar de seguir hasta el final del juego, nos detenemos y aplicamos la heurística.

Aunque esta estrategia no garantiza encontrar la solución óptima, reduce el espacio de búsqueda.

Otro ejemplo es el juego del Klotsky, donde el objetivo es ordenar una matriz 3×3 del 1 al 8. Partimos del estado inicial y exploramos cada posibilidad, generando un árbol de búsqueda. En este caso, la heurística busca acercarnos al estado objetivo.

1	2	3
5	4	6
7	8	

La principal desventaja de las heurísticas es que deben diseñarse con cuidado, ya que no siempre se pueden aplicar de manera general.

Otra heurística es, imaginemos tenemos un mapa con los nodos $N = \{A, B, C, D, E\}$ con los arcos que tienen la distancia real para ir de $E = \{(A, B, 725), (A, C, 100), (A, E, 50), (A, B,), (A, F), (B, F,), (E, D,), (C, D,), (F, D,), (A, D,)\}$ Entonces notamos que este problema es de orden factorial, este mapa tiene unas coordenadas para saber la distancia en línea recta de $A \rightarrow B$ y así entre todos los puntos, entonces digamos que inicialmente nos movemos a los siguientes y el próximo que expanda será el más corto, esta es una heurística que le da la idea de qué tan cerquita está.

8.2 Teoría de juegos

Se busca una solución cuando hay un oponente que responde con su propia estrategia, juegos son elegidos porque el mundo puede describirse con pocas reglas, de fácil de representación.

Son de información perfecta cuando los jugadores perciben el mundo en forma completa (*e.g. chess*), se asemejan mas al mundo real que un problema de búsqueda simple puesto el oponente introduce la incertidumbre y se debe manejar el problema de *contingencia*.

8.2.1 Algoritmo Min-Max

Para juegos de 02 agentes llamados MAX y MIN, MAX juega primero y busca ganar. Para definir formalmente el juego se debe establecer:

- **Estado Inicial:** Posicion del tablero y una indicación de quien debe jugar.
- **Operadores:** Normalmente constituyen las reglas del juego.
- **Prueba terminal:** Como termina y quien gana.
- **Función de Utilidad:** Si no se puede evaluar todo el espacio de soluciones se deberá disponer de una función de utilidad que evalúa la bondad de cada jugada.

8.2.2 Algoritmo MINIMAX

1. Generar todo el árbol hasta alcanzar los nodos terminales (o profundidad especificada).
2. Obtener el valor de utilidad a cada nodo terminal.
3. Retraer los valores de utilidad nivel por nivel, desde terminales hasta el nodo inicial. Los nodos en que le toca jugar a MAX (*nivel MAX*) se le asigna como valor de utilidad el **máximo** valor de la función de utilidad de todos los nodos hijos (la mejor jugada de MAX). A los nodos en que juega MIN (*nivel MIN*) se le asigna el **mínimo** valor de la función de los nodos hijos (*la mejor jugada de MIN*).
4. Seleccionar la jugada de más alto valor de utilidad.

Eficiencia

La complejidad del algoritmo $MINIMAX \in O(b^m)$ sea el factor de ramificación b y la profundidad donde se encuentra el nodo terminal mas próximo m .

Salvo que se ponga un límite la búsqueda se hace primero en profundidad, se deben explorar todos los nodos terminales (*si no la decisión es imperfecta y no se puede garantizar el resultado*).

8.2.3 Poda $\alpha - \beta$

Idéntico al Minimax pero evita expandir todos los nodos cuando el valor retornado por un nodo hace que este sea imposible de seleccionar.

Se supone que se alcanzó un primer nodo terminal o un límite de profundidad (*la búsqueda siempre se hace primero en profundidad*), entonces retrae valores Min y Max parciales, los valores parciales Max se los llama Alfa y los valores Min se los llama Beta.

Los valores que adopta Alfa son un límite inferior para los de Max y los valores de Beta son un límite superior para la elección de Min.

1. Los valores Alfa de los nodos Max son crecientes.
2. Los valores Beta de los nodos Min son decrecientes.

Se pueden escribir las siguientes reglas que permiten discontinuar la búsqueda:

1. Se puede discontinuar la búsqueda debajo de un nodo Min que tiene un valor Beta menor o igual al valor Alfa de cualquiera de sus Ancestros. El valor final retornado por este nodo es su valor Beta. Este valor puede no ser el mismo que el del algoritmo MinMAX pero la selección de la mejor movida será idéntica.
2. Se puede discontinuar la búsqueda debajo de un nodo Max que tiene un valor Alfa mayor o igual que el valor Beta de cualquiera de sus Ancestros Min. El valor final retornado de este nodo Max puede ser su valor Alfa.

Durante la búsqueda los valores Alfa y Beta son calculados de la siguiente forma:

- a) El valor Alfa actual de un nodo Max es igual al mayor de los valores Beta finales (*valor retornado*) de sus sucesores.
- b) El valor Beta actual de un nodo Min es igual al menor de los valores Alfa finales (*valor retornado*) de sus sucesores.

Cuando se utiliza la regla uno se dice que se realizo una poda Alfa del árbol de búsqueda y cuando se usa la regla 2 se hizo una poda Beta.

Eficiencia

Para realizar la poda Alfa-Beta al menos parte del árbol de búsqueda debe generarse a la máxima profundidad. Los valores alfa y beta se deben basar en valores estáticos de nodos terminales, se usa entonces, algún tipo de búsqueda en profundidad.

El valor retornado al nodo raíz es idéntico al valor estático de algún nodo terminal, si se encuentra ese nodo terminal primero, entonces la poda es máxima.

Si la profundidad de la solución es d y el factor de ramificación es b , el número nodos terminales es $N = b^d$.

Si la generación de nodos es afortunadamente la mejor, primero el valor máximo para los nodos Max y mínimo para Min (*ni de casualidad*).

El número nodos terminales es $N = 2b^{d/2} - 1$ para d par.

El número nodos terminales es $N = b^{(d+1)/2} + b^{d(d-1)/2}$ para d impar.

La eficiencia de Alfa-Beta permite explorar árboles del doble de profundidad, si disponemos de un método de ordenamiento eficiente de valor de utilidad.