18/19

ALGORITMOS

UNIVERSIDADE DA CORUÑA

FIC

CONTENIDO

ALGORITMOS

TEMA 1: INTRODUCCIÓN AL ANÁLISIS DE ALGORITMOS, NOTACIONES ASINTÓTICAS Y CÁLCULO DE LOS TIEMPOS DE EJECUCIÓN

1. ANÁLISIS DE ALGORTIMOS

1.1 Análisis de la eficiencia de los algoritmos

- **Objetivo**: Predecir el comportamiento del algoritmo ⇒ aspectos cuantitativos: tiempo de ejecución, cantidad de memoria.

- Disponer de una medida de su eficiencia:
  - "teórica"
  - no exacta: aproximación suficiente para comparar, clasificar
    ⇒ acotar T(n): tiempo de ejecución:
    n = tamaño del problema (a veces, de la entrada)
    $n \to \infty$: comportamiento asintótico
    ⇒ T(n) = O(f(n))
    f(n): una cota superior de T(n) suficientemente ajustada
    f(n) crece más deprisa que T(n)

- **Aproximación?**
  - Ignorar factores constantes:
    20 multiplicaciones por iteración → 1 operación por iteración
    ¿cuántas iteraciones? → iteraciones en función de n
  - Ignorar términos de orden inferior: n + cte → n

- **Ejemplo 1**:
  2 algoritmos (A1 y A2) para un mismo problema A
  - algoritmo A1: 100n pasos → un recorrido de la entrada
    T(n) = O(n) : algoritmo lineal
  - algoritmo A2: $2n^2 + 50$ pasos → n recorridos de la entrada
    $T(n) = O(n^2)$: algoritmo cuadrático
    ⇒ A1 lineal y A2 cuadrático:
  - Comparar: A2 "más lento" que A1,
    aunque con $n \leq 49$ sea más rápido
    ⇒ **A1 es mejor**
  - Clasificar: lineales, cuadráticos...
    Tasas de crecimiento características:
    $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), \ldots O(2^n), \ldots$

1.2 Notaciones Asintóticas

- **Objetivo**: Establecer un orden relativo entre las funciones, comparando sus tasas de crecimiento.

- **La notación** O:
  $T(n), f(n) : Z+ \rightarrow R+$
  Definición: $T(n) = O(f(n))$
  si ∃ constantes $c > 0$ y $n_0 > 0$: $T(n) \leq c * f(n) \forall n \geq n_0$
  n_0: umbral
  $T(n)\ es\ O(f(n)), T(n) \in O(f(n))$
  "la tasa de crecimiento de $T(n)$ ≤ que la de $f(n)$"
  → $f(n)$ es una cota superior de $T(n)$
- **Ejemplo**: ¿$5n^2 + 15 = O(n^2)$?
  $< c, n_0 >=< 6,4 >$ en la definición: $5n^2 + 15 \leq 6n^2 \forall n \geq 4$;
  ∃ infinitos $< c, n_0 >$ que satisfacen la desigualdad.

1.2.1 La notación O

- **Observación**:

Según la definición, T(n) podría estar muy por debajo:
¿$5n^2$ + 15 = O(n^3)?
< c, n_0 >=< 1,6 > en la definición: $5n^2$ + 15 ≤ $1n^3$ ∀n ≥ 6
pero es más preciso decir = O(n^2) ≡ ajustar cotas
⇒ **Para el análisis de algoritmos, usar las aproximaciones**:
$5n^2 + 4n \rightarrow O(n^2)$
$\log_2 n \rightarrow O(logn)$
$13 \rightarrow O(1)$

- **Observación**:
  La notación O también se usa en expresiones como $3n^2$+ O(n)

- **Ejemplo 3**:
  ¿Como se consigue una mejora más drástica, mejorando la eficiencia del algoritmo o mejorando el ordenador?

- **Ejemplo 4**: Ordenar 100.000 enteros aleatorios:
  * 17 s en un 386 + Quicksort
  * 17 min en un procesador 100 veces más rápido + Burbuja

Reglas prácticas para trabajar con la O:

- **Definición**: f(n) es **monótona creciente**
  si n1 ≥ n2 ⇒ f(n1) ≥ f(n2)
- **Teorema**: ∀c > 0,a > 1,f(n) monótona creciente:
  $(f(n))^c = O(a^{f(n)})$

≡ “Una función exponencial (ej: n^2) crece más rápido que una función polinómica (ej: n^2)”

$$\begin{cases} n^c = O(a^n) \\ (\log_a n)^c = O(a^{\log_a n}) \end{cases}$$
$$\rightarrow (\log n)^k = O(n)\ \forall \text{k cte.}$$

≡ “n crece más rápido que cualquier potencia de logaritmo”
≡ “los logaritmos crecen muy lentamente”

- **Suma y multiplicación**:
  T1(n) = O(f(n))∧T2(n) = O(g(n)) ⇒

$$\begin{cases} T_1(n) + T_2(n) = O\big(f(n) + g(n)\big) = \max(O\left(f(n), O\big(g(n)\big)\right) \\ T_1(n) * T_2(n) = O\big(f(n) * g(n)\big) \end{cases}$$

  Aplicación: $\begin{cases} (1) Secuencia:\ 2n^2 = O(n^2) \wedge 10n = O(n) \Rightarrow 2n^2 + 10n = O(n^2) \\ (2) Bucles \end{cases}$

  Observación: **No** extender la regla: ni resta, ni división ← relación ≤ en la definición de la O
  ... suficientes para ordenar la mayoría de las funciones.

1.2.2 Otras notaciones asintóticas

1. T(n), f(n) : Z+ → R+, **Definición**: O

2. **Definición**: T(n) = Ω(f(n))
   si ∃ constantes c y n_0: T(n) ≥ cf(n) ∀n ≥ n_0
   f(n): **cota inferior** de T(n) ≡ trabajo mínimo del algoritmo
   Ejemplo: 3n^2 = Ω(n^2): cota inferior más ajustada...
   pero 3n^2 = O(n^2) también! (O ∧Ω)

3. **Definición** : T(n) = Θ(f(n))
   si ∃ constantes $c1, c2\ y\ n_0$: $c1f(n) \leq T(n) \leq c2f(n)\ \forall n \geq n_0$
   f(n): **cota exacta** de T(n), del orden exacto
   Ejemplo: 5n$\log_2 n$ − 10 = Θ((nlogn):

$$\begin{cases} (1)\ demostrar\ O \rightarrow < c, n_0 > \\ (2)\ demostrar\ \Omega \rightarrow < c', n'_0 > \end{cases}$$

4. **Definición**: T(n) = o(f(n))
   si ∀ constante $C > 0, \exists n_0 > 0: T(n) < Cf(n)\ \forall n \geq n_0$∨
   $\equiv O \wedge \neg \Theta \equiv O \wedge \neg \Omega$
   f(n): **cota estrictamente superior** de T(n): $\lim_{n \to \infty} \frac{T(n)}{f(n)} = 0$

$T(n)$	tiempo$_1$ 1000 pasos/s	tiempo$_2$ 2000 pasos/s	tiempo$_3$ 4000 pasos/s	tiempo$_4$ 8000 pasos/s
$log_2 n$	0,010	0,005	0,003	0,001
n	1	0,5	0,25	0,125
$nlog_2 n$	10	5	2,5	1,25
$n^{1,5}$	32	16	8	4
n^2	1.000	500	250	125
n^3	1.000.000	500.000	250.000	125.000
$1,1^n$	10^{39}	10^{39}	10^{38}	10^{38}

Tabla: Tiempos de ejecución (en s) para 7 algoritmos de distinta complejidad (n=1000).

Ejemplos: $\frac{n}{\log_2 n} = o(n)$ $\frac{n}{10} \neq o(n)$

5. **Definición**: T(n) = ω(f(n))
   si ∀ constante C > 0, $\exists n_0 > 0$: T(n) > Cf(n) ∀n ≥ n_0 ↔ f(n) = o(T(n)) →
   f(n): **cota estrictamente inferior** de T(n)

6. **Notación OO** [Manber]: T(n) = OO(f(n)) si es O(f(n)) pero con constantes demasiado grandes para casos prácticos
   Ref: Ejemplo 2 (p. 4): B1 = O(n^2), B2 = OO($n^{1,8}$)

Reglas prácticas (Cont.):

- T(n) = $a_0 + a_1 n + a_2 n^2 + \cdots + a_k n^k$ ⇒ T(n) = Θ(n^k) (polinomio de grado k)
- **Teorema**: ∀c > 0,a > 1,f(n) monótona creciente:

$$\big(\mathrm{f(n)}\big)^{\mathrm{n}} = o(a^{f(n)})$$

≡ "Una función exponencial **crece más rápido** que una función polinómica"
→ no llegan a igualarse

1.3 Cálculo de los tiempos de ejecución

- Calcular O para T(n) ≡ numero de "pasos" → f(n)? ¿paso?
- **Modelo de computación**:
  - ordenador secuencial
  - instrucción ↔ paso (no hay instrucciones complejas)
  - entradas: tipo único ("entero") ´ → sec(n)
  - memoria infinita + "todo está en memoria"
- Alternativas: Un paso es...
  - **Operación elemental**:
    - Operación cuyo tiempo de ejecución está acotado superiormente por una constante que solo depende de la implementación →= O(1)
  - **Operación principal** [Manber]:
    - Operación representativa del trabajo del algoritmo: El número de operaciones principales que se ejecutan debe ser proporcional al número total de operaciones (verificarlo!).
    - **Ejemplo**: la comparación en un algoritmo de ordenación
- La hipótesis de la op. principal supone una aproximación mayor!
- En general, **usaremos la hipótesis de la operación elemental**.
- En cualquier caso, se ignora: lenguaje de programación, procesador, sistema operativo, carga... ⇒ Solo se considera el algoritmo, el tamaño del problema, ...
- **Debilidades**:
  - operaciones de coste diferente ("todo en memoria" ⇒ lectura en disco = asignación) → contar separadamente según tipo de instrucción y luego ponderar ≡ factores ≡ dependiente de la implementación ⇒ costoso y generalmente inútil
  - faltas de página ignoradas
  - etc.

  → Aproximación

Análisis de casos:

- Análisis de casos:

Consideramos distintas funciones para T(n):

$\begin{cases} T_{mejor}(n) \\ T_{medio}(n) \\ T_{peor}(n) \end{cases}$ ← en general, la más utilizada

← representativa, más complicada de obtener

$$T_{mejor}(n) \leq T_{medio}(n) \leq T_{peor}(n)$$

- ¿El tiempo de respuesta es crítico?
  → Sistemas de Tiempo Real

Ordenación por inserción:

```
procedimiento Ordenación por Inserción (var T[1..n])
      para i:=2 hasta n hacer x:=T[i];
            j:=i-1;
            mientras j>0 y T[j]>x hacer
                  T[j+1]:=T[j];
                  j:=j-1;
            fin mientras;
            T[j+1]:=x
      fin para
fin procedimiento
```

3	1	4	1	2	9	5	6	5	3
1	3	4	1	2	9	5	6	5	3
1	3	**4**	1	2	9	5	6	5	3
1	**1**	3	4	2	9	5	6	5	3
1	1	**2**	3	4	9	5	6	5	3
1	1	2	3	4	**9**	5	6	5	3
1	1	2	3	4	**5**	9	6	5	3
1	1	2	3	4	5	**6**	9	5	3
1	1	2	3	4	5	**5**	6	9	3
1	1	2	3	**3**	4	5	5	6	9

Análisis de los casos: Ordenación por inserción

- **Peor caso** → "insertar siempre en la primera posición"
  ≡ entrada en orden inverso
  ⇒ el bucle interno se ejecuta 1 vez en la primera iteración, 2 veces en la segunda, . . . , n−1 veces en la última:
  ⇒ $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ iteraciones del bucle interno

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

  ⇒ T(n) = n(n−1) 2 c1 + (n−1)c2 +c3 : polinomio de grado 2
  ⇒ $\boxed{T(n) = \Theta(n^2)}$

- **Mejor caso** → "no insertar nunca" ≡ entrada ordenada
  ⇒ el bucle interno no se ejecuta
  ⇒ T(n) = (n−1)c1 +c2 : polinomio de grado 1
  ⇒ $\boxed{T(n) = \Theta(n)}$

⇒ T(n) depende también del estado inicial de la entrada

Ordenación por selección:

```
procedimiento Ordenación por Selección (var T[1..n])
      para i:=1 hasta n-1
            minj:=i;
            minx:=T[i];
            para j:=i+1 hasta n hacer
                  si T[j] < minx entonces
                        minj := j;
                        minx := T[j];
                  fin si
            fin para
            T[minj] := T[i];
            T[i] := minx;
      fin para
fin procedimiento
```

3	1	4	1	2	9	5	6	5	3
1	3	4	1	2	9	5	6	5	3
1	**1**	4	3	2	9	5	6	5	3
1	1	**2**	3	4	9	5	6	5	3
1	1	2	**3**	4	9	5	6	5	3
1	1	2	3	**3**	9	5	6	5	4
1	1	2	3	3	**4**	5	6	5	9
1	1	2	3	3	4	**5**	6	5	9
1	1	2	3	3	4	5	**5**	6	9
1	1	2	3	3	4	5	5	**6**	9

Análisis de los casos: Ordenación por selección

- T(n) = Θ(n^2) sea cual sea el orden inicial (ejercicio) ↔ la comparación interna se ejecuta las mismas veces
  Empíricamente: T(n) no fluctúa más del 15%

algoritmo	mínimo	máximo
Inserción	0,004	5,461
Selección	4,717	5,174

Tabla: Tiempos (en segundos) obtenidos para n = 4000

- **Comparación**:

algoritmo	peor caso	caso medio	mejor caso
Inserción	Θ(n^2)	Θ(n^2)	**Θ($\boldsymbol{n}$)**
Selección	Θ(n^2)	Θ(n^2)	Θ(n^2)
Quicksort	O(n^2)	**O($\boldsymbol{nlogn}$)**	**O($\boldsymbol{nlogn}$)**

Análisis de los casos: exponenciación

- Potencia1: $x^n = x * x * \dots * x$ (bucle, n veces x)
  Operación principal: multiplicación
  ¿Número de multiplicaciones? f_1(n) = n−1 ⇒ T(n) = Θ(n)

- Potencia2 (recursivo):

$$x^n = \begin{cases} x^{\lfloor \frac{n}{2} \rfloor} * x^{\lceil \frac{n}{2} \rceil} & si\ n\ par \\ x^{\lfloor \frac{n}{2} \rfloor} * x^{\lceil \frac{n}{2} \rceil} * x & si\ x\ impar \end{cases}$$

  ¿Número de multiplicaciones? ¿f_2(n)?
  Cálculo de f_2(n)

$$\begin{cases} min: n\ par\ en\ cada\ llamada \rightarrow n = 2^k, k \in Z^+ \leftrightarrow \text{mejor caso} \\ max: n\ impar\ en\ cada\ llamada \rightarrow n = 2^k - 1, k \in Z^+ \leftrightarrow \text{peor caso} \end{cases}$$

  - Mejor caso: $f_2(2^k) = \begin{cases} 0 & si\ k = 1 \\ f_2(2^{k-1}) + 1 & si\ k > 1\ (1) \end{cases}$

$k = 0 \rightarrow f_2(1) = 0$
$1 \quad 2 \quad 1$
$2 \quad 4 \quad 2$
$3 \quad 8 \quad 3$
$\dots$

⇒ Hipótesis de inducción: $\boxed{f_2(2^\alpha) = \alpha}$: 0 ≤ α ≤ k −1

Paso inductivo:
(1)→ $f_2(2^k) = f_2(2^{k-1}) + 1 = (k-1) + 1 = k$

  - Peor caso: $f_2(2^k - 1) = \begin{cases} 0 & si\ k = 1 \\ f_2(2^{k-1} - 1) + 2 & si\ k > 1\ (2) \end{cases}$

→ relaciones de recurrencia

$k=$		$\rightarrow f_2($		$)=$	
	1		1		0
	2		3		2
	3		7		4
	4		15		6
	5		31		8
	6		63		10
	...				

⇒ Hipótesis de inducción:
$\boxed{f_2(2^{\alpha}-1)=:2(\alpha-1):}$ 1 ≤ α ≤ k −1

Paso inductivo:
(2)→ $f_2(2^k-1)=f_2(2^{k-1}-1)+2=2(k-1-1)+2=2(k-1)$

- n = 2^k (mejor caso):
  $f_2(2^k)$ = k para k ≥ 0
  → $f_2(n)=\log_2 n$ para n = 2^k y k ≥ 0 (ya que $\log_2 2^k$ = k)
  ⇒$\boxed{f_2(n)=\Omega(\log n)}$

- n = 2 k −1 (peor caso):
  $f_2(2^k-1)$ = 2(k-1) para k ≥ 1
  → $f_2(n)=2[\log_2(n+1)-1]$ para n = 2^k-1 y k ≥ 1
  ⇒$\boxed{f_2(n)=O(\log n)}$

- ⇒ $f_2(n)=\Theta(\log n)$
  Modelo de computación: operación principal = multiplicación
  ⇒$\boxed{T(n)=\Theta(\log n)}$

mejor caso ↔ Ω
peor caso ↔ O

Reglas para calcular O

1. operación elemental = 1 ↔ Modelo de Computación

2. **secuencia**: $S_1=O(f_1(n))\wedge S_2=O(f_2(n))\Rightarrow \boxed{S_1;S_2}=O(f_1(n)+f_2(n))=O(\max(f_1(n),f_2(n)))$
   También con Θ

3. **condición**: $B=O(f_B(n))\wedge S_1=O(f_1(n))\wedge S_2=O(f_2(n))\Rightarrow$
   si B **entonces** S_1 **sino** $S_2=O(\max(f_B(n),f_1(n),f_2(n)))$

   - Si $f_1(n)\neq f_2(n)\ y\ \max(f_1(n),f_2(n))>f_B(n)$ ↔ **Peor caso**
   - ¿Caso medio?
     → f(n): promedio de f_1 y f_2 ponderado con las frecuencias de cada rama
     → $O(\max(f_B(n),f(n)))$

4. **iteración**: $B;S=O(f_{B,S}(n))\wedge$ nºiter $=O(f_{iter}(n))$

   ⇒**mientras** B **hacer** S$=O(f_{B,S}(n)*f_{iter}(n))$

si el coste de las iteraciones no varía, sino: ∑ costes indiv.

⇒ **para** i ← x **hasta** y **hacer** S = $O(f_S(n) * n^{\underline{o}}\ iter)$
si el coste de las iteraciones no varía, sino: ∑ costes indiv.

- B es comparar 2 enteros = O(1); no iter = y −x +1

- Uso de las reglas:
  - análisis "de adentro hacia afuera"
  - analizar primero los subprogramas
  - recursividad: intentar tratarla como un ciclo, sino resolver relación de recurrencia

- **Ejemplo:** $\sum_{i=1}^{n} i^3$

```
  función suma (n:entero) : entero
{1}     s:=0;
{2}     para i:=1 hasta n hacer
{3}             s:=s+i*i*i;
{4}     devolver s
  fin función
```

Θ(1) en {3} y no hay variaciones ⇒ Θ(n) en {2} (regla 4) ⇒ T(n) = Θ(n) (regla 2)

- El razonamiento ya incluye las aproximaciones

Ordenación por selección:

```
  procedimiento Ordenación por Selección (var T[1..n])
{1}     para i:=1 hasta n-1 hacer
{2}             minj:=i; minx:=T[i];
{3}             para j:=i+1 hasta n hacer
{4}                     si T[j]<minx entonces
{5}                             minj:=j; minx:=T[j];
                        fin si
                fin para;
{6}             T[minj]:=T[i]; T[i]:=minx
        fin para
  fin procedimiento
```

- Θ(1) en {5} (regla 2)
  ⇒ O(max(Θ(1),Θ(1),0)) = Θ(1) en {4} (regla 3: **no estamos en peor caso**)

- S = Θ(1); no iter=n−i ⇒ Θ(n−i) en {3} (regla 4)

- Θ(1) en {2} y en {6} (regla 2)
  ⇒ Θ(n−i) en {2-6} (regla 2)

- S = Θ(n−i) **varía**: $\begin{cases} i = 1 & \rightarrow\ \Theta(\mathrm{n}) \\ i = n-1 & \rightarrow\ \Theta(1) \end{cases}$

⇒$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$ en {1} (regla 4)

$= (n-1)n - \frac{n(n-1)}{2}$: polinomio de grado 2

$\Rightarrow T(n) = \Theta(n^2)$ **en cualquier caso**

2. VERIFICACIÓN EMPÍRICA DE LA COMPLEJIDAD

- **Aplicación del "método empírico" al análisis de algoritmos:**
  medir tiempos de ejecución (experimentos sistemáticos)
  ⇒ tabla de tiempos para distintos valores de n
  ⇒ ¿O?

  Método empírico: Renacimiento, s. XVII (Galileo)
  "Mide lo que se pueda medir;
  lo que no se pueda... hazlo medible!"
- **Verificación empírica**: normalmente, se parte de una función f(n) candidata (obtenida mediante el análisis).
- **Aplicación**: Trabajo en prácticas

Medición de tiempos de ejecución:

Informe de resultados

- **Contexto**:
  - indicar **qué** se está midiendo:
    algoritmo, caso, características de la entrada, etc.
  - indicar **dónde** se está midiendo:
    id. ordenador del laboratorio (no debe ser un servidor!),
    características del ordenador personal

  ↔ ¿control del entorno de experimentación?
  - indicar unidades de tiempo: µs, ms, s...
  -
- **Tabla de tiempos**: m mediciones para distintos valores de n. Ejemplo (prácticas):

```
n           t(n)         t(n)/n^1.9       t(n)/n^2      t(n)/n^2.2
_____________________________________________________________________
 1000     1263.0000000      0.0025200      0.0012630     0.00031725
 2000     4926.0000000      0.0026335      0.0012315     0.00026930
 4000    18995.0000000      0.0027210      0.0011872     0.00022600
 8000    74409.0000000      0.0028560      0.0011626     0.00019268
16000   296213.0000000      0.0030463      0.0011571     0.00016693
32000  1209055.0000000      0.0033317      0.0011807     0.00014829
64000  4782141.0000000      0.0035309      0.0011675     0.00012765

                          subestimada      ajustada     sobrestimada
                                          cte:0.00118
```

```
f(n) = n^1.9 -> cota subestimada
g(n) = n^2   -> cota ajustada con constante : 0.00118 -> O(n^2)
h(n) = n^2.1 -> cota sobrestimada
```

- **Normas** en la obtención de tablas de tiempos (prácticas):
  - Los n deben seguir una progresión geométrica:
    → *2, *10 únicamente
  - Debe medirse un mínimo de 5 valores de n (m ≥ 5):

→ idealmente 7-8 mediciones

- **No debe haber tiempos nulos** (ti > 0)...
- ...ni tiempos muy pequeños medidos directamente
  ⇒ t_i ≥ umbral de confianza: ti ≥ 500µs
  → estrategia para la medición de tiempos pequeños
- No esperar mucho tiempo por un resultado
  → abortar mediciones e indicarlo en la tabla, si procede

Medición de tiempos pequeños:

```
leer_tiempo (ta);
repetir K veces:
      alg(n);
leer_tiempo (tb)
```

- K debe ser una potencia de 10
- en la tabla se pondrá (tb –ta)/K
- y se indicara con una nota al pie que esa medición corresponde a un tiempo promedio de K ejecuciones del algoritmo
- sólo válido para algoritmos que no modifican la entrada:
  Que hacer por ej. con un algoritmo de ordenación?
  → Ok para medir ordenación de una entrada ordenada. No usar en cualquier otro caso!

Ejemplo: “ordenación de un vector aleatorio”

```
inicializar(vector);
leer_tiempo (ta); alg(vector); leer_tiempo (tb);
t:=tb-ta;
si (t < 500) entonces {      % ‘‘umbral de confianza’’
      leer_tiempo (ta);
      repetir K veces: {
            inicializar(vector); alg(vector)
      };
      leer_tiempo (tb);
      t1:=tb-ta;             % debería estar por encima de 500!
      leer_tiempo (ta);
      repetir K veces:       % debe ser la misma constante K
            inicializar(vector);
      leer_tiempo (tb);
      t2:=tb-ta;
      t:=(t1-t2)/K
}
```

¿Valoración de esta solución?

- se mide más “ruido” (gestión del bucle...)
  → dudas sobre la calidad de la mediciones
- riesgo de tiempos negativos
  → comprobar las tablas, calcularlas varias veces. . .
- t1 también debería estar, a su vez, por encima del umbral de confianza
- …

A pesar de todo, la menos mala → “Norma” para las practicas

Determinar la complejidad del algoritmo

Punto de partida:

n	$t(n)$
n_1	t_1
n_2	t_2
n_3	t_3
...	...
n_m	t_m

Objetivo: encontrar f(n) **tal que** t(n)/f(n) → C **cte**. > 0 **donde** f(n) **sea una de las funciones características** ("típicas"): logn, n, nlogn, n^k, 2^n...
↔ una función con el mismo crecimiento!

- diremos entonces que f(n) **es una cota (superior) ajustada** (para t(n))
- t(n) = O(f(n))

→ ¿Cómo asegurarse de que la serie t(n)/f(n) → C > 0?

Técnica propuesta:

n	$t(n)$	$t(n)/f(n)$	$t(n)/g(n)$	$t(n)/h(n)$
n_1	t_1	$t_1/f(n_1)$	$t_1/g(n_1)$	$t_1/h(n_1)$
n_2	t_2	$t_2/f(n_2)$	$t_2/g(n_2)$	$t_2/h(n_2)$
n_3	t_3	$t_3/f(n_3)$	$t_3/g(n_3)$	$t_3/h(n_3)$
...	...	...	...	...
n_m	t_m	$t_m/f(n_m)$	$t_m/g(n_m)$	$t_m/h(n_m)$

Donde:

- f(n) es una cota **ligeramente subestimada** como cota superior ajustada
- g(n) es una cota **ajustada**: t(n) = O(g(n))
- h(n) es una cota **ligeramente sobrestimada**

Atención al orden de las funciones! < f(n),g(n),h(n) > ordenadas
Ejemplo: < n, **nlogn**, $n^{1,5}$ >; < $n^{0,8}$, n, $n^{1,2}$ >; ...
→ ¿Cómo diferenciar las 3 situaciones?

Estudio de la convergencia de una serie t(n)/f(n):

Dadas:

- t(n) la serie de mediciones obtenidas (tiempos)
- f(n) la cota con la que vamos a comparar las mediciones (una función característica)

⇒

- Si t(n)/f(n) → ∞ (diverge) cuando n → ∞:
  f(n) es una cota **subestimada**
- Si t(n)/f(n) → C > 0 cuando n → ∞:
  f(n) es una cota **ajustada**: t(n) = O(f(n))
- Si t(n)/f(n) → 0 (decrece) cuando n → ∞:
  f(n) es una cota **sobrestimada**

Presentación de los resultados: (en prácticas: ficheros de texto bien alineados)

n	$t(n)$	$t(n)/f(n)$	$t(n)/g(n)$	$t(n)/h(n)$
n_1	t_1*	$t_1/f(n_1)$	$t_1/g(n_1)$	$t_1/h(n_1)$
n_2	t_2*	$t_2/f(n_2)$	$t_2/g(n_2)$	$t_2/h(n_2)$
n_3	t_3	$t_3/f(n_3)$	$t_3/g(n_3)$	$t_3/h(n_3)$
...	...	...	...	...
n_m	t_m	$t_m/f(n_m)$	$t_m/g(n_m)$	$t_m/h(n_m)$
		subestimada	ajustada **Cte =** C	sobrestimada

*: tiempo promedio de K ejecuciones del algoritmo

Tabla i: estudio de la complejidad de $< alg, caso >$

Discusión: explicar dificultades, mediciones anómalas, cualquier duda en el análisis…
Conclusión: t(n) = O(g(n))

Precauciones:

- las cotas utilizadas deben indicarse explícitamente (y en orden)
- las constantes (K, C) deben indicarse explícitamente
  → C puede aproximarse mediante un intervalo
- comprobar recomendaciones sobre no de mediciones, progresión de los valores de n, validez de los resultados…
- todos los números de las series calculadas deben tener al menos tres cifras significativas
- comprobar recomendaciones sobre contexto (introducción, unidades de tiempo…), discusión y conclusiones
- ¿dificultades en el análisis? → repetir las mediciones varias veces y elegir las "mejores series"

3. Resolución de Recurrencias

Definición: Una sucesión es una aplicación del conjunto de los números naturales en un conjunto S. Se suele usar la notación a_n para denotar la imagen del número natural n, el término n-ésimo de la sucesión.

Ejemplos
La sucesión {4n + 1}, es de la forma

1, 5, 9, 13, . . .

Se llama **sucesión constante** a aquella cuyos términos son todos iguales. Los términos de la sucesión constante {2} son

2, 2, 2, 2, 2, . . .

Definición: Se llama relación de recurrencia para una sucesión $\{a_n\}$ a toda expresión matemática, generalmente una ecuación, que relaciona cada término a_n, a partir de uno dado, con los anteriores.

Ejemplo

La sucesión de Fibonacci, $\{F_n\}$ = {0, 1, 1, 2, 3, 5, 8, . . . }, puede definirse mediante la relación de recurrencia

$$F_n = F_n - 1 + F_n - 2 , n \geq 2$$

junto con las **condiciones iniciales**, $\{F_0 = 0,\ F_1 = 1\}$.

Con la misma relación de recurrencia:

$$a_n = a_n - 1 + a_n - 2 , n \geq 2$$

pero con otras condiciones iniciales $\{a_0 = 1, a_1 = 2\}$, nos queda la sucesión: $\{a_n\}$ = {1, 2, 3, 5, 8, . . . }

Definición: Resolver una relación de recurrencia es encontrar las sucesiones que la satisfacen, dando una fórmula explícita para el cálculo de su n-ésimo término.

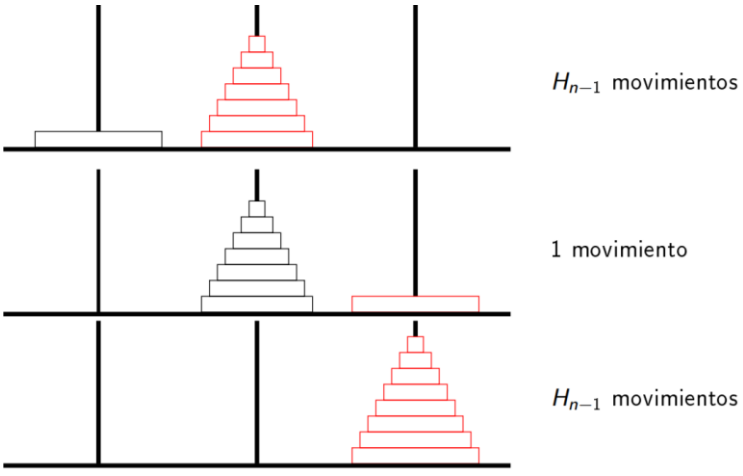
Ejemplo (Torres de Hanoi)

Objetivo: Trasladar la torre de discos a otro de los palos
Normas:

- en cada paso se mueve un único disco
- sólo puede moverse el que está en la parte superior de un montón
- no puede colocarse un disco encima de otro de menor tamaño

La sucesión $\{H_n\}$, donde $H_n$es el número de movimientos necesarios para resolver el juego de las torres de Hanoi con n discos, es solución de la relación de recurrencia

$$H_n = 2H_n-1 + 1$$



Los primeros términos de la sucesión son:

n	0	1	2	3	4	5	6	⋯
H_n	0	1	3	7	15	31	63	⋯

Parece que $H_n = 2^n - 1$

En efecto, la sucesión $a_n = 2^n - 1$, es una solución para la relación de recurrencia

$$a_n = 2a_{n-1} + 1,$$

puesto que

$$2^n - 1 = 2 \cdot (2^{n-1} - 1) + 1$$

OJO: Pero también la sucesión constante {−1, −1, −1, . . . }, es solución para esta misma relación de recurrencia, pues −1 = 2(−1) + 1. Evidentemente no es una solución para el problema de las torres de Hanoi.

3.1. Relaciones de Recurrencia Homogéneas

Definición: Una **relación de recurrencia lineal homogénea, con coeficientes constantes,** (RRLHCC), **de orden** *k* es una expresión de la forma

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

donde los coeficientes, $c_1, \ldots, c_k$, son números reales y $c_k \neq 0$

Ejemplos:

1. $F_n = F_{n-1} + F_{n-2}$ es una RRLHCC de orden 2.
2. $a_n = 2a_{n-1} + 3a_{n-2} - 5a_{n-3}$ es una RRLHCC de orden 3.
3. $a_n = na_{n-1}$ es una RRLH pero sus coeficientes no son constantes.
4. $a_n = 2a_{n-1} + 1$ es una RRLCC de orden 1 pero no homogénea.
5. $a_n = a_{n-1}a_{n-2}$ es una RRHCC de orden 2 pero no es lineal.

Para resolver la RRLHCC:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

buscaremos soluciones del tipo $a_n = r^n$, r ≠ 0. Substituyendo, tenemos:

$$r_n = c_1 r^{n-1} + c_2 r^{n-2} + \cdots + c_k r^{n-k}$$

Dividiendo por r^{n-k} y reordenando:

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \cdots - c_k = 0$$

$\{r^n\}$ es una solución de la relación de recurrencia si, y sólo si, r satisface la ecuación

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \cdots - c_k = 0,$$

que recibe el nombre de ecuación característica, y sus raíces el de raíces características.

Teorema (Solución de RRLHCC con raíces características distintas)

Sea $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$ una RRLHCC **tal que sus raíces características, $r_1, \ldots, r_k$, son todas reales y distintas**. Entonces, para cualesquiera números reales, $\alpha_1, \ldots, \alpha_k$,

- la sucesión $a_n = \alpha_1 r_1^n + \cdots + \alpha_k r_k^n$ es una solución para la relación de recurrencia
- cualquier solución es de esta forma, para algunos números reales $\alpha_1, \ldots, \alpha_k$

Teorema (Solución de RRLHCC con raíces características no distintas)

Sea $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$ una relación de recurrencia lineal, homogénea y con coeficientes constantes tal que sus raíces características, $r_1, \ldots, r_s$, son reales y con multiplicidades respectivas $m_1, \ldots, m_s$. Las soluciones son de la forma

$$\begin{aligned} an = &\left(\alpha_{10} + \alpha_{11} n + \cdots + \alpha_{1m_1-1} n^{m_1-1}\right) r_1^n \\ &+ \left(\alpha_{20} + \alpha_{21} n + \cdots + \alpha_{2m_2-1} n^{m_2-1}\right) r_2^n \\ &\cdots\cdots \\ &+ \left(\alpha_{s0} + \alpha_{s1} n + \cdots + \alpha_{sm_s-1} n^{m_s-1}\right) r_s^n \end{aligned}$$

para cualesquiera números reales α_{ij}

3.2. Relaciones de Recurrencia No Homogéneas

Definición: Una **relación de recurrencia lineal no homogénea, con coeficientes constantes,** (RRLnHCC), **de orden** *k* es una expresión de la forma

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + L(n)$$

donde los coeficientes, $c_1, \ldots, c_k$, son números reales, $c_k \neq 0$ y L(n) es una función de n (no nula)

Ejemplos:

1. $h_n = 2h_{n-1} + 1$
2. $a_n = 3a_{n-1} + 2^n$
3. $a_n = 3a_{n-1} - 2a_{n-2} + n2^n$

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + L(n)$$

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

relación de recurrencia lineal homogénea asociada

Teorema (Solución de una RRLnHCC)

Si $a_n^{(p)}$ es una solución particular de la RRLnHCC y $a_n^{(h)}$ es cualquier solución de la relación de recurrencia lineal homogénea asociada, entonces

$$a_n = a_n^{(h)} + a_n^{(p)}$$

es también solución de la relación de recurrencia no homogénea, y todas las soluciones son de esta forma, para alguna $a_n^{(h)}$.

Teorema (Soluciones particulares)

Dada la RRLnHCC: $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + L(n)$, donde $L(n) = (p0 + p_1 n + \cdots + p_t n^t) s^n$, entonces

1. Si s **no es** una de las raíces de la relación homogénea asociada, entonces

$$a_n^{(p)} = (\beta_0 + \beta_1 n + \cdots + \beta_t n^t) s^n ,$$

es una solución particular para $\beta_0, \ldots, \beta_t \in$ R.

2. Si s es una de las raíces de la relación homogénea asociada, con multiplicidad m, entonces

$$a_n^{(p)} = (\beta_0 + \beta_1 n + \cdots + \beta_t n^t) n^m s^n ,$$

es una solución particular para β0, . . . , βt ∈

3.3. Divide y Vencerás

TEMA 2: ESTRUCTURAS DE DATOS

1. PILAS COLAS Y LISTAS

1.1. Pilas

- Acceso limitado al último elemento insertado
- Operaciones básicas: **apilar**, **desapilar** y **cima**.
  - desapilar o cima en una pila vacía es un error en el TDA pila.
  - Quedarse sin espacio al apilar es un error de implementación.
- Cada operación debería tardar una cantidad constante de tiempo en ejecutarse.
  - Con independencia del número de elementos apiladas.

Pseudocódigo: Implementación a base de vectores

```
tipo Pila = registro
Cima_de_pila : 0..Tamaño_máximo_de_pila
Vector_de_pila : vector [1..Tamaño_máximo_de_pila]
                 de Tipo_de_elemento
fin registro

procedimiento Crear Pila ( P )
    P.Cima_de_pila := 0
fin procedimiento

función Pila Vacía ( P ) : test
    devolver P.Cima_de_pila = 0
fin función

procedimiento Apilar ( x, P )
    si P.Cima_de_pila = Tamaño_máximo_de_pila entonces
      error Pila llena
    sino
      P.Cima_de_pila := P.Cima_de_pila + 1;
      P.Vector_de_pila[P.Cima_de_pila] := x
fin procedimiento

función Cima ( P ) : Tipo_de_elemento
    si Pila Vac´ıa (P) entonces error Pila vacía
    sino devolver P.Vector_de_pila[P.Cima de Pila]
fin función

procedimiento Desapilar ( P )
    si Pila Vac´ıa (P) entonces error Pila vacía
    sino P.Cima_de_pila := P.Cima_de_pila - 1
fin procedimiento
```

Código C: pilas.h

```
#ifndef TAMANO_MAXIMO_PILA
#define TAMANO_MAXIMO_PILA 10
#endif
typedef int tipo_elemento;
typedef struct {
    int cima;
    tipo_elemento vector[TAMANO_MAXIMO_PILA];
} pila;

void crear_pila(pila *);
int pila_vacia(pila);
void apilar(tipo_elemento, pila *);
tipo_elemento cima(pila);
void desapilar(pila *);
/* ERRORES: cima o desapilar sobre la pila vacía apilar sobre la pila llena
*/
```

Código C: pilas.c

```
#include <stdlib.h>
#include <stdio.h>
#include "pilas.h"

void crear_pila(pila *p) {
    p->cima = -1;
}

int pila_vacia(pila p) {
    return (p.cima == -1);
}

void apilar(tipo_elemento x, pila *p) {
    if (++p->cima == TAMANO_MAXIMO_PILA) {
      printf("error: pila llena\n"); exit(EXIT_FAILURE);
    }
    p->vector[p->cima] = x;
}

tipo_elemento cima(pila p) {
    if (pila_vacia(p)) {
      printf("error: pila vacia\n");
      exit(EXIT_FAILURE);
    }
    return p.vector[p.cima];
}

void desapilar(pila *p) {
    if (pila_vacia(*p)) {
      printf("error: pila vacia\n");
      exit(EXIT_FAILURE);
    }
    p->cima--;
}
```

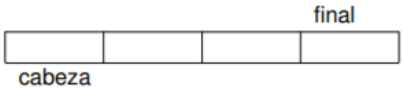
1.2. Colas

- Operaciones básicas: insertar, quitarPrimero y primero.
- Cada rutina debería ejecutarse en tiempo constante

Implementación circular a base de vectores

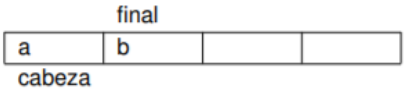
La implementación circular devuelve cabeza y fin al principio del vector cuando rebasan la última posición.

1. Crear_Cola (C)

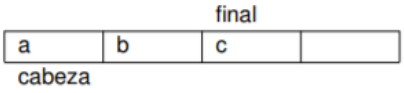


2. Insertar_en_Cola (a,C)

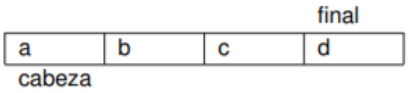
3. Insertar_en_Cola (b,C)



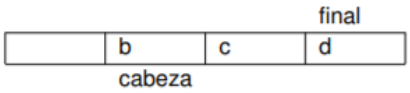
4. Insertar_en_Cola (c,C)



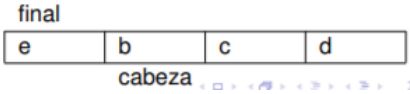
5. Insertar_en_Cola (c,C)



6. Quitar_Primero (C)



7. Insertar_en_Cola (e,C)



Pseudocódigo

```
tipo Cola = registro
    Cabeza_de_cola, Final_de_cola: 1..Tamaño_máximo_de_cola
    Tamaño_de_cola : 0..Tamaño_máximo_de_cola
    Vector_de_cola : vector [1..Tama˜no_m´aximo_de_cola]
                          de Tipo_de_elemento
fin registro

procedimiento Crear_Cola ( C )
    C.Tama˜no_de_cola := 0;
    C.Cabeza_de_cola := 1;
    C.Final_de_cola := Tamaño_máximo_de_cola
fin procedimiento

función Cola_Vacía ( C ) : test
    devolver C.Tamaño_de_cola = 0
fin función

procedimiento incrementar ( x ) (* privado *)
    si x = Tamaño_máximo_de_cola entonces x := 1
    sino x := x + 1
fin procedimiento
```

```
procedimiento Insertar_en_Cola ( x, C )
    si C.Tama˜no_de_Cola = Tama˜no_m´aximo_de_cola entonces
      error Cola llena
    sino
      C.Tamaño_de_cola := C.Tamaño_de_cola + 1;
      incrementar(C.Final_de_cola);
      C.Vector_de_cola[C.Final_de_cola] := x;
fin procedimiento

función Quitar_Primero ( C ) : Tipo_de_elemento
    si Cola_Vacía ( C ) entonces
      error Cola vacía
    sino
      C.Tamaño_de_cola := C.Tamaño_de_cola - 1;
      x := C.Vector_de_cola[C.Cabeza_de_cola];
      incrementar(C.Cabeza_de_cola);
    devolver x
fin función

función Primero ( C ) : Tipo_de_elemento
    si Cola_Vac´ıa ( C ) entonces
      error Cola vacía
    sino
      devolver C.Vector_de_cola[C.Cabeza_de_cola]
fin función
```

Código C: colas.h

```
#ifndef TAMANO_MAXIMO_COLA
#define TAMANO_MAXIMO_COLA 5
#endif
typedef int tipo_elemento;
typedef struct {
    int cabeza, final, tamano;
    tipo_elemento vector[TAMANO_MAXIMO_COLA];
} cola;
void crear_cola(cola *);
int cola_vacia(cola);
void insertar(tipo_elemento, cola *);
tipo_elemento quitar_primero(cola *);
tipo_elemento primero(cola);
/* ERRORES: quitar_primero o primero sobre una cola vacía insertar en una
cola llena */
```

Código C: colas.c

```
#include <stdlib.h>
#include <stdio.h>
#include "colas.h"

void crear_cola(cola *c) {
    c->tamano = 0;
    c->cabeza = 0;
    c->final = -1;
}

int cola_vacia(cola c) {
    return (c.tamano == 0);
}



void incrementar(int *x) { /* privado */
    if (++(*x) == TAMANO_MAXIMO_COLA)
    *x = 0;

}

void insertar(tipo_elemento x, cola *c) {
    if (c->tamano == TAMANO_MAXIMO_COLA) {
      printf("error: cola llena: %d\n", c->tamano);
      exit(EXIT_FAILURE);
    }
    c->tamano++;
    incrementar(&(c->final));
    c->vector[c->final] = x;
}

tipo_elemento primero(cola c) {
    if (cola_vacia(c)) {
      printf("error: cola vacia\n");
      exit(EXIT_FAILURE);
    }
    return(c.vector[c.cabeza]);
}

tipo_elemento quitar_primero(cola *c) {
    tipo_elemento x;
    if (cola_vacia(*c)) {
      printf("error: cola vacia\n");
      exit(EXIT_FAILURE);
    }
    c->tamano--;
    x = c->vector[c->cabeza];
    incrementar(&(c->cabeza));
    return x;
}
```

1.3. Listas

- Operaciones básicas:
  - **Visualizar** su contenido.
  - **Buscar** la posición de la primera ocurrencia de un elemento.
  - **Insertar** y **Eliminar** un elemento en alguna posición.
  - **Buscar_k_esimo**, que devuelve el elemento de la posición indicada.

Implementación de la lista a base de vectores

- Tiene que declararse el tamaño de la lista.
  - Exige sobrevaloración.
  - Consume mucho espacio.
- Complejidad computacional de las operaciones:
  - **Buscar_k_esimo**, tiempo constante.
  - **Visualizar** y **Buscar**, tiempo lineal.
  - **Insertar** y **Eliminar** son costosas.
    - Insertar o eliminar un elemento exige, en promedio, desplazar la mitad de los valores, O(n).
    - La construcción de una lista o la eliminación de todos sus elementos podría exigir un tiempo cuadrático.

Implementación de la lista a base de apuntadores

- Cada nodo apunta al siguiente; el ultimo no apunta a nada.
- La lista es un puntero al primer nodo (y al último).
- Complejidad computacional de las operaciones:
  - **Visualizar** y **Buscar**, tiempo lineal.
  - **Buscar_k_esimo**, tiempo lineal.
  - **Eliminar** realiza un cambio de apuntadores y una orden *dispose*, O(1).
    - Usa **Buscar_anterior** cuyo tiempo de ejecución es lineal.
  - **Insertar** tras una posición p requiere una llamada a *new* y dos maniobras con apuntadores, O(1).
    - Buscar la posición *p* podría llevar tiempo lineal.
  - Un nodo cabecera facilita la inserción y la eliminación al comienzo de la lista.

Implementación de listas doblemente enlazadas

- Cada nodo apunta al siguiente y al anterior.
- Duplica el uso de la memoria necesaria para los punteros.
- Duplica el coste de manejo de punteros al insertar y eliminar.
- La eliminación se simplifica.
  - No es necesario buscar el elemento anterior.

Pseudocódigo: Implementación con un nodo cabecera

```
tipo PNodo = puntero a Nodo
    Lista = PNodo
    Posición = PNodo
    Nodo = registro
      Elemento : Tipo_de_elemento
      Siguiente : PNodo
    fin registro

procedimiento Crear Lista ( L )
    nuevo ( tmp );
    si tmp = nil entonces error Memoria agotada
    sino
      tmpˆ.Elemento := { nodo cabecera };
      tmpˆ.Siguiente := nil;
      L := tmp
fin procedimiento

función Lista Vacía ( L ) : test
    devolver Lˆ.Siguiente = nil
fin función

función Buscar ( x, L ) : posición de la 1a ocurrencia o   nil
    p := Lˆ.Siguiente;
    mientras p <> nil y pˆ.Elemento <> x hacer
      p := pˆ.Siguiente;
    devolver p
fin función

función Último Elemento ( p ) : test { privada }
    devolver pˆ.Siguiente = nil
fin función

función Buscar Anterior ( x, L ) : posición anterior a x o a nil { privada }
    p := L;
    mientras pˆ.Siguiente <> nil y pˆ.Siguienteˆ.Elemento <> x hacer
      p := pˆ.Siguiente;
    devolver p
fin función


procedimiento Eliminar ( x, L )
    p := Buscar Anterior ( x, L );
    si Último Elemento ( p ) entonces error No encontrado
    sino tmp := pˆ.Siguiente;
      pˆ.Siguiente := tmpˆ.Siguiente;
      liberar ( tmp )
fin procedimiento

procedimiento Insertar ( x, L, p )
    nuevo ( tmp ); { Inserta despu´es de la posici´on p }
    si tmp = nil entonces
      error Memoria agotada
    sino
      tmpˆ.Elemento := x;
      tmpˆ.Siguiente := pˆ.Siguiente;
      pˆ.Siguiente := tmp;
fin procedimiento
```

Código C: listas.h

```
struct nodo {
    void *elem; /* ’void *’ es un apuntador ’generico’ */
    struct nodo *sig;
};

typedef struct nodo *posicion;
typedef struct nodo *lista;

lista crearlista();
int eslistavacia(lista l);
void insertar(void *e, posicion p); /*inserta e tras el nodo apuntado por p*/
posicion buscar(lista l, void *e, int (*comp)(const void *x, const void *y));
    /*la función comp devuelve un número mayor, igual o menor que cero,
    seg´un x sea mayor, igual, o menor que y*/
void borrar(lista l, void *e, int (*comp)(const void *x, const void *y));
posicion primero(lista l);
posicion siguiente(posicion p);
int esfindelista(posicion p);
void *elemento(posicion p);

/* Para recorrer los elementos de la lista:
    for(p=primero(l); !esfindelista(p); p=siguiente(p)) {
      //hacer algo con elemento(p)
    }
*/
```

Código C: listas.c

```
#include <stdlib.h>
#include <stdio.h>
#include "listas.h"

static struct nodo *crearnodo(){
    struct nodo *tmp = malloc(sizeof(struct nodo));
    if (tmp == NULL) {
      printf("memoria agotada\n"); exit(EXIT_FAILURE);
    }
    return tmp;
}


lista crearlista(){
    struct nodo *l = crearnodo();
    l->sig = NULL;
    return l;
}

int eslistavacia(lista l){
    return (l->sig == NULL);
}
```

```
void insertar(void *x, posicion p) {
    struct nodo *tmp = crearnodo();
    tmp->elem = x;
    tmp->sig = p->sig;
    p->sig = tmp;
}

posicion buscar(lista l, void *e, int (*comp)(const void *x, const void *y)){
    struct nodo *p = l->sig;
    while (p != NULL && 0!=(*comp)(p->elem, e))
      p = p->sig;
    return p;
}

static posicion buscarant(lista l, void *x,
                          int (*comp)(const void *, const void *)) {
    struct nodo *p = l;
    while (p->sig != NULL && 0!=(*comp)(p->sig->elem, x))
      p = p->sig;
    return p;
}

static int esultimoelemento(struct nodo *p) {
    return (p->sig == NULL);
}

void borrar(lista l, void *x, int (*comp)(const void *, const void *)) {
    struct nodo *tmp, *p = buscarant(l, x, comp);
    if (!esultimoelemento(p)) {
      tmp = p->sig;
      p->sig = tmp->sig;
      free(tmp);
    }
}

posicion primero(lista l) { return l->sig; }
posicion siguiente(posicion p) { return p->sig; }
int esfindelista(posicion p) { return (p==NULL); }
void *elemento(posicion p) { return p->elem; }
```

2. Árboles Binarios de Búsqueda y Montículos

2.1. Árboles binarios de búsqueda

- El camino de un nodo n1 a otro nk es la secuencia de nodos n1,n2,...,nk tal que ni es el padre de ni+1.

- La profundidad de un nodo n es la longitud del camino entre la raíz y n.
  - La raíz tiene profundidad cero.
- Para un árbol binario de búsqueda, el valor medio de la profundidad es O(logn).
  - Si la inserción en un ABB no es aleatoria, el tiempo computacional aumenta.
    - Para mantener el equilibrio: Arboles AVL, Splay Trees…
- La altura de *n* es el camino más largo de *n* a una hoja.
  - La altura de un árbol es la altura de la raíz.

Operaciones básicas:

- **Buscar**: devuelve la posición del nodo con la clave *x*.
- **Insertar**: coloca la clave *x*. Si ya estuviese, no se hace nada (o se "actualiza" algo).
- **Eliminar**: borra la clave *x*.
  - Si *x* esta en una hoja, se elimina de inmediato.
  - Si el nodo tiene un hijo, se ajusta un apuntador antes de eliminarlo.
  - Si el nodo tiene dos hijos, se sustituye x por la clave más pequeña, ˜ w, del subárbol derecho.
    - A continuación se elimina en el sub árbol derecho el nodo con w (que no tiene hijo izquierdo)

Eliminación perezosa:

- Si se espera que el número de eliminaciones sea pequeño, la eliminación perezosa es una buena estrategia.
  - Al eliminar un elemento, se deja en el árbol marcándolo como eliminado.
  - Habiendo claves duplicadas, es posible decrementar el campo con la frecuencia de apariciones.
  - Si una clave eliminada se vuelve a insertar, se evita la sobrecarga de asignar un nodo nuevo.
- Si el número de nodos reales en el árbol es igual al número de nodos "eliminados", se espera que la profundidad del árbol sólo aumente en uno (¿por qué?).
  - La penalización de tiempo es pequeña

Implementación de árboles binarios

```
tipo
    PNodo = ^Nodo
    Nodo = registro
             Elemento : TipoElemento
             Izquierdo, Derecho : PNodo
      fin registro
    ABB = PNodo

procedimiento CrearABB (var A)
    A := nil
fin procedimiento

función Buscar (x, A) : PNodo
    si A = nil entonces devolver nil
    sino si x = A^.Elemento entonces devolver A
    sino si x < A^.Elemento entonces devolver Buscar (x, A^.Izquierdo)
    sino devolver Buscar (x, A^.Derecho)
fin función

función BuscarMin (A) : PNodo
    si A = nil entonces devolver nil ç
    sino si A^.Izquierdo = nil entonces devolver A
    sino devolver BuscarMin (A^.Izquierdo)
fin función

procedimiento Insertar (x, var A)
    si A = nil entonces
      nuevo (A);
      si A = nil entonces error ‘‘sin memoria’’
      sino
          A^.Elemento := x;
          A^.Izquierdo := nil;
          A^.Derecho := nil
    sino si x < A^.Elemento entonces
      Insertar (x, A^.Izquierdo)
    sino si x > A^.Elemento entonces
      Insertar (x, A^.Derecho)
    { si x = A^.Elemento : nada }
fin procedimiento

procedimiento Eliminar (x, var A)
    si A = nil entonces error ‘‘no encontrado’’
    sino si x < A^.Elemento entonces
      Eliminar (x, A^.Izquierdo)
    sino si x > A^.Elemento entonces
      Eliminar (x, A^.Derecho)
    sino { x = A^.Elemento }
      si A^.Izquierdo = nil entonces
          tmp := A; A := A^.Derecho; liberar (tmp)
      sino si A^.Derecho = nil entonces
        tmp := A; A := A^.Izquierdo; liberar (tmp)
      sino tmp := BuscarMin (A^.Derecho);
            A^.Elemento := tmp^.Elemento;
            Eliminar (A^.Elemento, A^.Derecho)
fin procedimiento
```

Recorridos de un árbol

- En orden: Se procesa el subárbol izquierdo, el nodo actual y, por último, el subárbol derecho. O(n)

```
procedimiento Visualizar (A)
      si A <> nil entonces
            Visualizar (Aˆ.Izquierdo);
            Escribir (Aˆ.Elemento);
            Visualizar (Aˆ.Derecho)
fin procedimiento
```

- Post-orden: Ambos subárboles primero. O(n)

```
función Altura (A) : número
      si A = nil entonces devolver -1
      sino devolver 1 + max (Altura (Aˆ.Izquierdo), Altura
(Aˆ.Derecho))
fin función
```

- Pre-orden: El nodo se procesa antes. Ej: una función que marcase cada nodo con su profundidad. O(n)
- Orden de nivel: Todos los nodos con profundidad p se procesan antes que cualquier nodo con profundidad p +1. Se usa una cola en vez de la pila implícita en la recursión. O(n)

2.2. Montículos

Colas de prioridad

Permiten únicamente el acceso al mínimo (o máximo) elemento.
Operaciones básicas: **insertar**, **eliminarMin** (**eliminarMax**).
Implementaciones simples:

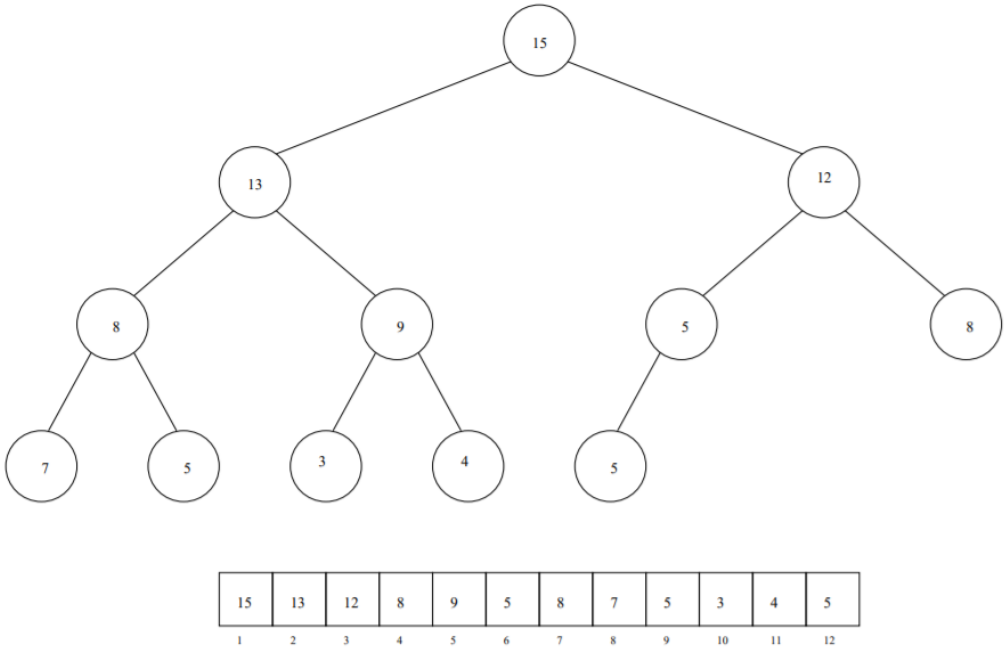
- Listas enlazadas efectuando inserciones al frente, O(1), y recorriendo la lista, O(n), para eliminar el mínimo (máximo).
- Listas ordenadas: inserciones costosas, O(n), eliminaciones eficientes, O(1).
- Árboles binarios de búsqueda: tiempo de ejecución medio O(logn) para ambas operaciones.
    - A pesar de que las eliminaciones no son aleatorias.
    - Se eliminan repetidamente nodos de un subárbol. No obstante, el otro subárbol es aleatorio y tendría a lo sumo el doble de elementos de los que debería. Y esto solo incrementa en uno la profundidad esperada.
- Montículos: ambas operaciones se realizan en O(logn) para el peor caso. No requieren apuntadores.

Propiedades estructurales de los montículos

Un montículo es un árbol binario completo: todos los niveles están llenos con la posible excepción del nivel más bajo, que se llena de izquierda a derecha.

- Un árbol binario completo de altura *h* tiene entre 2^h y $2^{h+1} - 1$ nodos.
  - Su altura es la parte entera de $\log_2 n$.
- Esta regularidad facilita su representación mediante un vector.
- Para cualquier elemento en la posición *i* del vector, el hijo izquierdo esta en la posición 2i, el hijo derecho en 2i +1, y el padre en i ÷2.
- El mínimo (o máximo) está en la raíz.
- Y como todo subárbol es también un montículo, todo nodo debe ser menor (mayor) o igual que todos sus descendientes.

Ejemplo de montículo de máximos



Implementación de montículos

```
tipo Montículo = registro
    Tamaño_montículo : 0..Tamaño_máximo
    Vector_montículo : vector [1..Tamaño_máximo] de Tipo elemento
fin registro

procedimiento Inicializar Montículo ( M )
    M.Tamaño monticulo := 0
fin procedimiento

función Montículo_Vacío ( M ) : test
    return M.Tamaño_monticulo = 0
fin función

procedimiento Flotar ( M, i ) { privado }
    mientras i > 1 y M.Vector_montículo[i div 2] < M.Vector_montículo[i]
    hacer
      intercambiar M.Vector_montículo[i div 2] y M.Vector_montículo[i];
      i := i div 2
    fin mientras
fin procedimiento
```

```
procedimiento Insertar ( x, M )
    si M.Tamaño_monticulo = Tamaño_máximo entonces
      error Monticulo lleno
    sino M.Tamaño_monticulo := M.Tamaño_monticulo + 1;
      M.Vector monticulo[M.Tama˜no monticulo] := x;
      Flotar ( M, M.Tamaño_monticulo )
fin procedimiento

procedimiento Hundir ( M, i ) { privado }
    repetir
      HijoIzq := 2*i;
      HijoDer := 2*i+1;
      j := i;
      si HijoDer <= M.Tamaño_monticulo y
          M.Vector_montículo[HijoDer] > M.Vector_montículo[i]
      entonces i := HijoDer;
      si HijoIzq <= M.Tamaño_monticulo y
        M.Vector_montículo[HijoIzq] > M.Vector_montículo[i]
      entonces i := HijoIzq;
      intercambiar M.Vector_montículo[j] y M.Vector_montículo[i];
    hasta j=i {Si j=i el nodo alcanz´o su posici´on final}
fin procedimiento

función EliminarMax ( M ) : Tipo_elemento
    si Montículo Vacío ( M ) entonces
      error Monticulo vacío
    sino
      x := M.Vector_montículo[1];
      M.Vector_montículo[1] := M.Vector_montículo[M.Tamaño_monticulo];
      M.Tamaño_monticulo := M.Tamaño_monticulo - 1;
      si M.Tamaño_monticulo > 0 entonces
            Hundir ( M, 1);
      devolver x
fin función
```

- Creación de montículos en tiempo lineal, O(n):

```
procedimiento Crear_Montículo ( V[1..n], M )
    Copiar V en M.Vector_montículo;
    M.Tamaño_montículo := n;
    para i := M.Tamaño_montículo div 2 hasta 1 paso -1
      Hundir(M, i);
    fin para
fin procedimiento
```

- El número de intercambios está acotado por la suma de las alturas de los nodos.
- Se demuestra mediante un argumento de marcado del árbol.
- Para cada nodo con altura h, marcamos h aristas:
- bajamos por la arista izquierda y después sólo por aristas derechas.
- Así una arista nunca se marca 2 veces.

3. TABLAS DE DISPERSIÓN

Objetivo: realizar inserciones, eliminaciones y búsquedas en tiempo promedio constante.
Estructura de datos ideal: un vector con tamaño fijo N.

- Una función de dispersión establece la correspondencia de cada clave con algún número en el intervalo [0...N −1].
- Esta función tiene que ser fácil de calcular, y asegurar que dos claves distintas se correspondan con celdas diferentes.
    - Como esto último es imposible, buscamos una función que distribuya homogéneamente las claves entre las celdas.
- Resta escoger una función y decidir el tamaño de la tabla y qué hacer cuando dos claves caen en la misma celda.
    - Si al insertar un elemento, este se dispersa en el mismo valor que un elemento ya insertado tenemos una colisión y hay que resolverla.

Las tablas de dispersión se usan para representar diccionarios en los que se busca una clave y se devuelve su definición.

3.1. Funciones de dispersión

- Toda función de dispersión debe:
    - calcularse de forma sencilla, y
    - distribuir uniformemente las claves.
- Por ejemplo, si las claves son números enteros, clave mod N es una función buena, salvo que haya propiedades indeseables:
    - Si N fuese 100 y todas las claves terminasen en cero, esta función de dispersión sería una mala opción.
        - Es buena idea asegurarse de que el tamaño de la tabla sea un numero primo.
    - Si las claves fuesen enteros aleatorios, esta función sería muy simple y distribuiría las claves con uniformidad.
- Por lo regular, las claves son cadenas de caracteres.
    - La longitud y las propiedades de las claves influirán en la elección de una buena función de dispersión.

- Una opción es sumar los valores ASCII de los caracteres.

```
función Dispersión1 (Clave, TamañoClave): Índice
    valor := ascii(Clave[1]);
    para i := 2 hasta TamañoClave hacer
       valor := valor + acii(Clave[i])
    fin para
    devolver valor mod N
fin función
```

- Es una función fácil de implementar, y se ejecuta con rapidez.
- Pero si el tamaño de la tabla es grande, esta función no distribuye bien las claves.
    - Por ejemplo, si N = 10007 y las claves tienen 8 caracteres, la función sólo toma valores entre 0 y 1016 ´ = 127 · 8

- En esta función de dispersión se supone que la clave tiene al menos tres caracteres:

```
función Dispersión2 (Clave, TamañoClave): Índice
  devolver (ascii(Clave[1]) + 27*ascii(Clave[2]) + 729*ascii(Clave[3])) mod N
fin función
```

- Si los primeros caracteres son aleatorios y el tamaño de la tabla es 10007, esperaríamos una distribución bastante homogénea.
- Desafortunadamente, los lenguajes naturales no son aleatorios.
    - Aunque hay 27^3 = 17576 combinaciones posibles, en un diccionario el número de combinaciones diferentes que nos encontramos es menor que 3000.
    - Sólo un porcentaje bajo de la tabla puede ser aprovechada por la dispersión.

- En la función de dispersión que sigue intervienen todos los caracteres en la clave y se puede esperar una buena distribución:

```
función Dispersión3 (Clave, TamañoClave): Índice
    valor := ascii(Clave[1]);
    para i := 2 hasta TamañoClave hacer
      valor := (32*valor + acii(Clave[i])) mod N
    fin para
    devolver valor
fin función
```

- El código calcula una función polinómica con base en la regla:

$$\sum_{i=0}^{LongitudClave-1} 32^i * ascii(calve[LongitudClave - i])$$

    - Para “abcd” p. ej. (323 · ascii(a) +322 · ascii(b) +32 · ascii(c) +ascii(d)) mod N

- La multiplicación por 32 es el desplazamiento de cinco bits.
- Con lenguajes que permitan el desbordamiento se aplicaría mod una sola vez justo antes de volver.
- Si las claves son muy grandes, no se usan todos los caracteres.

3.2. Dispersión abierta

Resolución de colisiones: dispersión abierta

- La solución consiste en tener una lista de todos los elementos que se dispersan en un mismo valor.
- Al buscar, usamos la función de dispersión para determinar qué lista recorrer.
- Al insertar, recorremos la lista adecuada.
    - Si el elemento resulta ser nuevo, se inserta al frente o al final de la lista.
- Además de listas, se podría usar cualquier otra estructura para resolver las colisiones, como un árbol binario de búsqueda.

- El factor de carga, λ, de una tabla de dispersión es la relación entre el número de elementos en la tabla y su tamaño.

$$\lambda = \frac{Número\ de\ claves\ en\ la\ tabla}{N}$$

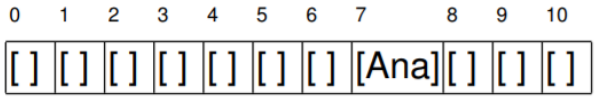
- La longitud media de una lista es λ.
    - En dispersión abierta, la regla es igualar el tamaño de la tabla al número de elementos esperados, (λ = 1).
- El esfuerzo al realizar una búsqueda es:
    1) el tiempo constante necesario para evaluar la función de dispersión, O(1), mas
    2) el tiempo necesario para recorrer la lista, O(λ).
        - En una búsqueda infructuosa el promedio de nodos recorridos es O(λ)
        - En una búsqueda con éxito, O(λ/2)

Ejemplo de dispersión abierta

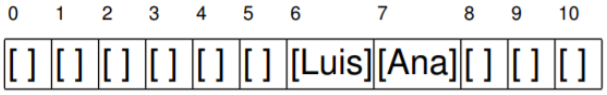
- Valores de la función de dispersión:

hash(Ana,11) =7	hash(Luis,11)=6	hash(Jose,11)=7
hash(Olga,11)=7	hash(Rosa,11)=6	hash(Ivan,11)=6

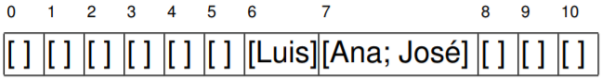
- Tabla después de insertar Ana:



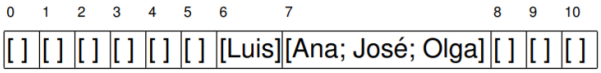
- Tabla después de insertar Luís:



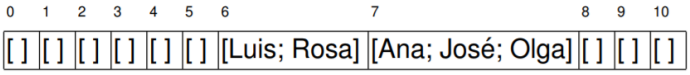
- Tabla después de insertar José:



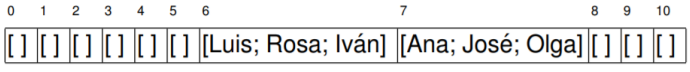
- Tabla después de insertar Olga:



- Tabla después de insertar Rosa:



- Tabla después de insertar Iván:



Pseudocódigo

```
tipo
    Índice = 0..N-1
    Posición = ^Nodo
    Lista = Posición
    Nodo = registro
       Elemento : TipoElemento
       Siguiente : Posición
    fin registro
    TablaDispersión = vector [Índice] de Lista

procedimiento InicializarTabla (T)
    para i := 0 hasta N-1 hacer
       CrearLista(T[i])
    fin para
fin procedimiento

función Buscar (Elem, Tabla): Posición
    i := Dispersión(Elem);
    devolver BuscarLista(Elem, Tabla[i])
fin función

procedimiento Insertar (Elem, Tabla)
    pos := Buscar(Elem, Tabla); {No inserta repetidos}
    si pos = nil entonces
       i := Dispersión(Elem);
       InsertarLista(Elem, Tabla[i])
fin procedimiento
```

3.3. Dispersión cerrada

- En un sistema de dispersión cerrada, si ocurre una colisión, se buscan celdas alternativas hasta encontrar una vacía.
    - Se busca en sucesión en las celdas: d0(x),d1(x),d2(x)... donde:
      di(x) = (dispersión(x) +f(i)) mod N, con f(0) = 0
    - La función f es la estrategia de resolución de las colisiones.
        - Determinará si la dispersión cerrada es lineal, cuadrática o doble.
- Como todos los datos se guardan en la tabla, esta tiene que ser más grande para la dispersión cerrada que para la abierta.
- En general, para la dispersión cerrada el factor de carga λ debe estar por debajo de 0,5
- La eliminación estándar no es realizable con dispersión cerrada.
    - La celda ocupada pudo haber causado una colisión en el pasado.
- Por ejemplo, con exploración lineal, si las claves “x” e “y” se dispersan a la misma posición (p. ej. 2)

4. GRAFOS

Un grafo es un par G = (V,A).

- V es el conjunto de vértices o nodos.
- A es el conjunto de aristas.
    - Cada arista es un par (v,w) ∈ V.
    - Si el par esta ordenado, entonces el grafo es dirigido.

Principales representaciones de grafos dirigidos:

- Matriz de adyacencia.
- Listas de adyacencia.

4.1. Matriz de adyacencia

- Es una matriz bidimensional.
- Para cada arista (u,v), se pone a[u,v] = 1; en caso contrario, el contenido es 0.
- Si la arista tiene un peso asociado,
    - se pone en a[u,v] el peso, y
    - se usa un peso muy grande o muy pequeño como centinela indicando la inexistencia de aristas.
- Requerimiento de espacio: $\Theta(|V|^2)$.
    - Resulta adecuado para grafos densos,
    - pero prohibitivo si el grafo es disperso.

4.2. Listas de adyacencia

- Para cada vértice mantenemos una lista de todos sus vértices adyacentes.
    - La representación consistirá en un vector de listas de adyacencia.
- Requerimiento de espacio: $\Theta(|A|+|V|)$.
    - Buena solución para grafos dispersos.
- Si el grafo no fuese dirigido,
    - Cada arista (u,v) aparecería en dos listas, duplicándose el espacio en uso.

Consideraciones

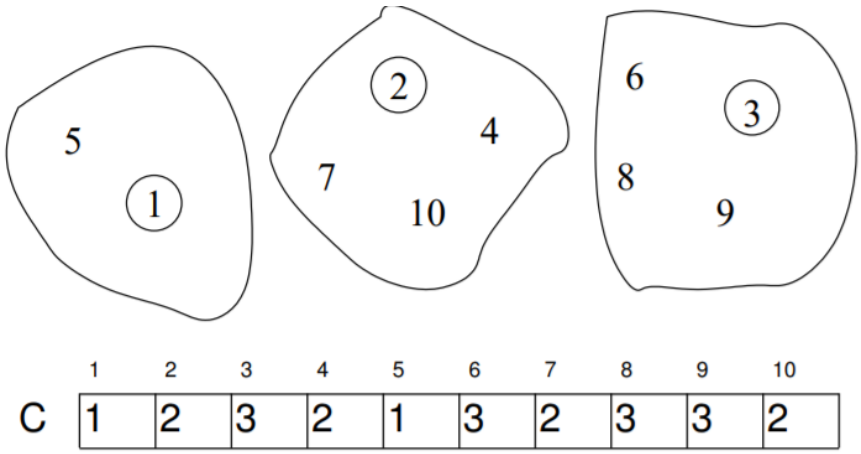
- Problema común en los algoritmos de grafos: encontrar los nodos adyacentes a un nodo dado.
    - Ambas representaciones consiguen buenos resultados,
        - recorriendo una fila o columna de la matriz de adyacencia, o
        - recorriendo la lista de adyacencia apropiada.
- En la mayoría de aplicaciones, los vértices tienen nombres, desconocidos en tiempo de compilación, en vez de números.
    - La forma más sencilla de dar una correspondencia entre nombres y números es usar una tabla de dispersión.

5. Conjuntos Disjuntos

5.1. Primer enfoque

Representación de conjuntos disjuntos

- Todos los elementos se numeran de 1 a n.
- Cada subconjunto tomara su nombre de uno de sus elementos, su representante, p. ej. el valor más pequeño.
- Mantenemos en un vector el nombre del subconjunto disjunto de cada elemento



Operaciones válidas

- La representación inicial es una colección de n conjuntos, Ci.
  - Cada conjunto tiene un elemento diferente, Ci ∩ Cj = 0
  - Así, al principio se tiene Ci = {i}.
- Hay dos operaciones válidas.
  - La búsqueda devuelve el nombre del conjunto de un elemento dado.
  - La unión combina dos subconjuntos que contienen a y b en un subconjunto nuevo, destruyéndose los originales.

Pseudocódigo

```
tipo
      Elemento = entero;
      Conj = entero;
      ConjDisj = vector [1..N] de entero

función Buscar1 (C, x) : Conj
      devolver C[x]
fin función
```

- La búsqueda es una simple consulta O(1).
  - El nombre del conjunto devuelto por búsqueda es arbitrario.
  - Todo lo que importa es que búsqueda(x)=búsqueda(y) si y solo si x e y están en el mismo conjunto.

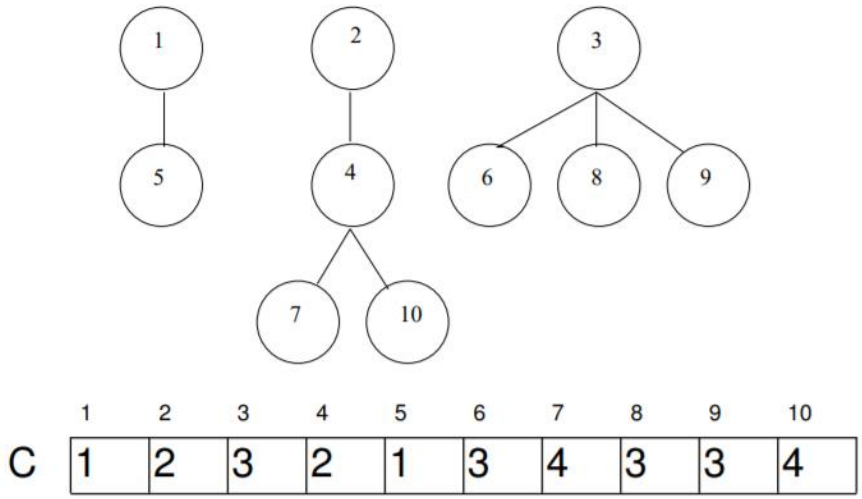
```
procedimiento Unir1 (C, a, b)
      i := min (C[a], C[b]);
      j := max (C[a], C[b]);
      para k := 1 hasta N hacer
            si C[k] = j entonces C[k] := i
      fin para
fin procedimiento
```

- La unión toma O(n). No importa, en lo que concierne a corrección, qué conjunto retiene su nombre.
- Una secuencia de n−1 uniones (la máxima, ya que entonces todo estaría en un conjunto) tomaría $O(n^2)$.
- La combinación de m búsquedas y $n-1$ uniones toma $O(m+n^2)$.

5.2. Segundo enfoque

Se utiliza un árbol para caracterizar cada subconjunto.

- La raíz nombra al subconjunto.
- La representación de los árboles es fácil porque la única información necesaria es un apuntador al padre.
- Cada entrada p[i] en el vector contiene el padre del elemento i.
    - Si i es una raíz, entonces p[i]=i



Pseudocódigo

```
función Buscar2 (C, x) : Conj
      r := x;
      mientras C[r] <> r hacer
            r := C[r]
      fin mientras;
      devolver r
fin función
```

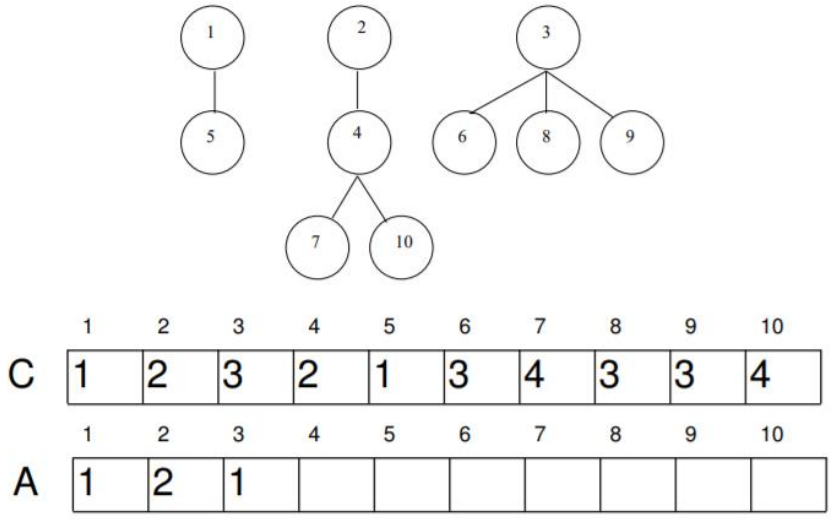
- Una búsqueda sobre el elemento x se efectúa devolviendo la raíz del árbol que contiene x.
- La búsqueda de un elemento x es proporcional a la profundidad del nodo con x.
    - En el peor caso es O(n)

```
procedimiento Unir2 (C, raíz1, raíz2)
            { supone que raíz1 y raíz2 son raíces }
    si raíz1 < raíz2 entonces C[raíz2] := raíz1
    sino C[raíz1] := raíz2
fin procedimiento
```

- La unión de dos conjuntos se efectúa combinando ambos arboles: apuntamos la raíz de un árbol a la del otro.
- La unión toma O(1).
- La combinación de m búsquedas y n−1 uniones toma $O(m \cdot n)$.

5.2.1. Unión por alturas

- Las uniones anteriores se efectuaban de modo arbitrario.
- Una mejora sencilla es realizar las uniones haciendo del árbol menos profundo un subárbol del árbol más profundo.
    - La altura se incrementa solo cuando se unen dos árboles de igual altura.



Pseudocódigo

```
procedimiento Unir3 (C, A, raíz1, raíz2) { supone que raíz1 y raíz2
son raíces }
    si A[raíz1] = A[raíz2] entonces
      A[raíz1] := A[raíz1] + 1;
      C[raíz2] := raíz1
    sino si A[raíz1] > A[raíz2] entonces C[raíz2] := raíz1
    sino C[raíz1] := raíz2
fin procedimiento
```

- La profundidad de cualquier nodo nunca es mayor que $\log_2(n)$.
  - Todo nodo esta inicialmente a la profundidad 0.
  - Cuando su profundidad se incrementa como resultado de una unión, se coloca en un árbol al menos el doble de grande.
    - Así, su profundidad se puede incrementar a lo más, $\log_2(n)$ veces.
- El tiempo de ejecución de una búsqueda es O(log(n)).
- Combinando m búsquedas y n−1 uniones, **O(m · log(n) +n)**.

5.2.2. Compresión de caminos

La compresión de caminos se ejecuta durante búsqueda.

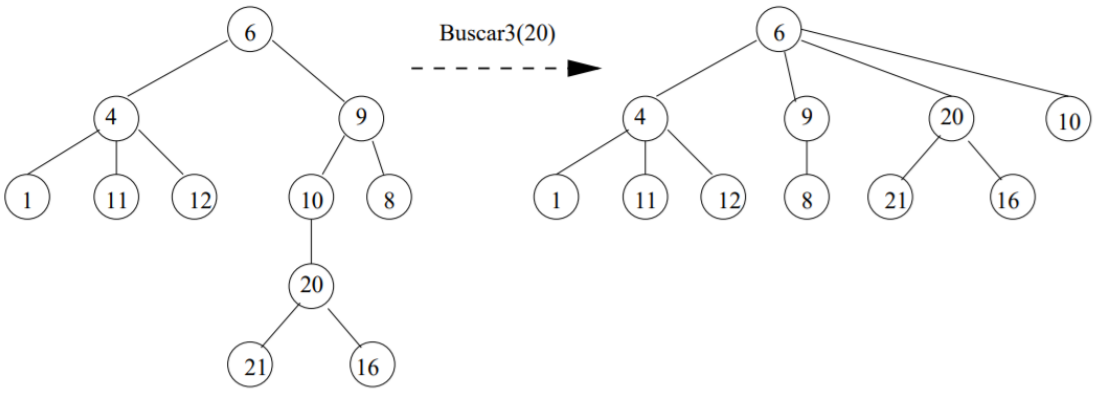
- Durante la búsqueda de un dato x, todo nodo en el camino de x a la raíz cambia su padre por la raíz.
- Es independiente del modo en que se efectúen las uniones.

```
función Buscar3 (C, x) : Conj
      r := x;
      mientras C[r] <> r hacer
            r := C[r]
      fin mientras;
      i := x;
      mientras i <> r hacer
            j := C[i]; C[i]:= r; i := j
      fin mientras;
      devolver r
fin función
```

Rangos

- La compresión de caminos no es totalmente compatible con la unión por alturas.
  - Al buscar, la altura de un árbol puede cambiar.
- Las alturas almacenadas para cada árbol se convierten en alturas estimadas (rangos).
  - Pero la unión por rangos es tan eficiente, en teoría, como la unión por alturas.

Ejemplo de compresión de caminos



TEMA 3: ALGORITMOS SOBRE SECUENCIAS Y CONJUNTOS DE DATOS

1. ALGORITMOS SOBRE SECUENCIAS Y CONJUNTOS DE DATOS (VISTOS EN TGR)

1.1. Suma de la Subsecuencia Máxima

- **Problema de la Suma de la Subsecuencia Máxima**
  a1..an → $\sum_{k=i}^{j} a_k$ maxima?
  Ejemplo: SSM(−2,11,−4,13,−5,−2) = 20 [2..4]

- **SSM recursiva**: estrategia Divide y Vencerás
  Divide la entrada en 2 mitades → 2 soluciones recursivas
  Vence usando las 2 soluciones → solución para entrada original

  La SSM puede estar: $\begin{cases} en\ la\ 1ª\ mitad \\ en\ la\ 2ª\ mitad \\ entre\ las\ 2\ mitades \end{cases}$

  Las dos primeras soluciones son las obtenidas recursivamente.
  La 3ª solución se obtiene sumando:
  - la SSM de la 1a mitad que incluye el extremo derecho, y
  - la SSM de la 2a mitad que incluye el extremo izquierdo.

```
    función SSM ( a[1..n] ) : valor          función interfaz
        devolver SSM recursiva (a, 1, n)
    fin función

    función SSM recursiva (var a[1..n], izq, der) : valor
{1}     si izq = der entonces
{2}         si a[izq] > 0 entonces
{3}             devolver a[izq]              caso base: si >0, es SSM
            sino
{4}             devolver 0
            fin si
        sino
{5}         Centro := (izq + der) div 2 ;
{6}         Primera solución := SSM recursiva (a, izq, Centro);
{7}         Segunda solución := SSM recursiva (a, Centro + 1, der);
{8}         Suma máxima izquierda := 0 ; Suma izquierda := 0 ;
{9}         para i := Centro hasta izq paso -1 hacer
{10}            Suma izquierda := Suma izquierda + a[i] ;
{11}                si Suma izquierda > Suma máxima izquierda entonces
{12}                    Suma máxima izquierda := Suma izquierda
            fin para ;
{13}        Suma máxima derecha := 0 ; Suma derecha := 0 ;
{14}        para i := Centro + 1 hasta der hacer
{15}            Suma derecha := Suma derecha + a[i] ;
{16}            si Suma derecha > Suma m´axima derecha entonces
{17}                Suma máxima derecha := Suma derecha
            fin para ;
{18}        devolver max (Primera solución, Segunda solución, Suma máxima
                          izquierda + Suma máxima derecha)
        fin si
    fin función
```

- **Análisis**:
  Caso base: {1-4} ⇒ T(1) = Θ(1)
  Ciclos {9-12} y {14-17} : Θ(n) en conjunto: a1..an
  Llamadas recursivas {6} y {7} : T(n/2) cada una (aprox.)
  Resto = Θ(1): se puede ignorar frente a Θ(n)

  Relación de recurrencia: $\begin{cases} T(1) = 1 \\ T(n) = 2T\left(\frac{n}{2}\right) + n, n > 1 (*) \end{cases}$

  1. Razonando con inducción:

$$T(\begin{matrix}2\\4\\8\\16\\\dots\end{matrix}) = \begin{matrix}4\\12\\32\\80\\\end{matrix} = \begin{matrix}2 * 2\\4 * 3\\8 * 4\\16 * 5\\\end{matrix}$$

⇒ $T(n) = n(k+1)$ para $n = 2^k$: *demostrar la hipótesis*
⇒ $T(n) = n(log_2 n + 1) = \Theta(nlogn)$

  2. Manejando proyecciones

a) dividir (*) por n:	$\frac{T(n)}{n}$	$= \frac{T(n/2)}{n/2} + 1$
b) proyectar ($n = 2^k$):	$\frac{T(n/2)}{n/2}$	$= \frac{T(n/4)}{n/4} + 1$
	$\frac{T(n/4)}{n/4}$	$= \frac{T(n/8)}{n/8} + 1$
	...	
	$\frac{T(2)}{2}$	$= \frac{T(1)}{1} + 1$
c) sumar:	$\frac{T(n)}{n}$	$= T(1) + log_2 n \Rightarrow T(n) = \Theta(nlogn)$

  3. Aplicando teoremas

Teorema de resolución de recurrencias Divide y Vencerás

$T(n) = lT(n/b) + cn^k, n > n_0,$
con $l \geq 1$, $b \geq 2$, $c > 0 \in R$, $k \geq 0 \in N$, $n_0 \geq 1 \in N$
$\{l = 2, b = 2, c = 1, k = 1, n_0 = 1\}$: caso $l = b^k$
$\Rightarrow T(n) = \Theta(n^k logn) \Rightarrow T(n) = \Theta(nlogn)$

- **Observacion**: pasar el vector *a* **por referencia** (var), sino:

Sea R(n): nº de copias de a: $\begin{cases} R(1) = 0 \\ R(n) = 2R\left(\frac{n}{2}\right) + 2, n > 1 \end{cases}$

⇒ R(n) = 2n − 2 copias ∗ Θ(n) cada una ⇒ T(n) = $\Theta(n^2)$
También la complejidad espacial sería cuadrática!

- **SSM en línea**:
  -Acceso secuencial
  -Respuesta para la subsecuencia parcial
  ⇒ Algoritmo en línea
  Análisis: espacio constante (no se necesita memorizar la entrada) y tiempo lineal
  → mejor imposible
  Ejercicio: modificar el algoritmo para asegurar espacio constante

```
    función SSM en línea ( a[1..n] ) : <i, j, valor>
{1}     i:=1 ; EstaSuma:=0 ; SumaMax:=0 ; MejorI:=0 ; MejorJ:=0 ;
{2}     para j := 1 hasta n hacer
{3}         EstaSuma := EstaSuma + a[j] ;
{4}         si EstaSuma > SumaMax entonces
{5}             SumaMax := EstaSuma ;
{6}             MejorI := i ;
{7}             MejorJ := j
{8}         sino si EstaSuma < 0 entonces
{9}             i := j+1 ;
{10}            EstaSuma := 0
            fin si
        fin para ;
{11}    devolver < MejorI, MejorJ, SumaMax>
    fin función
```

1.2. Búsqueda Binaria

- Ejemplo de algoritmo logarítmico
- Dados x y un vector ordenado a1, a2, ... an de enteros,

Devolver: $\begin{cases} i \; si \; \exists \; a_i = x \\ "elemento \; no \; encontrado" \end{cases}$

➔ Comparar x y a_{medio}, con medio = (i +j)div2, siendo $a_i, \dots, a_j$ el espacio de búsqueda:

1. $x = a_{medio}$: terminar(interrupción)
2. $x > a_{medio}$: seguir buscando en $a_{medio+1} \dots a_j$
3. $x < a_{medio}$: seguir buscando en $a_i \dots a_{medio-1}$

- ¿nº iter? ↔ evolución del tamaño *d* del espacio de búsqueda
- Invariante: $d = j - i + 1$

¿Cómo decrece d? $\begin{cases} i \leftarrow \text{medio} + 1 \\ j \leftarrow \text{medio} - 1 \end{cases}$

- Peor caso: se alcanza la terminación “normal” del bucle ≡ i > j

```
    función Búsqueda Binaria (x, a[1..n]) : posición
    {a: vector ordenado de modo no decreciente}
{1}     i := 1 ; j := n ;                    {espacio de búsqueda: i..j}
{2}     mientras i <= j hacer
{3}         medio := (i + j) div 2 ;
{4}         si a[medio] < x entonces
{5}             i := medio + 1
{6}         sino si a[medio] > x entonces
{7}             j := medio - 1
{8}         sino devolver medio           {se interrumpe el bucle}
        fin mientras; ´
{9}     devolver "elemento no encontrado"    {fin normal del bucle}
    fin función
```

Búsqueda binaria: Análisis del peor caso

- Sea $< d, i, j >$ iteración $< d', i', j' >$:

1. i ← medio + 1:

$$i' = (i + j) div 2 + 1$$
$$j' = j$$
$$d' = j' - i' + 1 = j - (i + j)\ div\ 2 - 1 + 1 \leq j - \frac{i+j-1}{2} = \frac{j-i+1}{2} = \frac{d}{2}$$
$$\rightarrow \boxed{d\ 0 \leq d/2}$$

2. i ← medio - 1:

$$i' = i$$
$$j' = (i + j)\ div\ 2 - 1$$
$$d' = j' - i' + 1 = (i + j)\ div\ 2 - i - 1 + 1 \leq \frac{i+j}{2} - i < \frac{j-i+1}{2} = \frac{d}{2}$$
$$\rightarrow \boxed{d\ 0 < d/2} (decrece\ mas\ rápido)$$

- ¿T(n)? Sea d_I: d después de la I-ésima iteración

$$\begin{cases} d_o = n \\ d_I = \frac{d_{I-1}}{2}\ \forall\ \mathrm{I} \geq 1 \end{cases}$$ (inducción) → $d_I \leq \frac{\mathrm{n}}{2^{\mathrm{I}}}$

hasta $d < 1 \rightarrow I = [\log_2 n] + 1 = O(logn)$ iteraciones
Cada iteración es Θ(1) (reglas) ⇒ $T(n) = O(logn)$

Razonamiento alternativo: pensar en version recursiva

$$T(n) = \begin{cases} 1 & si\ n = 0, 1 \\ T\left(\frac{n}{2}\right) + 1 & si\ n > 1 \end{cases}$$

- Teorema Divide y Vencerás: I = 1, b = 2, c = 1, k = 0, n_0 = 1
- Caso I = b k ⇒ T(n) = Θ($n^k \log n$) → T(n) = Θ($\log n$)

Observaciones:

- Pensar en versión recursiva puede ser otro recurso útil
- es Divide y Vencerás? → Algoritmos de reducción´ (I = 1)
- T(n) = Θ($\log n$) ↔ los datos ya están en memoria (Modelo de Computación)

2. ALGORITMOS DE ORDENACIÓN

2.1. Ordenación por Inserción

```
procedimiento Ordenación por Inserción (var T[1..n])
        para i:=2 hasta n hacer
                x:=T[i];
                j:=i-1;
                mientras j>0 y T[j]>x hacer
                        T[j+1]:=T[j];
                        j:=j-1
                fin mientras;
                T[j+1]:=x
        fin para
fin procedimiento
```

- peor caso: max *i* comparaciones para cada *i* ⇒ $\sum_{i=2}^{n} = \Theta(n^2)$
- mejor caso: min 1 comparación para cada *i* (entrada ordenada) ⇒ Θ(n)
- ¿caso medio?

⇒ Cota inferior (Ω) para los algoritmos de ordenación que intercambian elementos adyacentes: inserción, selección, burbuja...

Observación: ¿Inserción intercambia elementos adyacentes? → abstracción

Sea T[1..n] la entrada del algoritmo:
Definición: **inversión** ≡ cualquier (i,j) : i < j ∧T[i] > T[j]
Ejemplo: 3 1 4 1 5 9 2 6 5 3
→ inversiones: (3,1),(3,1),(3,2),...,(5,3)
Sea *I* el número de inversiones: ´ "medida del desorden"
En el ejemplo, I = 15
Intercambiar 2 elementos adyacentes elimina una inversión
En el ejemplo, I = 15 ⇒ 15 intercambios para ordenar hasta I = 0 ≡ vector ordenado
⇒ Inserción = O(I +n)

- Inserción = $\begin{cases} O(n)\ si\ I = 0\ (mejor\ caso) \vee \mathrm{I} = \mathrm{O(n)}\ (*) \\ O(n^2)\ si\ I = O(n^2)\ (peor\ caso) \end{cases}$ (*) Nuevo resultado

- Caso medio ⇒ ¿Imedio en una entrada?

  Hipótesis = $\begin{cases} \text{sin } duplicados \\ permutaciones\ equiprobables \end{cases}$ ⇒ ¿Imedio en una permutación?

  Teorema: Imedio = n(n−1)/4
  Demostración: sean ´ T[1..n] el vector, Ti [1..n] el vector inverso:
  cualquier (x,y) es inversión en T o en Ti
  Nº total de (x,y) con y > x
  = $(n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i = n(n-1)/2$
  equiprobabilidad ⇒ Tmedio tiene la mitad de esas inversiones

  ⇒ Caso medio de Inserción: I = $O(n^2)$ ⇒ T(n) = $O(n^2)$

Teorema: cualquier algoritmo que ordene intercambiando elementos adyacentes requiere un tiempo $\Omega(n^2)$ en el caso medio.
Demostración:

Imedio = n(n−1)/4 = $\Omega(n^2)$
cada intercambio elimina solo una inversión
⇒ Ω(n 2) intercambios

¿Como conseguir un trabajo $o(n^2) \equiv \neg\Omega(n^2)$ ≡ "bajar de n^2"?

- Intercambiar elementos alejados ⇒ deshacer varias inversiones a la vez:
  Ordenación de Shell

2.2. Ordenación de Shell

- 1er algoritmo de ordenación que baja de $O(n^2)$ en el peor caso
- Secuencia de **incrementos** ≡ distancias para intercambios: Naturales, ordenados descendentemente: $h_t, \ldots h_k, h_{k-1}, \ldots h1 = 1$
- t iteraciones: en la iteración *k* utiliza el incremento h_k
  Postcondición = $\{\forall i, T[i] \leq T[i + h_k]\}$
  ≡ los elementos separados por h_k posiciones están ordenados
  → **vector h_k-ordenado**
  Trabajo de la iteración *k*: h_k ordenaciones por Inserción
- Propiedad:
  un vector h_k-ordenado que se h_{k-1}-ordena sigue estando h_k-ordenado
- Problema: ¿secuencia optima de incrementos?
  incrementos de Shell: $h_t = \left[\frac{n}{2}\right], h_k = [h_k + \frac{1}{2}]$

```
procedimiento Ordenación de Shell (var T[1..n])
    incremento := n;
    repetir
        incremento := incremento div 2;
        para i := incremento+1 hasta n hacer
            tmp := T[i];
            j := i;
            seguir := cierto;
            mientras j-incremento > 0 y seguir hacer
                si tmp < T[j-incremento] entonces
                    T[j] := T[j-incremento];
                    j := j-incremento
                sino seguir := falso ;
            T[j] := tmp
    hasta incremento = 1
fin procedimiento
```

- Otros incrementos también funcionan

Ejemplo: n = 13 → 5,3,1 en vez de Shell (6,3,1)

ini	81	94	11	96	12	35	17	95	28	58	41	75	15
(5)	35	17	**11**	28	12	41	75	**15**	96	58	81	94	**95**
(3)	28	12	11	35	15	41	58	17	94	75	81	96	95
(1)	11	12	15	17	28	35	41	58	75	81	94	95	96

- **Teorema**: Shell con incrementos de Shell es $\Theta(n^2)$ (peor caso).

Demostración:

1. ¿$\Omega(n^2)$?
   n = 2^k ⇒ incrementos pares excepto el ultimo (= 1)
   Peor situación: los n/2 mayores están en las posiciones pares
   Ejemplo (el más favorable dentro de esta situación):

1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16

   ¿Mas favorable? → 8, 4 y 2-ordenado; todo el trabajo: 1-ordenar
   El i-ésimo menor está en la posición 2 ´ i −1, i ≤ n/2 (ej: 8 en posición 15)
   → hay que moverlo i −1 veces (ej: 8 → 7 desplazamientos)
   ⇒ colocar menores: $\sum_{i=1}^{n/2} i - 1 = \Omega(n^2)$

2. ¿$\Omega(n^2)$?
   Trabajo realizado en iteración *k* con el incremento h_k:
   h_k ordenaciones por Inserción sobre n/h_k elementos cada una
   = $h_k O(\left(\frac{n}{h_k}\right)^2) = O(h_k \left(\frac{n}{h_k}\right)^2) = O(\frac{n^2}{h_k})$
   En el conjunto de iteraciones del algoritmo:
   $T(n) = O(\sum_{i=1}^{t} n^2/h_i) = O(n^2 \sum_{i=1}^{t} 1/h_i)) = O(n^2)$

   Observación 1: ≠ $O(n^3)$ (← "3 bucles anidados")
   Observación 2: Bajar de $O(n^2)$ en peor caso? → otros incrementos

- **Otros incrementos**

incrementos	peor caso	caso medio
Hibbard: $1,3,7..,2^k-1$	$\Theta(n^{3/2})$ (teo[b])	$O(n^{5/4})$ (sim[c])
Sedgewick[a]: $1,5,19,41,109...$	$O(n^{4/3})$ (sim[c])	$O(n^{7/6})$ (sim[c])

Tabla: Ordenación de Shell con distintos incrementos

a: varias secuencias de incrementos
b: demostración formal (teorema)
c: comprobación empírica (simulaciones)

- **Conclusión**: código sencillo y resultados muy buenos en la práctica

2.3. Ordenación por montículo

```
procedimiento Ordenación por Montículos (var T[1..n])
    Crear montículo (T, M);
    para i := 1 hasta n hacer
        T[n-i+1] := Obtener mayor valor (M);
        Eliminar mayor valor(M)
```

```
    fin para
fin procedimiento
```

Para crear un montículo a partir de un vector:

```
procedimiento Crear montículo (V[1..n], var M)
{ V[1..n]: entrada: vector con cuyos datos se construirá el montículo
M: entrada/salida: montículo a crear }
    Copiar V[1..n] en M[1..n];
    para i := n div 2 hasta 1 paso -1 hacer
        hundir(M,i)
    fin para
fin procedimiento
```

- ¿Como mejorar la complejidad espacial (y algo T(n))?
  → utilizar la misma estructura. Ejemplo:

entrada	4	3	7	9	6	5	8
Crear Mont.	9	6	8	3	4	5	7
Eliminar(9)	8	6	7	3	4	5	**9**
Eliminar(8)	7	6	5	3	4	**8**	9
Eliminar(7)	6	4	5	3	**7**	8	9
Eliminar(6)	5	4	3	**6**	7	8	9
Eliminar(5)	4	3	**5**	6	7	8	9
Eliminar(4)	3	**4**	5	6	7	8	9
Eliminar(3)	**3**	4	5	6	7	8	9

- **Teorema**: La ordenación por montículos es $O(n \log n)$. Demostración:
  Crear Montículo es $O(n)$, y n Eliminar es $O(n \log n)$
- **Observación**: Incluso en el peor caso es $O(n \log n)$, pero en la practica es más lento que Shell con incrementos de Sedgewick.

2.4. Ordenación por fusión

```
procedimiento Fusión ( var T[Izda..Dcha], Centro:Izda..Dcha )
{fusiona los subvectores ordenados T[Izda..Centro] y T[Centro+1..Dcha] en
T[Izda..Dcha], en T[Izda..Dcha], utilizando un vector auxiliar
Aux[Izda..Dcha] }
    i := Izda ; j := Centro+1 ; k := Izda ;
    {i, j y k recorren T[Izda..Centro], T[Centro+1..Dcha] y
     Aux[Izda..Dcha] respectivamente}
    mientras i <= Centro y j <= Dcha hacer
        si T[i] <= T[j] entonces Aux[k] := T[i] ; i := i+1
        sino Aux[k] := T[j] ; j := j+1 ;
        k := k+1 ;
    {copia elementos restantes del subvector sin recorrer}
    mientras i <= Centro hacer
        Aux[k] := T[i] ; i := i+1 ; k := k+1 ;
    mientras j <= Dcha hacer
        Aux[k] := T[j] ; j := j+1 ; k := k+1 ;
```

```
    para k := Izda hasta Dcha hacer
        T[k] := Aux[k]
fin procedimiento
```

- mergesort
- O bien, ordenación por intercalación.
- Utiliza un algoritmo de **Fusión** de un vector cuyas mitades están ordenadas para obtener un vector ordenado.
- El procedimiento Fusión es lineal (n comparaciones).
- Ordenación: algoritmo Divide y Vencerás
    - Divide el problema en 2 mitades, que
    - se resuelven recursivamente;
    - Fusiona las mitades ordenadas en un vector ordenado.
- **Mejora**: Ordenación por Inserción para vectores pequeños: n < umbral, que se determina empíricamente.

```
procedimiento Ordenación por Fusión Recursivo ( var T[Izda..Dcha] )
    si Izda+UMBRAL < Dcha entonces
        Centro := ( Izda+Dcha ) div 2 ;
        Ordenación por Fusión Recursivo ( T[Izda..Centro] ) ;
        Ordenación por Fusi´on Recursivo ( T[Centro+1..Dcha] ) ;
        Fusión ( T[Izda..Dcha], Centro )
    sino Ordenación por Inserción ( T[Izda..Dcha] )
fin procedimiento

procedimiento Ordenación por Fusión ( var T[1..n] )
    Ordenación por Fusión Recursivo ( T[1..n] );
fin procedimiento
```

- **Análisis** de la versión puramente recursiva (UMBRAL = 0):
  $T(n) = T(n/2) + T(n/2) + O(n)$ (Fusion)

$$n = 2^k \Rightarrow \begin{cases} T(1) = O(1) & = 1 \\ T(n) = 2T\left(\frac{n}{2}\right) + O(n) = 2T\left(\frac{n}{2}\right) + n, n > 1 \end{cases}$$

  Teorema Divide y Vencerás: $I = 2,\ b = 2,\ c = 1,\ k = 1, n_0 = 1$
  caso $I = b^k \Rightarrow T(n) = \Theta(nlogn)$
- Podría mejorarse la complejidad espacial (= 2n: vector auxiliar)
  → El algoritmo adecuado es quicksort
- **Observación**: Importancia de balancear los subcasos en Divide y Vencerás: Si llamadas recursivas con vectores de tamaño $n-1$ y 1
  $\Rightarrow T(n) = T(n-1) + T(1) + n = O(n^2)$
  ... pero ya no sería Ordenación por Fusión.

2.4. Ordenación rápida (Quicksort)

Paradigma de Divide y Vencerás.
Con respecto a Fusión:

- más trabajo para construir las subinstancias (pivote...),
- pero trabajo nulo para combinar las soluciones.

Selección del pivote en T[i..j]:

- Objetivo: obtener una partición lo más balanceada posible
  ⇒ **¿Mediana?** Inviable
- Usar el **primer valor** del vector (T[i]):
  Ok si la entrada es aleatoria.
  Pero elección muy desafortunada con entradas ordenadas o parcialmente ordenadas (caso bastante frecuente)
  → $O(n^2)$ para no hacer nada...
- Usar un valor elegido al azar (pivote aleatorio):
  Más seguro, evita el peor caso detectado antes, pero depende del generador de números aleatorios (eficiencia vs. coste).
- Usar la **mediana de 3 valores**: $T[i],\ T[j],\ T[(i\ +j)div2]$

```
procedimiento Mediana 3 ( var T[i..j] )
    centro := ( i+j ) div 2 ;
    si T[i] > T[centro] entonces intercambiar ( T[i], T[centro] ) ;
    si T[i] > T[j] entonces intercambiar ( T[i], T[j] ) ;
    si T[centro] > T[j] entonces intercambiar ( T[centro], T[j] ) ;
    intercambiar ( T[centro], T[j-1] )
fin procedimiento
```

- **Estrategia de partición**:
  Ejemplo: → Hipótesis: sin duplicados

entrada	8	1	4	9	6	3	5	2	7	0
Mediana 3	0	1	4	9	**7**	3	5	2	6	8

```
procedimiento Qsort ( var T[i..j] )
    si i+UMBRAL <= j entonces
        Mediana 3 ( T[i..j] ) ;
        pivote := T[j-1] ; k := i ; m := j-1 ;      {sólo con Mediana 3}
        repetir
            repetir k := k+1 hasta T[k] >= pivote ;
            repetir m := m-1 hasta T[m] <= pivote ;
            intercambiar ( T[k], T[m] )
        hasta m <= k ;
        intercambiar ( T[k], T[m] ) ;     {deshace el último intercambio}
        intercambiar ( T[k], T[j-1] ) ;          {pivote en posición k}
        Qsort ( T[i..k-1] ) ;
        Qsort ( T[k+1..j] )
fin procedimiento

procedimiento Quicksort ( var T[1..n] )
    Qsort ( T[1..n] ) ;
    Ordenación por Inserción ( T[1..n] )
fin procedimiento
```

- **Estrategia de partición**:
  Ejemplo (Cont.):

entrada	8	1	4	9	6	3	5	2	7	0
Mediana 3	0	1	4	9	**7**	3	5	2	6	8
iteración 1	0	1	4	**2**	7	3	5	**9**	6	8
iteración 2	0	1	4	2	**5**	3	**7**	9	6	8
iteración 3	0	1	4	2	5	**7**	**3**	9	6	8
corrección	0	1	4	2	5	**3**	**7**	9	6	8
final	0	1	4	2	5	3	6	9	**7**	8

- **Observaciones** sobre intercambiar:
  - Mejor deshacer un intercambio que incluir un test en el bucle
  - Evitar llamadas a funciones
- La estrategia de partición depende de la selección del pivote [Brassard & Bratley] → Mediana 3 sin sentido si UMBRAL < 3 (de hecho, el algoritmo propuesto falla; ejercicio: corregirlo)
- **Ejercicio**: escribir Quicksort con pivote aleatorio

- **Considerar valores repetidos**:
  ↔ ¿Parar o no parar cuando $T[k]$ = pivote o $T[m]$ = pivote?
  - Uno de los índices se detiene y el otro no:
    ⇒ los valores idénticos al pivote van al mismo lado ≡ desbalanceo
    Caso extremo (*): todos los valores son idénticos ⇒ $O(n^2)$ ⇒ Hacer lo mismo
  - Parar los 2 índices: (*) → muchos intercambios inútiles, pero los índices se cruzan en la mitad ≡ partición balanceada, $O(n \log n)$
  - No parar ninguno: (*) → evitar que sobrepasen $[i..j]$, pero sobretodo no se produce ningún intercambio ´ ≡ desbalanceo, $O(n^2)$

- **Vectores pequeños**:
  - Recursividad ↔ muchas llamadas (en las hojas) con vectores pequeños, que serán mejor tratados por Inserción, si nos aseguramos que *I* es $O(n)$.
  - ⇒ Utilizar un **umbral** para determinar los casos base. Su valor optimo se encuentra empíricamente: entre n = 12 y n = 15
  - Otra mejora: hacer una única llamada a Inserción con todo el vector: El *I* total es igual a la suma de los *I* locales.

- **Análisis**: pivote aleatorio y sin umbral

$$\Rightarrow \begin{cases} T(0) = T(1) = 1 \\ T(n) = T(z) + T(n-z-1) + cn, \ \ n > 1 \end{cases}$$

  - **Peor caso**: p siempre es el menor o el mayor elemento
    $\Rightarrow T(n) = T(n-1) + cn, n > 1$
    $\Rightarrow \boxed{T(n) = O(n^2)}$
  - **Mejor caso**: p siempre coincide con la mediana
    $\Rightarrow T(n) = 2T(n/2) + cn, n > 1$
    $\Rightarrow \boxed{T(n) = O(nlogn)}$

- **Caso medio**:
  Sea z: tamaño de la parte izquierda;
  Cada valor posible para z (0, 1, 2, ..., n − 1) es equiprobable: p = 1/n
  $\leftrightarrow\ T(z)\ =\ T(n-z\ -1)\ =\ 1/n \sum_{x=0}^{n-1} T(x)$
  $\Rightarrow\ T(n) = \frac{2}{n}[\sum_{x=0}^{n-1} T(x)]\ + cn, n\ >\ 1$
  $\Rightarrow \boxed{T(n)\ =\ O(nlogn)}$
  (calculo similar al de la profundidad media de un ABB = $O(logn)$)

- **Algoritmos aleatorios**: El peor caso ya no es una entrada particular, sino que depende de la secuencia de números aleatorios obtenida durante la ejecución.
  ¿Mejor caso? ¿Caso medio?
  → Otros problemas: calidad de los números aleatorios...

TEMA 4: ALGORITMOS VORACES

1. CARACTERÍSTICAS / EL PROBLEMA DE LA MOCHILA I

Devolver el cambio
Sistema monetario M: monedas de denominación (valor) 1, 2, 5, 10, 20, 50, 100, 200
Problema: pagar exactamente n unidades de valor con un mínimo de monedas:

```
const M = [1, 2, 5, 10, 20, 50, 100, 200]          {denominaciones}
función Devolver cambio (n) : conjunto de monedas
    S := conjunto vacío;             {la solución se construye en S}
    ss := 0;                              {suma de las monedas de S}
    mientras ss <> n hacer   {bucle voraz}
        x := mayor elemento de M : ss + x <= n;
        si no existe tal elemento entonces
            devolver "no hay soluciones";
        S := S U        {una moneda de valor x};
        ss := ss + x;
    fin mientras;
    devolver S
fin función
```

- ¿Por qué funciona? ⇒ M adecuado y número suficiente de monedas
- **No funciona con cualquier** *M*: Ejemplo: *M* = {1,4,6}, n = 8 → {6,1,1} en vez de {4,4}
  Este problema se resolverá con Programación Dinámica
- La función Devolver cambio es **voraz** (algoritmos ávidos, greedy)
  ¿Por qué voraz?
  - Selecciona el mejor candidato que puede en cada iteración, sin valorar consecuencias.
  - Una vez seleccionado un candidato, decide definitivamente:
    - aceptarlo, o
    - rechazarlo

    sin evaluación en profundidad de alternativas, sin retroceso...

  → Algoritmos sencillos tanto en su diseño como implementación. Cuando la técnica es adecuada, se obtienen algoritmos eficientes

Características de los algoritmos voraces

- Resuelven **problemas de optimización**:
  En cada fase, toman una decisión (selección de un candidato), satisfaciendo un óptimo local según la información disponible, esperando así, en conjunto, satisfacer un óptimo global.
- Manejan un **conjunto de candidatos** *C*:
  En cada fase, retiran el candidato seleccionado de *C*, y si es aceptado se incluye en S, el conjunto donde se construye la solución ≡ candidatos aceptados
- **4 funciones** (no todas aparecen explícitamente en el algoritmo):
  1. ¿S es **Solución**?
  2. ¿S es **Factible**? (¿nos lleva hacia una solución?)
  3. **Selección**: determina el mejor candidato
  4. **Objetivo**: valora S (está relacionada con Selección)

→ Encontrar S: Solución que optimiza Objetivo (max/min)

Esquema de los algoritmos voraces

```
función Voraz (C:conjunto): conjunto
    S := conjunto vacío;       {la solución se construye en S}
    mientras C <> conjunto vacío y no solución(S) hacer {bucle voraz}
        x := seleccionar(C);
        C := C-{x};
        si factible (SU{x}) entonces S := SU{x}
    fin mientras;
    si solución (S) entonces devolver S
    sino devolver "no hay soluciones"
fin función
```

- Diseño de un algoritmo voraz:
  1. adaptar el esquema al problema
  2. introducir mejoras (ejemplo: en Devolver cambio, añadir div)
- **Problema**: Asegurarse (demostrar) que la técnica funciona. No siempre funciona - ejemplo: "tomar la calle principal"

El problema de la mochila I

- $n\ objetos: i = 1 \dots n \begin{cases} peso\ w_i > 0 \\ valor\ v_i > 0 \end{cases}$

  Problema: cargar una mochila de capacidad W (unidades de peso), maximizando el valor de su carga.
- **Versión I**: los objetos se pueden fraccionar, y no se pierde valor ≡ fracción $x_i, 0 \leq x_i \leq 1$

  ⇒ el objeto i contribuye:

  en $x_i w_i$ al peso de la carga, limitado por *W*;

  en $x_i v_i$ al valor de la carga, que se quiere maximizar.

  ⇒ $\boxed{max\ \sum_{i=1}^{n} x_i v_i\ \ con\ la\ restricción\ \sum_{i=1}^{n} x_i w_i \leq W}$
- + Hipótesis: $\sum_{i=1}^{n} w_i > W$, sino la solucion es trivial

  ⇒ en óptimo, $\sum_{i=1}^{n} x_i w_i = W$

```
función Mochila 1 ( w[1..n], v[1..n], W): objetos[1..n]
    para i := 1 hasta n hacer
        x[i] := 0;        {la solución se construye en x}
    peso := 0;
    {bucle voraz:}
    mientras peso < W hacer
        i := el mejor objeto restante; {1}
        si peso+w[i] <= W entonces
            x[i] := 1;
            peso := peso+w[i]
        sino
            x[i] := (W-peso)/w[i];
            peso := W
        fin si
    fin mientras;
    devolver x
fin función
```

Ejemplo: mochila de capacidad *W* = 100 y 5 objetos:

	1	2	3	4	5	
v_i	20	30	66	40	60	
w_i	10	20	30	40	50	$(\sum_{i=1}^{n} w_i > W)$

¿Cuál es la función de Selección adecuada? (solo una es correcta!)

1. ¿Objeto más valioso? ↔ $v_i\ max$
2. ¿Objeto más ligero? ´ ↔ $w_i\ min$
3. ¿Objeto más rentable? ´ ↔ $v_i/w_i\ max$

Objetos	1	2	3	4	5	Objetivo $(\sum_{i=1}^{n} x_i v_i)$
v_i	20	30	66	40	60	
w_i	10	20	30	40	50	
v_i/w_i	2,0	1,5	2,2	1,0	1,2	
x_i (v_i max)	0	0	1	0,5	1	146
x_i (w_i min)	1	1	1	1	0	156
x_i (v_i/w_i max)	1	1	1	0	0,8	164

- **Teorema**: Si los objetos se seleccionan **por orden decreciente de** v_i/w_i , el algoritmo Mochila 1 encuentra la solución óptima. Demostración por absurdo.
- **Análisis**:
  inicialización: $\Theta(n)$;
  bucle voraz: $O(1) * n$ (peor caso) → $O(n)$
  + ordenación: $O(nlogn)$
- **Mejora**: con un montículo
  inicialización: + $O(n)$ (Crear montículo);
  bucle voraz: $O(logn)\ *\ n$ (peor caso) → $O(nlogn)$
  → pero mejores T(n)
- **Ejercicio**: pseudocódigo de ambas versiones

2. ORDENACIÓN TOPOLÓGICA

- **Definición**:
  Ordenación de los nodos de un grafo dirigido acíclico:
  ∃ camino $v_i, \ldots, v_j\ \Rightarrow\ v_j$ aparece despues de v_i
- **Aplicación**: sistema de prerrequisitos (llaves) en una titulación *(u,v)* ≡ *u* debe aprobarse antes de acceder a *v*
  → grafo acíclico, sino la ordenación no tiene sentido
- **Observación**: La ordenación topológica no es única.
- **Definición**: Grado de entrada de *v* = número de aristas *(u,v)*
- **Algoritmo**: en cada iteración, buscar nodo de grado 0, enviarlo a la salida y eliminarlo junto a las aristas que partan de él.
  + Hipótesis: el grafo ya está en memoria, listas de adyacencia
  $G = (N, A), |N| = n, |A| = m, 0 \leq m \leq n(n-1)$

```
función Ordenación topológica 1 (G:grafo): orden[1..n]
    Grado Entrada [1..n] := Calcular Grado Entrada (G);
    para i := 1 hasta n hacer Número Topológico [i] := 0;
    contador := 1;
    mientras contador <= n hacer
        v := Buscar nodo de grado 0 sin n´umero topol´ogico asignado; {*}
        si v no encontrado entonces
            devolver error "el grafo tiene un ciclo"
        sino
            Número Topológico [v] := contador;
            incrementar contador;
            para cada w adyacente a v hacer
                Grado Entrada [w] := Grado Entrada [w] - 1
        fin si
    fin mientras;
    devolver Número Topológico
fin función
```

- **Mejora**: estructura para nodos cuyo grado de entrada sea 0
  - {*} puede devolver cualquiera de ellos
  - al decrementar un grado, decidir si se incluye el nodo

  → pila o cola
- **Ejemplo**: evolución de Grado Entrada

nodo							
1	0						
2	1	0					
3	2	1	1	1	0		
4	3	2	1	0			
5	1	1	0				
6	3	3	3	3	2	1	0
7	2	2	2	1	0		
Insertar	1	2	5	4	3,7	-	6
Eliminar	1	2	5	4	3	7	6

```
función Ordenación topológica 2 (G:grafo): orden[1..n]
    Grado Entrada [1..n] := Calcular Grado Entrada (G);
  { para i := 1 hasta n hacer Número Topológico [i] := 0; }
    Crear Cola (C); contador := 1 ;
    para cada nodo v hacer
        si Grado Entrada [v] = 0 entonces Insertar Cola (v, C);
    mientras no Cola Vacía (C) hacer
        v := Eliminar Cola (C);
        Número Topológico [v] := contador; incrementar contador;
        para cada w adyacente a v hacer
            Grado Entrada [w] := Grado Entrada [w] - 1;
            si Grado Entrada [w] = 0 entonces Insertar Cola (w, C)
        fin para
    fin mientras;
    si contador <= n entonces devolver error "el grafo tiene un ciclo"
    sino devolver Número Topológico
fin función
```

- **Análisis**: $O(n+m)$ con listas de adyacencia
  Peor caso: grafo denso $[m \rightarrow n(n-1)]$ y visita todas las aristas
  Mejor caso: grafo disperso $[m \rightarrow 0, m \rightarrow n]$
- **Ejercicios**: ¿Calcular Grado Entrada (G) es O(n+ m)?
  Contrastar el algoritmo con la función voraz.

3. ÁRBOL EXPANDIDO MÍNIMO

- **a. e. m.**, árbol de expansión, árbol de recubrimiento mínimo
  Sea G = (N,A) conexo, no dirigido, pesos ≥ 0 en las aristas
- **Problema**: T subconjunto de A tal que G' = (N, T) conexo, peso (∑ pesos de T) mínimo y |T| mínimo.
- $|N| = n \Rightarrow |T| \geq n-1$;
  pero, si $|T| > n-1 \Rightarrow \exists$ ciclo
  → podemos quitar una arista del ciclo
  $\Rightarrow |T| = n-1 \wedge G'$ conexo ⇒ árbol (e. m.)
- **Aplicación**: instalación de cableado: ¿solución más económica?
- Técnica voraz:
  - Candidatos: aristas → S: conjunto de aristas
  - Solución?: S = T?
  - Factible?: (N,S) sin ciclos (ej: S vacío es Factible)

- **Definición**: una arista parte de un conjunto de nodos ⇔ uno de sus extremos esta en el conjunto (no parte ⇔ sus 2 extremos están dentro/fuera del conjunto)
- **Lema**: sean G = (N, A) un grafo conexo, no dirigido, pesado;
  B un subconjunto (estricto) de N;
  T un subconjunto (estricto) de A, Factible,
  sin aristas que partan de B;
  (*u, v*): la arista más corta que parte de B
  ⇒ TU{(*u, v*)} es Factible

- → Algoritmos de **Kruskal** y **Prim**

Algoritmo de Kruskal

- Inicialmente: T vacío
- Invariante: (N,T) define un conjunto de componentes conexas (i. e. subgrafos, arboles)
- Final: solo una componente conexa: el a. e. m.
- Selección: lema → **arista más corta...**
- Factible?: **...que una componentes conexas distintas**
- Estructuras de datos:
  - "grafo": aristas ordenadas por peso
  - árboles: Conjuntos Disjuntos (buscar(x), fusionar(A, B))

Ejemplo:

arista	(1,2)	(2,3)	(4,5)	(6,7)	(1,4)	(2,5)	(4,7)	(3,5)	...
peso	1	2	3	3	4	4	4	5	

paso	selección	componentes conexas
ini	-	1 2 3 4 5 6 7
1	(1,2)	1,2 3 4 5 6 7
2	(2,3)	1,2,3 4 5 6 7
3	(4,5)	1,2,3 4,5 6 7
4	(6,7)	1,2,3 4,5 6,7
5	(1,4)	1,2,3,4,5 6,7
6	(2,5) **rechazada**	1,2,3,4,5 6,7
7	(4,7)	1,2,3,4,5,6,7

```
función Kruskal ( G =(N,A) ) : árbol
    Ordenar A según longitudes crecientes;
    n := |N|;
    T := conjunto vacío;
    inicializar n conjuntos, cada uno con un nodo de N;
    {bucle voraz:}
    repetir
        a := (u,v) : arista más corta de A aún sin considerar;
        Conjunto U := Buscar (u);
        Conjunto V := Buscar (v);
        si Conjunto U <> Conjunto V entonces
            Fusionar (Conjunto U, Conjunto V);
            T := T U {a}
        fin si
    hasta |T| = n-1;
    devolver T
fin función
```

- **Teorema**: Kruskal calcula el árbol expandido mínimo. Demostración: inducción sobre |T|, utilizando el lema anterior
- **Análisis**: $|N| = n \wedge |A| = m$
  ordenar A: $O(mlogm) \equiv O(mlogn) : n-1 \leq m \leq n(n-1)/2$
  + inicializar n conjuntos disjuntos: $\Theta(n)$
  + 2m buscar (peor caso) y n−1 fusionar (siempre): $O(2m\alpha(2m,n)) = O(mlogn)$
  + resto: O(m) (peor caso)
  ⇒$\boxed{T(n) = O(mlogn)}$
- **Mejora**: utilizar un montículo de aristas en vez de ordenarlas. No cambia la complejidad del peor caso pero se obtienen mejores tiempos (ejercicio).

Algoritmo de Prim:

- Kruskal: bosque que crece hasta convertirse en el a. e. m.
  Prim: **un único árbol** que va creciendo hasta alcanzar todos los nodos.
- Inicialización: B = {nodo arbitrario} = {1}, *T* vacío
- Selección: arista más corta que parte de B:
  $(u, v), u \in B \wedge v \in N - B$
  ⇒ se añade (*u*, *v*) a *T* y *v* a *B*
- Invariante: T define en todo momento un a.e.m. del subgrafo (B,A)
- Final: B = N (Solución?)

```
función Prim 1 ( G =(N,A) ) : árbol
    T := conjunto vacío;
    B := un nodo de N;
    mientras B <> N hacer
        a := (u,v): arista más corta que parte de B
                    (u pertenece a B y v no);
        T := T U {a};
        B := B U {v}
    fin mientras;
    devolver T
fin función
```

- **Ejemplo** (el mismo que para Kruskal):

paso	selección	*B*
ini	-	1
1	(1,2)	1,2
2	(2,3)	1,2,3
3	(1,4)	1,2,3,4
4	(4,5)	1,2,3,4,5
5	(4,7)	1,2,3,4,5,7
6	(7,6)	1,2,3,4,5,6,7 = *N*

- **Observación**: No se producen rechazos.
- **Teorema**: Prim calcula el árbol expandido mínimo.
  Demostración: inducción sobre |T|, utilizando el lema anterior

- **Implementación**

$$L: matriz\ de\ adyacencia \equiv \mathrm{L[i,j]} = \begin{cases} distancia\ si\ \exists(\mathrm{i,j}) \\ \infty\ \text{sino} \end{cases}$$

→ matriz simétrica (≡ desperdicio de memoria)

Para cada nodo i ∈ N − B:
    Mas Próximo [i] : nodo ∈ B más próximo
    Distancia Mínima [i] : distancia de (i, Mas Próximo [i])
Para cada nodo i ∈ B:
    Distancia Mínima [i] = − 1

```
función Prim 2 ( L[1..n,1..n] ) : árbol
    Distancia Mínima [1] := -1;
    T := conjunto vacío;
    para i := 2 hasta n hacer
        Más Próximo [i] := 1;
        Distancia Mínima [i] := L[i,1]
    fin para;
    repetir n-1 veces: {bucle voraz}
        min := infinito;
        para j := 2 hasta n hacer
            si 0 <= Distancia Mínima [j] < min entonces
                min := Distancia Mínima [j];
                k := j
            fin si
        fin para;
        T := T U        { ( Más Próximo [k], k ) };
        Distancia Mínima [k] := -1;         {añadir k a B}
        para j := 2 hasta n hacer
            si L[j,k] < Distancia Mínima [j] entonces
                Distancia Mínima [j] := L[j,k];
                Más Próximo [j] := k
            fin si
        fin para
    fin repetir;
    devolver T
fin función
```

- **Análisis**:
  inicialización = Θ(n)
  bucle voraz: n−1 iteraciones, cada para anidado = Θ(n)
  $\Rightarrow T(n) = \Theta(n^2)$
- ¿Posible mejora con un montículo?
  → $O(mlogn)$, igual que Kruskal (ejercicio)

Continuación árbol expandido mínimo

- **Comparación:**

		Prim $\Theta(n^2)$	**Kruskal** $O(m\log n)$
Grafo denso:	$m \to n(n-1)/2$	$\Theta(n^2)$	$O(n^2 \log n)$
Grafo disperso:	$m \to n$	$\Theta(n^2)$	$O(n \log n)$

Tabla: Complejidad temporal de los algoritmos de Prim y Kruskal

- **Observaciones**: Existen algoritmos más eficientes, más sofisticados… más recientes (Kruskal es de 1956, Prim de 1957 [1930]). Son ejemplos importantes de aplicación de la técnica voraz.
- **Ejercicios**: Contrastar Prim y Kruskal con la función voraz. Completar Prim y Kruskal para que devuelvan el peso del a. e. m.

3. CAMINOS MÍNIMOS

- **Problema**: Dado un grafo G = (N, A) dirigido,
  con longitudes en las aristas ≥ 0,
  con un nodo distinguido como origen de los caminos (el nodo 1):
  → encontrar los caminos mínimos entre el nodo origen y los demás nodos de N
  ⇒ **algoritmo de Dijkstra**

- Técnica voraz:
  - 2 conjuntos de nodos:

$$\begin{cases} S \equiv \text{seleccionados: camino mínimo establecido} \\ C \equiv \text{candidatos: los demás} \end{cases}$$

  - invariante: N = SUC
  - inicialmente, S = 1 → final: S = N: función solución
  - Selección : nodo de C con menor distancia conocida desde 1 → existe una información provisional sobre distancias mínimas

- **Definición**: Un camino desde el origen a un nodo *v* es especial si todos sus nodos intermedios están en S.

- ⇒ D: vector con longitudes de caminos especiales mínimos;
  Selección de v ↔ **el camino especial mínimo** 1..v **es también camino mínimo** (se demuestra!).
  Al final, D contiene las longitudes de los caminos mínimos.

- **Implementación**:
  - N = 1, 2, ..., n (1 es el origen)
  - $\begin{cases} L[i,j] \geq 0 \text{ si } (i,j) \in A \\ = \infty \text{ sino} \end{cases}$ Matriz de adyacencia, no simétrica

Algoritmo de Dijkstra

```
función Dijkstra ( L[1..n,1..n] ) : vector[1..n]
    C := { 2, 3, ..., n};
    para i := 2 hasta n hacer
        D[i] := L[1,i];                              {1}
    {bucle voraz:}
    repetir n-2 veces:
        v := nodo de C que minimiza D[v];
        C := C-{v};
        para cada w en C hacer
            D[w] := min ( D[w], D[v]+L[v,w] )        {2}
    fin repetir;
    devolver D
fin función
```

- **Ejemplo**:

paso	selección	C	D[2]	D[3]	D[4]	D[5]
ini	-	2,3,4,5	50	30	100	10
1	5	2,3,4	50	30	20	10
2	4	2,3	40	30	20	10
3	3	2	35	30	20	10

Tabla: Evolución del conjunto *C* y de los caminos mínimos

- **Observación**: 3 iteraciones = n – 2

- ¿Calcular también los nodos intermedios?
  → vector P[2..n]:
  P[v] ≡ nodo que precede a v en el camino mínimo
  → Seguir precedentes hasta el origen

  ```
  {1} D[i] := L[1,i]; P[i] := 1

  {2} si D[w] > D[v]+L[v,w] entonces
          D[w] := D[v]+L[v,w];
          P[w] := v
      fin si
  ```

  … y devolver P junto con D

- **Teorema**: Dijkstra encuentra los caminos mínimos desde el origen hacia los demás nodos del grafo. Demostración por inducción.

- **Análisis**: $|N| = n, |A| = m, L[1..n, 1..n]$
  Inicialización= $\Theta(n)$
  ¿Selección de v?
  → "implementacion rápida": recorrido sobre C
  ≡ examinar $n-1, n-2, \ldots, 2$ valores en *D*, $\sum = \Theta(n^2)$
  Para anidado: $n-2, n-3, \ldots, 1$ iteraciones, $\sum = \Theta(n^2)$
  $\boxed{T(n) = \Theta(n^2)}$

- **Mejora**: si el grafo es disperso ($m << n^2$),
  utilizar listas de adyacencia
  → ahorro en para anidado: recorrer lista y no fila o columna de L

- **Análisis** (Cont.):
  ¿Como evitar $\Omega(n^2)$ en selección?
  → C: montículo min, ordenado según $D[i]$
  ⇒ inicialización en $O(n)$
  $C := C - v$ en $O(logn)$
  Para anidado: modificar $D[w] = O(logn) \equiv$ flotar
  ¿Nº de veces? Maximo 1 vez por arista (peor caso)
  En total:
  - extraer la raíz $n-2$ veces (siempre)
  - modificar un máximo de *m* veces un valor de *D* (peor caso)
  ⇒$\boxed{T(n) = O((m+n)logn)}$

- **Ejercicio**: escribir el pseudocódigo

- **Observación**: "implementación rápida" preferible si grafo denso

TEMA 5: DISEÑO DE ALGORITMOS POR INDUCCIÓN

1. DIVIDE Y VENCERÁS

- **Descomponer** el caso a resolver en subcasos del mismo problema, resolverlos, independientemente entre sí (recursivamente), y **combinar** las soluciones de los subcasos para obtener la solución del caso original.

- **Ejemplos vistos**: Suma de la Subsecuencia Máxima (función SSM recursiva), Mergesort, Quicksort, Búsqueda Binaria

- **Esquema para la técnica**: considerar
  - un problema (ejemplo: ordenación)
  - un algoritmo ad-hoc (capaz de resolver ese problema), sencillo y eficiente para casos pequeños del problema (ej: Inserción)

  → Esquema: **función Divide y Vencerás**

- **Ejercicio**: contrastarla con los ejemplos vistos

```
función Divide y Vencerás (x): solución
    si x es suficientemente pequeño entonces devolver ad-hoc(x)
    sino
        descomponer x en casos más pequeños x1, x2, ..., xa;
        para i := 1 hasta a hacer
            yi := Divide y Vencerás (xi)
        fin para;
        combinar los yi para obtener una solución y de x;
        devolver y
    fin si
fin función
```

- Características de *a* (nº de subcasos):
  - pequeño: 2 en Quicksort, Mergesort...
  - independiente de la entrada Caso particular: a = 1 ⇒ algoritmo de reducción

    → el paso de recursivo a iterativo supondrá un ahorro en tiempo (constante) y en espacio (pila de ejecución, Ω(n))

    Ejemplo: búsqueda binaria

- **Principales problemas**:
  - descomposición y combinación ¿posibles? ¿eficientes?
  - subcasos en lo posible del mismo tamaño: n/b,

    donde *b* es una constante distinta, a priori, de *a*

    → **importancia de balancear el tamaño de los subcasos**
  - ¿umbral a partir del cual hay que utilizar el algoritmo ad-hoc?

- **Análisis**:
  1. Reglas para el cálculo de la complejidad
     ⇒ relación de recurrencia
  2. ¿la ecuación de recurrencia cumple las condiciones **del teorema de resolución de recurrencias Divide y Vencerás**?
     Ejemplos:
     - en Mergesort, se puede utilizar el teorema en la demostración general a todos los casos
     - en Quicksort, solo se puede utilizar en el mejor caso (caso poco probable!)
- Diferenciar la técnica de diseño Divide y Vencerás del teorema de resolución de recurrencias que lleva su nombre

2. PROGRAMACIÓN DINÁMICA

Programación dinámica: Motivación

- **Divide y Vencerás** → riesgo de llegar a tener un gran número de subcasos idénticos ≡ ineficiencia!
- Ejemplo: **La sucesión de Fibonacci** [1202]
  - Leonardo de Pisa [1170-1240]
  - se define inductivamente del modo siguiente:

$$\begin{cases} \text{fib}(0) = 0 \\ \text{fib}(1) = 1 \\ \text{fib(n)} = \text{fib(n} - 1) + \text{fib(n} - 2) \text{ si n} \geq 2 \end{cases}$$

  ↔ sección áurea: $s1 \geq s2, s = s1 + s2$
  s1: segmento áureo de $s \equiv s_1^2 = s.s_2$
  ⇒ s2: segmento áureo de $s_1 \equiv s_2^2 = (s_1 - s_2)s_1$
  → ley de armonía (arquitectura, arte)...

Fibonacci 1

Algoritmo Fibonacci 1:

```
función fib1(n): valor
    si n < 2 entonces devolver n
    sino devolver fib1(n-1) + fib1(n-2)
    fin si
fin función
```

- $\boxed{T(n) = \Theta(\Phi^n)}$, donde $\Phi = (1 + \sqrt{5})/2$ (Cf. resolución de recurrencias)
- fib1(5) produce 3 llamadas a fib1(0), 5 llamadas a fib1(1),... en total, 15 llamadas

Programación dinámica

- **Programación Dinámica**: resolver cada subcaso una sola vez, guardando las soluciones en una tabla de resultados, que se va completando hasta alcanzar la solución buscada.
  ⇒ Técnica ascendente, opuesta a la descendente de Divide y Vencerás

- Ejemplo: **Algoritmo Fibonacci 2**

```
función fib2(n): valor
    i := 1; j := 0;
    para k := 1 hasta n hacer
        j := i+j; i := j-i
    fin para;
    devolver j
fin función
```

- $\boxed{T(n) = \Theta(n)}$ y espacio en $\Theta(1)$

Fibonacci 3

Algoritmo Fibonacci 3: $\boxed{\text{T(n) = O(logn)}}$

```
función fib3(n): valor
    i := 1; j := 0; k := 0; h := 1;
    mientras n > 0 hacer
        si n es impar entonces
            t := j*h;
            j := i*h + j*k + t;
            i := i*k + t
        fin si;
        t := h^2;
        h := 2*k*h + t;
        k := k^2 + t;
        n := n div 2
    fin mientras;
    devolver j
fin función
```

2.1. Coeficientes Binomiales

- $\binom{n}{k} = \begin{cases} 1 \text{ si } k=0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} \text{ si } 0 < k < n \\ 0 \text{ sino} \end{cases}$

- Ejemplo: Teorema de Newton

$$(1+x)^n = 1 + \binom{n}{1}x + \binom{n}{2}x^2 + \cdots + \binom{n}{n-1}x^{n-1} + x^n$$

- **Problema**: Dados $0 \leq k \leq n \rightarrow$ ¿ $\binom{n}{k}$?

```
función C(n, k): valor
    si k = 0 ó k = n entonces devolver 1
    sino devolver C(n-1, k-1) + C(n-1, k)
    fin si
fin función
```

Divide y Vencerás: muchos cálculos se repiten ≡ suma de 1's
(como en fib1) → $\Omega\left(\binom{n}{k}\right)$

- **Programacion Dinámica**:
→ Tabla de resultados intermedios: triángulo de Pascal

	0	1	2	...	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
...						
$n-1$					$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n						$\binom{n}{k}$

- $\boxed{T(n) = \Theta(nk)}$ y la complejidad espacial también
- ¿Mejora? La complejidad espacial puede ser Θ(k)
  ↔ manejar una sola línea del triángulo de Pascal
- **Ejercicio**: escribir el pseudocódigo

2.2. Devolver el cambio

- **Problema**: el algoritmo voraz es eficiente pero no "funciona" siempre:
$M = \{1,4,6\}, n = 8 \Rightarrow ¿S = \{4,4\}?$
- Dado $M = \{v_1, v_2, \ldots, v_m\}, v_i > 0$: denominaciones de monedas
Objetivo: pagar exactamente n unidades de valor, con |S| mínimo
Hipótesis: ∃ suministro ilimitado de monedas
- Programación Dinámica ↔ tabla $c[1..m, 0..n]$
$c[i,j]$ = nº mínimo de monedas para pagar j $(0 \leq j \leq n)$ utilizando monedas de denominación $v_1 \ldots v_i$ $(1 \leq i \leq m)$
- $|S| = c[m,n]$
- Construcción de la tabla:
$c[i,0] = 0$
$c[i,j]$

$$\min \begin{cases} c[i-1,j]: no\ utilizar\ una\ moneda\ más\ de\ v_i \leftrightarrow \text{i} > 1 \\ 1 + c[i, j - v_i]: \text{utilizar una moneda mas de ´ vi} \leftrightarrow \text{j} \geq \text{v} \end{cases}$$

Caso particular: $i = 1 \wedge j < v_1 \Rightarrow c[i,j] = +\infty$ ≡ no hay solución

```
función monedas (n): número de monedas
    const v[1..m]=[1,4,6]          {denominaciones de las monedas}
    {se construye una tabla c[1..m, 0..n]}
    para i := 1 hasta m hacer c [i,0] := 0;
    para i := 1 hasta m hacer
        para j := 1 hasta n hacer
            si i = 1 y j < v[i] entonces c[1,j] := infinito
            sino si i = 1 entonces c[1,j] := 1 + c[1, j-v[1] ]
            sino si j < v[i] entonces c[i,j] := c[i-1,j]
            sino c[i,j] := min ( c[i-1, j], 1 + c[i, j-v[i] ] )
            fin si
        fin para
    fin para;
    devolver c[m,n]
fin función
```

- **Ejemplo**: $M = \{1,4,6\}$ ¿$c[3,8]$?

n	0	1	2	3	4	5	6	7	8
$\{1\}$	0	1	2	3	4	5	6	7	8
$\{1,4\}$	0	1	2	3	1	2	3	4	2
$\{1,4,6\}$	0	1	2	3	1	2	1	2	2

- **Análisis**: $\boxed{T(n) = \Theta(mn)}$

- **Problema**: ¿Conjunto de monedas?
  ⇒ Algoritmo voraz sobre c: camino $c[m,n] \rightarrow c[0,0]$
  m "pasos" hacia arriba ≡ "no utilizar más´ vi "
  $+c[m,n]$ "saltos" hacia la izquierda ≡ "utilizar una vi más"
  ⇒ $\boxed{\Theta(m + c[m,n])}$ adicional a la construcción de la tabla

2.3. El Problema de la Mochila II

- Versión II: los objetos **no** se pueden fraccionar

$$\equiv xi \in \begin{cases} 0 \equiv \text{dejar} \\ 1 \equiv \text{tomar} \end{cases}$$

- ¿Qué pasa con el algoritmo voraz? Ejemplo: $n = 3$, $W = 9$

Objetos	1	2	3	
v_i	9	7	5	
w_i	6	5	4	
v_i/w_i	1,5	1,4	1,25	Objetivo ($\sum_{i=1}^{n} x_i v_i$)
x_i (voraz)	1	0	0	9
x_i (óptimo)	0	1	1	12

⇒ ¡Ha dejado de funcionar!

- **Programación Dinámica** ↔ tabla $v[1..n, 0..W]$
  $v[i,j]$ = valor de carga máxima para capacidad j $(0 \leq j \leq W)$ considerando los objetos $1..i$ $(1 \leq i \leq n)$.

- Construcción de la tabla:

$$v[i,j] = max\begin{cases} v[i-1,j] : \text{ no añadir el objeto i} \\ v[i-1, j-w_i] + v_i : \text{ añadir el objeto i} \end{cases}$$

- **Observación**: a diferencia del caso de las monedas, cada objeto solo se puede incluir "una vez" en la carga de la mochila.

- **Ejercicio**: ¿algoritmo?

- **Ejemplo**:

	v_i	w_i	0	1	2	3	4	5	6	7	8	9
$\{1\}$	9	6	**0**	0	0	0	0	0	9	9	9	9
$\{1,2\}$	7	5	0	0	0	0	0	**7**	9	9	9	9
$\{1,2,3\}$	5	4	0	0	0	0	5	7	9	9	9	**12**

- **Análisis**: $\boxed{T(n) = \Theta(nW)}$

- **Problema**: ¿Composición de la carga?
  ⇒ Recorrido sobre *v*: camino $v[n, W] \rightarrow v[0,0]$
  máximo n "pasos" hacia arriba ≡ "no incluir el objeto i"
  + máximo ´ W "saltos" hacia arriba y a la izquierda ≡ "incluir el objeto i"
  ⇒ $\boxed{\Theta(n+W)}$: trabajo adicional a la construcción de la tabla

2.4. Conclusión

- **Principio de optimalidad**: La Programación Dinámica se utiliza para resolver problemas de optimización que satisfacen el principio de optimalidad: "En una secuencia optima de decisiones toda subsecuencia ha de ser también optima"
- ¡No siempre es aplicable! Ejemplo: hallar el camino simple más largo entre dos nodos.

TEMA 6: EXPLORACIÓN DE GRAFOS

1. RECORRIDOS SOBRE GRAFOS

1.1. Recorrido en profundidad

- Recursividad
- $G = (N, A)$: grafo no dirigido → recorrido a partir de cualquier $v \in N$:

```
procedimiento rp (v)          {v no ha sido visitado anteriormente}
    marca[v] := visitado;
    para cada w adyacente a v hacer
        si marca[w] != visitado entonces rp(w)
fin procedimiento
```

- El recorrido en profundidad asocia a un grafo conexo un árbol de recubrimiento
- **Análisis**: n nodos, m aristas
  - se visitan todos los nodos (n llamadas recursivas) ⇒ $\Theta(n)$
  - en cada visita se inspeccionan todos los adyacentes ⇒ $\Theta(m)$

$$\boxed{T(n) = \Theta(n + m)}$$

- Ejemplo: **numerar en preorden** → Añadir al principio de rp:

```
visita := visita+1;
preorden[v] := visita;
```

- Para recorrer completamente el grafo (incluso no conexo):

```
procedimiento recorridop (G=(N,A))
    para cada nodo v hacer marca[v] := no visitado;
    para cada nodo v hacer
        si marca[v] != visitado entonces rp(v)
fin procedimiento
```

- Grafo dirigido: la diferencia está en la definición de *adyacente*...
  árbol → "bosque de recubrimiento"

- Ejemplo: **ordenación topológica** (grafo dirigido acíclico)
  → Añadir al final de rp:

```
escribir(v);
```

  e invertir el resultado

- Versión no recursiva: usa una Pila

```
procedimiento rp2 (v)
    Crear Pila (P); marca[v] := visitado; Apilar(v,P);
    mientras no Pila Vacía (P) hacer
        mientras exista w adyacente a Cima(P): marca[w] != visitado
        hacer
            marca[w] := visitado; Apilar (w,P)
        fin mientras;
        Desapilar (P)
    fin mientras
fin procedimiento
```

1.2. Recorrido en anchura

- Diferencia: al llegar a un nodo v, primero visita todos los nodos adyacentes → no es "naturalmente recursivo"

```
procedimiento ra (v)
    Crear Cola (C); marca[v] := visitado; Insertar Cola (v,C);
    mientras no Cola Vacía (C) hacer
        u := Eliminar Cola (C);
        para cada nodo w adyacente a u hacer
            si marca[w] != visitado entonces
                marca[w] := visitado; Insertar Cola (w,C)
            fin si
        fin para
    fin mientras
fin procedimiento
```

- Grafo conexo → árbol de recubrimiento
- $T(n) = \Theta(n+ m)$
- Exploración parcial de grafos, camino más corto entre 2 nodos...

2. JUEGOS DE ESTRATEGIA

- Ejemplo: variante del **Juego de Nim**
  - Un montón de *n* palillos
  - 2 jugadores, alternativamente
  - 1ª jugada: coger [*1..n−1*] palillos
  - jugadas siguientes: coger [*1..2* ∗ jugada anterior] palillos
  - Objetivo: coger el ultimo palillo

- Grafo:

$$\begin{cases} \text{nodo} \leftrightarrow \text{situación: } < \text{quedan i palillos, se pueden coger j} > \\ \text{arista} \leftrightarrow \text{jugada: nª de palillos} \end{cases}$$

- Ejemplo: desde i = 5, j = 4 (i.e. jugada anterior = 2) y juego yo

- Marcado de nodos:
  - 2 tipos de nodos:

$$\begin{cases} \text{situación de derrota} \\ \text{situación de victoria} \equiv \text{“victoria segura si juego bien”} \end{cases}$$

  - Situación final (de derrota): < 0,0 >

- Marcado de jugadas:
  Distinguir las jugadas de victoria

- Reglas de marcado: desde < 0,0 >
  - marcar situación de victoria si ∃ sucesor situación de derrota
  - marcar situación de derrota si todos los sucesores son situación de victoria

TEMA 7: COMPLEJIDAD COMPUTACIONAL

1. INTRODUCCIÓN. *P* Y *NP*

1.1. Algoritmia y complejidad computacional

Algoritmia: diseño de algoritmos específicos, lo más eficientes posibles, para un problema dado

- Ejemplo: la ordenación por selección es $O(n^2)$, la ordenación por montículos es $O(nlog(n))$.

Complejidad computacional: estudio del problema, considerando globalmente todos los algoritmos posibles para resolverlo (¡incluso los que no conocemos!)

- Ejemplo: ¿cuál es la mejor complejidad temporal que podemos conseguir para ordenar un array de tamaño n?
- Sabemos que podemos hacerlo en $O(nlog(n))$; pero ¿podemos hacerlo más rápido?
- Si la respuesta es ”sí”, podemos demostrarlo encontrando un algoritmo más rápido. Pero si es ”no”, buscar algoritmos mejores sería perder el tiempo. Necesitamos razonar sobre el problema en su conjunto, más allá de algoritmos concretos → complejidad computacional.

1.2. Problemas tratables e intratables

Nos centraremos en un aspecto dado: qué problemas son tratables en la práctica.

Se considera que un problema es tratable si se puede resolver en tiempo polinómico $O(n^k)$ para algún *k*. El conjunto de estos problemas se llama clase de complejidad *P*.

- Por ejemplo, sabemos que multiplicar dos números, ordenar una lista de valores, buscar el camino mínimo entre dos ciudades en un mapa, están en *P*.

Para otros problemas, no conocemos algoritmos en tiempo polinómico. Ejemplo: problema de la suma de subconjuntos.

- Dado un conjunto de n enteros (por ejemplo, $\{4, -2, 6, 3, 1, -28, 10, -17\}$). ¿Existe un subconjunto cuya suma sea 0?
- En este caso, sí: $4 + 3 + 10 + (-17) = 0$.

```
función Suma_Subconjuntos({i1,...,in}) : bool
    visto := false
    para cada subconjunto {x1,...,xk } ⊆ {i1,...,in} {O(n2^n)}!
        si x1+x2+...+xk = 0 entonces visto := true {O(n)}
    fin para;
    devolver visto
fin función
```

Para este problema no se conoce algoritmo polinómico → no sabemos si está en *P*.

- En teoría podría estar: si existe algoritmo polinómico pero no se ha descubierto (improbable).

Este problema pertenece a una clase de complejidad llamada *NP*

- Ojo: nada que ver con ”no polinómico”, sino con ”non-deterministic polynomial time” (veremos qué significa)

1.3. EL problema de *P* y *NP*

Informalmente, *NP* es el conjunto de problemas cuya solución se puede comprobar en tiempo polinómico.

- $P \subseteq NP$: si tenemos un algoritmo polinómico para averiguar la solución, podemos usarlo para comprobarla (ej. suma de la subsecuencia máxima).
- Pero hay problemas cuya solución es fácil de comprobar (*NP*) pero difícil de obtener.
    - Suma de subconjuntos: podemos comprobar si un subconjunto es solución sumando sus elementos ($O(n)$) pero encontrar nosotros la solución es otra historia...
- La "pregunta del millón" (literalmente) es si todos estos problemas están también en *P* o no, es decir, si *P* = *NP* o no.

- Se cree que *P* ≠ *NP*; pero hasta ahora no se ha logrado demostrarlo.
- Si efectivamente *P* ≠ *NP*, nunca encontraremos soluciones polinómicas al problema de las sumas de subconjuntos (y otros problemas prácticos que están en *NP*), porque sabremos que no existen (veremos por qué).
- Si *P* = *NP*, existirían soluciones polinómicas al problema de las sumas de subconjuntos y todos los demás problemas de N P.

2. MÁQUINAS DE TURING

Para estudiar la complejidad de los problemas se suele usar un modelo abstracto de computador llamado máquina de Turing:

- Es muy sencillo...
- ...pero tiene la misma capacidad de cómputo que otros sistemas mucho más complejos (todo lo que se puede computar en un lenguaje de programación como Pascal o C se puede computar en una máquina de Turing).

Una máquina de Turing es una máquina que manipula símbolos en una cinta según una tabla de reglas. Se compone de:

- Una cinta infinitamente larga (por la derecha) donde se pueden leer/escribir símbolos,
- Una cabeza lectora/escritora que apunta a una posición de la cinta y se puede mover,
- Un control de estado finito, es decir:
    - Un registro almacena el estado en que está la máquina en un momento dado,
    - La máquina utiliza el estado actual, y el símbolo leído de la cinta, para decidir qué hacer, de acuerdo con unas reglas.

Cada una de las reglas es de la siguiente forma: "si el estado actual es q0 y el símbolo leído por la cabeza lectora es s0, entonces escribir el símbolo s1 en la cinta, mover la cabeza lectora hacia [izquierda| derecha], y pasar al estado q1".
Según las reglas que fijemos, podemos conseguir que la máquina de Turing calcule distintas funciones.
Con este mecanismo tan simple podemos computar cualquier función que podamos computar en un lenguaje de programación (C, Java, etc.) Hay distintas variantes, todas equivalentes (cinta infinita hacia los dos lados, varias cintas, posibilidad de que la cabeza se quede en el sitio, etc.)

Ejemplo 1:

Determinar si la longitud de una cadena es par o impar.

- Estados: $\{q_0, q_1, q_P^f, q_I^f\}$
- Alfabeto (símbolos que pueden aparecer en la cinta): {0, 1, *b*}
    - A *b* lo llamaremos el símbolo en blanco. Consideraremos que la cinta siempre empieza por un símbolo en blanco (marca de inicio), a continuación tiene la cadena que se quiere procesar, y el resto de la cinta está relleno de símbolos en blanco. Suponemos también que la cabeza lectora empieza sobre el primer símbolo de la cadena (sin contar el símbolo en blanco que marca el inicio).

Determinar si la longitud de una cadena es par o impar. Reglas:

Si el estado es	y leemos un	: escribir un	, pasar al estado	y mover hacia
q_0	0	0	q_1	$\rightarrow$
q_0	1	1	q_1	$\rightarrow$
q_1	0	0	q_0	$\rightarrow$
q_1	1	1	q_0	$\rightarrow$
q_0	*b*	*b*	q_P^f	$=$
q_1	*b*	*b*	q_I^f	$=$

- Esta máquina indica si la cadena es par o impar según si termina su ejecución en q_P^f (estado de aceptación) o q_I^f (al no haber reglas para estos estados, se termina la ejecución al llegar a ellos).
- Equivalentemente, podríamos borrar la cadena de la cinta y escribir 0 ´o 1 para indicar par o impar.

Ejemplo 2:

Determinar si una cadena es un palíndromo (se lee igual al derecho que al revés)

- Estados: {Lee1o, Leido0, Leido1, Comprobar0, Comprobar1, Volver, Aceptar}
- Alfabeto (símbolos que pueden aparecer en la cinta): {0, 1, b}

Determinar si una cadena es un palíndromo. Reglas:

Si el estado es	y leemos un	: escribir un	, pasar al estado	y mover hacia
Lee1o	0	*b*	*RecienLeido0*	$\rightarrow$
Lee1o	1	*b*	*RecienLeido1*	$\rightarrow$
Lee1o	*b*	*b*	*Aceptar*	$=$
RecienLeido0	0/1	0/1	*Leido0*	$\rightarrow$
RecienLeido1	0/1	0/1	*Leido1*	$\rightarrow$
RecienLeido0	*b*	*b*	*Aceptar*	$=$
RecienLeido1	*b*	*b*	*Aceptar*	$=$
Leido0	0/1	0/1	*Leido0*	$\rightarrow$
Leido1	0/1	0/1	*Leido1*	$\rightarrow$
Leido0	*b*	*b*	*Comprobar0*	$\leftarrow$
Leido1	*b*	*b*	*Comprobar1*	$\leftarrow$
Comprobar0	0	*b*	*Volver*	$\leftarrow$
Comprobar1	1	*b*	*Volver*	$\leftarrow$
Volver	0/1	0/1	*Volver*	$\leftarrow$
Volver	*b*	*b*	*Leer1o*	$\rightarrow$

- Si la cadena es un palíndromo, terminamos en el estado Aceptar.

Estas máquinas de Turing reconocen determinados lenguajes, es decir, conjuntos de cadenas.

- Lenguaje formado por las cadenas de longitud par.
- Lenguaje formado por los palíndromos.

Estructuras de datos como arrays, árboles, grafos... también se pueden representar como cadenas, así que las máquinas de Turing se pueden usar en problemas con estructuras de datos complejas.

Todos los problemas con respuesta sí/no se pueden ver como problemas de determinar si una cadena pertenece a un lenguaje (p.ej. el conjunto de las cadenas que representan conjuntos de enteros que tienen un subconjunto que suma 0).

2.1. *P* y *NP* con Máquinas de Turing

La definición original de *P* y *NP* es:

- Clase de complejidad *P*: Conjunto de lenguajes (o sea, problemas sí/no) que se pueden resolver con una máquina de Turing que termina su ejecución en un número de pasos acotado por una función polinómica, $O(n^k)$.
- Clase de complejidad *NP*: Conjunto de lenguajes (o sea, problemas sí/no) que se pueden resolver con una máquina de Turing no determinista que termina su ejecución en un número de pasos acotado por una función polinómica, $O(n^k)$.

Una máquina de Turing no determinista es una que puede tomar varios caminos a la vez ante una situación dada:

Si el estado es	y leemos un	: escribir un	, pasar al estado	y mover hacia
q_1	0	q_2	0	$\rightarrow$
q_1	0	q_3	1	$\leftarrow$

Todo lo que se puede computar con una máquina de Turing no determinista se puede hacer también en una determinista...

- ...pero no con la misma eficiencia, por supuesto: de ahí que pueda haber problemas que están en *NP* pero no en *P*.

Los ordenadores que nosotros manejamos son deterministas.

- Lo análogo a una máquina de Turing no determinista "en la vida real" sería un procesador con una cantidad infinita de núcleos. Por supuesto, no existe (si existiera, sería fácil escribir un algoritmo multihilo que resolviese el problema de la suma de subconjuntos en tiempo polinómico).

3. *NP*-COMPLETITUD

Problemas *NP*-completos

Existe un subconjunto de problemas en *NP*, llamados problemas *NP*–completos, que tienen la siguiente curiosa propiedad:

- Si cualquiera de ellos está en *P*, entonces todos los problemas de *NP* están en *P*. Es decir, bastaría con encontrar un algoritmo polinómico para uno solo de ellos para demostrar que *P* = *NP*.
- De lo anterior se deduce que, si *P* ≠ *NP*, ninguno de estos problemas puede estar en *P*. Es decir, si se demostrara que *P* ≠ *NP*, sabríamos de un golpe que para todos estos

problemas (que son muchos) nunca encontraremos una solución polinómica. (y, dado que se han encontrado varios miles de problemas *NP*-completos y muchos de ellos tienen gran relevancia práctica, esto hace ver por qué el problema de *P* y *NP* se considera uno de los "problemas del milenio"...) ¿Cómo se ha llegado a estas conclusiones?

Reducibilidad

Decimos que un problema A es reducible en tiempo polinómico a un problema B si, dado un algoritmo que resuelve B, podemos obtener un algoritmo que resuelve A añadiendo, como mucho, un factor polinómico a su complejidad.

- Intuitivamente: construimos un algoritmo para A haciendo una llamada a función al algoritmo para B.
- Esto quiere decir que, en términos de complejidad computacional, A es al menos tan "fácil" como B (por ejemplo, si B estaba en *P*, A también estará en *P*).

Un problema es *NP*-completo si está en *NP* y cualquier problema en *NP* es reducible a él.

- Por eso si un problema *NP*-completo estuviera en *P*, todos los problemas de *NP* estarían en *P* (serían reducibles a él).

Ejemplos de problemas *NP*-completos

Algunos ejemplos de problemas *NP*-completos:

- Suma de subconjuntos.
- Problema del viajante: recorrer todas las ciudades de un mapa sin repetir ninguna, y volviendo al punto de partida.
- Determinar si los países de un mapa se pueden colorear con k colores (k > 4) sin que países adyacentes tengan el mismo color.
- Dada una lista de personas y sus enemistades, ¿cuál es el máximo número de personas que podemos invitar a una boda sin que nadie se lleve mal con nadie?
- Satisfacibilidad booleana: determinar si una expresión de lógica proposicional (como las condiciones de un if) es verdadera para alguna combinación de valores de las variables
- Resolver un sudoku.
- Etc...

Más sobre el problema de *P* y *NP*

Según lo que hemos visto, bastaría con encontrar una solución polinómica a uno cualquiera de estos problemas *NP*-completos para demostrar que *P* = *NP*.

- Que tras tantos años intentándolo con tantos problemas distintos, no se haya logrado, es una de las razones por las que muchos investigadores piensan que *P* ≠ *NP*.
- Otro argumento es más filosófico: *"Si P = NP, el mundo sería muy diferente de como normalmente suponemos que es. Los "saltos creativos" no tendrían ningún valor especial, no habría diferencias importantes entre resolver un problema y reconocer la solución una vez encontrada. Cualquiera que pudiese apreciar una sinfonía sería Mozart, cualquiera que pudiese entender una demostración paso a paso sería Gauss, cualquiera que pudiese reconocer una buena estrategia de inversión sería Warren Buffett..."* (Scott Aaronson, MIT)