Diseño y Análisis de Algoritmos

Ana X. Ezquerro

ana.ezquerro@udc.es, GitHub

Grado en Ciencia e Ingeniería de Datos
Universidad de A Coruña (UDC)

Curso 2020-2021

Tabla de Contenidos

Bloque I
Análisis de Algoritmos

Tema 1: Análisis de Algoritmos

1.1. Análisis de la eficacia de los algoritmos

El objetivo del análisis de la eficiencia de los algoritmos es **predecir el comportamiento** de un algoritmo en aspectos cuantitativos (tiempo de ejecución, cantidad de memoria usada, etc). Para llevar a cabo este análisis, necesitamos disponer una **medida de la eficiencia de los algoritmos** con las siguientes características:

- Que sea una **medida teórica**.
- Que no sea una medida exacta, sino una **aproximación** suficiente para comparar y clasificar.
  - Ignorar factores constantes.
  - Ignorar términos de orden inferior.
- Que describa el **comportamiento asintótico** del tiempo de ejecución $T(n)$, donde n es el tamaño del problema y cuando n tiende a infinito.
- Se escribe como "$T(n) = O(f(n))$", donde $f(n)$ es una cota superior de $T(n)$ (crece más deprisa que $T(n)$).

Para realizar esta **aproximación**, vamos a seguir los siguientes criterios:

- Ignorar factores constantes.
- Ignorar términos de orden inferior.

Tasas de crecimiento características: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, ..., $O(2^n)$.

1.2. Notaciones asintóticas: $f(n) : \mathbb{Z}^+ \longrightarrow \mathbb{R}^+$

Notación O

$$T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ tal que } T(n) \leq cf(n), \ \forall n \geq n_0$$

- n_0 es el umbral.
- La tasa de crecimiento de $T(n)$ es menor o igual que la de $f(n)$.
- $f(n)$ es cota superior de $T(n)$.
- $f(n)$ es monótona creciente $\iff n_1 \geq n_2 \Longrightarrow f(n_1) \geq f(n_2)$

Teoremas para la notación O

Sea $f(n)$ una cota superior monótona creciente, $\forall c > 0, a > 1$, se cumple que:

- $f(n)^c = O(a^{f(n)})$
- Una función exponencial (a^n) crece más rápido que una función polinómica (n^c): $n^c = O(a^n)$
- $(\log n)^k = O(n), \forall k \in \mathbb{R}$
- n crece más rápido que cualquier potencia de logaritmo.
- Los logaritmos crecen muy lentamente.

Suponemos $T_1(n) = O(f(n))$ y $T_2(n) = O(g(n))$.

- $T_1(n) + T_2(n) = O\big(f(n) + g(n)\big) = \text{máx}\left(O(f(n)), O(g(n))\right)$
- $T_1(n)T_2(n) = O(f(n)g(n))$

1.3. Otras notaciones

Notación Ω

Se dice que $f(n)$ es una **cota inferior** de $T(n)$ (trabajo mínimo del algoritmo):

$$T(n) = \Omega(f(n)) \iff \exists c, n_0 \in \mathbb{R} \text{ tal que } T(n) \geq cf(n), \ \forall n \geq n_0$$

Notación Θ

Se dice que $f(n)$ es **cota exacta** de $T(n)$ (orden exacto):

$$T(n) = \Theta(f(n)) \iff \exists c_1, c_2, n_0 \in \mathbb{R} \text{ tal que } c_1 f(n) \leq T(n) \leq c_2 f(n), \ \forall n \geq n_0$$

Notación o

Se dice que $f(n)$ es una **cota estrictamente superior** de $T(n)$:

$$T(n) = o(f(n)) \iff \forall C > 0, \exists n_0 > 0 \text{ tal que } T(n) < Cf(n), \ \forall n \geq n_0$$

Expresión equivalente:

$$\begin{aligned} T(n) = o(f(n)) &\equiv T(n) = O(f(n)) \wedge \neg\Theta(f(n)) \\ &\equiv T(n) = O(f(n)) \wedge \neg\Omega(f(n)) \equiv \lim_{n\to 0} \frac{T(n)}{f(n)} = 0 \end{aligned}$$

Notación ω

Se dice que $f(n)$ es una **cota estrictamente inferior** de $T(n)$:

$$T(n) = \omega(f(n)) \iff \forall C > 0, \exists n_0 > 0 \text{ tal que } T(n) > cf(n), \ \forall n \geq n_0$$

Notación OO [Manber]

$$T(n) = OO(f(n)) \iff T(n) = O(f(n)) \text{ pero con constantes demasiado grandes para casos prácticos}$$

1.4. Cálculo de los tiempos de ejecución

El tiempo de ejecución $T(n)$ se puede abstraer como el número de "pasos"que se ejecutan en el algoritmo. Un *paso* lo podemos definir como:

- Una **operación elemental** cuyo tiempo de ejecución está acotado por una constante que no depende de n.
- Una **operación principal**, que es una operación representativa del trabajo del algoritmo.

En cualquier caso, se da por supuestas las siguientes condiciones:

- Memoria infinita.
- Independencia del lenguaje de programación.
- Independencia de las prestaciones del computador.

Para determinar la O para algoritmos, debemos tener en cuenta el tiempo de ejecución $T(n)$ que depende de la entrada. Debido a esta dependencia de la entrada, no existe una única función que defina el tiempo de ejecución, sino varios. Vamos a considerar tres:

$$\begin{cases} T_{\text{mejor}}(n) & \\ T_{\text{medio}}(n) & \text{representativa} \\ T_{\text{peor}}(n) & \text{general, más usada} \end{cases} \qquad T_{\text{mejor}}(n) \leq T_{\text{medio}}(n) \leq T_{\text{peor}}(n)$$

1.5. Ejemplos. Ordenación

```
function InsertionSort (var T[1..n]):
    for i:=2 to n:
        x := T[i];
        j := i-1;
        while j > 0 and T[j] > x:
            T[j+1] := T[j];
            j := j-1;
        T[j+1]:= x;
```

```
function SelectionSort (var T[1..n])
    for i:=1 to n-1:
        minj := i;
        minx := T[i];
        for j:= i+1 to n:
            if T[j] < minx:
                minj := j;
                minx := T[j];
        T[minj] := T[i];
        T[i] := minx;
```

Ordenación por inserción (`InsertionSort`)

Análisis del peor caso:

- Los elementos de la entrada están en orden inverso.
- En el bucle interno siempre hay que insertar un elemento en la primera posición.
- El bucle interno se ejecuta 1 vez en la primera iteración, 2 veces en la segunda,..., $n-1$ veces en la última. Número total de iteracciones:

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

- Tiempo de ejecución total:

$$T(n) = c_1\frac{n(n-1)}{2} + c_2(n-1) + c_3 \Longrightarrow T(n) = \Theta(n^2)$$

Análisis del mejor caso:

- Entrada ordenada, no insertar nunca.
- El bucle interno no se ejecuta.
- Tiempo de ejecución: $T(n) = c_1(n-1) + c_2 \Longrightarrow T(n) = \Theta(n)$.

Ordenación por selección (`SelectionSort`)

- Bucle interno:

$$\Theta(n-i) = \begin{cases} i = 1 & \Theta(n) \\ i = n-1 & \Theta(1) \end{cases}$$

- Bucle externo + interno:

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

- $T(n) = \Theta(n^2)$ sin importar la entrada.
- La comparación interna se ejecuta las mismas veces.

Comparativa

	Peor caso	Caso medio	Mejor caso
Inserción	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Selección	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Quicksort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

1.6. Exponenciación (potencia recursiva)

Existen dos formas de calcular la potencia de un número x elevado a otro número n:

- Potencia 1: $x \cdot x \cdot ... \cdot x = x^n$
  - $n-1$ multiplicaciones.
  - $f(n) = n-1 \Longrightarrow T(n) = \Theta(n)$
- Potencia 2 (recursiva):

$$x^n = \begin{cases} x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} & \text{si } n \text{ par} \\ x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} \cdot x & \text{si } n \text{ impar} \end{cases}$$

  - Mejor caso: $n = 2^k, \ \ k \in \mathbb{Z}^+ \Longrightarrow n$ es par en cada llamada recursiva.
  - Peor caso: $n = 2^k + 1, \ \ k \in \mathbb{Z}^+ \Longrightarrow n$ es impar en cada llamada recursiva.

Veremos a continuación cuál es la complejidad del algoritmo recursivo de la potencia 2.

Análisis del mejor caso

Número de multiplicaciones en el mejor caso:

$$f(2^k) = \begin{cases} 0 & k = 0 \\ f(2^{k-1}) + 1 & k > 0 \end{cases}$$

- Hipótesis de inducción: $f(2^\alpha) = \alpha$ para $0 \leq \alpha \leq k-1$.
- Paso inductivo: $f(2^k) = f(2^{k-1}) + 1 = (k-1) + 1 = k$

Mejor caso: $f(2^k) = f(2^{k-1}) + 1 = k$

Análisis del peor caso

Número de multiplicaciones en el peor caso:

$$f(2^k - 1) = \begin{cases} 0 & k = 1 \\ f(2^{k-1} - 1) + 2 & k > 1 \end{cases}$$

- Hipótesis de inducción: $f(2^\alpha - 1) = 2(\alpha - 1)$, para $1 \leq \alpha \leq k-1$.
- Paso inductivo:

$$\begin{aligned} f(2^k - 1) &= f(2^{k-1} - 1) + 2 \\ &= 2(k - 1 - 1) + 2 = 2(k-1) \end{aligned}$$

Peor caso: $f(2^k - 1) = 2(k-1)$

Complejidad de la potencia recursiva

Mejor caso: $n = 2^k$

- $f(2^k) = k$ para $k \geq 0$.
- $f(n) = \log_2 n$ para $n = 2^k$.

Mejor caso: $f(n) = \Omega(\log n)$

Peor caso: $n = 2^k - 1$

- $f(2^k - 1) = 2(k-1)$ para $k \geq 1$.
- $f(n) = 2[\log_2(n+1) - 1]$ para $n = 2^k - 1$.

Peor caso: $f(n) = O(\log n)$

Por tanto en el caso medio: $T(n) = \Theta(\log n)$

1.7. Reglas para calcular O

1. **Operación elemental**: $O(1)$

2. **Secuencia**. Sean $S_1 = O(f_1(n))$ y $S_2 = O(f_2(n))$ dos secuencias:

$$S_1 : S_2 = O(\text{máx}(f_1(n), f_2(n)))$$

3. **Condición**. Sea $B = O(f_B(n))$ la expresión booleana, $S_1 = O(f_1(n))$ y $S_2 = O(f(n))$.

$$\text{if } B : S_1 \text{ else: } S_2 = O(\text{máx}(f_B(n)), f_1(n), f(n))$$

   - Si $f_1(n) \neq f_2(n)$ y $\text{máx}(f_1(n), f_2(n)) > f_B(n)$, la regla sólo vale para el peor caso.
   - Caso medio: $O(\text{máx}(f_B(n), f(n)))$.
   - $f(n)$ es el promedio de f_1 y f_2 ponderado con las frecuencias de cada rama.

4. **Iteración**: Sea $B : S = O(f_{B,S}(n))$ la expresión booleana seguida de una secuencia a ejecutar si esta se cumple, m el número de iteraciones y $O(f_{\text{iter}}(n))$ la complejidad del número de iteraciones.
   - Si el coste de las iteraciones no varía (sino, suma de los costes individuales)

$$\text{while } B : S = O(f_{B,S}(n) * f_{\text{iter}}(n))$$

$$\text{for } i = x \text{ to } y : S \;=\; O(f_S(n) * m)$$

   - En el caso `for`, B es comparar 2 enteros: $O(1)$, por eso no se tiene en cuenta en la expresión final.

Bloque II
Paradigmas de diseño algorítmico

Tema 2: Divide y Vencerás

2.1. Recursividad

Un **algoritmo recursivo** expresa la solución a un problema en términos de una o varias llamadas a sí mismo (*llamadas recursivas*). En una solución recursiva:

1. Un problema de tamaño n se descompone en uno o varios problemas similares de tamaño $k < n$.
2. Cuando el tamaño de los problemas es lo bastante pequeño se resuelven directamente.
3. La solución al problema se construye combinando las soluciones a los problemas más pequeños en los que se ha descompuesto.

En un algoritmo recursivo deben aparecer:

- **Caso base**: Solución directa del problema para tamaños pequeños ($n \leq n_0$).
- **Caso general**: Solución recursiva basada en la solución de problemas más pequeños ($\Phi(n)$ resuelto como combinación de $\Phi(k)$ con $k < n$).

La **prueba de corrección** de un algoritmo recursivo equivale a una **demostración por inducción**:

1. Caso base: El método $\Phi(n)$ funciona cuando $n \leq n_0$.
2. Hipótesis de inducción: Se supone que el método $\Phi(n)$ funciona $\forall k < n$.
3. Prueba de inducción: Se demuestra $\Phi(n)$ a partir de $\Phi(k)$ con $k < n$.

Si el caso base resuelve el problema de tamaño menor que n_0 y la expresión general de la solución resuelve el problema a partir de las soluciones a problemas estrictamente más pequeños, entonces el algoritmo es correcto.

2.2. Recurrencia

Cuando se analizan algoritmos recursivos se puede describir la función que mide su coste temporal $T(n)$ mediante una recurrencia.

Una **recurrencia** es una ecuación o desigualdad que describe una función en términos del propio valor de sí misma para argumentos más cercano a algún caso base, para el cual la función está definida explícitamente.

Resolver una recurrencia significa encontrar la expresión explícita que define la función. Por ejemplo, si tenemos una función de recurrencia de la forma:

$$T(n) = \begin{cases} n & \text{si } n \in \{0,1,2\} \\ 5T(n-1) - 8T(n-2) + 4T(n-3) & \text{en caso contrario} \end{cases}$$

La solución viene dada por la expresión: $T(n) = 2^{n+1} - n2^{n-1} - 2$.

Veremos que las recurrencias se usan ampliamente para la determinación del coste de ejecución de algoritmos recursivos tales como los desarrollados con la técnica de diseño *Divide y Vencerás*.

2.3. Divide y Vencerás

La técnica **Divide y Vencerás** se basa en descomponer el problema en ℓ subproblemas más pequeños.

- Tamaño del problema original: n.
- Tamaño del i-ésimo subproblema: $m_i < n$ (típicamente $\sum_{i=1}^{\ell} m_i \leq n$).
- Los ℓ subproblemas disjuntos se resuelven por separado aplicando el mismo algoritmo.
- El número de subproblemas ℓ debe ser pequeño, independientemente de la entrada, y a ser posible **balanceado**.
- La solución al problema original se obtiene combinando las soluciones a los ℓ subproblemas.
- Cuando el problema original genera un único subproblema se habla de **algoritmos de simplificación**.

La **hipótesis** es que:

- Resolver un problema más pequeño tiene un coste menor.
- La solución de los casos muy pequeños ($n \leq n_0$) es trivial y tiene un coste fijo (**algoritmo ad-hoc**).

Pseudocódigo Divide y Vencerás

```
function Divide and Conquer (x): solution
    if x is sufficiently small:
        s = ad-hoc(x);
    else:
        decompose x into smaller cases x1, x2, ..., xl;
        for xi in x1, x2....:
            si = Divide and Conquer (xi);
        combine the si's to obtain a solution s of x;
    return s;
```

Aspectos de diseño

- Algoritmo recursivo: Algoritmo Divide y Vencerás + División + Combinación.
- Algoritmo ad-hoc: Algoritmo básico o específico que resuelve problemas de tamaño pequeño.

$$T_{\text{dyv}} = \begin{cases} T_{\text{ad-hoc}(n)} & n \leq n_0 \\ T_{\text{dividir}}(n,\ell) + \sum_{i=1}^{\ell} T_{\text{dyv}}(m_i) + T_{\text{combinar}}(n,\ell) & n > n_0 \end{cases}$$

Determinación del umbral

- Su valor optimo depende de la implementación.
- No existen restricciones sobre este valor.

Método empírico:

- Se implementa el algoritmo ad-hoc y el algoritmo Divide y Vencerás.
- Se resuelven ambos algoritmos para distintos valores de n.
- Se toma como n_0 el valor de n donde $T_{\text{ad-hoc}}(n) \approx T_{\text{dyv}}(n)$.

Método teórico: La ideal del método experimental se traduce en:

$$\begin{cases} T_{\text{ad-hoc}}(n) = h(n) & n \leq n_0 \\ T_{\text{dyv}}(n) = \ell T_{\text{dyv}}(n/b) + g(n) & n > n_0 \end{cases}$$

Teóricamente, el umbral n_0 será óptimo cuando ambos tiempos de ejecución coincidan:

$$h(n) = \ell h(n/b) + g(n), \quad n = n_0$$

Método híbrido:

1. Se calculan constantes utilizando el enfoque empírico.
2. Se calcula el umbral utilizando el enfoque teórico.
3. Se prueban valores alrederl del umbral teórico para determinar el umbral óptimo.

2.4. Teorema de resolución de recurrencias *Divide y Vencerás*

Sea una recurrencia de *Divide y Vencerás* de la forma:

$$T(n) = \ell T(n/b) + cn^k, \quad n \geq n_0$$

con $\ell \geq 1$, $b \geq 2$, $k \geq 0$, $n_0 \geq 1 \in \mathbb{N}$ y $c \in \mathbb{R}^+$.

Cuando n/n_0 es potencia exacta de b (es decir, $n \in \{b^i n_0\}, i \in \mathbb{Z}^+$) se aplica:

$$T(n) = \begin{cases} O(n^k) & \text{si } \ell < b^k \\ O(n^k \log n) & \text{si } \ell = b^k \\ O(n^{\log_b \ell}) & \text{si } \ell > b^k \end{cases}$$

2.5. Ejemplos de algoritmos *Divide y Vencerás*

2.5.1. El problema del Skyline

Problema: Dada una secuencia de triplas (l_i, h_i, r_i) para $i = 1, ..., n$, que representan un conjunto de edificios rectangulares. Determinar la silueta que forman los edificios en el horizonte (*skyline*).

Función recursiva

```
function Skyline(var B[1...n]): list
    if n=1:
        SK := (l1, h1, r1, 0);                  ⟶ Θ(1)
    else:
        half := n div 2;                        ⟶ Θ(1)
        SK1  := Skyline(B[1..half])             ⟶ T(⌊n/2⌋) ≈ T(n/2)
        SK2  := Skyline(B[half+1..n]);          ⟶ T(⌊n/2⌋) ≈ T(n/2)
        SK   := Mix(SK1, SK2);                  ⟶ O(n)
    return SK
```

El coste total de la solución recursiva planteada para el problema del skyline cumple la siguiente recurrencia:

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

Y como $\ell = 2$, $b = 2$ y $k = 1$ en el teorema de Divide y Vencerás:

$$T(n) = O(n \log n)$$

2.5.2. El problema de los puntos más cercanos

Problema: Dados n puntos en el plano, encontrar la pareja de puntos con la menor distancia euclídea entre ellos.

- Algoritmo de fuerza bruta: Se comprueban todos los pares de puntos ($T(n) = O(n^2)$)
- Algoritmo Divide y Vencerás:

1. **Divide**: Establecer la línea vertical L que divide el espacio en dos mitades iguales.
2. **Vencerás**: Llamadas recursivas para encontrar los pares de puntos más cercanos a la izquierda y derecha de L.
3. **Combinar soluciones**: Utilizando $\delta = \min(\delta_{\text{left}}, \delta_{\text{right}})$ (el mínimo de las distanncias mínimas en el subespacio izquierdo y el subespacio derecho generados por L), se busca el par de puntos más cercanos con la restricción de que cada uno esté en un subespacio distinto y a una distancia δ de la recta L.

Para combinar soluciones, seguimos los siguientes pasos:

1. Ordenar los elementos de la franja 2δ en función de su coordenada y.
2. Comprobar sólo aquellos puntos que están a menos de 8 posiciones en la lista en función de y.
3. Propiedad: Si p_1 y p_2 son puntos de la franja 2δ tales que $(p_1, p_1) < \delta$, para encontrarlos no tendremos más que calcular 7 distancias por punto si los ordenamos por su coordenada y.
4. Demostración: En un rectángulo $2\delta \times \delta$ hay a lo sumo 8 puntos a una distancia menor que δ.

Función recursiva

```
function Closest Pair (P[1..n]): δ
    Obtener la recta L con la mitad de los puntos
    a un lado y la mitad al otro lado                    ⟶ O(n log n)

    δ1 = Closest Pair(left, half);                       ⟶ T(n/2)
    δ2 = Closest Pair(right, half);                      ⟶ T(n/2)
    δ  = min(δ1, δ2);

    Obtener todos los puntos más cercanos a L que δ      ⟶ O(n)

    Ordena los puntos por su coordenada y                ⟶ O(n log n)

    Escanear todos los puntos ordeandos y comparar
    sus distancias con los 7 más próximos                ⟶ O(n)

    if any distance < δ:
        update δ

    return δ
```

El tiempo de ejecución del algoritmo viene dado por:

$$T(n) \leq 2T(n/2) + O(n \log n)$$

Y como $\ell = 2$, $b = 2$, $n^k = n \log n$:

$$T(n) = O(n \log n)$$

Mejora: No reordenar desde cero los puntos de la franja 2δ, sino que cada llamada recursiva devuelva los puntos ordenados por coordenada x y coordenada y. Después se ordenan las listas ordenadas en $O(n)$.

$$T(n) = O(n \log n)$$

Multiplicación de Karatsuba-Ofman

El método tradicional para la **multiplicación de dos números enteros** de n dígitos es $\Theta(n^2)$. El procedimiento de Karatsuba-Ofman permite la multiplicación de números de gran tamaño de forma más efciente.

Sean x e y dos números enteros en alguna base b. Para cualquier entero positivo m ($m < n$), x e y se pueden descomponer de la siguiente forma:

$$x = x_1 b^m + x_0 \iff x = x_1 | x_0$$

$$y = y_1 b^m + y_0 \iff y = y_1 | y_0$$

Por tanto el producto es:

$$\begin{aligned} xy &= (x_1 b^m + x_0)(y_1 b^m + y_0) \\ &= z_2 b^{2m} + z_1 b^m + z_0 \end{aligned}$$

donde $z_2 = x_1 y_1$, $z_1 = x_1 y_0 + x_0 y_1$ y $z_0 = x_0 y_0$.

Esta fórmula requieren cuatro multiplicaciones de números más pequeños, pero con el método de Karatsuba se puede calcular empleando sólo tres:

$$\begin{cases} z_2 = x_1 y_1 \\ z_0 = x_0 y_0 \\ z_1 = (x_1 + x_0)(y_1 + y_0) - z_0 - z_2 \end{cases}$$

Y el procedimiento se puede resolver de forma recursiva para cada producto de menor tamaño.

Multiplicación de Karatsuba-Ofman

```
function Karatsuba(num1, num2): num
    if num1 < 10 or num2 < 10:
        return num1 * num2;
    else:
        m       :=  max(sizeBase10(num1), sizeBase10(num2));     ⟶ Θ(1)
        half    :=  m/2;                                         ⟶ Θ(1)
        x1, x0  :=  splitAt(num1,half);                          ⟶ O(n)
        y1, y0  :=  splitAt(num2,half);                          ⟶ O(n)
        z0      :=  Karatsuba(x0, y0);                           ⟶ T(n/2)
        z1      :=  Karatsuba(x0+x1, y0+y1);                     ⟶ T(n/2)
        z2      :=  Karatsuba(x1, y1);                           ⟶ T(n/2)
    return ( z2*10^(2*half) ) + ( (z1-z2-z0) * 10^half ) + (z0);
```

El coste de la solución recursiva de la multiplicación Karatsuba-Ofman cumple la siguiente recurrencia:

$$T(n) = 3T(n/2) + O(n)$$

Y como $\ell = 3$, $b = 2$ y $k = 1$, se obtiene:

$$T(n) = O(n^{\log_2 3})$$

2.6. Resumen de los ejemplos

- Problema del Skyline: $T(n) = 2T(n/2) + O(n) \implies T(n) = O(n \log n)$
- Problema de los puntos más cercanos: $T(n) \leq 2T(n/2) + O(n \log n) \implies T(n) = O(n \log n)$
- Multiplicación de Karatsuba-Ofman: $T(n) = 3T(n/2) + O(n) \implies T(n) = O(n^{\log_2 3})$

Tema 3: Algoritmos voraces

3.1. Definición y características

El esquema voraz es un esquema simple y de los más utilizados. Forma parte de los algoritmos de **búsqueda local** y es utilizado típicamente para resolver problemas de optimización. Son fáciles de inventar, fáciles de implementar y cuando funcionan son eficientes.

Características del problema a resolver:

- **Función objetivo** a maximizar o minimizar.
- **Dominio**. Conjunto de valores posibles para cada una de las variables de la función objetivo.
- **Restricciones** a los valores del dominio que toman las variables de la función objetivo.
- La solución debe ser expresable en forma de secuencia de decisiones.
- **Función solución** que permite determinar cuándo una secuencia de decisiones es solución para el problema.
- **Función factible** que permite determinar si una secuencia de decisiones viola o no las restricciones.
- **Objetivo**: Encontrar una solución tal que el valor de la función objetivo sea óptimo.

Proceso secuencial:

1. Tomar una decisión siempre bajo el mismo criterio (**función selección**).
2. Comprobar si se puede incorporar a la secuencia de decisiones (**función factible**).
3. Comprobar si se ha alcanzado el objetivo (**función solución**).
4. Repetir hasta alcanzarlo.

Para cada problema:

- Identificar el objetivo (maximizar o minimizar).
- Determinar la función selección.
- Definir la función factible.
- Comprobar si se ha alcanzado la solución.

Formato de los algoritmos voraces

```
function Greedy(C:set): set
    S := ∅;                                  % La solución se construye en S
    while C <> ∅ and not solution(S):        % C es el conjunto de candidatos
        x := select(C);
        C := C -{x};
        if feasible(S ∪ {x}):
            S := S ∪ {x};
        if solution(S):
            return S;
        else
            return "there are no solutions"
```

3.2. El problema de devolver el cambio

Problema: Suponiendo que tenemos un conjunto C de monedas de distinto tipo queremos pagar una cantidad n utilizando el menor número de monedas posible.

Devolver el cambio

```
function MakeChange(n, C): SetOfCoins
    % C es el conjunto de monedas de distinto tipo
    S := ∅;                                  % la solución se construye en S
    ss := 0;                                   % ss es la suma de monedas de S
    while ss <> n:
        x := Largest item in C such that ss + x <= n;
        if there is no such item:
            return "No solutions";
        S := S ∪ a coin with value x;
        ss = ss + x;
    return S;
```

¿Por qué `MakeChange` funciona?

- Con los valores proporcionados de las monedas y una cantidad suficiente de cada una, el algoritmo obtiene una solución óptima al problema.
- No funciona con cualquier C.

Características voraces de `MakeChange`

- Elige el mejor candidato en cada paso sin valorar las consecuencias.
- Una vez que un candidato se incluye en la solución, se queda para siempre.
- Una vez que un candidato se excluye de la solución, nunca se vuelve a considerar.

3.3. El problema de la mochila

Problema: Cargar una mochila de capacidad W maximizando el valor de carga con n objetos, los cuales tiene un peso w_i y valor v_i asociado (para $1 \leq i \leq n$).

- Los objetos se pueden fraccionar sin perder su valor.
- Podemos decidir cargar solo una fracción x_i del objeto i (para $0 \leq x_i \leq 1$)
- El objeto i contribuye:
  - $x_i w_i$ al peso de la carga, limitado por W.
  - $x_i v_i$ al valor de la carga que se quiere maximizar.
  - El problema se puede expresar como:

$$\text{máx} \sum_{i=1}^{n} x_i v_i \quad \text{sujeto a} \quad \sum_{i=1}^{n} x_i w_i \leq W$$

- Hipótesis: $\sum_{i=1}^{n} w_i > W$.
- La solución óptima llena completamente la mochila: $\sum_{i=1}^{n} x_i w_i = W$

Problema de la mochila

```
function knapsack (w[1..n], v[1..n], W):  array [1..n]
    for i := 1 to n:
        x[i] := 0;             /* Initialization */
    weight := 0;
    while weight < W:         /* Greedy loop */
        i := the best remaining object;
        if weight + w[i] <= W:
            x[i] := 1;
            weight := weight + w[i];
        else:
            x[i] := (W - weight) / w[i];
            weight  := W;
    return x;
```

Funciones selección plausibles

- Elegir el objeto más valioso de los restantes ($\text{máx}\, v_i$).
- Elegir el objeto más ligero de los restantes ($\text{mín}\, w_i$).
- Elegir el objeto con la mayor relación valor/peso ($\text{máx}\, v_i/w_i$).

Teorema. Si los objetos se seleccionan en orden decreciente v_i/w_i el algoritmo encuentra la solución óptima (demostración por reducción al absurdo).

Análisis de la complejidad

- Inicialización: $O(n)$.

- Bucle voraz: $O(n)$
- Ordenación de los objetos: $O(n \log n)$

Complejidad: $T(n) = O(n \log n)$

Mejora: Mantener los objetos en un montículo.

- Inicialización (crear el montículo): $O(n)$.
- Bucle voraz $O(\log n) * n$ en el peor caso. Esto es: $O(n \log n)$
- Se consigue un mejor $T(n)$.

3.4. Ordenación topológica

La ordenación topológica consiste en la ordenación de los vértices de un **grafo acíclo** tal que, si existe un camino de v_i a v_j, entonces v_j aparece después de v_i en la ordenación.

- La ordenación topológica **no** es única.
- El grado de entrada de un vértice v es el número de aristas (u, v) que inciden en él.
- Hipótesis de partida. El grafo se encuentra en memoria representado mediante listas de adyacencia.
- $G = (V, E)$, $|V| = n$, $|E| = m$, para $0 \leq m \leq n(n-1)$.

El algoritmo de la ordenación topológica se define de la siguiente manera:

1. Encontrar un vértice sin aristas de entrada.
2. Imprimir el vértice.
3. Eliminar ese vértice dle grafo junto con sus aristas.

Ordenación topológica 1

```
function TopologicalSorting1 (G: graph): order [1..n]
    Indegree [1..n] := Obtaining Indegree (G);
    for i := 1 to n:
        TopologicalNum [i] := 0;c
    counter := 1;
    while counter <= n:
        /* No Topological number assigned  */
        v := FindNewVertexOfIndegreeZero(...);
        if v not found:
            return error "the graph has a cycle";
        else:
            TopologicalNum[v] := counter;
            counter ++;
            for each w adjacent to v:
                Indegree [w] := Indegree [w] - 1;
    returnTopologicalNum
```

Mejoras en el algoritmo

Una posible mejora sería mantener todos los vértices no asignados de grado 0 en una caja especial.

1. La función `FindNewVertexOfIndegreeZero` devuelve (y elimina) cualquier vértice de la caja.
2. Cuando se decrementa el grado de los vértices adyacentes, se comprueba si hay algún vértice de grado 0 y se introduce en la caja.
3. Para implementar la caja se puede usar una pila o una cola.

Ordenación topológica 2

```
function TopologicalSorting2 (G: graph): order [1..n]
    Indegree [1..n] := Obtaining Indegree (G);
    { for i := 1 to n:  TopologicalNum [i] := 0; }
    counter := 1;
    CreateQueue(Q);
    for each v:
        if Indegree[v] = 0:
            EnQueue(v,Q);
    while not EmptyQueue(Q):
        v := Dequeue(Q);
        TopologicalNumber[v] := counter;  /* Assing next number */
        counter ++;
        for each w adjacent to v:
            Indegree [w] := Indegree [w] - 1;
            if Indegree[w] = 0:
                EnQueue(w,Q);
    if counter <= n:
        return error "The graph has a cylce";
    else:
        return TopologicalNum;
```

Análisis de la complejidad

- Utilizando listas de adyacencia: $O(n+m)$
- Pero caso: el grafo es denso y se visitan todas las aristas ($m \to n(n-1)$).
- Mejor caso: el grafo es disperso ($m \to n$).

3.5. Árbol expandido mínimo

Sea $G = (V, E)$ un grafo conexo, no dirigido, con aristas de longitud no negativa.

Objetivo: Encontrar un subconjunto T de E tal que todos los nodos siguen conectados cuando se utilizan solamente las aristas de T y la suma de las longitudes de las aristas de T sea la menor posible.

Sea $G' = (V, T)$ el grafo parcial formado por los nodos de G y las aristas de T donde n es el número de nodos en V. Recordemos que:

- Un grafo conexo con n nodos debe tener **al menos** $n-1$ aristas.

- Un grafo conexo con n nodos y más de $n-1$ aristas contiene al menos un ciclo. Por tanto se puede eliminar una de ellas sin desconectar G'.

Conclusión: T debe tener exactamente $n-1$ aristas y ya que G' se denomina **árbol expandido mínimo** para el grafo G.

Obtención del árbol expandido mínimo

- Los candidatos son las aristas de G.
- Un conjunto de aristas es **solución** si constituye un árbol expandido mínimo para los nodos de V.
- Un conjunto de aristas es **factible** si no incluye ciclos.
- La función **selección** que se utiliza varía con el algoritmo.
- La función **objetivo** a minimizar es la longitud total de las aristas de la solución.
- Un conjunto de aristas factible es **prometedor** si se puede extender para producir no sólo una solución sino la **solución óptima**.

Lema. Sea $G = (V, E)$ un grafo conexo no dirigido donde se conocen la longitud de sus aristas, y:

- $B \subset V$ es un subconjunto estricto de los nodos de G.
- $T \subseteq E$ es un conjunto de aristas prometedor tal que ninguna arista de T parte de B.
- v la arista más corta que parte de B.

Entonces $T \cup v$ es prometedor.

3.6. Algoritmo de Kruskal

1. El conjunto T está vacío y el grafo parcial $G' = (V, T)$ consiste en n componentes conexas (un nodo en cada componente conexa).
2. Se examinan las aristas de G en orden creciente de longitud.
3. Si una arista une dos nodos en componentes conexas diferentes, se añade a T y se fusionan esas dos componentes conexas como una. En otro caso se rechaza la arista.
4. El algoritmo finaliza cunado solo existe una componente conexa.

Algoritmo de Kruskal

```
function Kruskal (G=(V,E): graph): tree
    /* Inicialización */
    Sort E by increasing length;
    T := { };
    Initialize n sets, each contains a different element of V;
    repeat until |T| = n-1;
        a := (u,v) with smallest length not considered yet;
        Uset := find(u);
        Vset := find(v);
        if Uset <> Vset:        /* u y v en diferentes componentes conexas */
            merge(Uset,Vset);
            T := T U (u,v)
    return T
```

Análisis del algoritmo de Kruskal

1. Ordenar m aristas por su peso: $\Theta(m \log m) \Rightarrow \Theta(m \log n)$ porque $n - 1 \leq m \leq n(n-1)/2$.
2. Inicializar los n conjuntos disjuntos: $\Theta(n)$.
3. Todas las operaciones `find` y `merge`: $\Theta(2m\alpha(2m, n))$:
   - α es una función de crecimiento lento (existen como mucho $2m$ operaciones `find` y $n-1$ operaciones `merge`).
   - Como $\Theta(\alpha(2m, n)) \subset \Theta(m \log n) \Longrightarrow \Theta(m \log n)$.
4. En el peor de los casos, $\Theta(m)$ para el resto de las operaciones.

Complejidad: $T(n) = \Theta(m \log n)$

Mejora: Mantener las aristas en un montículo invertido. No cambia el análisis del peor caso, pero obtiene mejores tiempos de ejecución.

3.7. Algoritmo de Prim

Sea B un conjunto de nodos y T un conjunto de aristas:

1. Inicialmente B contiene un único nodo arbitrario y T está vacío.
2. En cada paso la arista (u, v) de menor peso tal que $u \in B \wedge v \in V - B$ se añade a T y v se añade a B.
3. T constituye en cualquier momento un árbol expandido mínimo para los nodos de B.
4. Se continúa mientras $B \neq V$

Algoritmo de Prim

```
function Prim 1 (G = (V,E): graph): tree
    /*  Inicialización */
    T := { };
    B := an arbitrary member of V;
    while B <> V:
        a := (u,v);                /* Arista más corta que parte de B */
        T := T U a;
        B := B U v;
    return T
```

Implementación con matrices de adyacencia

Una matriz de adyacencia L indica la longitud de cada arista del nodo i al nodo j (con $L[i,j] = \infty$ si la arista del nodo i al nodo j no existe).

- Se numeran los nodos de G de 1 a n tal que $V = \{1, ..., n\}$.
- Para cada nodo $i \in V - B$:
  - `nearest[i]` indica el nodo en B más cercano al nodo i.
  - `mindist[i]` indica la distancia de i a `nearest[i]`.
- Para cada nodo $i \in B$ se establece `mindist[i] = -1` con el fin de saber si un nodo está en B o no.

Algoritmo de Prim (matrices de adyacencia)

```
function Prim 2 (L[1..n,1..n]): tree
    /* Inicialización */
    mindist[1] := -1;
    T := { };
    for i := 2 to n:
        nearest[i] := 1;
        mindist[i] := L[i,1]
    repeat n-1 times:                    /* Bucle voraz */
        min := infinity;
        for j := 2 to n:
            if 0 <= mindist[j] < min:
                min := mindist[j];
                k := j
        T := T U { nearest{k}, k};
        mindist[k] := -1;                /* Añade k a B */
        for j := 2 to n:
            if L[j,k] < mindist[j]:
                mindist[j] := L[j,k];
                nearest[j] := k
    return T
```

Interpretación del pseudocógido

Inicialización:

1. Se toma el nodo 1 y se establece su `mindist` a -1 (pues ya estará en B).
2. Se crea el conjunto T de aristas vacío.
3. Los $n-1$ nodos restantes establecen su `nearest` (nodo más cercano en B) a 1, pues en B sólo está el nodo 1; y su `mindist` a $L[i,1]$.

Bucle voraz: Se repite $n-1$ veces (por cada nodo restante):

1. Se establece un marcador `min` como ∞.
2. Se busca en $V-B$ el nodo con la distancia más corta a algún nodo en B. Dicho nodo se llamará k.
3. Se anota en T la arista que va desde el nodo k hasta el nodo más cercano en B de k (`nearest[k]`).
4. Se establece `mindist[k]` a -1 (pues ahora k pertenece al conjunto B).
5. Se abre otro bucle para los nodos en $V-B$ que actualiza el array `mindist` y `nearest` para que los nodos que aún no están en B consideren al nodo k recién añadido.

Análisis del algoritmo de Prim

1. Inicialización: $\Theta(n)$.
2. Bucle voraz: $n-1$ iteraciones.
3. En cada iteración el bucle interno `for` tarda $\Theta(n)$.

Complejidad: $T(n) = \Theta(n^2)$

Mejora: Mantener las aristas en un montículo invertido ($\Theta(m \log n)$).

3.8. Kruskal vs Prim

Algoritmos	Complejidad	Grafo denso $m \to n(n-1)/2$	Grafo disperso $m \to n$
Kruskal	$\Theta(m \log n)$	$\Theta(n^2 \log n)$	$\Theta(n \log n)$
Prim	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$

3.9. Algoritmo de Dijkstra

Sea $G = (V, E)$ un grafo dirigido donde V es el conjunto de nodos y E es el conjunto de aristas dirigidas, cada arista tiene una **longitud** no negativa y uno de los nodos se designa como **nodo origen**.

Problema: Determinar la longitud del camino mínimo desde el origen de cada uno de los nodos del grafo.

Solución voraz

Consideremos dos conjuntos de nodos de V: los seleccionados (S) y los candidatos (C). S contiene aquellos nodos ya elegidos. La distancia mínima desde el origen se conoce para cada nodo de S. C contiene el resto de los nodos cuya distancia mínima al origen es desconocida todavía.

1. Inicialmente S contiene solamente el nodo origen. Al final contiene todos los nodos.
2. En cada paso se elige el nodo de C cuya distancia al origen es mínima.
3. Se dispone de información temporal sobre distancias mínimas.

Formulación del algoritmo de $Dijkstra$

Camino especial: Camino desde el origen a cualquier otro nodo tal que los nodos intermedios pertenecen a S.

1. Los nodos de G se numeran de 1 a n tal que $V = \{1, ..., n\}$. Suponemos que el nodo origen es el 1 y disponemos de la matriz de adyacencia L.
2. En cada paso del algoritmo, un array D mantiene la longitud del camino mínimo especial a cada nodo del grafo.
3. Cuando se añade un nuevo nodo v a S, el camino mínimo especial a v es también el camino más corto de todos los caminos a v.
4. Al final, D proporciona la solución al problema de los caminos mínimos.

Algoritmo de Dijkstra

```
function Dijkstra (L[1..n,1..n]): array[1..n]
    C := {2,3,..,n};
    for i := 2 to n:
        D[i] := L[1,i];
    repeat n-2 times:                  /* Bucle voraz */
        v := some element of C minimizing D[v];
        C := C - {v};
        for each w in C:
            D[w] := min(D[w], D[v]+ L[v,w])
    return D;
```

- Para encontrar **por dónde pasa un camino**, añadir un segundo array $P[2, ..., n]$ donde $P[v]$ contiene el nodo que precede a v en el camino mínimo.
- Para encontrar el **camino completo** se siguen los elementos de P hacia atrás desde el nodo destino al nodo origen.

Análisis del algoritmo de Dijkstra

Sea $G = (V, E)$ un grado de n nodos y m aristas y L la matriz de adyacencia:

1. Inicialización: $\Theta(n)$.

2. Bucle `repeat` requiere revisar todos los elementos de C: $\Theta(n^2)$
3. Bucle `for` revisa todos los elementos en C: $\Theta(n^2)$.

$$\text{Complejidad:} \quad T(n) = \Theta(n^2)$$

Mejora: Si el grafo es disperso, mn^2, es preferible representarlo mediante listas de adyacencia.

- Ahorro de tiempo en el bucle `for` interno, considerando solo los nodos adyacentes a v.
- Recorre una lista, no una fila y una columna.

Algoritmo de Dijkstra con montículos

Para obtener v en D que minimza la distancia con el nodo origen:

- Utilizar un montículo invertido con un nodo por cada elemento v de C, ordenado por $D[v]$.
- El elemento v de C que miniza $D[v]$ estará en la raíz.

Aplicando este método, la complejidad del algoritmo se reduciría de la siguiente manera:

1. Inicializar un montículo: $\Theta(n)$.
2. Eliminar la raíz del montículo: $O(\log n)$
3. Bucle `for` interior examina cada nodo w en C adyacente a v y comprueba si $D[v] + L[v, w] < D[w]$. Si es así, modifica $D[w]$ y flota w por el montículo ($O(\log n)$).
   - Elimina la raíz del montículo $n - 2$ veces.
   - Flota como mucho m nodos.

$$\text{Complejidad:} \quad O((m + n) \log n)$$

Tema 4: Programación dinámica

4.1. Motivación de la Programación Dinámica

La técnica **Divide y Vencerás** tiene el riesgo de llegar a tener un gran número de subcasos que empeoren la eficiencia del algoritmo.

En Programación Dinámica nuestro objetivo es resolver cada subcaso una única vez, guardando las soluciones en una tabla de resultados que se va completando hasta alcanzar la solución buscada. Se trata de una **técnica ascendente** opuesta a la descendente de Divide y Vencerás.

Principio de optimalidad: Programación Dinámica se utiliza para resolver problemas de optimización que satisfacen el principio de optimalidad: "*En una secuencia de decisiones toda subsecuencia ha de ser también óptima*".

- **Algoritmo Fibonnaci 1**: $T(n) = \Theta(\Phi^n)$, donde $\Phi = \frac{1+\sqrt{5}}{2}$, y espacio usado $\Theta(n)$
- **Algoritmo Fibonnaci 2**: $T(n) = \Theta(n)$ y espacio usado $\Theta(1)$.
- **Algoritmo Fibonnaci 3**: $T(n) = O(\log n)$

```
function fib1(n):
    if n < 2:
        return n;
    else:
        return fib1(n-1) + fib1(n-2);

function fib2(n):
    i := 1;
    j := 0;
    for k := 1 to n:
        j := i+j;
        i := j-i;
    return j;
```

```
function fib3(n):
    i := 1; j := 0; k := 0; h := 1;
    while n > 0:
        if n es impar:
            t := j*h;
            j := i*h + j*k + t;
            i := i*k + t;
        t := h^2;
        h := 2*k*h + t;
        k := k^2 + t;
        n := n div 2
    return j;
```

4.2. El problema de los coeficientes binomiales

El objetivo del problema de los coeficientes binomiales es encontrar los coeficientes que aparecen al elevar un binomio a una potencia.

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

Resolución recursiva con Divide y Vencerás: $\Omega\left(\binom{n}{k}\right)$

La solución se puede encontrar con **programación dinámica** usando como tabla de resultados el **triángulo de**

Pascal.

Resumen del algoritmo

- Complejidad del algoritmo: $T(n) = O(nk)$.
- Complejidad espacial: $O(nk)$.
- Mejora de la complejidad espacial con una línea del triángulo de Pascal: $\Theta(k)$.

4.3. Problema de devolver el cambio

Dado $M = \{v_1, v_2, ..., v_m\}$, con $v_i > 0$, un conjunto de monedas con distintos valores, el objetivo es pagar exactamente una cantidad n con la mínima cantidad posible de monedas.

La solución se puede encontrar con *programación dinámica*, creando una tabla de m filas y $n + 1$ columnas, donde el valor de la fila i y columna j es el mínimo número de monedas para pagar una cantidad j (donde $j = 0, 1, ..., n$) utilizando las monedas $v_1, v_2, ..., v_i$. Una vez creada la tabla, la cantidad mínima de monedas a usar está determinada por el valor de la tabla de la última fila y última columna $c[m, n]$.

Construcción de la tabla

1. Las celdas $c[i, 0] = 0$

2. Se va completando. Si $i \geq 1$ y $j \geq v_i$ (es decir, si el número del tipo de monedas con las que pagar es mayor que 0 y si la cantidad a pagar j es mayor que el valor de la moneda de tipo v_i):

$$c[i,j] \quad = \quad \text{mín} \begin{cases} c[i-1,j] & \text{no utilizar una moneda más de } v_i \\ 1 + c[i, j - v_i] & \text{utilizar una moneda más de } v_i \end{cases}$$

3. Caso particular: Si $i = 1$ y $j < v_1$ entonces $c[i, j] = \infty$. No hay solución (no se puede pagar una cantidad menor al valor de la moneda).

Devolver el cambio

```
function coins (n):              # devuelve número de monedas
    const v[1..m]=[1,4,6];       # denominaciones de las monedas

    # se construye una tabla c[1..m, 0..n]
    for i := 1 to m:
        c[i,0] := 0;

    for i := 1 to m:
        for j := 1 to n:
            if i = 1 and j < v[i]:
                c[1,j] := infinito;
            else if i = 1:
                c[1,j] := 1 + c[ 1, j-v[1] ];
            else if j < v[i]:
                c[i,j] := c[i-1,j];
            else:
                c[i,j] := min ( c[i-1, j], 1 + c[ i, j-v[i] ] );
    return c[m,n];
```

El conjunto de monedas que se deben escoger

La solución del problema (el conjunto de monedas que se deben escoger) viene dado por el algoritmo voraz que comienza en $c[m,n]$ hasta $c[0,0]$ buscando de dónde sale cada resultado. Al final, el número de saltos hacia atrás que se tomen en una fila k determinada indica el número de monedas v_k que se deben usar. Esto tiene un coste de $O(m + c[m,n])$.

Complejidad del algoritmo: $\Theta(mn)$

4.4. Problema de la Mochila

Dados n objetos con unos pesos y valores asociados y una carga W máxima que soporta nuestra mochila. El objetivo es llenar la mochila con objetos maximizando el valor total.

Con Programación Dinámica este problema se soluciona siguiendo el siguiente algoritmo:

1. Creación de una tabla c de n filas y $W+1$ columnas de la forma: $c[1...n, 0...W]$ donde cada $c[i,j]$ será el valor de carga máxima para la capacidad j considerando los objetos $1, ..., i$ (para $0 \leq j \leq W$ y $1 \leq i \leq n$).
2. La tabla se va construyendo siguiendo la siguiente regla:

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

3. Observación: A diferencia del caso de las monedas, cada objeto sólo se puede incluir una vez en la mochila.

Complejidad del algoritmo: $T(n) = \Theta(nW)$

Para obtener la composición de la carga se debe realizar el siguiente recorrido de $v[n, W]$ a $v[0, 0]$:

1. Mirar de dónde procede el $v[n, W]$.
2. Si procede del elemento de arriba (es igual al elemento de arriba), no hacer nada.
3. Si procede del elemento n posiciones a la izquierda en la fila superior, añadir el objeto n al conjunto de objetos que se introducen en la mochila.

El coste adicional de realizar este recorrido es: $T(n) = O(n + W)$.

4.5. Análisis Sintáctico

Objetivo: Obtener la estructura de una oración (árbol sintáctico) a partir de una gramática.

Algoritmo CYK: Junta pares de subárboles formando un árbol más grande usando reglas gramaticales. Considerando n palabras, s símbolos y r reglas gramaticales:

- Complejidad: $O(rn^3)$
- Espacio ocupado: $O(sn^2)$

Bloque III

Estructuras de datos, algoritmos básicos y complejidad

Tema 5: Búsqueda en memoria principal

La búsqueda es una de las operaciones principales realizaas sobre un conjunto de datos (conocida también como recuperación o *searching*). Consiste en encontrar la posición de un elemento entre un conjunto de elementos dado.

La elección del algoritmo depende de la forma de organización de los datos. Podemos distinguir la búsqueda en memoria principal y la búsqueda en memoria secundaria.

subsection

Búsqueda lineal **Algoritmo básico (sin centinela):** Recorre la estructura hasta encontrar el elemento buscado x o hasta el final. Es eficiente cuando el conjunto de datos es pequeño.

- Mejor caso (x se encuentra en la 1ª posición): $T(n) = 1$.
- Peor caso (x no está en la estructura o está en la última posición): $T(n) = n$
- Caso medio (x se encuentra en el medio de la estructura): $T(n) = \frac{n}{2}$.

Complejidad: $T(n) = \Theta(n)$

Algoritmo con centinela (búsqueda secuencial rápida): Se introduce el elemento buscado (**centinela**) en la última posición de la estructura, de forma que siempre se asegura su encuentro. Este algoritmo reduce el número de comparaciones a la mitad (ahorra una operación de comparación y una operación lógica `AND`). Es eficiente cuando el conjunto de datos es pequeño.

- Mejor caso (x se encuentra en la 1ª posición): $T(n) = 1$.
- Peor caso (x se encuentra en la penúltima o última posición): $T(n) = n$.
- Caso medio (x se encuentra en el medio de la estructura): $T(n) = \frac{n}{2}$.

Complejidad: $T(n) = \Theta(n)$

Búsqueda lineal sin centinela

```
function Lineal search (T[1..n],x):
    i := 1;
    while i <= n and T[i] <> x:
        i = i + 1;
    return i;
```

Búsqueda lineal con centinela

```
function Lineal search2 (T[1..n],x):
    T[n+1] := x
    i := 1;
    while T[i] <> x:
        i = i + 1;
    return i;
```

5.1. Búsqueda binaria

La **búsqueda binaria** se basa en la estrategia *Divide y Vencerás*, la división sucesiva del espacio ocupado por los datos en sucesivas mitades. Es un algoritmo eficiente cuando se tratan grandes conjuntos de datos, pero necesita que los datos estén ordenados. El proceso es el siguiente:

1. Examinar el elemento central de la lista.
2. Si es el elemento buscado, termina la búsqueda.
3. Si no es el elemento buscado, se determina si es mayor o menor al central.
4. Se elige la nueva mitad a evaluar y se repite el proceso.

Análisis de la complejidad:

- Mejor caso (x está en el medio de la estructura): $T(n) = 1$
- Peor caso (el nº de iteraciones es el menor entero k tal que $2^k = n$): $T(n) = O(\log n)$.
- Aplicando el teorema Divide y Vencerás: $T(n) = T(n/2) + 1 \Longrightarrow \ell = 1, b = 2, k = 0$.

Complejidad: $T(n) = O(\log n)$.

Función de búsqueda binaria

```
function Binary search (T[1..n],x):
    left := 1;
    right := n;
    while left <= right:
        middle := (left + right) / 2;
        if T[middle] = x:
            return middle
        else:
            if T[middle] > x:
                right := middle - 1;
            else:
                left := middle + 1;
    return 0;
```

Función de búsqueda binaria (versión recursiva)

```
function Binary searchR (T[1..n], x, left, right): boolean
    if left > right:
        return False;
    else:
        middle := (left + right) / 2;
        if T[middle] = x:
            return True
        else:
            if T[middle] > x:
                return Binary searchR(T[1..n], x, left, middle-1);
            else:
                return Binary searchR(T[1..n], x, middle+1, right);
```

5.2. Árboles de búsqueda

- La altura del árbol es el factor limitante.
- En un árbol aleatorio: $T(n) = O(\log n)$.
- En un árbol degenerado o sesgado: $T(n) = O(n)$.

Función de un árbol binario de búsqueda

```
function Find BST (T,x): tree
    if IsEmptyTree(T):
        return nil;
    else:
        if Root(T) = x:
            return T
        else:
            if Root(T) > x:
                Find BST(LeftChild(T),x);
            else:
                Find BST(RightChild(T,x);
```

5.3. Tablas de dispersión

La dispersión o *hashing* es una técnica utilizada para realizar operaciones de inserción, borrado y búsqueda en un tiempo medio constante. La estructura de datos central es la **tabla de dispersión**.

Una tabla de dispersión ideal es un array de tamaño fijo que almacena claves (una clave es una cadena con un valor asociado). La representación de la tabla de dispersión que se utiliza se denomina **diccionarios**.

5.3.1. Funciones de dispersión

La **función de dispersión** es una función que asocia cada clave con un número del rango $[0, ..., \texttt{Tablesize} - 1]$ y loa almacena en la celda adecuada.

1. **Suma de valores ASCII** de los caracteres de la clave.
   - Función fácil de implementar y rápida.
   - Si el tamaño de la tabla es grande, la función no distribuye correctamente las claves.
2. **Suma de valores ASCII** promediando cada valor con un coeficiente.
   - El número posible de combinaciones de caracteres no es el mismo que el número de posibles cadenas que se pueden tomar, pues los lenguajes no son aleatorios.
   - El porcentaje usado de la tabla sería mucho menor.
3. **Función polinomial** con la regla de Horner:

$$\sum_{i=0}^{\texttt{KeySize}-1} \text{ascii}(\texttt{Key}[\texttt{KeySize} - i]) * 32^i$$

   - Multiplicar por 32 resulta en un desplazamiento de 5 bits.
   - Para evitar el desbordamiento, la operación `mod` se realiza en cada iteración.
   - Para los lenguajes que permiten desbordamiento, la operación `mod` se realizaría justo antes de devolver el valor.
   - En la práctica, para las claves largas no utiizar todos los caracteres.

Función de dispersión (Suma de valores ASCII)

```
function Hash (Key, KeySize): Index
    value := ascii(Key[1]);
    for i := 2 to KeySize:
        value := value + ascii(Key[i]);
    return value mod Tablesize;
```

Función de dispersión (Suma de valores ASCII promediada)

```
function Hash (Key, KeySize):Index
    return (ascii(Key[1]) + 27*ascii(Key[2]) + 729*ascii(Key[3])) mod Tablesize
```

Función polinomial

```
function Hash (Key, KeySize): Index
    value := ascii(Key[1]);
    for i := 2 to KeySize:
        value := ( 32 * value + ascii(Key[i])) mod Tablesize;
    return value
```

5.4. Resolución de colisiones

5.4.1. Dispersión abierta

Mantener en una lista todos los elementos que se dispersan en el mismo valor:

- Para buscar, se utiliza la función de dispersión para determinar qué lista se recorre y se recorre la lista devolviendo la posición del elemento.
- Para insertar, se recorre la lista adecuada para determinar si existe el elemento. Si el elemento es nuevo, se inserta al principio/final de la lista.
- Se puede utiizar cualquier estructura de datos para resolver colisiones.
- Si el tamaño de la tabla es grande y la función de dispersión es buena, las listas serán cortas.

Factor de carga λ

$$\lambda = \frac{\text{Nº de claves en la tabla}}{\texttt{TableSize}} = \begin{cases} \text{“Factor de carga”} \ (\lambda \approx 1) \\ \text{“Tamaño medio de una lista”} \end{cases}$$

1. Coste de una búsqueda: $\underbrace{O(1)}_{\text{Hash(X)}} + \underbrace{O(\lambda)}_{\text{Recorrer lista}}$
2. Búsqueda sin éxito = promedio de λ enlaces recorridos: $O(\lambda)$
3. Búsqueda con éxito = promedio de $1 + \lambda/2$ enlaces recorridos: $O(\lambda/2)$

5.4.2. Dispersión cerrada

Se basa en probar la sucesión de celdas:

$$h_i(X) = [\text{Hash}(X) + F(i)] \ \texttt{mod Tablesize}, \qquad \text{con } F(0) = 0$$

donde F es la función estrategia de resolución de colisiones.

- A medida que los datos se almacenan en la tabla se necesita un tamaño de tabla mayor.
- Generalmente el factor de carga suele estar por debajo de $\lambda = 0.5$.
- El borrado estándar no es factible. Se necesita *borrado perezoso*.

Exploración lineal: $F(i) = i$

- Mientras el tamaño de la tabla sea suficientemente grande siempre encuentra una celda vacía, pero el tiempo para encontrar la celda puede ser elevado.
- Asumiendo `TableSize` grande e independencia, el número de pruebas por término medio es $1/(1-\lambda)$.
- Problemas:
  - La independencia de operaciones no se cumple y se forman bloques de celdas ocupadas.
  - Efecto del agrupamiento primario: cualquier clave dispersada en el agrupamiento necesitará varios intentos para resolver la colisión y finalmente acabará en ese agrupamiento.
- El número de pruebas esperado es: $\frac{1}{2}(1 + 1/(1-\lambda)^2)$
- Inserciones y búsquedas sin éxito necesitan el mismo número de pruebas.

- Búsquedas con éxito: $\frac{1}{2}(1 + 1/(1 - \lambda))$

Exploración cuadrática: $F(i) = i^2$

- Tabla **medio vacía** y `TableSize` es primo: la inserción de un nuevo elemento está garantizada.
- Más de la mitad de la tabla llena: no se garantiza encontrar una celda libre.
- Los elementos que se dispersan a la misma posición probarán las mismas celdas alternativas (agrupamiento secundario).

Exploración doble: $F(i) = i * \text{hash}_2(X), \ \ i \neq 0$

- Es importante asegurar que se pueden intentar todas las celdas.
- Función correca: $\text{hash}_2(X) = R - (X \ \texttt{mod} \ R)$ con R primo menor que `TableSize`.
- Importante: `TableSize` número primo.

Tema 6: Búsqueda en memoria secundaria

Los algoritmos y estructuras para procesr datos en memoria secundaria deben considerar:

- Coste de transferencia entre memoria principal y secundaria.
- Coste de transferencia por bloque de registros.
- Método eficiente de búsqueda depende de las características del sistema de computación (arquitectura y sistema operativo).

6.1. Árboles n-arios de búsqueda

- Almacena los nodos del árbol en disco y los apuntadores son las direcciones de disco.
- Para un fichero de n registros, se necesitan $\log_2 n$ búsquedas en disco.
- Para minimizar el número de accesos a disco se agrupan los nodos en páginas.
- La organización óptiama en pagíans es difícil de obtener durante la construcción del árbol.
- La ubicación secuencial almacena los nodos por orden de entrada (no por su localidad en el árbol), pero obtiene peor rendimiento en la búsqueda.

6.2. Árboles B

Los árboles B son árboles de búsqueda ordenados.

- **Grado**: Número máximo de hijos que puede tener un nodo (también denominado **orden**).
- En un árbol de grado m, un nodo puede tener:
  - Entre $\lceil (m/2) - 1 \rceil$ y $m - 1$ claves.
  - Entre $m/2$ y m hijos (excepto la raíz, que puede tener menos).
- Todas las hojas están al mismo nivel (equilibrio, altura logarítmica respecto al número de claves almacenadas).
- La profundidad es el número máximo de consultas parra encontrar una clave.

Criterios para dimensionar el orden de un árbol B

- Limitar su profundidad para que el número máximo de consultas sea pequeño.
- Cuanto mayor sea el orden, menor será la profundidad.
- Para indexar ficheros en disco interesa minimizar el número de accesos utilizando el tamaño del cluster como orden (teniendo en cuenta la arquitectura del disco).

6.2.1. Operaciones sobre el árbol B

División

- Situación: Se intenta insertar una clave en un nodo lleno.
- Solución: Reestructurar el árbol/subçarbol para que se pueda insertar elementos en él.

Fusión

- Situación: Borraruna clave de un nodo y éste se queda con un número de claves inferior al mínimo.
- Solución: Reestructurar el árbol llevando las claves de ese nodo a sus hermanos.

Búsqueda del elemento x

- Se desciende desde la raíz hasta el nodo que contenga el elemento o hasta las hojas (búsqueda sin éxito).
- En cada nodo se busca en la lista de claves. Si no se encuentra, se pasa la búsqueda al hijo asociado entre la mayor clave menor que x y a la menor clave mayor que x, o al último hijo si el valor buscado es mayor que todas las claves.

Inserción del elemento x

- Se desciende hasta la hoja que debería contener elemento.
- Se inserta en la posición adecuada de la lista de claves.
- Si se supera el número máximo de claves, el nodo se divide, llevando su clave a la posición media del padre.
- El proceso se repite mientras se supere el número máximo de claves en el nodo.

Borrado de un nodo interno

- Se desciende desde la raíz hasta el nodo que contenga el elemento a borrar.
- Se intercambia con el **máximo del hijo izquierdo** o con el **mínimo del hijo derecho** (se escoge el hijo con más claves).
- Se borra el elemento en el hijo en el que se hizo el intercambio.

Borrado de un nodo hoja

- Se elimina de la lista de claves.
- Si el número de claves es inferior al mínimo:
  - Se intenta una transferencia con el hermano que contenga más claves..
  - Si no es posible, se fusiona con el hermano.
  - La fusión toma un elemento del padre, por lo que éste a su vez puede necesitar transferencias o fusiones.
  - El proceso se repite mientras sea necesario.

6.2.2. Variantes

Árboles con prerecorrido

- Antes de insertar se realiza una búsqueda que divide todos los nodos llenos.
- Número máximo de claves: $2m + 1$.

Árboles B+

- Sólo las hojas contienen elementos. Los nodos interiores contienen claves para dirigir la búsqueda (los nodos hoja también tienen claves)
- Las hojas se implementan mediante una lista doblemente enlazada.

Árboles B*

- Número mínimo de claves: $2/3$ de la capacidad.
- Se fusionan 3 nodos en 2, y se dividen 2 nodos en 3.

Tema 7: Ordenación Interna y Externa

Inversión: Cualquier par (i, j) tal que $i < j$ y $T[i] > T[j]$.

Teorema: Intercambiar dos elementos adyacentes elimina una inversión.

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

7.1. Ordenación por Inserción (`InsertionSort`)

```
function InsertionSort (var v[1..n]):
    for i:=2 to n:
        x := v[i];
        j := i-1;
        while j>0 and v[j]>x:
            v[j+1] := v[j];
            j := j-1;
        v[j+1] := x;
```

Análisis de `InsertionSort`

- Peor caso: $T(n) = \Theta(n^2)$
- Mejor caso: $T(n) = \Theta(n)$
- Caso medio: $T(n) = O(n^2)$

- Sea I el número de inversiones de un array, la complejidad queda determinada por $O(I + n)$.
- Peor caso: $T(n) = \sum_{i=2}^{n} i = \Theta(n^2)$
  - Entrada ordenada de forma inversa.
  - i comparaciones por cada iteración del bucle externo.
  - $I = O(n^2)$
- Mejor caso: $T(n) = \Theta(n)$
  - Entrada ordenada ($I = 0$).
  - Número de inversiones lineal ($I = O(n)$).
- Caso medio: $T(n) = O(n^2)$
  - Suponiendo que el array no tiene duplicados y cualquier permutación es equiprobable. Sea $T[1, ..., n]$ un array y $T_i[1, ..., n]$ su inverso, la suma de inversiones en ambos arrays es igual al número de pares de T.
  - El número de pares de un array de tamaño n es: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
  - Por la hipótesis de equiprobabilidad, el número de inversiones en uno de los arrays será la mitad del total:

$$T(n) = \frac{n(n-1)}{4} = O(n^2)$$

7.2. Ordenación de Shell (`ShellSort`)

- Basado en una secuencia de t incrementos (distancias con las que realizar inversiones). Deben ser números enteros, positivos, decrecientes y el último siempre debe ser 1.
- Bucle de t iteraciones para cada incremento.
- En la iteración k se utiliza el incremento h_k y una vez terminada los elemenos separados por h_k posiciones están ordenados (vector h_k-ordenado).
- **Teorema**: Un vector h_k ordenado que se h_{k-1} ordena, sigue estando h_k ordenado (donde $h_k > h_{k-1}$).
- Incrementos de Shell: $h_k = \lfloor h_{k-1}/2 \rfloor$

`ShellSort` (incrementos de Shell)

```
function ShellSort (var v[1..n]):
    gap := n;
    repeat
        gap := gap / 2;
        for i := gap+1 to n:
            tmp := v[i];
            j := i;
            keepgoing := true;
            while j-gap > 0 and keepgoing:
                if tmp < v[j-gap]:
                    v[j] := v[j-gap];
                    j := j-gap;
                else:
                    keepgoing := false;
            v[j] := tmp;
    until gap = 1;
```

7.2.1. Otros incrementos

	Incrementos	Caso medio	Peor caso
Hibbard	$1, 3, 7, ..., 2^k - 1$	$O(n^{5/4})$ (simulación)	$\Theta(n^{3/2})$ (teorema)
Sedgewick	$1, 5, 19, 41, 109...$	$O(n^{7/6})$ (simulación)	$O(n^{4/3})$ (simulación)

7.2.2. Análisis del peor caso

Vamos a demostrar que la complejidad del peor caso de la ordenación Shell con incrementos de Shell está acotada inferiormente por $\Omega(n^2)$ y superiormente por $O(n^2)$:

Mejor situación:

- Los $n/2$ mayores valores están en las posiciones pares y los $n/2$ menores valores en las impares, pero ordenados entre sí.
- El i-ésimo menor elemento está en la posición $2i-1$, $i \leq n/2$.
- Todas las inversiones se realizan cuando gap=1y para ordenar el elemento i se necesitan $i-1$ inversiones.
- Complejidad: $\sum_{i=1}^{n/2} i - 1 = \Omega(n^2)$

Peor situación:

- El trabajo realizado en cada iteración k con el incremento h_k son h_k ordenaciones por inserción sobre n/h_k elementos cada una.

$$h_k \cdot O\left(\frac{n}{h_k}\right)^2 = O\left(\frac{n^2}{h_k}\right)$$

- En el conjunto de iteraciones del algoritmo:

$$T(n) = O\left(\sum_{i=1}^{t} \frac{n^2}{h_i}\right) = O(n^2)$$

7.3. Ordenación por montículos (`heapSort`)

Ordenación por montículos

```
function HeapSort (var v[1..n]):
    h := createHeap(v);
    for i := 1 to n:
        v[i] := getMin(h);
        removeMin(h);
```

Creación del montículo

```
function createHeap (var v[1..n]):
    copy v[1..n] to h[1..n];
    for i := n / 2 to 1 by -1:
        sink(h,i);
```

- Árbol binario equilibrado donde el padre es siempre menor que los hijos.
- Para mejorar la complejidad espacial, usar el mismo array para crear el montículo.
- **Complejidad de `sink`**: $T(n) = O(\log n)$ (altura del árbol).
- **Complejidad de `createHeap`**: $T(n) = O(n)$
  - Nº de intercambios de `sink(h,i)` $\leq$ altura del nodo i.
  - Nº de intercambios totales $\leq$ suma de las alturas de los nodos $= O(n)$
- **Complejidad**: $T(n) = O(n \log n)$
- El peor caso sigue siendo $O(n \log n)$, pero tiene peores tiempos que `ShellSort` con incrementos de Sedgewick.

7.4. Ordenación por fusión (MergeSort)

- Algoritmo basado en el paradigma Divide y Vencerás:
  1. Divide el problema en 2 mitades.
  2. Se resuelve recursivamente.
  3. Fusión de las mitades ordenadas en un vector ordenado.
- Fusionar un vector cuyas mitades están ordenadas para obtener un vector ordenado (función lineal).
- Mejora: Ordenación por inserción para vectores pequeños (determinar un umbral empíricamente).

Ordenación por fusión MergeSort

```
function Merge (var T[i..j], center):
    k := i;         # recorre T[i..center]
    l := center+1; # recorre T[center..j]
    m := i;         # recorre Aux[i..j]
    while k <= center and l <= j:
        if T[k] <= T[l]:
            Aux[m] := T[k];
            k := k+1;
        else:
            Aux[m] := T[l];
            l := l+1 ;
        m := m+1 ;
    while k <= center:
        Aux[m] := T[k];
        k := k+1;
        m := m+1;
    while l <= j:
        Aux[m] := T[l];
        l := l+1;
        m := m+1;
    for m := i to j:
        T[m] := Aux[m];
```

MergeSort recursivo

```
function RecursiveMergeSort (var T[i..j]):
    if i+THRESHOLD < j:
        center := ( i+j ) div 2;
        RecursiveMergeSort (T[i..centre]);
        RecursiveMergeSort (T[centre+1..j]);
        Merge ( T[i..j], center );
    else:
        InsertionSort ( T[i..j] );
```

Si suponemos que THRESHOLD=0:

$$T(n) = \begin{cases} T(1) = O(1) & n = 1 \\ 2T(n/2) + O(n) & n > 1 \end{cases} \implies T(n) = \Theta(n \log n)$$

7.5. Ordenación rápida (QuickSort)

Se basa en el paradigma Divide y Vencerás:

1. Elegir un pivote.
2. Clasificar el resto de los elementos del array poniendo los menores que el pivote a la izquierda y los mayores a la derecha.
3. Llamadas recursivas para ordenar la parte izquierda y la parte derecha.
4. El array resultante estará ordenado.

Selección del pivote:

- Pivote ideal: Aquel que tiene el mismo número de elementos mayores que menores que él en el array (mediana).
- Primer valor del array (si la entrada está ordenada o parcialmente ordenada: $T(n) = O(n^2)$).
- Pivote aleatorio (depende del generador de números aleatorios).
- Mediana de tres valores.

Selección de la mediana de tres valores

```
function Median3 (varT[i..j] )
    center := ( i+j ) / 2 ;
    if T[i] > T[center]:
        swap ( T[i], T[center] );
    if T[i] > T[j]:
        swap ( T[i], T[j] );
    if T[center] > T[j]:
        swap ( T[center], T[j] ) ;
    swap ( T[center], T[j-1] );
```

QuickSort

```
function Qsort (var T[i..j] ):
    if i+THRESHOLD <= j:
        Median3 ( T[i..j] );
        pivot := T[j-1];
        k := i;
        m := j-1;
        repeat
            repeat k := k+1 until T[k] >=  pivot;
            repeat m := m-1 until T[m] <= pivot;
            swap ( T[k], T[m] );
        until m <= k;
        swap ( T[k], T[m] );        # deshace el último intercambio
        swap ( T[k], T[j-1] );      # pivote en posición k
        Qsort ( T[i..k-1] );
        Qsort ( T[k+1..j] );

function Quicksort ( var T[1..n] ):
    Qsort ( T[1..n] );
    InsertionSort ( T[1..n] );
```

- En caso de encontrar duplicados del pivote, lo mejor es parar ambos índices.
- Utilizar un umbral para determinar los casos base que se solucionan on `InsertionSort`.
- Mejora: Hacer una única llamada a `InsertionSort` con todo el vector ($T(n) = O(n)$)

7.5.1. Análisis de `QuickSort`

- Suponemos un pivote aleatorio y sin umbral:

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

  donde z (número de elementos del array menores que el pivote) depende del pivote.

- Peor caso (el pivote es siempre el menor o el mayor elemento): $T(n) = T(n-1) + cn = O(n^2)$.
- Mejor caso (el pivote coincide con la mediana): $T(n) = 2T(n/2) + cn = O(n \log n)$
- Caso medio: $T(n) = O(n \log n)$

$$\text{Promedio de los posibles valores de } z: \quad \frac{1}{n} \sum_{x=0}^{n-1} T(x)$$

  Como el promedio de los posibles valores de z es el mismo que el promedio de los posibles valores de $n - z - 1$, concluimos que, en términos de valores esperados, $T(z) = T(n-z-1)$. Por tanto:

$$\text{En términos medios:} \quad T(n) = 2T(z) + cn = O(n \log n)$$

7.6. Ordenación externa

7.6.1. Ordenación externa por fusión

Tenemos una base de datos no ordenada de tamaño n en memoria externa:

1. Dividir la base de datos en k trozos tales que quepan en RAM ($k/n < m$).
2. Cargar cada trozo en memoria RAM y ordenarlo mediante ordenación interna.
3. Tomar k buffers de memoria RAM de tamaño $m/(k+1)$ y leer los primeros $m/(k+1)$ elementos de cada trozo a cada buffer.
4. Aplicar un `MergeSort` a k bandas, e ir almacenando el resultado de la fusión (los elementos ordenados) en un buffer de salida de tamaño $m/(k+1)$.
5. Cuando el buffer de salida se llena, se escribe al fichero de salida.
6. Cuando uno de los buffers de entrada se llena, se rellena con más datos (otros $m/(k+1)$ elementos del trozo).
7. No es un algoritmo in-place.

7.6.2. Ordenación externa por distribución

- Encontrar una serie de pivotes para particionar la lista de datos en memoria externa.
- Se ordena recursivamente cada partición.
- Problema: determinar el número de pivotes + encontrar pivtoes que produzcan particiones equilibradas.

Tema 8: Grafos

8.1. Introducción: Terminología

Un **grafo** $G = \{V, E\}$ consiste en un conjunto de vértices V y un conjunto de aristas E.

- Si las aristas (v, w) tiene un orden, entonces el grafo es dirigido (digrafos).
- Las aristas pueden incorporar un tercer elemento denominado *peso* o *coste*.
- Un **vértice** w es adyacente a v si y sólo si $(v, w) \in E$.

Un **camino** en un grafo es una secuencia de n vértices $w_1, w_2, ..., w_n$ tal que $(w_i, w_i + 1) \in E$ para $1 \leq i < n$.

- La **longitud** es el número de aristas del camino: $(n - 1)$-
- Un **camino simple** es un camino con todos sus vértices diferentes.

Un **ciclo** en un grafo dirigido es un camino de longitud al menos 1 tal que el vértice inicial es el vértice final $(w_1 = w_n)$.

- **Grafo conexo**: Grafo no dirigido en el que existe un camino desde cada vértice a cualquier otro vértice.
- **Grafo fuertemente conexo**: Grafo dirigido en el que exite un camino desde cada vértice a cualquier otro vértice.
- **Grafo completo**: Grafo en el que existe una arista entre cada par de vértices.

8.1.1. Matriz de adyacencia

La matriz de adyacencia A es un array bidimensional donde cada arista (u, v) se establece en la matriz como $A[u, v] = 1$ (y el resto de entradas tendrán valor 0). Si la arista tiene un peso asociado se establecerá la posición $A[u, v]$ con ese peso.

Las necesidades de espacio de almacenamiento son $\Theta(|V|^2)$ (adecuada si el grafo es denso).

8.1.2. Lista de adyacencia

La lista de adyacencia es un array para cada vértice u de un grafo donde se mantienen los vértices adyacentes. Se necesita un espacio de almacenamietno $\Theta(|E| + |V|)$ y es una solución más adecuada si el grafo es disperso.

Si el grafo es **no dirigido** cada arista (u, v) aparecerá en dos listas de adyacencia, por lo que el espacio de almacenamiento necesario será el doble.

8.2. Recorrido en profundidad

Sea $G = (V, E)$ un **grafo no dirigido** cuyos nodos queremos visitar. Para realizar un recorrido en profundidad elegimos $v \in V$ como nodo inicial y aplicamos el siguiente algoritmo:

1. **Situación inicial**: Tenemos un grafo y un array donde vamos marcando los nodos visitados.
2. **Inicialización del algoritmo**: Marcamos todos los nodos como no visitados.
3. Para cada nodo v que no ha sido visitado se llama a la función recursiva `dfs`.
4. A la función recursiva `dfs` (recorrido en profundidad) se le introduce siempre un nodo v que no ha sido visitado.
5. En la función recursiva `dfs` se marca el nodo v como visitado.
6. Se recorren todos los nodos adyacentes a v y se vuelve a llamar a la función `dfs` para aquellos que no han sido visitados.
7. El algoritmo finaliza cuando todos los nodos han sido visitados.

Recorrido en profundidad

```
function dfsearch (Graph):
    for each v in V:
        mark[v] := not-visited;
        for each v in V:
            if mark[v] <> visited:
                return dfs(v)
```

Función recursiva

```
function dfs(v):
    mark[v] := visited;
    for each w adjacent to v:
        if mark[w] <> visited:
            return dfs(w)
```

8.2.1. Análisis del recorrido en profundidad

Sea n el número de nodos y m el número de aristas:

- Cada nodo se visita una única vez (n llamadas a `dfs`): $\Theta(n)$.
- En cada visita se revisa la marca de los nodos adyacentes. Como mucho: $\Theta(m)$.

$$T(n) = \Theta(n + m)$$

- El recorrido en profundidad de un **grafo conexo** asocia un árbol expandido mínimo al grafo.
- En **grafos dirigidos** la diferencia consiste en la interpretación de adyacencia. El nodo w es adyacente al nodo v si y sólo si existe una arista dirigida (v, w).
- En **grafos no conexos** las aristas utilizadas para visitar todos los nodos pueden formar un bosque de varios árboles.
- Aplicaciones: recorrido en preorden de un grafo, ordenación topológica (grafo dirigido acíclico).

8.2.2. Versión no recursiva

1. Se crea una pila S, se marca el nodo v como visitado y se mete en la pila.
2. Se repiten iterativamente estos pasos hasta que la pila esté vacía.
   - Mientras que haya algún nodo w no visitado adyacente al nodo que está en la cima de la pila S, meter el nodo w en la pila.
   - Sacar el nodo de la cima de la pila.

Recorrido en anchura (recursivo)

```
function dfs2(v): mark[1..n]
    CreateStack(S);
    mark[v] := visited;
    Push(v,S);
    while not EmptyStack(S):
        while there is a node w adjacent to Top(S) that mark[w] != visited:
            mark[w] := visited;
            Push(w,S);
        Pop(S);
```

8.3. Recorrido en anchura

El recorrido en anchura se basa en visitar **todos** los vecinos de un nodo v.

1. Se crea una cola Q, se marca el nodo v como visitado y se mete en la cola.
2. Mientras que haya elementos en la cola se repiten estos pasos de forma iterativa:
   - Se retira el elemento u de la cola (obteniendo este una posición en el recorrido).
   - Por cada nodo w no visitado adyacente a u, se marca como visitado y se mete en la cola.

Recorrido en anchura

```
function bfs(v)
    CreateQueue(Q);
    mark[v] := visited;
    Enqueue(v,Q);
    while not EmptyQueue(Q):
        u := Dequeue(Q);
        for each node w adjacent to u:
            if mark[w] <> visited:
                mark[w] := visited;
                Enqueue(w,Q);
```

Posibles aplicaciones:

- Asociar un árbol a un grafo.
- Para grafos conexos se obtiene el árbol expandido mínimo.
- Exploración parcial de un grafo.
- Encontrar el camino más corto desde un nodo a otro.

$$T(n) = \Theta(n+m)$$

8.4. Juegos de estrategia. El juego de Nim

Suponemos que tenemos un montón de palillos (al menos dos) en la mesa entre dos jugadores. El primer jugador puede retirar tantos palillos como quiera, pero debe **retirar al menos uno** y debe **dejar al menos uno**. Cada jugador en su turno debe retirar **al menos un palillo** y **como mucho el doble que su oponente**. El jugador que retire el último palillo gana.

Para formalizar este proceso vamos a representar el juego mediante un grafo dirigido.

- Cada nodo $< i, j >$ correponde a una posición en el juego donde:
  - i es el número de palillos que permanecen en la mesa.
  - j es el número máximo de palillos que se pueden retirar.
- Cada arista se corresponde con un movimiento de una posición a otra y el **peso** será el número de palillos retirados.
- Tipos de nodos:
  - Posiciones perdedoras: $< 0, 0 >$ es la situación final perdedora.
  - Posiciones ganadoras.

Objetivo: Identificar los movimientos a través de los cuales podemos llegar a las posiciones ganadoras.

- El nodo $< i, j >$ representa una situación ganadora si existe un sucesor en una situación perdedora.
- El nodo $< i, j >$ representa una situación perdedora si todas las sucesoras son posiciones ganadoras.

Función ganadora

```
function Winning(i,j): boolean
    /* Hipotesis: 0 <= j <= *i/
    for k := 1 to j:
        if not Winning(i-k, min(2k,i-k)):
            return true
    return false
```

8.4.1. Juego de Nim con programación dinámica

La solución que nos da la programación dinámica es anotar en una tabla V las situaciones del juego. De esta forma, una situación ganadora aparecerá anotada en la tabla como `V[i,j]=true`.

Para calcular si una posición $< i, j >$ es ganadora se debe calcular antes:

- $V[r, s],\ 1 \leq s \leq r \leq i$ (las filas y columnas de V anteriores a la fila i y columna i). Esto se traduce en mirar las situaciones donde quedan menos palillos que i.
- $V[i, s],\ 1 \leq s < j$ (columnas de V anteriores a la columna j en la fila i). Esto se traduce en mirar las situaciones donde quedan i palillos y se pueden quitar hasta $j - 1$ palillos.
- $V[0, 0] =$ `false`.

Función ganadora con Programación Dinámica

```
function DP_Winning(n):
    for i := 1 to n:
        for j := 1 to n:
            k := 1;
            while k < j and V[i-k, min(2k,i-k)]:
                k := k + 1;
            V[i,j] := not V[i-k, min(2k,i-k)];
```

8.4.2. Juego de Nim con la función `memoria`

Esta implementación del juego de Nim consiste en marcar los nodos ya visitados en un array booleano bidimensional denominado `known` de tamaño $n+1 \times n+1$ y de índices de 0 a n.

Inicialización

```
V[0,0] := false;
known[0,0] := true;
for i := 1 to n:
    for j := 1 to i:
        known[i,j] := false
```

Nim con memoria

```
function Nim (i,j)
    if known[i,j]:
        return V[i,j];
    known[i,j] := true;
    for k := 1 to j:
        if not Nim(i-k, min(2k,i-k)):
            V[i,j] := true;
            return true;
    V[i,j] := false;
    return false;
```

8.5. Algoritmos de retroceso o *backtracking*

En un **grafo implícito** las partes relevantes del grafo se pueden construir a medida que la búsqueda progresa.

- Si la búsqueda tiene éxito, se ahorra tiempo de computación y espacio al no construir el grafo completo.
- La búsqueda tiene éxito si se puede definir completamente una solución. En este caso, el algoritmo puede parar o continuar buscando soluciones alternativas.
- La búsqueda no tiene éxito si en algún estado la solución parcial construida no se puede completar. En este caso, la búsqueda retrocede eliminando los elementos que fueron añadidos en cada estado.

8.5.1. El problema de la mochila

Supongamos que se dispone de n tipos de objetos y un número adecuado de cada uno. Para $i = 1, ..., n$, un objeto de tipo i tiene peso $w_i \geq 0$ y un valor $v_i \geq 0$. La mochila puede soportar un peso máximo W.

Objetivo: Lenar la mochila maximizando el valor de los objetos incluidos respetando la limitación de capacidad y con la **restricción** de que no se puede fraccionar un objeto.

Solución: Utilizando *backtracking* explorando el grafo implícito.

1. Inicialmente la solución parcial está vacía.

2. El algoritmo de *backtracking* explora el árbol como la búsqueda en profundidad y construye soluciones parciales que se van extendiendo.

3. En cada paso hacia atrás, se elimina el último elemento incluido en la solución parcial.

Pseudocódigo: Cada llamada calculará el valor de la mejor carga utilizando los objetos de i a n cuyo peso no exceda el valor r.

Función `backpack`

```
function backpack(i,r): value
    b := 0;
    for k :=i to n:
        if w[k] <= r:
            b := max(b, v[k]+backpack(k,r-w[k]));
    return b;
```

8.5.2. El problema de las 8 reinas

Objetivo: Colocar las ocho reinas en un tablero de ajedrez de tal forma que ninguna de ellas se amenace.

Solución obvia: Probar todas las combinaciones posibles y determinar cuál es la solución ($\binom{64}{8} = 4426165368$ posiciones a comprobar).

- **Mejora 1**: No colocar más de una reina en la misma fila.
  - 8^8 posiciones para comprobar.
  - Se reduce el número de posiciones a comprobar un vector de 8 elementos.
  - El algoritmo encuentra una solución comprobando solamente 1299852 posiciones.
- **Mejora 2**: No colocar más de una reina en la misma fila ni en la misma columna.
  - $8! = 40320$ posiciones a comprobar.
  - Se consideran 2830 posiciones.
- **Problema**: Ninguna realiza una comprobación hasta que las 8 reinas están colocadas.

Una **solución k-prometedora** (para $k = 0, ..., 8$) es un vector $v[1, ..., k]$ de enteros entre 1 y 8 si y sólo si ninguna de las k reinas colocadas en las posiciones $\{(1, v[1]), ..., (k, v[k])\}$ amenaza a ninguna de las demás.

$$v \text{ es } k\text{-prometedor} \iff v[i] - v[j] \notin \{i - j, 0, j - i\}, \ \forall 1 \leq i, j \leq k$$

Sea N un conjunto de vectores k-prometedores y $G = (N, A)$ un grafo digirido tal que:

$$(U, V) \in A \iff \exists k, 0 \leq k < 8 \text{ tal que } \begin{cases} U \text{ es } k\text{-prometedor} \\ V \text{ es } (k+1)\text{-prometedor} \\ U[i] = V[i], \forall i \in [1, .., k] \end{cases}$$

- El grafo es un árbol y su raíz es un vector vacío ($k = 0$).

- Las hojas son soluciones ($k = 8$) o finales sin salida.
- La solución está determinada mediante un recorrido en profundidad.
- Nº de nodos del árbol <8.
- Es suficiente con explorar 114 nodos para encontrar la primera solución.

El grafo es un árbol y su raíz es un vector vacío ($k = 0$). Las hojas son soluciones ($k = 8$) o finales sin salida ($k < 8$). La solución está determinada a partir de una exploración del árbol mediante un recorrido en profundidad.

Pseudocódigo: Llamada incial `queens(0, , , )`.

Función de `Queens`

```
function queens (k, col, diag45, diag135):
    /*  sol[1..k] is k-prometedor;
        col := sol[i]:          1 ≤ i ≤ k;
        diag45 := sol[i]-i+1:   1 ≤ i ≤ k;
        diag135 := sol[i]+i-1:  1 ≤ i ≤ k; */
    if k = 8:
        write sol    /* Un vector 8-prometedor es solución */
    else
        for j := 1 to 8:
            if j not in col and (j-k) not in diag45 and (j+k) not in diag135:
                sol[k+1] := j;
                queens (k+1, col+{j}, diag45 + {j-k}, diag135 + {j+k})
```

8.6. Ramificación y poda (*Branch and Bound*)

La técnica de ramificación y poda es una generalización de la técnica de retroceso *backtracking*. Se basa en explorar un **grafo dirigido implícito** y realizar un recorrido sistemático por un árbol de estados de un problema usando una estrategia de **ramificación** y técnicas de **poda** para eliminar nodos que no llevan a soluciones óptimas (en cada nodo se estima cotas del beneficio que se puede obtener a partir del mismo). Suelen ser algoritmo de orden exponencial en su peor caso.

Backtracking

1. Al generar un nuevo hijo del nodo en curso, este hijo pasa a ser el nodo en curso.
2. Los únicos nodos vivos son los que están en el camino de la raíz al nodo en curso.
3. La función poda comprueba únicamente si un nodo en concreto puede llevar a la solución o no.

Branch & Bound

1. Se generan todos los hijos del nodo en curso antes de que cualquier otro nodo vivo pase a ser el nuevo nodo en curso.
2. Puede haber más nodos vivos que se almacenan en una lista de nodos vivos.
3. Se acota el valor de la solución a la que nos puede conducir un nodo concreto:
   - Podar el árbol si sabemos que no nos va a llevar a una solución mejor de la que ya tenemos.
   - Establecer el orden de ramificación para comenzar explorando las ramas más prometedoras del árbol.

8.6.1. Descripción del *Branch and Bound*

1. Se explora un árbol comenzando a partir de un problema raíz y su región factible.
2. Se aplican funciones de **acotación** al problema raíz, para el que se establecen cotas inferiores y/o superiores.
3. A medida que se explora el árbol y la región factible del problema raíz, si la estimación de la cota local en un "camino" cumple las condiciones establecidas (dadas por las cotas inferiores y superiores), encontramos la solución óptima del problema y la búsqueda termina.
4. Si se encuentra una solución óptima para un nodo concreto, este será una solución factible pero no necesariamente su óptimo global.
5. Cuando en un nodo su cota local es peor que el mejor valor conocido en la región, no puede existir un óptimo global en el subespacio de la región factible asociada a ese nodo y, por tanto, ese nodo debe ser eliminado (poda).
6. La búsqueda prosigue hasta que:
   - Se examinan o podan todos los nodos.
   - Se cumple algún criterio preestableceido sobre el mejor valor encontrado y las cotas locales de los subproblemas aún no resueltos.

8.6.2. Estimadores y cotas

	Problema de maximización	Problema de minimización
Valor	Beneficio	Coste
Cota local (CL)	Cota superior (CL $\geq$ Óptimo local)	Cota inferior (CL $\leq$ Óptimo local)
Cota global (CG)	Cota inferior (CG $\leq$ Óptimo global)	Cota superior (CG $\geq$ Óptimo global)

Cota local

- Asegura que no se alcanzará nada mejor al expandir un nodo determinado. Se calcula localmente para cada nodo i.
- El **Óptimo Local (i)** es el coste/beneficio de la mejor solución que se podría alcanzar al expandir el nodo i.
- La cota local debe ser mejor o igual que **Óptimo Local (i)**.
- Cuanto más cercana sea la cota local a **Óptimo Local (i)**, mejor será la cota y más se podará el árbol.
- Debemos matener un equilibrio entre la eficiencia del cálculo de la cota y su calidad.

Cota global

- Es el valor de la mejor solución estudiada hasta el momento (una estimación del óptimo global).
- Debe ser peor o igual al coste/beneficio de la solución óptima.
- Inicialmente se le puede asignar el valor obtenido por un algoritmo voraz o, en su defecto, el peor valor posible.
- Se actualiza siempre que se alcanza una solución que mejore su valor actual.
- Cuanto más cercana sea al coste/beneficio óptimo, más se podará el árbol, por lo que es importante encontrar buenas soluciones cuanto antes.

8.6.3. Estrategia de poda

- Se podan aquellos nodos que no cumplan las restricciones implícitas (soluciones parciales no factibles).
- Se podan aquellos nodos cuya cota local sea peor que la cota global.
- La poda no perderá ninguna solución óptima

8.6.4. Estrategia de ramificación

Se utiliza una **lista de nodos vivos**: nodos generados pero aún no explorados, pendientes de tratar por B&B.

- Búsqueda ciega (estrategia LIFO con una pila o estrategia FIFO con una cola).
- Búsqueda informada: Explorando primero los nodos más prometedores (Estrategia de Menor Coste o Estrategia de Máximo Beneficio). En caso de empate, utilizar una estrategia LIFO o FIFO.
- En cada nodo se dispone:
  - Cote inferior de coste/beneficio.
  - Coste/beneficio estimado.
  - Cota superior de coste/beneficio.
- El árbol se ramificará según los valores estimados y se poda según los valores de la cotas.

8.6.5. Observaciones

- Sólo se comprueba el citerio de poda cuando se introduce o se saca un nodo de la lista de nodos vivos.
- Si un descendiente de un nodo es una solución final, no se introduce en la lista de nodos vivos, sino que se comprueba si es mejor solución que la actual. Si lo es, se actualiza la menor de las cotas superiores hasta el momento y se guarda como mejor solución hasta el momento.

Cuando la cota superior y la cota inferior tienen el mismo valor, tenemos tres opciones:

- Opción A: No podar (no sabemos si tiene la solución).
- Opción B. Usar dos variables de poda.
  - CI: Cota inferior actual de una solución parcial.
  - voa: Valor óptimo actual de una solución encontrada.

  - Podar el nodo x si CS(x) <CI o si CS(x) <voa.

- Opción C: Generar directamente el nodo solución usando el método utilizado para calcular la cota. Si CI(x) = CS(x), entonces el nodo x es una solución para calcular la cota.

8.6.6. Tiempo de ejecución

El tiempo de ejecución de un algoritmo B&B depende del número de nodos recorridos (y de la efectividad de la poda) y del tiempo empleado en cada nodo (tiempo necesario para hacer **estimaciones de coste** y para **gestionar la lista de nodos vivos** en función de la estrategia de ramificación).

En el peor caso, el tiempo del algoritmo B& B será igual al de un algoritmo *backtracking*, o incluso peor si se tiene en cuenta el tiempo que requiere la lista de nodos vivos. En promedio, no obstante, se suelen obtener mejoras con respecto a la técnica de retroceso.

Para que un algoritmo B&B sea eficiente:

- Hacer estimaciones de coste muy precisas (mejor poda pero más tiempo para realizar las estimaciones).
- Hacer estimaciones de coste poco precisas (poco tiempo para realizar estimaciones pero el número de nodos explorado es muy elevado).

Se debe buscar un equilibrio entre la precisión de las cotas y el tiempo empleados en calcularlas.

Bloque IV
Problemas NP-Completos

Tema 9: NP-Completo y NP-Difícil

9.1. Introducción a $\mathcal{P}$ y $\mathcal{NP}$

Algoritmia: Diseño de algoritmos específicos y lo más eficientes posibles para un problema dado.

Complejidad computacional: Estudio del problema, considerando todos los algoritmos posibles para resolverlo (los conocidos y los desconocidos).

Clase de complejidad $\mathcal{P}$: Conjunto de problemas que se pueden resolver en tiempo polinómico $O(n^k)$ (problemas tratables).

Clase de complejidad $\mathcal{NP}$: Conjunto de problemas cuya solución se puede comprobar en tiempo polinómico.

- Está demostrado que $\mathcal{P} \subset \mathcal{NP}$. Es decir, si tenemos un algoritmo polinómico para obtener la solución y podemos usarlo para comprobarla.
- Pregunta: ¿$\mathcal{P} = \mathcal{NP}$?
- Si $\mathcal{P} \neq \mathcal{NP}$ nunca encontraremos soluciones polinómicas a problemas que están en $\mathcal{NP}$ porque no existen.
- Si $\mathcal{P} = \mathcal{NP}$ existen soluciones polinómicas a todos los problemas de $\mathcal{NP}$.

9.2. Máquinas 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 donde se leen/escriben símbolos.
- Una **cabeza lectora/escritora** que apunta a una posición de la cinta y se puede mover.
- Un **control de estado finito**: un registro que almacena el estado en el que está la máquina en un momento dado.
- La máquina usa el estado actual y el simbolo leído de la cinta para decidir qué hacer de acuerdo con unas reglas.
- Cada regla tiene la siguiente forma: "Si el estado actual es q_0 y el símbolo leído por la cabeza lectora es s_0, entonces escribir el símbolo s_1 en la cinta y mover la cabeza lectora a la izquierda/derecha y pasar al estado q_1".
- Según las reglas fijadas, se puede conseguir que la máquina de Turing calcule distintas funciones y se puede computar cualquier función computable en un lenguaje de programación.
- Las máquinas de Turing reconocen determinados lenguajes (conjuntos de cadenas).
- Todos los problemas con respuesta sí/no se pueden ver como problemas de determinar si una cadena pertenece a un lenguaje.

La definición original de $\mathcal{P}$ y $\mathcal{NP}$ es:

- **Clase de complejidad** $\mathcal{P}$: Conjunto de lenguajes (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** $\mathcal{NP}$: Conjunto de lenguajes (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 puede tomar varios caminos a la vez ante una situación dada. 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.

9.3. Problemas $\mathcal{NP}$-completos

Los problemas $\mathcal{NP}$-completos son un subconjunto de problemas de $\mathcal{NP}$ que tienen las siguientes propiedades:

- Si cualquiera de ellos está en $\mathcal{P}$, entonces todos los problemas de $\mathcal{NP}$ están en $\mathcal{P}$. Bastaría con encontrar algún algoritmo polinómico para *uno solo* de ellos para demostrar que $\mathcal{P} = \mathcal{NP}$.
- Si $\mathcal{P} \neq \mathcal{NP}$ entonces ninguno de estos problemas puede estar en $\mathcal{P}$.

9.3.1. 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.

Teorema: Un problema es $\mathcal{NP}$-completo si está en $\mathcal{NP}$ y cualquier problema en $\mathcal{NP}$ es reducible a él. Por eso, si un problema $\mathcal{NP}$-completo estuviera en $\mathcal{P}$, todos los problemas de $\mathcal{NP}$ estarían en $\mathcal{P}$.

Ejemplos: suma de subconjuntos, el problema del viajante, la satisfacibilidad booleana, resolver un sudoku, etc.

Tema 10: Algoritmos heurísticos y aproximados

10.1. Algoritmos heurísticos

Los algoritmos heurísticos son métodos eficeitnes para encontrar buenas soluciones a los problemas $\mathcal{NP}$. Se utilizan principalemnte cuando no se conoce un método exacto para la resolución del problema o ese método se conoce pero es inviable computacionalmente.

- Suelen estar basados en una regla de sentido común.
- Normalmente obtienen una solución próxima a la óptima (incluso pueden llegar a obtener la óptima en algunos casos).
- Puede haber casos especiales para los que la solución encontrada sea muy mala o incluso no encontrar solución.

Propiedades de un algoritmo heurístico

- Ser eficiente. El esfuerzo computacional debe ser realista y adecuado para obtener la solución.
- Ser bueno. La solución debe estar, en promedio, cerca del óptimo.
- Ser robusto. La probabilidad de obtener una mala solución (lejos del óptimo) debe ser baja.

10.1.1. Coloreado de un grafo

El coloreado de un grafo de la forma $G = (N, A)$ es un problema $\mathcal{NP}$-completo clásico. No existe ningún algoritmo de tiempo polinómico que lo resuelva de forma óptima.

Objetivo: Colorear todos los nodos de forma queno haya dos nodos adyacentes con el mismo color y utilizando el mínimo número de colores.

Algoritmo heurístico voraz sencillo:

1. Se toma un color no usado y un nodo no coloreado.
2. Se recorren los nodos sin colorear, pintando del color actual todos los que sea posible.
3. Se repite el proceso hasta colorear todos los nodos.

Coloreado de un grafo

```
function ColourGraphGreedy(G=(N,A): graph):
    NoColoreados = { node list };
    while NoColoreados <> { }:
        nodo = First(NoColoreados);
        color = color no utilizado;
        Colorea(nodo);
        for nnodo in NoColoreados:
            if nnodo no tiene vecinos de ese color:
                Colorea(nnodo)
```

Una heurística puede encontrar una buena solución no muy distinta de la óptima. Sin embargo, hay grafos que hacen mala esta heurística. Son grafos que se pueden colorear con k colores donde la heurística encontrará la solución empleando c colores (siendo c/k tan grande como se quiera). Por ejemplo:

- Sea un grafo con $2n$ nodos numerados.
- Cuando i es impar, el nodo i es adyacente a todos los nodos pares excepto $i+1$.
- Cuando i es par, el nodo i es adyacente a todos los nodos impares excepto $i-1$.

El algoritmo `ColourGraphGreedy` no es un algoritmo aproximado, pues no podemos establecer una cota del error en la solución proporcionada, ya que crece con el número de nodos del grafo. Para un grafo bipartito de n nodos la solución óptima es 2 colores, pero el algoritmo puede llegar a utilizar $n/2$ colores.

Análisis de la complejidad En el peor caso tenemos un grafo competo donde todos los nodos están conectados entre sí:

Peor caso: $O(n^3)$

- El bucle externo se realiza n veces y en cada iteración sólo se colorea un nodo: $O(n)$.
- El bucle interno se realiza para todos los nodos no coloreados: $O(n)$.
- Dentro del bucle interno, para cada nodo igual se tiene que comprobar sus $n-1$ vecinos: $O(n)$.

10.1.2. El problema del viajante

Objetivo: Encontrar el itinerario más corto que permita a un viajante recorrer un conjunto de ciudades pasando una única vez por cada una y regresando a la ciudad de partida.

- Problema $\mathcal{NP}$-completo clásico.
- Los algoritmos óptimos conocidos son exponenciales.
- El problema se suele representar mediante un grafo no dirigido o una matriz de distancias.
- Solución: Ciclo Hamiltoniano más corto del grafo.

Solución heurística sencilla:

1. Ir desplazándose desde cada ciudad a la ciudad más próxima no visitada aún.
2. Finalmente regresar a la ciudad desde la que se comenzó el viaje.

El problema del viajante

```
function SalesmanGreedy(G=(N,A): graph):
    Cities = { node list };
    city = First(Cities);
    travel = { city };
    while Cities <> { }:
        for c in Cities:
            otherCity = Nearest from city;
        Remove(Cities, otherCity);
        Add(travel, otherCity);
        city = otherCity;
    Add(travel, First(travel));
```

Bondad de la solución obtenida:

- Se pueden encontrar casos para los que la solución de la heurística sea muy mala.
- El algoritmo no es un algoritmo aproximado ya que no es posible acotar el error máximo.

Complejidad: $O(n^2)$

- El bucle exterior se realiza una vez por cada ciudad: $O(n)$.
- Encontrar la ciudad más próxima es recorrer todas las ciudades que faltan por visitar: $O(n)$.

10.1.3. Clasificación de los algoritmos heurísticos

- **Algoritmos de descomposición**.
  - Descompone el problema original en subproblemas más sencillos de resolver.
  - Programación lineal (principio de descomposición).
- **Algoritmos inductivos**. Generalizan las propiedades o técnicas más fáciles de analizar, que se pueden aplicar al problema completo.
- **Algoritmos de reducción**.
  - Introducir como restricciones del problema propiedades que cumplen las buenas soluciones.
  - Objetivo: Restringir el espacio de soluciones del problema.
  - Riesgo: Dejar fuera soluciones óptimas del problema original.
- **Algoritmos constructivos**.
  - Construir paso a paso solución del problema.
  - Métodos deterministas: Mejor elección en cada iteración.

- **Algoritmos de búsqueda local**.
  - Comienzan con una solución del problema y la mejoran progresivamente.
  - Fin: Para una solución no existe ninguna solución accesible que la mejore.
  - Método de ramificación y poda.

10.1.4. Calidad de la heurística

- Comparación con la solución óptima.
  - Se dispone de un método para obtener la solución óptima para un número reducido de ejemplos.
  - Se calcula el promedio de las desviaciones de la solución heurñistica frente a la óptima de cada ejemplo.
- Comparación con una cota del problema.
  - La bondad de esta medida depende de la bondad de la cota que debería ser conocida.
- Comparación con un método exacto truncado.
  - Se establece un límite de iteraciones o de tiempo de ejecución.
  - Se satura un nodo en un problema de maximización cuando su cota inferior sea menor o igual que la cota superior global más un cierto valor α.
- Comparación con otros algoritmos heurísticos.
  - La bondad de esta medida depende de la bondad de los heurísticos con los que se compara.
- Análisis del peor caso.
  - Se consideran los ejemplos más desfavorables.
  - Se acota la máxima desviación respecto del óptimo del problema para cualquier ejemplo.
  - Los resultados no son representativos del caso medio.
  - El análisis puede ser complicado para heurísticos sotisficados.

10.2. Algoritmos aproximados

Son algoritmos utilizados para encontrar soluciones aproximadas a problemas de optimización.

- A menudo se asocian con problemas $\mathcal{NP}$-completos.
- A diferencia de los algoritmos heurísticos...
  - Los algoritmos heurísticos encuentran soluciones razonablemente buenas en tiempos razonablemente rápidos.
  - Los algoritmos aproximados encuentran soluciones de calidad con tiempos de ejecución acotados.
- Se utilizan donde los algoritmos exactos son costosos debido al tamaño de la entrada.

10.2.1. El problema del viajante

Existe un algoritmo aproximado para este problema y se aplica cuando:

1. El grafo es completo.
2. La matriz de distancias verifica que para cualquier trío de nodos A, B y C:

$$\text{distancia}(A, C) \leq \text{distancia}(B, C) + \text{distancia}(C, A)$$

Si el grafo cumple las condiciones, sabemos que:

- Eliminar una arista de un ciclo de longitud c da lugar a un camino de longitud c' donde $c' < c$.
- Todo camino es un árbol expandido de longitud menor que la del ciclo.
- La longitud del camino (c') será mayor que el árbol expandido mínimo (r). Es decir, $r \leq c'$.
- Por tanto cualquier ciclo es más largo que el árbol expandido mínimo ($r < c$)
- Si recorremos todos los nodos utilizando las aristas del AEM, se pasa dos veces por cada arista y la distancia recorrida es $2r$.
- Si tomamos "atajos", por la propiedad métrica del grafo, la distancia d recorrida será menor o igual que $2r$.

$$d \leq 2r$$

- El itinerario verificará que $d \leq 2r < 2c$

El algoritmo aproximado considera:

- Encontrar el árbol expandido mínimo del grafo con el Algoritmo de Prime ($O(n^2)$).
- Ordenación de los nodos en preorden ($O(n)$).
- Eficiencia: $O(n^2)$.

El itinerario obtenido nunca será más de dos veces más largo que el que habríamos obtenido utilizando el algoritmo óptimo:

$$d_{\text{aprox}} \leq 2d_{\text{óptimo}}$$

10.2.2. El problema de la mochila

Problema: Cargar una mochila de capacidad W (unidades de carga) maximizando el valor de su carga.

El algoritmo voraz puede ser útil si se garantiza que su error relativo está bajo control. Sin embargo, puede ser arbitrariamente malo. Sea un entero $x > 2$ y dos objetos donde $w_1 = 2$, $v_1 = 2$, $w_2 = x$, $v_2 = x$ y $W = x$, los algoritmos seleccionan el objeto 1 con rendimiento 2 y ninguno más, cuando la solución óptima es seleccionar el objeto 2 con rendimiento x.

Suponiendo que:

1. No tenemos objetos cuyos pesos superen W.

2. La mochila no puede contener todos los objetos a la vez.
3. Tenemos los objetos ordenados por peso de mayor a menor.
4. La solución óptima es opt y la aproximada es $\tilde{\text{opt}}$ y se obtiene con:

Aproximación del problema de la mochila

```
function approxKnap(w[1..n],v[1..n],W):
    biggest = max{v[i]| 1 ≤ i ≤ n}
    return max(biggest,knapsack1(w,v,W))
```

Sea ℓ el menor entero tal que $\sum_{i=1}^{\ell} w_i > W$. Suponiendo que $W' = \sum_{i=1}^{\ell} w_i$, la solución óptima a un problema donde la capacidad de la mochila fuese W' sería:

$$\text{opt}' = \sum_{i=1}^{\ell} v_i, \implies \text{opt} \leq \text{opt}'$$

Si desarrollamos:

$$\tilde{\text{opt}} = \max(\texttt{biggest}, \texttt{knapsack1}) \geq \frac{\texttt{biggest} + \texttt{knapsack1}}{2}$$
$$\geq \frac{v_\ell + \sum_{i=1}^{\ell-1} v_i}{2} = \frac{\sum_{i=1}^{\ell} v_i}{2} = \frac{\text{opt}'}{2}$$

Queda por tanto demostrado que la solución está siemrpe dentro de un factor de 2 de la solución óptima.

10.2.3. Llenado de cajas

- Disponemos de n objetos numerados de 0 a $n-1$ ordeandos de menor a mayor peso.
- Disponemos de C cajas idénticas, cada una capaz de soportar un peso P

Objetivo: Meter objetos en C cajas de capacidad P maximizando el número de objetos.

- Existe una heurística voraz óptima con una caja: Meter los objetos empezando por los de menor peso.
- Con 2 cajas, la solución heurística fallará como máximo en un objeto.
  - Suponemos que tenemos una caja que soporta el peso $2P$ y se va llenando con los objetos de menor a mayor peso y la llenamos hasta con t objetos.
  - Si partimos la caja por la mitad, puede darse la casualidad de que el objeto j quede "dividido" y no se pueda introducir en ninguna de las mitades de la caja.
  - Por tanto esta solución para dos cajas de peso P sólo fallará en un objeto.
- Para un número de cajas C, la solución aproximada tendrá, a lo sumo, $C-1$ objetos menos que la solución óptima.

Complejidad del algoritmo

- Ordenación de los objetos por peso decreciente: $O(n \log n)$.
- Algoritmo voraz que va tomando objetos uno a uno y guardándolos en cajas: $O(n)$.

Complejidad: $O(n \log n)$

10.2.4. Llenado de cajas (algoritmo 2)

Objetivo: Meter objetos en C cajas de capacidad P minimizando el número de cajas.

Suponemos que metemos los objetos empezando por los de menor peso y contamos las cajas que vamos llenando. Si b es la solción óptima y s es la solución aproximada, se cumple que s/b está acotado. Es imposible obtener una cota del error en términos de una constante aditiva.

Supongamos que tenemos $n = 2k$ objetos y C cajas de capacidad $2P$, donde k es par y $P > 3(k-1)$. Donde los pesos de los objetos son: $P-k, P-(k-1), ..., P-1, P+1, ..., P+(k-1), P+k$, la **solución óptima** viene dada por meter el objeto más pesado y más ligero en una caja.

Bloque V
Apéndice

A: Complejidades de los algoritmos

A.1. Análisis de Algoritmos

Exponenciación recursiva

$$x^n = \begin{cases} x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} & \text{si } n \text{ par} \\ x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} \cdot x & \text{si } n \text{ impar} \end{cases}$$

$$f(n) = \begin{cases} f(2^k) = f(2^{k-1}) + 1 = k & n \text{ par, } k > 0 \\ f(2^k - 1) = f(2^{k-1} - 1) + 2 = 2(k-1) & n \text{ impar, } k > 1 \end{cases}$$

- Mejor caso (n par): $T(n) = \Omega(\log n)$
- Peor caso (n impar): $T(n) = O(\log n)$
- Caso medio: $T(n) = \Theta(\log n)$

A.2. Divide y Vencerás

$T(n) = \ell T(n/b) + cn^k, \quad n \geq n_0$

$$T(n) = \begin{cases} O(n^k) & \text{si } \ell < b^k \\ O(n^k \log n) & \text{si } \ell = b^k \\ O(n^{\log_b \ell}) & \text{si } \ell > b^k \end{cases}$$

	Tiempo de ejecución	Complejidad
Problema del Skyline	$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$
Problema de los puntos más cercanos	$T(n) \leq 2T(n/2) + O(n \log n)$	$T(n) = O(n \log n)$
Multiplicación de Karatsuba-Ofman	$T(n) = 3T(n/2) + O(n)$	$T(n) = O(n^{\log_2 3})$

A.3. Algoritmo Voraces

- El problema de la mochila: $T(n) = O(n \log n)$

- Ordenación topológica: $T(n) = O(n+m)$ (listas de adyacencia)
  - Peor caso: $m \to (n-1)n$
  - Mejor caso: $m \to n$.
- Árbol expandido mínimo:
  - Algoritmo Kruskal: $T(n) = \Theta(m \log n)$
  - Algoritmo Prim: $T(n) = \Theta(n^2)$
- Caminos mínimos.
  - Algoritmo Dijkstra: $T(n) = \Theta(n^2)$.
  - Dijkstra montículos: $T(n) = O((m+n)\log n)$

Algoritmos	Complejidad	Grafo denso $m \to n(n-1)/2$	Grafo disperso $m \to n$
Kruskal	$\Theta(m \log n)$	$\Theta(n^2 \log n)$	$\Theta(n \log n)$
Prim	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$

A.4. Programación Dinámica

- Sucesión de Fibonacci:
  - Algoritmo recursivo: $T(n) = \Theta(\Phi^n)$, $\Phi = \frac{1+\sqrt{5}}{2}$.
  - Programación Dinámica: $T(n) = \Theta(n)$.
  - Más eficiente: $T(n) = O(\log n)$.
- Coeficientes binomiales:
  - Divide y Vencerás: $T(n) = \Omega\left(\binom{n}{k}\right)$
  - Programación Dinámica: $T(n) = O(nk)$
  - Mejora con PD: $T(n) = \Theta(k)$
- Devolver el cambio: $T(n) = \Theta(mn) + \underbrace{O(m + c[m,n])}_{\text{Determinar solución}}$
- Problema de la mochila: $T(n) = \Theta(nW) + \underbrace{O(n+W)}_{\text{Determinar solución}}$
- Análisis sintáctico con algoritmo CYK: $\underbrace{O(rn^3)}_{\text{Complejidad}}$ y $\underbrace{O(sn^2)}_{\text{Espacio ocupado}}$

A.5. Búsqueda en memoria principal y secundaria

Búsqueda lineal

- Algoritmo sin centinela: $T(n) = \Theta(n)$
  - Mejor caso: $T(n) = 1$.
  - Peor caso: $T(n) = n$.
  - Caso medio: $T(n) = n/2$.
- Algoritmo con centinela: $T(n) = \Theta(n)$
  - Mejor caso: $T(n) = 1$.
  - Peor caso: $T(n) = n$.
  - Caso medio: $T(n) = n/2$.

Búsqueda binaria

- Mejor caso: $T(n) = 1$
- Peor caso: $T(n) = O(\log n)$
- Teorema Divide y Vencerás: $T(n) = T(n/2) + 1 \implies T(n) = O(\log n)$.

Árbol de búsqueda

- Árbol aleatorio: $T(n) = O(\log n)$
- Árbol degenerado o sesgado: $T(n) = O(n)$

Tablas hashing (resolución de colisiones)

Dispersión abierta

$$\lambda = \frac{\text{Nº de claves en la tabla}}{\texttt{TableSize}} = \begin{cases} \text{“Factor de carga”} \ (\lambda \approx 1) \\ \text{“Tamaño medio de una lista”} \end{cases}$$

- Búsqueda: $\underbrace{O(1)}_{\text{Hash(X)}} + \underbrace{O(\lambda)}_{\text{Recorrer lista}}$
- Búsqueda sin éxito = promedio de λ enlaces recorridos: $O(\lambda)$
- Búsqueda con éxito = promedio de $1 + \lambda/2$ enlaces recorridos: $O(\lambda/2)$

Dispersión cerrada

Sucesión de celdas: $h_i(x) = [\text{Hash}(X) + F(i)] \ \texttt{mod TableSize}$

F: Estrategia de resolución de colisiones.

1. Exploración lineal: $F(i) = i$.

- Nº de pruebas teóricas (`TableSize` grande e independencia): $1/(1+\lambda)$
- Nº pruebas esperado: $\dfrac{1+1/(1+\lambda)^2}{2}$
- Inserciones y búsquedas sin éxito: mismo número de pruebas.
- Búsqueda con éxito: $\dfrac{1+1/(1+\lambda)}{2}$

2. Exploración cuadrática: $F(i) = i^2$.
3. Exploración doble: $F(i) = i * \text{hash}_2(X), \ \ i \neq 0$.
   - Función correcta: $\text{hash}_2(X) = R - (X \ \texttt{mod} \ R)$
   - R primo menor que `TableSize`.
   - `TableSize` primo.

A.6. Ordenación Interna y Externa

	Mejor caso	Caso medio	Peor caso
`InsertionSort`	$O(n)$	$O(n^2)$	$O(n^2)$
`ShellSort` (incrementos Shell)			$\Theta(n^2)$
`ShellSort` (incrementos Hibbard)		$O(n^{5/4})$ (sim)	$O(n^{3/2})$
`ShellSort` (incrementos Sedgewick)		$O(n^{7/6})$ (sim)	$O(n^{4/3})$ (sim)
`HeapSort`	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
`MergeSort`	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
`QuickSort`	$O(n \log n)$	$O(n \log n)$	$O(n^2)$