
UNIVERSIDAD DE GRANADA

# Algorítmica

*Temario de la asignatura*

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

# Licencia

# Algorítmica

Ismael Sallami Moreno

*https://elblogdeismael.github.io*

# Índice general

Capítulo 1

# La eficiencia de los algoritmos

Tema 1 del programa. Cómo se mide el coste de un algoritmo sin ejecutarlo, la notación asintótica y la resolución de las ecuaciones de recurrencia que aparecen al analizar algoritmos recursivos.

## 1.1 Por qué no se mide con un cronómetro

Medir el tiempo de ejecución da un número que depende de la máquina, del compilador, de la carga del sistema y de los datos concretos. Sirve para comparar dos implementaciones del mismo algoritmo en la misma máquina, y no sirve para comparar algoritmos.

El análisis teórico cuenta **operaciones elementales** en función del tamaño de la entrada. Da un resultado independiente de la máquina y permite predecir qué pasa al crecer el problema, que es la pregunta que importa.

### 1.1.1 Tamaño de la entrada y operación elemental

| Problema | Tamaño | Operación elemental |
|---|---|---|
| Ordenar un vector | número de elementos | comparación entre elementos |
| Buscar en un vector | número de elementos | comparación |
| Multiplicar matrices | dimensión | multiplicación escalar |
| Recorrer un grafo | vértices y aristas | visita de un vértice o arista |
| Comprobar si un número es primo | **número de dígitos**, no el valor | división |

La última fila no es un detalle: si el tamaño se midiera como el valor de $n$, el algoritmo que prueba divisores hasta $\sqrt{n}$ parecería polinómico, y en términos del número de dígitos es exponencial. La elección del tamaño cambia la conclusión.

### 1.1.2 Los tres casos

| Caso | Qué mide |
|---|---|
| Mejor | la entrada más favorable |
| Peor | la más desfavorable |
| Medio | la esperanza sobre una distribución de entradas |

El **peor caso** es el que se usa por omisión, porque es una garantía: el algoritmo nunca tardará más. El caso medio exige suponer una distribución de las entradas, y esa suposición suele ser discutible.

La ordenación rápida es el ejemplo que lo ilustra: $O(n \log n)$ en el caso medio y $O(n^2)$ en el peor, y aun así se usa porque el peor caso es raro con un pivote elegido bien. La ordenación por mezcla garantiza $O(n \log n)$ siempre, y sin embargo es más lenta en la práctica por su constante y por su uso de memoria. La notación asintótica no lo dice todo.

## 1.2 Notación asintótica

### 1.2.1 Las tres cotas

| Notación | Definición | Significado |
|---|---|---|
| $f \in O(g)$ | $\exists c, n_0 : f(n) \leq c\, g(n)$ para todo $n \geq n_0$ | cota superior |
| $f \in \Omega(g)$ | $\exists c, n_0 : f(n) \geq c\, g(n)$ para todo $n \geq n_0$ | cota inferior |
| $f \in \Theta(g)$ | $f \in O(g)$ y $f \in \Omega(g)$ | orden exacto |

Decir que un algoritmo es $O(n^2)$ afirma que **no tarda más** que un múltiplo de $n^2$. Es cierto y poco informativo si además es $O(n)$: la cota superior no obliga a ser ajustada. Cuando se conoce el orden exacto se escribe $\Theta$, y en la práctica se dice $O$ queriendo decir $\Theta$.

Las constantes y los términos de orden inferior se descartan:

$$3n^2 + 5n + 100 \in \Theta(n^2)$$

porque para $n$ grande el término cuadrático domina. Eso también avisa de su límite: **para $n$ pequeño las constantes deciden**, y un algoritmo $\Theta(n^2)$ con constante pequeña puede batir a uno $\Theta(n \log n)$ hasta cierto tamaño. Por eso las bibliotecas cambian a ordenación por inserción en los tramos cortos.

### 1.2.2 Jerarquía de órdenes

De menor a mayor crecimiento:

$$\Theta(1) \subset \Theta(\log n) \subset \Theta(n) \subset \Theta(n \log n) \subset \Theta(n^2) \subset \Theta(n^3) \subset \Theta(2^n) \subset \Theta(n!)$$

Lo que significa en tiempo real, suponiendo un microsegundo por operación:

| $n$ | $n$ | $n \log n$ | $n^2$ | $2^n$ |
|---:|---:|---:|---:|---:|
| 10 | 10 µs | 33 µs | 100 µs | 1 ms |
| 100 | 100 µs | 664 µs | 10 ms | $4 \cdot 10^{16}$ años |
| 1 000 | 1 ms | 10 ms | 1 s | — |
| 1 000 000 | 1 s | 20 s | 11,6 días | — |

La conclusión del tema está en esa tabla: **un algoritmo exponencial es inutilizable a partir de tamaños ridículos**, y ninguna máquina más rápida lo arregla. Duplicar la velocidad del ordenador permite resolver un problema exponencial con un elemento más.

### 1.2.3 Reglas de cálculo

| Regla | Enunciado |
|---|---|
| Suma | $\Theta(f) + \Theta(g) = \Theta(\text{máx}(f, g))$ |
| Producto | $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$ |
| Constantes | $\Theta(c \cdot f) = \Theta(f)$ |
| Instrucciones seguidas | se suman |
| Bucles anidados | se multiplican |

```
for (int i = 0; i < n; i++)          // n vueltas
    for (int j = 0; j < n; j++)      // n vueltas
        s += m[i][j];                // Theta(1)
```

Total: $\Theta(n^2)$.

```
for (int i = 1; i < n; i *= 2)       // log n vueltas
    for (int j = 0; j < n; j++)      // n vueltas
        s++;
```

Total: $\Theta(n \log n)$. La clave es que el primer bucle **multiplica** en vez de sumar, así que el número de vueltas es logarítmico.

Y un caso que engaña:

```
for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
        s++;
```

El interior no da $n$ vueltas sino $n - i$. La suma es $\sum_{i=0}^{n-1}(n-i) = n(n+1)/2$, es decir $\Theta(n^2)$. El resultado es el mismo orden, y la constante es la mitad.

## 1.3 Ecuaciones de recurrencia

Un algoritmo recursivo tiene un coste que se define en términos de sí mismo, y resolver esa recurrencia es obtener el orden.

### 1.3.1 Método de sustitución

Se expande la recurrencia hasta ver el patrón. Para la búsqueda binaria:

$$T(n) = T(n/2) + c, \qquad T(1) = c$$

$$T(n) = T(n/2) + c = T(n/4) + 2c = \cdots = T(n/2^k) + kc$$

El caso base se alcanza con $n/2^k = 1$, es decir $k = \log_2 n$, y queda

$$T(n) = c \log_2 n + c \in \Theta(\log n)$$

### 1.3.2 Árbol de recursión

Se dibuja el árbol de llamadas y se suma el coste por niveles. Para la ordenación por mezcla, $T(n) = 2T(n/2) + \Theta(n)$:

```
nivel 0:            n                       -> n
nivel 1:      n/2       n/2                 -> n
nivel 2:   n/4  n/4   n/4  n/4              -> n
   ...
nivel log n: 1 1 1 ... 1                    -> n
```

Cada nivel cuesta $\Theta(n)$ y hay $\log_2 n$ niveles, así que $T(n) \in \Theta(n \log n)$.

### 1.3.3 Teorema maestro

Para recurrencias de la forma

$$T(n) = a\,T(n/b) + f(n), \qquad a \geq 1,\ b > 1$$

se compara $f(n)$ con $n^{\log_b a}$:

| Caso | Condición | Resultado |
|---|---|---|
| 1 | $f(n) \in O(n^{\log_b a - \varepsilon})$ | $T(n) \in \Theta(n^{\log_b a})$ |
| 2 | $f(n) \in \Theta(n^{\log_b a})$ | $T(n) \in \Theta(n^{\log_b a} \log n)$ |
| 3 | $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ y se cumple la regularidad | $T(n) \in \Theta(f(n))$ |

La intuición: el caso 1 es que domina el trabajo de las hojas, el 3 que domina el de la raíz, y el 2 que todos los niveles cuestan lo mismo.

Aplicado:

| Recurrencia | $a$ | $b$ | $n^{\log_b a}$ | $f(n)$ | Caso | Resultado |
|---|---|---|---|---|---|---|
| $T(n) = 2T(n/2) + n$ | 2 | 2 | $n$ | $n$ | 2 | $\Theta(n \log n)$ |
| $T(n) = T(n/2) + 1$ | 1 | 2 | 1 | 1 | 2 | $\Theta(\log n)$ |
| $T(n) = 4T(n/2) + n$ | 4 | 2 | $n^2$ | $n$ | 1 | $\Theta(n^2)$ |
| $T(n) = 3T(n/2) + n^2$ | 3 | 2 | $n^{1,58}$ | $n^2$ | 3 | $\Theta(n^2)$ |
| $T(n) = 7T(n/2) + n^2$ | 7 | 2 | $n^{2,81}$ | $n^2$ | 1 | $\Theta(n^{2,81})$ |

La última fila es el algoritmo de Strassen para multiplicar matrices, y explica por qué mejora al método clásico de $\Theta(n^3)$.

El teorema **no cubre todos los casos**: si $f(n)$ cae entre dos casos sin cumplir ninguno —por ejemplo $f(n) = n \log n$ con $n^{\log_b a} = n$— hay que recurrir al árbol de recursión.

### 1.3.4 Recurrencias por sustracción

Para $T(n) = a\,T(n-b) + f(n)$ el comportamiento es distinto y mucho peor:

| Condición | Resultado |
|---|---|
| $a = 1$ | $\Theta(n \cdot f(n))$ |
| $a > 1$ | $\Theta(a^{n/b} \cdot f(n))$: **exponencial** |

De ahí que el Fibonacci recursivo ingenuo, con $T(n) = T(n-1) + T(n-2) + c$, sea exponencial: dos llamadas y el tamaño decrece restando. Dividir el problema es lo que da órdenes logarítmicos; restarle una constante, no.

## 1.4 Eficiencia en espacio

Además del tiempo se analiza la memoria. Un algoritmo **in situ** usa $\Theta(1)$ de memoria adicional.

| Algoritmo | Tiempo | Espacio adicional |
|---|---|---|
| Ordenación por inserción | $\Theta(n^2)$ | $\Theta(1)$ |
| Ordenación rápida | $\Theta(n \log n)$ medio | $\Theta(\log n)$ por la pila |
| Ordenación por mezcla | $\Theta(n \log n)$ | $\Theta(n)$ |
| Ordenación por montículo | $\Theta(n \log n)$ | $\Theta(1)$ |

La memoria de la pila de recursión cuenta, y a menudo se olvida: un algoritmo recursivo «sin memoria adicional» consume un marco por nivel de recursión.

Y hay un intercambio explícito entre los dos recursos: la programación dinámica del tema 5 gasta memoria para no repetir cálculos, y la memorización convierte un algoritmo exponencial en uno polinómico a cambio de una tabla.

## 1.5 Cotas inferiores del problema

Distinto del coste de un algoritmo: es lo que **ningún** algoritmo puede mejorar.

La ordenación por comparaciones tiene cota inferior $\Omega(n \log n)$, y la demostración es el árbol de decisión. Un algoritmo que solo compara elementos recorre un camino en un árbol binario cuyas hojas son las $n!$ permutaciones posibles; la altura mínima de ese árbol es $\log_2(n!) \in \Theta(n \log n)$.

La consecuencia es que la mezcla y el montículo son **óptimos** en su modelo, y que mejorarlos exige salir del modelo: la ordenación por conteo y la radix son lineales precisamente porque no comparan, sino que usan el valor de la clave como índice.

Saber que existe una cota inferior evita perder tiempo buscando algo imposible, y es la razón por la que las cotas se estudian junto a los algoritmos. El tratamiento de la eficiencia y de las recurrencias está en Brassard y Bratley, 1997 y en Cormen et al., 2022, y el planteamiento de la asignatura en Verdegay Galdeano, 2017.

Capítulo 2

# Algoritmos divide y vencerás

Tema 2 del programa. Partir un problema en subproblemas del mismo tipo, resolverlos por separado y combinar sus soluciones.

## 2.1 El esquema

```
funcion dyv(P):
    si P es suficientemente pequeño:
        devolver solucion_directa(P)
    dividir P en subproblemas P1, ..., Pk
    para cada Pi: Si = dyv(Pi)
    devolver combinar(S1, ..., Sk)
```

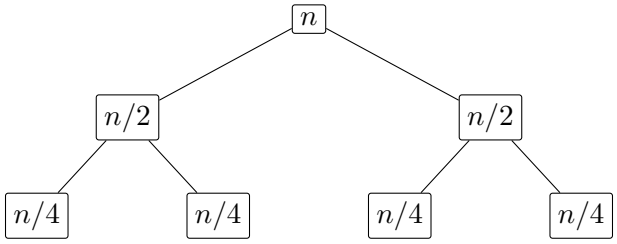
Tres piezas, y el coste sale de cómo se reparten:

| Pieza | Qué hace |
|---|---|
| Umbral y caso directo | resolver sin recursión los problemas pequeños |
| División | partir en subproblemas del mismo tipo y menor tamaño |
| Combinación | construir la solución a partir de las parciales |

Su coste responde a la recurrencia del tema 1:

$$T(n) = a\,T(n/b) + f(n)$$

con $a$ el número de subproblemas, $b$ el factor de reducción y $f(n)$ el coste de dividir y combinar.




Cuándo compensa: cuando el coste de dividir y combinar es bajo frente al de resolver el problema entero, y cuando los subproblemas son **independientes**. Si se solapan, resolverlos por separado repite trabajo, y eso es lo que la programación dinámica del tema 5 viene a corregir.

## 2.2 El umbral

La recursión no se lleva hasta un elemento: por debajo de cierto tamaño el coste de las llamadas supera al de resolver directamente.

```
void ordenar(int v[], int izq, int der) {
    if (der - izq < UMBRAL) {
        insercion(v, izq, der);      // directo
        return;
    }
    // ... dividir y combinar
}
```

El umbral se determina midiendo, y suele estar entre 10 y 50 elementos. Es la razón por la que las bibliotecas combinan la ordenación rápida con la inserción, y la primera comprobación cuando un divide y vencerás resulta más lento de lo esperado.

## 2.3 Búsqueda binaria

El caso más simple: un subproblema, y la mitad del anterior.

```
int buscar(const int v[], int izq, int der, int x) {
    if (izq > der) return -1;
    int centro = izq + (der - izq) / 2;
    if (v[centro] == x)     return centro;
    if (v[centro] < x)      return buscar(v, centro + 1, der, x);
    return buscar(v, izq, centro - 1, x);
}
```

$$T(n) = T(n/2) + \Theta(1) \implies T(n) \in \Theta(\log n)$$

Con $a = 1$ no hay ramificación: es una **reducción**, no una división. `izq + (der - izq) / 2` en vez de `(izq + der) / 2` evita el desbordamiento cuando los índices son grandes.

## 2.4 Ordenación por mezcla

```
void mezclar(int v[], int izq, int centro, int der) {
    vector<int> aux(der - izq + 1);
    int i = izq, j = centro + 1, k = 0;
    while (i <= centro && j <= der)
        aux[k++] = (v[i] <= v[j]) ? v[i++] : v[j++];
    while (i <= centro) aux[k++] = v[i++];
    while (j <= der)    aux[k++] = v[j++];
    for (int t = 0; t < k; t++) v[izq + t] = aux[t];
}

void ordenarMezcla(int v[], int izq, int der) {
    if (izq >= der) return;
    int centro = izq + (der - izq) / 2;
    ordenarMezcla(v, izq, centro);
```

```
    ordenarMezcla(v, centro + 1, der);
    mezclar(v, izq, centro, der);
}
```

$$T(n) = 2T(n/2) + \Theta(n) \implies T(n) \in \Theta(n \log n)$$

Aquí la división es trivial —partir por la mitad— y el trabajo está en la combinación. `<=` en la comparación de `mezclar` es lo que hace el algoritmo **estable**: ante dos elementos iguales se toma antes el de la izquierda. Con `<` dejaría de serlo, y ese detalle importa cuando se ordena por un segundo criterio.

Su coste en memoria es $\Theta(n)$ por el vector auxiliar, y es su desventaja frente a la ordenación rápida.

## 2.5 Ordenación rápida

Aquí el trabajo está en la división y la combinación es gratuita.

```
int particion(int v[], int izq, int der) {
    int pivote = v[der];
    int i = izq - 1;
    for (int j = izq; j < der; j++) {
        if (v[j] <= pivote) swap(v[++i], v[j]);
    }
    swap(v[i + 1], v[der]);
    return i + 1;
}

void ordenarRapida(int v[], int izq, int der) {
    if (izq >= der) return;
    int p = particion(v, izq, der);
    ordenarRapida(v, izq, p - 1);
    ordenarRapida(v, p + 1, der);
}
```

| Caso | Reparto | Recurrencia | Coste |
|---|---|---|---|
| Mejor | mitades iguales | $2T(n/2) + \Theta(n)$ | $\Theta(n \log n)$ |
| Medio | reparto razonable | — | $\Theta(n \log n)$ |
| Peor | uno vacío y otro de $n-1$ | $T(n-1) + \Theta(n)$ | $\Theta(n^2)$ |

El peor caso ocurre con el vector **ya ordenado** y el pivote al extremo, que es justo la entrada que más aparece en la práctica. Se evita eligiendo el pivote como mediana de tres, o al azar. Con pivote aleatorio el peor caso deja de depender de la entrada y pasa a depender de la suerte, lo que impide construir un caso adverso a propósito.

Frente a la mezcla: la rápida es in situ salvo la pila, tiene mejor constante y no garantiza el orden en el peor caso; la mezcla garantiza $\Theta(n \log n)$ y es estable, a cambio de $\Theta(n)$ de memoria.

## 2.6 Selección: el $k$-ésimo menor

La misma partición, y solo se recurre **a un lado**:

```
int seleccion(int v[], int izq, int der, int k) {
    if (izq == der) return v[izq];
    int p = particion(v, izq, der);
    int posicion = p - izq + 1;
    if (k == posicion) return v[p];
    if (k <  posicion) return seleccion(v, izq, p - 1, k);
    return seleccion(v, p + 1, der, k - posicion);
}
```

$$T(n) = T(n/2) + \Theta(n) \implies T(n) \in \Theta(n)$$

Encontrar la mediana **sin ordenar** cuesta lineal en el caso medio. Con el algoritmo de la mediana de medianas se consigue $\Theta(n)$ también en el peor caso, a costa de una constante grande.

## 2.7 Multiplicación rápida

Dos casos donde divide y vencerás bate al algoritmo directo.

**Karatsuba**, para enteros grandes. El método clásico multiplica dos números de $n$ dígitos en $\Theta(n^2)$. Partiéndolos por la mitad harían falta cuatro productos:

$$x = x_1 B + x_0, \quad y = y_1 B + y_0$$
$$xy = x_1 y_1 B^2 + (x_1 y_0 + x_0 y_1) B + x_0 y_0$$

Karatsuba observa que el término central se obtiene con **un** producto más:

$$x_1 y_0 + x_0 y_1 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0$$

Con tres productos en vez de cuatro:

$$T(n) = 3T(n/2) + \Theta(n) \implies T(n) \in \Theta(n^{\log_2 3}) = \Theta(n^{1{,}585})$$

**Strassen**, para matrices. El método clásico es $\Theta(n^3)$; partir en bloques da ocho productos, y Strassen los reduce a siete:

$$T(n) = 7T(n/2) + \Theta(n^2) \implies T(n) \in \Theta(n^{\log_2 7}) = \Theta(n^{2{,}807})$$

Los dos son el mismo truco: **cambiar multiplicaciones por sumas**, porque el número de subproblemas es lo que domina la recurrencia. Y los dos tienen el mismo problema práctico: su constante es grande, así que solo compensan a partir de tamaños considerables.

## 2.8 Cuándo no aplicar el esquema

| Situación | Qué pasa |
|---|---|
| Los subproblemas se solapan | se repite trabajo; usar programación dinámica |
| La combinación cuesta más que resolver | no se gana nada |
| El problema no se puede partir | no aplica |
| Se lleva la recursión hasta $n = 1$ | la sobrecarga domina; hay que poner umbral |

La primera fila es la que separa este tema del 5. Fibonacci escrito como divide y vencerás es exponencial porque `fib(n-1)` y `fib(n-2)` comparten casi todo su trabajo, y ningún ajuste del esquema lo arregla. El desarrollo de esta técnica y sus análisis está en Brassard y Bratley, 1997, Cormen et al., 2022 y Verdegay Galdeano, 2017.

Capítulo 3

# Algoritmos voraces

Tema 3 del programa. Construir la solución tomando en cada paso la opción que parece mejor en ese momento, sin reconsiderar.

## 3.1 El esquema

```
funcion voraz(C):            // C es el conjunto de candidatos
    S = vacio
    mientras C no este vacio y S no sea solucion:
        x = seleccionar(C)   // el mejor segun la funcion de seleccion
        C = C - {x}
        si factible(S + {x}):
            S = S + {x}
    si S es solucion: devolver S
    si no: no hay solucion
```

Cinco componentes, y describirlos es la forma de plantear cualquier problema de este tema:

| Componente | Qué es |
|---|---|
| Conjunto de candidatos | de dónde se elige |
| Función de selección | cuál se toma en cada paso |
| Función de factibilidad | si añadirlo puede llevar a una solución |
| Criterio de solución | cuándo se ha terminado |
| Función objetivo | qué se optimiza |

La característica que lo define: **una vez tomada una decisión no se revisa**. De ahí que sea rápido y de ahí que no siempre acierte.

## 3.2 Cuándo es correcto

Un algoritmo voraz da la solución óptima si el problema tiene dos propiedades:

- **Subestructura óptima.** La solución óptima contiene soluciones óptimas de sus subproblemas.
- **Propiedad de elección voraz.** Existe una solución óptima que contiene la primera elección voraz.

Comprobarlas exige demostración. Suponer que un voraz funciona porque parece razonable es el error característico del tema, y por eso los contraejemplos de más abajo son parte del contenido.

## 3.3 Cambio de monedas

Devolver una cantidad con el menor número de monedas, tomando siempre la de mayor valor que quepa.

```
vector<int> cambio(int cantidad, const vector<int> &valores) {
    vector<int> uso(valores.size(), 0);
    for (size_t i = 0; i < valores.size(); i++) {   // ordenados de mayor a menor
        uso[i] = cantidad / valores[i];
        cantidad %= valores[i];
    }
    return uso;   // valido solo si cantidad acabo en 0
}
```

Con el sistema de monedas del euro es óptimo. **Y no lo es en general**:

| Sistema | Cantidad | Voraz | Óptimo |
|---|---|---|---|
| 1, 5, 10, 20, 50 | 30 | $20 + 10$ (2 monedas) | 2 monedas |
| 1, 4, 6 | 8 | $6 + 1 + 1$ (3 monedas) | $4 + 4$ (2 monedas) |
| 1, 3, 4 | 6 | $4 + 1 + 1$ (3 monedas) | $3 + 3$ (2 monedas) |

Las dos últimas filas son el contraejemplo: el mismo algoritmo, un sistema de monedas distinto, y deja de ser óptimo. Para un sistema arbitrario hace falta programación dinámica.

## 3.4 Mochila fraccionaria

Objetos con peso $p_i$ y valor $v_i$, y una mochila de capacidad $W$. Se pueden partir los objetos.

Selección: por **densidad de valor** $v_i/p_i$ decreciente.

```
struct Objeto { double peso, valor; };

double mochilaFraccionaria(vector<Objeto> obj, double W) {
    sort(obj.begin(), obj.end(), [](const Objeto &a, const Objeto &b) {
        return a.valor / a.peso > b.valor / b.peso;
    });

    double total = 0.0;
    for (const Objeto &o : obj) {
        if (W <= 0) break;
        double tomar = min(o.peso, W);
        total += o.valor * (tomar / o.peso);
        W -= tomar;
    }
    return total;
}
```

Coste $\Theta(n \log n)$, dominado por la ordenación, y **es óptimo**: siempre existe una solución óptima que empieza tomando todo lo posible del objeto de mayor densidad.

**La versión 0-1, en la que los objetos no se pueden partir, no la resuelve el voraz.** Con capacidad 10 y objetos $(6, 30)$, $(5, 20)$ y $(5, 20)$, la densidad elige el primero y llega a 50; el óptimo

son los dos últimos, 40... y con $(6, 30)$, $(5, 25)$, $(5, 25)$ la densidad elige el primero, 30 + nada, mientras el óptimo es 50. Es el mismo problema con una restricción más, y cambia de técnica: va a programación dinámica.

## 3.5 Planificación de tareas

**Con plazos y penalizaciones.** Tareas de duración unitaria, cada una con un plazo y un beneficio si se cumple. Se ordenan por beneficio decreciente y cada una se coloca lo más tarde posible dentro de su plazo. Óptimo, y se demuestra con la teoría de matroides.

**Minimizar el tiempo medio de espera.** Con tareas de duración distinta en un solo servidor, ordenarlas por duración creciente minimiza la espera media. La demostración es un intercambio: si dos tareas consecutivas están en el orden contrario, intercambiarlas no empeora.

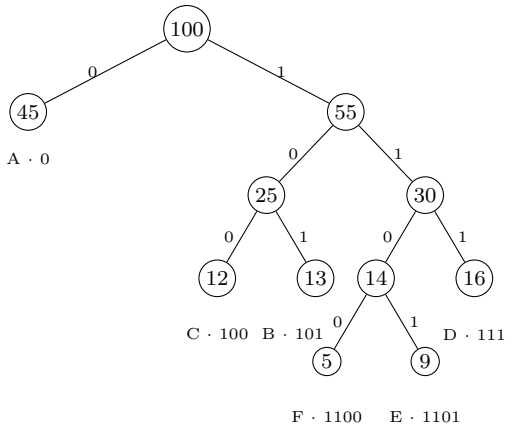
Es el mismo resultado que la planificación SJF de sistemas operativos, y aquí se demuestra en vez de afirmarse.

## 3.6 Códigos de Huffman

Compresión sin pérdida: los símbolos frecuentes reciben códigos cortos.

```
1. Cada simbolo es un arbol de un nodo, con su frecuencia.
2. Mientras quede mas de un arbol:
      tomar los dos de menor frecuencia
      unirlos bajo una raiz con la suma de sus frecuencias
3. El arbol resultante define el codigo: izquierda 0, derecha 1.
```

Con frecuencias A:45, B:13, C:12, D:16, E:9, F:5:




El símbolo más frecuente queda a profundidad 1 y los raros abajo, que es exactamente lo que minimiza la longitud media. Coste $\Theta(n \log n)$ con una cola de prioridad, y **es óptimo** entre los códigos de prefijo.

Un **código de prefijo** es aquel en el que ningún código es prefijo de otro, y eso es lo que permite descodificar sin separadores: al recorrer el árbol desde la raíz, cada hoja alcanzada es un símbolo completo.

## 3.7 Árbol de recubrimiento mínimo

Conectar todos los vértices de un grafo con el menor peso total.

**Kruskal.** Ordena las aristas por peso y añade la siguiente si no forma ciclo.

```
sort(aristas.begin(), aristas.end(), porPeso);
ConjuntosDisjuntos cd(n);
for (const Arista &a : aristas) {
    if (cd.buscar(a.u) != cd.buscar(a.v)) {
        cd.unir(a.u, a.v);
        arbol.push_back(a);
    }
}
```

Coste $\Theta(E \log E)$. La estructura de **conjuntos disjuntos** con unión por rango y compresión de caminos hace que comprobar el ciclo cueste prácticamente constante.

**Prim.** Parte de un vértice y añade en cada paso la arista de menor peso que conecta el árbol con un vértice de fuera. Coste $\Theta(E \log V)$ con montículo.

| | Kruskal | Prim |
|---|---|---|
| Crece | por aristas sueltas, varios fragmentos | desde un único árbol |
| Estructura clave | conjuntos disjuntos | cola de prioridad |
| Mejor con | grafos dispersos | grafos densos |

Los dos son óptimos, y la demostración es la misma: la **propiedad del corte**. Para cualquier partición de los vértices, la arista de menor peso que la cruza pertenece a algún árbol de recubrimiento mínimo.

## 3.8 Dijkstra

Camino más corto desde un origen, con pesos no negativos. Es voraz: en cada paso fija definitivamente el vértice no visitado con menor distancia.

```
vector<int> dijkstra(const Grafo &g, int origen) {
    vector<int> dist(g.n, INF);
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> cola;
    dist[origen] = 0;
    cola.push({0, origen});

    while (!cola.empty()) {
        auto [d, u] = cola.top(); cola.pop();
        if (d > dist[u]) continue;                  // entrada obsoleta
        for (const auto &[v, peso] : g.ady[u]) {
            if (dist[u] + peso < dist[v]) {
                dist[v] = dist[u] + peso;
                cola.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

Coste $\Theta((V + E) \log V)$.

**Falla con pesos negativos**, y la razón es exactamente la del tema: al fijar un vértice supone que no hay forma de llegar más barato, y una arista negativa posterior lo desmiente. La decisión no se revisa, así que el error se queda. Para pesos negativos está Bellman-Ford, que sí revisa: relaja todas las aristas $V - 1$ veces.

## 3.9 Resumen

| Problema | Voraz óptimo | Por qué |
|---|---|---|
| Mochila fraccionaria | sí | se puede partir el objeto |
| Mochila 0-1 | **no** | la elección puede desperdiciar capacidad |
| Cambio con euros | sí | por el sistema de monedas |
| Cambio general | **no** | contraejemplo con 1, 4, 6 |
| Huffman | sí | óptimo entre los códigos de prefijo |
| Kruskal y Prim | sí | propiedad del corte |
| Dijkstra con pesos $\geq 0$ | sí | ninguna arista puede abaratar lo ya fijado |
| Dijkstra con pesos negativos | **no** | la decisión fijada deja de valer |

La lección del tema: un voraz es rápido y hay que **demostrar** que acierta. Las tres filas en negrita son problemas donde el voraz da una respuesta razonable y equivocada, que es la peor combinación. El desarrollo de la técnica y sus demostraciones está en Brassard y Bratley, 1997, Cormen et al., 2022 y Kleinberg y Tardos, 2005.

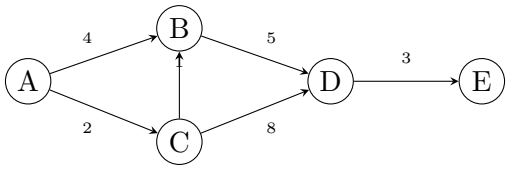
Capítulo 4

# Algoritmos para la exploración de grafos

Tema 4 del programa. Recorrer un grafo, y usar ese recorrido para explorar el espacio de soluciones de un problema: vuelta atrás y ramificación y poda.

## 4.1 Grafos

Un grafo $G = (V, E)$ son vértices y aristas. Puede ser dirigido o no, y ponderado o no.




### 4.1.1 Representación

| | Matriz de adyacencia | Listas de adyacencia |
|---|---|---|
| Espacio | $\Theta(V^2)$ | $\Theta(V + E)$ |
| ¿Existe la arista $(u, v)$? | $\Theta(1)$ | $\Theta(\deg u)$ |
| Recorrer los vecinos de $u$ | $\Theta(V)$ | $\Theta(\deg u)$ |
| Conviene con | grafos densos | grafos dispersos |

Los grafos reales son casi siempre dispersos, así que las listas son la opción por omisión. La diferencia no es menor: con un millón de vértices, la matriz pide un billón de posiciones.

## 4.2 Recorridos

### 4.2.1 En anchura

Visita por niveles, con una cola.

```
void anchura(const Grafo &g, int origen) {
    vector<bool> visitado(g.n, false);
    queue<int> q;
    visitado[origen] = true;
    q.push(origen);
```

```
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : g.ady[u]) {
            if (!visitado[v]) {
                visitado[v] = true;
                q.push(v);
            }
        }
    }
}
```

Coste $\Theta(V + E)$. Marcar al **encolar** y no al desencolar es lo que evita que un vértice entre varias veces en la cola.

Su propiedad útil: en un grafo **sin pesos** encuentra el camino con menos aristas, porque explora por niveles. Es Dijkstra sin cola de prioridad, y para ese caso es mejor que Dijkstra.

### 4.2.2 En profundidad

Avanza todo lo posible antes de retroceder, con una pila o con recursión.

```
void profundidad(const Grafo &g, int u, vector<bool> &visitado) {
    visitado[u] = true;
    for (int v : g.ady[u]) {
        if (!visitado[v]) profundidad(g, v, visitado);
    }
}
```

Coste $\Theta(V + E)$. Los tiempos de descubrimiento y de finalización que se pueden anotar en cada vértice clasifican las aristas —de árbol, de retroceso, de avance y cruzadas—, y de ahí salen tres resultados:

| Aplicación | Cómo |
|---|---|
| Detección de ciclos | existe una arista de retroceso |
| Orden topológico | el orden inverso al de finalización |
| Componentes fuertemente conexas | Tarjan o Kosaraju, dos recorridos |

| | Anchura | Profundidad |
|---|---|---|
| Estructura | cola | pila o recursión |
| Memoria | hasta $\Theta(V)$ por nivel | proporcional a la profundidad |
| Encuentra el camino más corto sin pesos | sí | no |
| Adecuada para | proximidad al origen | ciclos, topológico, conectividad |

## 4.3 Exploración del espacio de soluciones

Muchos problemas no tienen un algoritmo eficiente conocido, y hay que explorar las posibilidades. Ese espacio se modela como un **árbol de decisiones**: cada nivel fija una componente de la solución.

Recorrerlo entero es exponencial. Las dos técnicas del tema consisten en **no recorrerlo entero**.

## 4.4 Vuelta atrás

Recorrido en profundidad del árbol de soluciones, abandonando una rama en cuanto se sabe que no lleva a ninguna solución válida.

```
funcion vuelta_atras(solucion_parcial):
    si es_solucion_completa(solucion_parcial):
        registrar
        return
    para cada candidato c:
        si es_prometedor(solucion_parcial + c):
            vuelta_atras(solucion_parcial + c)
            deshacer c          // el paso que da nombre a la tecnica
```

`deshacer` es lo que distingue la técnica: el estado se modifica al descender y se restaura al volver, así que no hace falta copiar la solución parcial en cada llamada.

### 4.4.1 Las ocho reinas

Colocar ocho reinas en un tablero sin que se ataquen.

```cpp
bool seguro(const vector<int> &col, int fila, int c) {
    for (int f = 0; f < fila; f++) {
        if (col[f] == c) return false;                  // misma columna
        if (abs(col[f] - c) == abs(f - fila)) return false;  // diagonal
    }
    return true;
}

void reinas(vector<int> &col, int fila, int n, int &soluciones) {
    if (fila == n) { soluciones++; return; }
    for (int c = 0; c < n; c++) {
        if (seguro(col, fila, c)) {
            col[fila] = c;
            reinas(col, fila + 1, n, soluciones);
            // deshacer: col[fila] se sobrescribe en la vuelta siguiente
        }
    }
}
```

Representar el tablero como un vector con la columna de cada fila ya elimina por construcción las colocaciones con dos reinas en la misma fila. **Elegir bien la representación poda más que cualquier comprobación posterior**: el espacio pasa de $\binom{64}{8} \approx 4 \cdot 10^9$ a $8^8 = 16.777.216$, y la poda lo deja en unos 2.000 nodos.

### 4.4.2 Otros problemas del esquema

| Problema | Decisión por nivel | Poda |
|---|---|---|
| Sudoku | valor de una casilla | el valor ya está en fila, columna o bloque |
| Coloreado de grafos | color de un vértice | un vecino ya tiene ese color |

| Problema | Decisión por nivel | Poda |
|---|---|---|
| Suma de subconjuntos | incluir o no un elemento | la suma parcial ya se pasa |
| Laberinto | dirección | casilla visitada o muro |
| Ciclo hamiltoniano | siguiente vértice | ya visitado, o no adyacente |

## 4.5 Ramificación y poda

Para problemas de **optimización**, no de satisfacción. Se explora con una función que estima el mejor valor alcanzable desde cada nodo, y se descartan los nodos cuya estimación es peor que la mejor solución encontrada.

| | Vuelta atrás | Ramificación y poda |
|---|---|---|
| Objetivo | encontrar soluciones válidas | encontrar la óptima |
| Recorrido | en profundidad | por prioridad, con cota |
| Estructura | pila o recursión | cola de prioridad |
| Poda | por factibilidad | por factibilidad **y** por cota |

La **cota** es la pieza central:

- En un problema de **maximización**, la cota es superior: si la mejor estimación de un nodo no supera la mejor solución ya encontrada, la rama se descarta.
- En uno de **minimización**, al revés.

Una cota es válida si nunca es pesimista: en maximización debe ser mayor o igual que el óptimo alcanzable. Una cota optimista de más poda poco; una que no lo sea lo bastante **descarta el óptimo**, y ese es el error grave de la técnica.

### 4.5.1 Mochila 0-1

Que el tema 3 no podía resolver con un voraz.

```
cota(nodo) = valor acumulado
           + valor de los objetos completos que aún caben
           + fraccion del siguiente objeto      <- optimista a proposito
```

La cota usa la solución **fraccionaria**, que es siempre mayor o igual que la entera. Por eso es válida, y por eso es buena: está cerca del óptimo real, así que poda mucho.

```
struct Nodo { int nivel, valor, peso; double cota; };

int mochila01(const vector<Objeto> &obj, int W) {
    // objetos ordenados por densidad decreciente
    priority_queue<Nodo> cola;          // ordenada por cota
    int mejor = 0;
    cola.push(raiz);

    while (!cola.empty()) {
        Nodo n = cola.top(); cola.pop();
        if (n.cota <= mejor) continue;          // poda por cota
        // ramificar: incluir el objeto n.nivel, o no incluirlo
```

```
    }
    return mejor;
}
```

Explorar primero el nodo de mejor cota es lo que hace que aparezca pronto una solución buena, y una solución buena poda mucho más. De ahí que la estructura sea una cola de prioridad y no una pila.

### 4.5.2 El viajante de comercio

El problema clásico. El espacio son $(n-1)!$ rutas, así que enumerarlas es imposible más allá de unas quince ciudades.

Con ramificación y poda se resuelven instancias mucho mayores, usando como cota la suma de la arista más barata de cada ciudad no visitada. Y sigue siendo exponencial en el peor caso: la técnica reduce el árbol, no la complejidad.

## 4.6 Sobre la dificultad

El viajante, la mochila 0-1, el coloreado y el ciclo hamiltoniano son **NP-completos**: no se conoce algoritmo polinómico, y si apareciera uno para cualquiera de ellos los resolvería todos.

Saberlo cambia lo que se busca. Para un problema NP-completo las salidas razonables son tres, y ninguna es «encontrar el algoritmo eficiente»:

| Salida | Qué da |
|---|---|
| Exacta con poda | el óptimo, para tamaños moderados |
| Aproximación | solución con garantía de calidad, en tiempo polinómico |
| Heurística o metaheurística | solución buena sin garantía |

La primera es lo que este tema enseña. Las otras dos son materia de cursos posteriores. La exploración de grafos y estas técnicas están desarrolladas en Cormen et al., 2022, Horowitz et al., 2007 y Skiena, 2020.

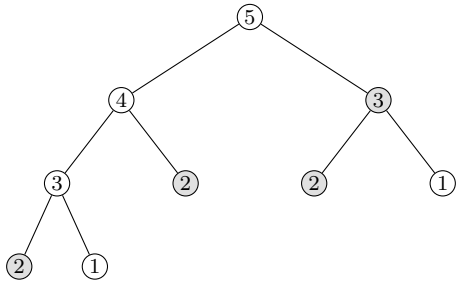
Capítulo 5

# Algoritmos basados en programación dinámica

Tema 5 del programa. Resolver cada subproblema una sola vez y guardar el resultado, para los casos en que divide y vencerás repite trabajo.

## 5.1 El problema que resuelve

Divide y vencerás supone que los subproblemas son independientes. Cuando **se solapan**, resolverlos por separado repite el mismo cálculo un número exponencial de veces.




En el árbol de `fib(5)`, los nodos sombreados se recalculan. Con $n$ grande el mismo valor se recalcula millones de veces, y de ahí el coste exponencial.

## 5.2 Las dos propiedades

Un problema admite programación dinámica si cumple:

| Propiedad | Qué significa |
|---|---|
| Subestructura óptima | la solución óptima se construye con soluciones óptimas de subproblemas |
| Subproblemas solapados | el mismo subproblema aparece muchas veces |

La primera la comparte con divide y vencerás y con los voraces. **La segunda es la que decide**: si no hay solapamiento, guardar resultados no aporta nada y divide y vencerás es mejor.

## 5.3 Los dos enfoques

### 5.3.1 Descendente, con memorización

Se conserva la recursión y se guarda lo ya calculado.

```cpp
long long fib(int n, vector<long long> &memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    return memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
}
```

De $\Theta(2^n)$ a $\Theta(n)$ con tres líneas. Cada subproblema se resuelve una vez y se consulta después.

### 5.3.2 Ascendente, con tabla

Se resuelven los subproblemas en orden creciente de tamaño, sin recursión.

```cpp
long long fib(int n) {
    if (n <= 1) return n;
    long long anterior = 0, actual = 1;
    for (int i = 2; i <= n; i++) {
        long long siguiente = anterior + actual;
        anterior = actual;
        actual = siguiente;
    }
    return actual;
}
```

| | Descendente | Ascendente |
|---|---|---|
| Estructura | recursión con tabla | bucles |
| Subproblemas resueltos | solo los que hacen falta | todos |
| Sobrecarga | llamadas y pila | ninguna |
| Ahorro de memoria | difícil | fácil: basta con guardar las filas necesarias |
| Escribirlo | inmediato desde la recursión | exige decidir el orden |

El ascendente permite el ahorro de memoria que se ve en el ejemplo: la tabla de $n$ posiciones se reduce a dos variables, porque cada valor solo depende de los dos anteriores. Es el mismo truco que aplica la mochila.

## 5.4 Cómo se plantea

Cuatro pasos, y son los mismos para cualquier problema del tema:

1. **Caracterizar la estructura** de la solución óptima.
2. **Definir recursivamente** el valor óptimo.
3. **Calcularlo** de forma ascendente o con memorización.
4. **Reconstruir** la solución a partir de la tabla, si hace falta.

El paso 4 se olvida a menudo: la tabla da el **valor** óptimo, no la solución que lo alcanza. Reconstruirla exige recorrer la tabla hacia atrás decidiendo qué opción produjo cada valor, o guardar esa decisión al llenarla.

## 5.5 Mochila 0-1

El problema que quedó pendiente en los temas 3 y 4.

$$M[i][w] = \begin{cases} 0 & \text{si } i = 0 \text{ o } w = 0 \\ M[i-1][w] & \text{si } p_i > w \\ \max\big(M[i-1][w],\ v_i + M[i-1][w-p_i]\big) & \text{en otro caso} \end{cases}$$

```
int mochila(const vector<int> &peso, const vector<int> &valor, int W) {
    int n = peso.size();
    vector<vector<int>> M(n + 1, vector<int>(W + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            M[i][w] = M[i - 1][w];                      // no incluirlo
            if (peso[i - 1] <= w) {
                M[i][w] = max(M[i][w],
                              valor[i - 1] + M[i - 1][w - peso[i - 1]]);
            }
        }
    }
    return M[n][W];
}
```

Coste $\Theta(nW)$ en tiempo y en memoria. Recorriendo `w` **de mayor a menor** se puede usar un solo vector en lugar de la matriz, y la memoria baja a $\Theta(W)$.

Una advertencia sobre ese coste: $\Theta(nW)$ **no es polinómico** en el tamaño de la entrada, porque $W$ se codifica en $\log W$ bits. Se llama pseudopolinómico, y es la razón por la que el problema sigue siendo NP-completo pese a tener este algoritmo.

### 5.5.1 Reconstruir qué objetos se toman

```
int w = W;
for (int i = n; i > 0; i--) {
    if (M[i][w] != M[i - 1][w]) {      // el objeto i se tomo
        seleccionados.push_back(i - 1);
        w -= peso[i - 1];
    }
}
```

## 5.6 Otros problemas del tema

### 5.6.1 Subsecuencia común más larga

$$L[i][j] = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ L[i-1][j-1] + 1 & \text{si } x_i = y_j \\ \max(L[i-1][j],\ L[i][j-1]) & \text{en otro caso} \end{cases}$$

Coste $\Theta(nm)$. Es la base de `diff` y de las medidas de similitud entre cadenas.

### 5.6.2 Distancia de edición

El menor número de inserciones, borrados y sustituciones para convertir una cadena en otra:

$$D[i][j] = \min\big(D[i-1][j]+1,\ D[i][j-1]+1,\ D[i-1][j-1]+[x_i \neq y_j]\big)$$

Coste $\Theta(nm)$. Se usa en correctores ortográficos y en alineamiento de secuencias biológicas.

### 5.6.3 Multiplicación de una cadena de matrices

Con qué paréntesis se multiplica una cadena de matrices para hacer menos operaciones escalares. El orden importa: multiplicar $A_{10\times100}$, $B_{100\times5}$ y $C_{5\times50}$ como $(AB)C$ cuesta 7.500 operaciones y como $A(BC)$, 75.000. **Diez veces**, sin cambiar el resultado.

$$M[i][j] = \min_{i\leq k<j}\big(M[i][k] + M[k+1][j] + d_{i-1}d_k d_j\big)$$

Coste $\Theta(n^3)$.

### 5.6.4 Floyd-Warshall

Caminos mínimos entre **todos** los pares de vértices:

```
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (d[i][k] + d[k][j] < d[i][j])
                d[i][j] = d[i][k] + d[k][j];
```

Coste $\Theta(V^3)$, y **admite pesos negativos**, que es lo que Dijkstra no podía. El bucle de `k` va por fuera: es el subproblema —«caminos que solo pasan por los primeros $k$ vértices»— y ponerlo dentro produce un resultado incorrecto sin que nada falle.

### 5.6.5 Otros

| Problema | Recurrencia sobre | Coste |
|---|---|---|
| Cambio de monedas general | cantidad restante | $\Theta(nC)$ |
| Subsecuencia creciente más larga | posición final | $\Theta(n^2)$, o $\Theta(n \log n)$ |
| Corte de varillas | longitud restante | $\Theta(n^2)$ |
| Bellman-Ford | número de aristas del camino | $\Theta(VE)$ |
| Distribución de tareas | conjunto ya asignado | exponencial en el número de tareas |

El cambio de monedas cierra el círculo con el tema 3: el voraz fallaba con el sistema 1, 4, 6, y esta recurrencia da el óptimo para cualquier sistema.

## 5.7 Comparación de las cuatro técnicas

| Técnica | Cuándo | Coste típico | Garantía |
|---|---|---|---|
| Divide y vencerás | subproblemas independientes | $\Theta(n \log n)$ | óptimo |
| Voraz | elección local demostrablemente óptima | $\Theta(n \log n)$ | óptimo si se demuestra |
| Programación dinámica | subproblemas solapados | polinómico o pseudopolinómico | óptimo |
| Vuelta atrás y poda | no hay algoritmo eficiente | exponencial en el peor caso | óptimo |

Y el mismo problema recorre las cuatro, que es la mejor forma de fijarlas:

| Problema | Técnica que lo resuelve | Por qué no las otras |
|---|---|---|
| Mochila fraccionaria | voraz | la densidad decide y se puede partir |
| Mochila 0-1 | programación dinámica | el voraz falla; los subproblemas se solapan |
| Mochila 0-1 con $W$ enorme | ramificación y poda | la tabla no cabe |
| Ordenar | divide y vencerás | los subproblemas son independientes |
| Cambio con euros | voraz | el sistema de monedas lo permite |
| Cambio general | programación dinámica | el voraz da contraejemplos |

La pregunta que guía la elección: **¿los subproblemas se solapan?** Si no, divide y vencerás. Si sí, programación dinámica. Y antes de las dos, comprobar si un voraz demostrable resuelve el problema, porque será más rápido. El desarrollo de esta técnica y sus problemas está en Cormen et al., 2022, Brassard y Bratley, 1997 y Roughgarden, 2022.

Capítulo 6

# Temario práctico

Las prácticas de la asignatura: analizar la eficiencia, y diseñar e implementar algoritmos con cada una de las técnicas estudiadas.

## 6.1 Análisis de la eficiencia

La práctica que da sentido al tema 1: comprobar que el orden teórico se corresponde con lo que la máquina hace.

### 6.1.1 Cómo se mide

```
#include <chrono>
using namespace std::chrono;

auto t0 = high_resolution_clock::now();
algoritmo(v, n);
auto t1 = high_resolution_clock::now();
double ms = duration<double, std::milli>(t1 - t0).count();
```

Reglas que la práctica exige:

- **Reloj monótono**, no la hora del día, que da saltos al ajustarse.
- **Descartar la primera medida**: carga las cachés y las páginas.
- **Repetir y quedarse con la mediana**, no con la media: un pico del sistema contamina la media.
- **Compilar con `-O2`** y anotarlo. Medir sin optimizar mide otra cosa.
- **Cuidado con el compilador**: si el resultado no se usa, la llamada puede desaparecer entera. Se evita imprimiendo o acumulando el resultado.
- **Anotar la máquina**: `lscpu`, compilador y opciones. Sin eso la medida no es reproducible.

### 6.1.2 Ajuste de la curva

Se mide para tamaños crecientes y se ajusta a la forma teórica:

| Orden esperado | Al duplicar $n$, el tiempo | Cómo se comprueba |
|---|---|---|
| $\Theta(\log n)$ | crece una constante | $t$ frente a $\log n$ es una recta |
| $\Theta(n)$ | se duplica | $t/n$ constante |
| $\Theta(n \log n)$ | algo más que el doble | $t/(n \log n)$ constante |
| $\Theta(n^2)$ | se cuadruplica | $t/n^2$ constante |
| $\Theta(n^3)$ | se multiplica por 8 | $t/n^3$ constante |

**Dividir el tiempo por la función teórica** es la comprobación que vale: si la hipótesis es correcta, el cociente tiende a una constante. Es más fiable que mirar la forma de la gráfica.

Y hay que medir hasta tamaños suficientes: por debajo de cierto punto las constantes dominan y la curva no dice nada.

### 6.1.3 El caso peor hay que construirlo

Un algoritmo con casos distintos no se mide con datos aleatorios y ya está:

| Algoritmo | Peor caso | Cómo se construye |
|---|---|---|
| Ordenación rápida con pivote al extremo | $\Theta(n^2)$ | vector ya ordenado |
| Ordenación por inserción | $\Theta(n^2)$ | vector en orden inverso |
| Búsqueda lineal | $\Theta(n)$ | el elemento no está |

Medir la ordenación rápida solo con vectores aleatorios da $\Theta(n \log n)$ y oculta que existe un caso cuadrático. La práctica pide medir los tres casos.

## 6.2 Prácticas por técnica

### 6.2.1 Divide y vencerás

- Implementar mezcla y rápida, y comparar tiempos con los tres tipos de entrada.
- Medir el efecto del **umbral**: por debajo de qué tamaño compensa la inserción. Sale una curva con mínimo, y ese mínimo es el valor que las bibliotecas usan.
- Selección del $k$-ésimo menor, comparada con ordenar y tomar el elemento $k$. La diferencia entre $\Theta(n)$ y $\Theta(n \log n)$ se ve con $n$ grande.

### 6.2.2 Voraces

- Cambio de monedas con el sistema del euro y con un sistema que produzca contraejemplo. **Comprobar que el voraz falla** es parte del ejercicio.
- Mochila fraccionaria frente a mochila 0-1 con el mismo voraz, midiendo cuánto se aleja del óptimo.
- Kruskal y Prim sobre el mismo grafo, con densidad creciente, para ver dónde se cruzan sus tiempos.
- Huffman: comprimir un texto y medir la razón de compresión frente a la entropía.

### 6.2.3 Exploración de grafos

- Anchura y profundidad sobre el mismo grafo, comparando el orden de visita y la memoria que consume cada uno.
- Ocho reinas, contando **nodos explorados** con poda y sin poda. La diferencia es de varios órdenes de magnitud, y es la medida que demuestra el valor de la poda.
- Ramificación y poda sobre la mochila 0-1 y sobre el viajante, midiendo nodos explorados según la calidad de la cota.

### 6.2.4 Programación dinámica

- Fibonacci en tres versiones: recursiva ingenua, con memorización e iterativa. Medir hasta donde la primera aguante, que suele ser $n = 40$.
- Mochila 0-1 con tabla, y con el ahorro a un solo vector.
- Subsecuencia común más larga y distancia de edición, con reconstrucción de la solución y no solo del valor.
- Floyd-Warshall frente a Dijkstra ejecutado desde cada vértice, para ver cuándo compensa cada uno.

## 6.3 El informe

Lo que se entrega con cada práctica:

1. Planteamiento: qué problema y qué técnica, con la justificación.
2. Diseño: la recurrencia o el esquema, y el análisis teórico del coste.
3. Implementación, con la complejidad de cada función documentada.
4. Casos de prueba, incluidos los límite y el peor caso construido a propósito.
5. Medidas con su dispersión, y el ajuste a la curva teórica.
6. **Comparación entre lo teórico y lo medido, con la explicación de las diferencias.**

El punto 6 es el que distingue una práctica hecha de una práctica entregada. Las diferencias tienen casi siempre una de estas causas:

| Diferencia observada | Causa habitual |
|---|---|
| Más lento de lo esperado con $n$ grande | fallos de caché: el conjunto de trabajo dejó de caber |
| Más rápido de lo esperado | el compilador eliminó código, o los datos eran favorables |
| Escalones en la curva | límites de los niveles de caché |
| Mucha dispersión entre repeticiones | carga del sistema, o escalado de frecuencia |

## 6.4 Cómo se compila y se comprueba

```
g++ -Wall -Wextra -std=c++17 -O2 -o programa programa.cpp
./programa

# Durante el desarrollo, sin optimizar y con comprobaciones
g++ -Wall -Wextra -std=c++17 -g -fsanitize=address,undefined -o dbg programa.cpp
./dbg
valgrind --leak-check=full ./dbg
```

Dos versiones a propósito: la optimizada para medir y la instrumentada para comprobar que el programa es correcto. Medir con los desinfectantes activados no tiene sentido, y comprobar con `-O2` esconde errores de memoria.

Los guiones y problemas de estas prácticas siguen el planteamiento de Verdegay Galdeano, 2017 y Brassard y Bratley, 1997; las implementaciones de referencia están en Sedgewick, 2001 y Skiena, 2020.

# Bibliografía


Brassard, G., & Bratley, P. (1997). *Fundamentos de Algoritmia*. Prentice Hall.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4.ª ed.). MIT Press.

Horowitz, E., Sahni, S., & Rajasekaran, S. (2007). *Computer Algorithms*. Computer Science Press.

Kleinberg, J., & Tardos, É. (2005). *Algorithm Design*. Addison-Wesley.

Roughgarden, T. (2022). *Algorithms Illuminated: Omnibus Edition*. Soundlikeyourself Publishing.

Sedgewick, R. (2001). *Algorithms in C* (3.ª ed.). Addison-Wesley.

Skiena, S. S. (2020). *The Algorithm Design Manual* (3.ª ed.). Springer.

Verdegay Galdeano, J. L. (2017). *Lecciones de Algorítmica*. Editorial Técnica AVICAM.