Universidad
Rey Juan Carlos

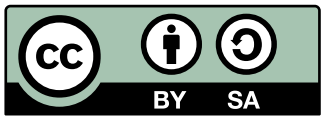
Exámenes de años anteriores y ejercicios adicionales

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

13 de mayo de 2024

Autor: Manuel Rubio Sánchez



Índice de contenidos

Exámenes con soluciones

Exámenes sin soluciones

Ejercicios adicionales

Exámenes con soluciones

Universidad
Rey Juan Carlos

Prueba de evaluación

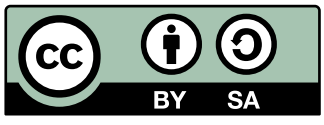
Soluciones examen análisis – mayo 2014-2015

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2014-2015 – 10 de abril de 2015

Prueba de Análisis de Algoritmos

Valor: 30 % de la nota final. Duración: **1 hora y 45 minutos**

Ejercicio 1 [3 puntos]

Demostrar **mediante la definición** de complejidad asintótica Ω, sin usar límites, si se verifica:

$$n^2 - 3n + 1 \in \Omega(n^2)$$

Solución:

La definición de Ω es:

$$\Omega(g(n)) = \Big\{ f(n) : \exists\ c > 0 \text{ y } n_0 > 0 \ /\ 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0 \Big\}$$

Por tanto, tenemos que ver si es posible encontrar una pareja de constantes $c > 0$ y $n_0 > 0$ de forma que se cumpla la definición. Empezamos analizando:

$$n^2 - 3n + 1 \geq cn^2$$

Escogemos un valor de c lo suficientemente pequeño, pero positivo. En este caso $c = 1/2$ resulta ser suficiente. Por tanto, tenemos:

$$n^2 - 3n + 1 \geq \frac{1}{2}n^2$$

$$\frac{1}{2}n^2 - 3n + 1 \geq 0$$

$$n^2 - 6n + 2 \geq 0$$

La función de la izquierda es una parábola convexa, así que ya podemos ver que va a ser posible encontrar un valor para n_0. Para ello, analizamos la mayor raíz del polinomio, que es:

$$\frac{6 + \sqrt{36 - 8}}{2} = \frac{6 + \sqrt{28}}{2} = 3 + \sqrt{7}$$

Por tanto, podemos escoger para n_0 cualquier valor superior o igual a $3 + \sqrt{7}$. Por ejemplo, $n_0 = 6$, ya que $\sqrt{7} < 3$. De esta manera hemos encontrado las dos constantes c y n_0 que satisfacen la definición, y podemos afirmar que $n^2 - 3n + 1 \in \Omega(n^2)$.

Ejercicio 2 [4 puntos]

Determinar el orden de complejidad (no es necesario hallar las constantes) de la siguiente función recursiva:

$$T(n) = 4T(n-1) - 5T(n-2) + 2T(n-3) + 1 + n^2 + n2^n$$

Solución:

Aplicamos el método general de resolución de recurrencias. Primero pasamos los términos con T del lado derecho al izquierdo, y expresamos el resto de términos de la derecha como polinomios por potencias elevadas a n:

$$T(n) - 4T(n-1) + 5T(n-2) - 2T(n-3) = (1+n^2)1^n + n2^n$$

El polinomio característico es:

$$(x^3 - 4x^2 + 5x - 2)(x-1)^3(x-2)^2$$

Factorizamos el primer término aplicando el método de Ruffini y agrupamos términos:

$$(x-1)^2(x-2)(x-1)^3(x-2)^2 = (x-1)^5(x-2)^3$$

Por tanto, la función $T(n)$ tiene la siguiente expresión:

$$T(n) = C_1 + C_2 n + C_3 n^2 + C_4 n^3 + C_5 n^4 + C_6 2^n + C_7 n2^n + C_8 n^2 2^n$$

Finalmente, el orden de complejidad es:

$$T(n) \in \theta(n^2 2^n)$$

Ejercicio 3 [6 puntos]

La fórmula para considerar todas las operaciones que se llevan a cabo en un bucle de tipo FOR o WHILE es:

$$T_{\text{bucle}} = 1_{\text{inicialización}} + \sum^{n}(1_{\text{comparación}} + T_{\text{cuerpo}} + 1_{\text{incremento}}) + 1_{\text{última comparación}}$$

Utilízala para hallar el número de operaciones ($T(n)$, simplificada) del siguiente código:

```
1  for (int i=0; i<n; i++)
2      for(j=i;j<=n;j++){
3          int k=0;
4          while (k<j){
5              procesa(i,j,k);   // una operación
6              k++;
7          }
8      }
```

Se considera que las inicializaciones, comparaciones, e incrementos siempre necesitan una sola operación.

Solución:

El código consta de 3 bucles, que podemos descomponer de la siguiente manera:

$$T(n) = 1 + \sum_{i=0}^{n-1}(1 + T_{\text{For 1}} + 1) + 1$$

$$T_{\text{For 1}} = 1 + \sum_{j=i}^{n}(1 + T_{\text{For 2}} + 1) + 1$$

$$T_{\text{For 2}} = 1 + \sum_{k=0}^{j-1}(1 + 1 + 1) + 1 = 2 + 3j$$

Sustituyendo en $T_{\text{For 1}}$ tenemos:

$$T_{\text{For 1}} = 2 + \sum_{j=i}^{n}(4 + 3j)$$

$$= 2 + \sum_{j=i}^{n} 4 + 3\sum_{j=i}^{n} j = 2 + \sum_{j=i}^{n} 4 + 3\left[\sum_{j=1}^{n} j - \sum_{j=1}^{i-1} j\right]$$

$$= 2 + 4(n - i + 1) + 3\left[\frac{n(n+1)}{2} - \frac{(i-1)i}{2}\right]$$

Simplificando obtenemos:

$$T_{\text{For 1}} = \frac{3n^2 + 11n + 12 - 3i^2 - 5i}{2}$$

Sustituyendo en la expresión para el primer bucle tenemos:

$$T(n) = 2 + \sum_{i=0}^{n-1} \left(2 + \frac{3n^2 + 11n + 12 - 3i^2 - 5i}{2}\right)$$

$$T(n) = 2 + \frac{1}{2} \sum_{i=0}^{n-1} \left(3n^2 + 11n + 16 - 3i^2 - 5i\right)$$

$$T(n) = 2 + \frac{3}{2}n^3 + \frac{11}{2}n^2 + 8n - \frac{3}{2} \sum_{i=0}^{n-1} i^2 - \frac{5}{2} \sum_{i=0}^{n-1} i$$

Para terminar el problema hay que resolver los dos últimos sumatorios:

$$T(n) = 2 + \frac{3}{2}n^3 + \frac{11}{2}n^2 + 8n - \frac{3}{2}\frac{(2(n-1)+1)(n-1)n}{6} - \frac{5}{2}\frac{(n-1)n}{2}$$

$$T(n) = 2 + \frac{3}{2}n^3 + \frac{11}{2}n^2 + 8n - \frac{2n^3 - 3n^2 + n}{4} - \frac{5n^2 - 5n}{4}$$

Finalmente:

$$T(n) = \frac{4n^3 + 20n^2 + 36n + 8}{4} = n^3 + 5n^2 + 9n + 2 \in \theta(n^3)$$

Ejercicio 4 [6 puntos]

Halla una expresión no recursiva para la siguiente recurrencia por el método de **expansión de recurrencias**:

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

donde n es una potencia de 2 ($n = 2^k$, para $k = 1, 2, \ldots$), y donde $T(1) = 1$. Indica además el orden de $T(n)$.

Solución:

Procedemos a expandir la recurrencia:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\left[2T(n/4) + \frac{n}{2}\right] + n = 4T(n/4) + 2n \\ &= 4\left[2T(n/8) + \frac{n}{4}\right] + 2n = 8T(n/8) + 3n \\ &= 8\left[2T(n/16) + \frac{n}{8}\right] + 3n = 16T(n/16) + 4n \\ &\vdots \\ &= 2^i T(n/2^i) + in \end{aligned}$$

Se llega al caso base cuando $n/2^i = 1$. Es decir, cuando $n = 2^i$, e $i = \log_2 n$. Sustituyendo:

$$T(n) = nT(1) + n\log_2 n = n + n\log_2 n \in \theta(n \log n)$$

Ejercicio 5 [6 puntos]

Halla una expresión no recursiva para la siguiente recurrencia por el método **general de resolución recurrencias**:

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

donde n es una potencia de 2 ($n = 2^k$, para $k = 1, 2, \ldots$), y donde $T(1) = 1$. Indica además el orden de $T(n)$.

Solución:

Para poder aplicar el método primero tenemos que hacer el cambio de variable $n = 2^k$:

$$T(n) = T(2^k) = 2T(2^k/2) + 2^k = 2T(2^{k-1}) + 2^k$$

A continuación hacemos un cambio de función $t(k) = T(2^k)$:

$$T(n) = T(2^k) = t(k) = 2t(k-1) + 2^k$$

El polinomio característico de la recurrencia es:

$$(x-2)(x-2) = (x-2)^2$$

Por tanto, la recurrencia tiene la siguiente forma:

$$t(k) = C_1 2^k + C_2 k 2^k$$

Para hallar las constantes necesitamos un segundo caso base. Procedemos a calcular $T(2) = 2T(1) + 2 = 4$. Los correspondientes casos base para $t(k)$ son:

$$T(1) = T(2^0) = t(0) = 1$$
$$T(2) = T(2^1) = t(1) = 4$$

Por tanto, el sistema de ecuaciones a resolver es:

$$\left.\begin{array}{rcrcl} t(0) & = & C_1 & = & 1 \\ t(1) & = & C_1 2^1 + C_2 2^1 & = & 4 \end{array}\right\}$$

Las soluciones son $C_1 = 1$ y $C_2 = 1$. Por tanto, $t(k) = 2^k + k2^k$. Deshaciendo el cambio de variable, donde $k = \log_2 n$, obtenemos:

$$t(k) = T(2^k) = T(n) = n + n\log_2 n \in \theta(n \log n)$$

Ejercicio 6 [5 puntos]

Halla una expresión no recursiva para la siguiente recurrencia por el método de **expansión de recurrencias**:

$$T(n) = 1 + \sum_{i=0}^{n-1} T(i)$$

donde $T(0) = 1$.

Solución:
En primer lugar podemos reescribir la fórmula de la siguiente manera:

$$T(n) = 1 + T(n-1) + \sum_{i=0}^{n-2} T(i) \tag{1}$$

A partir de ahora se puede resolver de varias formas.

- Primera forma (difícil). Expandimos $T(n-1)$, quedando:

$$T(n) = 1 + 1 + \sum_{i=0}^{n-2} T(i) + \sum_{i=0}^{n-2} T(i) = 2 + 2\left[\sum_{i=0}^{n-2} T(i)\right]$$

Ahora extraemos $T(n-2)$ del sumatorio, y lo expandimos:

$$T(n) = 2 + 2\left[T(n-2) + \sum_{i=0}^{n-3} T(i)\right] = 2 + 2\left[1 + \sum_{i=0}^{n-3} T(i) + \sum_{i=0}^{n-3} T(i)\right]$$

$$= 2 + 2\left[1 + 2\sum_{i=0}^{n-3} T(i)\right] = 4 + 4\left[\sum_{i=0}^{n-3} T(i)\right]$$

Hacemos lo mismo un paso más:

$$T(n) = 4+4\left[T(n-3) + \sum_{i=0}^{n-4} T(i)\right] = 4+4\left[1 + 2\sum_{i=0}^{n-4} T(i)\right] = 8+8\left[\sum_{i=0}^{n-4} T(i)\right]$$

Tras una serie de pasos (en este caso usamos la variable j ya que la i ya aparece en el sumatorio) tenemos:

$$T(n) = 2^{j-1} + 2^{j-1}\left[\sum_{i=0}^{n-j} T(i)\right]$$

Alcanzamos el caso base cuando el sumatorio solo suma un término $T(0)$. Esto sucede cuando $n = j$. Sustituyendo:

$$T(n) = 2^{n-1} + 2^{n-1}T(0) = 2 \cdot 2^{n-1} = 2^n \in \theta(2^n)$$

- Segunda forma (fácil). Por la definición de T tenemos:

$$1 + \sum_{i=0}^{n-2} T(i) = T(n-1)$$

Por tanto, sustituyendo en (1) tenemos una nueva expresión simplificada de la recurrencia:

$$T(n) = 2T(n-1)$$

Ahora procederíamos a resolverla:

$$T(n) = 2T(n-1) = 4T(n-2) = 8T(n-3) = \cdots = 2^i T(n-i)$$

El caso base se alcanza cuando $n-i = 0$, es decir, cuando $n = i$. Sustituyendo:

$$T(n) = 2^n T(0) = 2^n \in \theta(2^n)$$

Universidad
Rey Juan Carlos

Prueba de evaluación

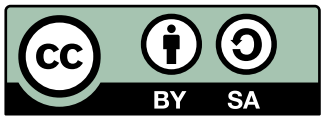
Soluciones examen análisis – mayo 2015-2016

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2015-2016 – 17 de marzo de 2016

Prueba de Análisis de Algoritmos

Valor: 30 % de la nota final. Duración: **1 hora y 15 minutos**

Soluciones

Soluciones

Ejercicio 1 [1.5 puntos]

Indicad la definición matemática de $\mathcal{O}$, y usadla para demostrar (sin usar límites), si se verifica:

$$5n \in \mathcal{O}(n \log(n))$$

Solución:

La definición de $\mathcal{O}$ es:

$$\mathcal{O}(g(n)) = \Big\{ f(n) : \exists\ c > 0 \text{ y } n_0 > 0 \ /\ 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0 \Big\}$$

Por tanto, tenemos que ver si es posible encontrar una pareja de constantes $c > 0$ y $n_0 > 0$ de forma que se cumpla la definición. Empezamos analizando:

$$5n \leq cn \log(n)$$

Escogemos un valor de c lo suficientemente grande. En este caso $c = 5$ resulta adecuado para continuar:

$$5n \leq 5n \log(n)$$

$$0 \leq 5n \log(n) - 5n$$

$$0 \leq 5n(\log(n) - 1)$$

En este caso, sea cual sea la base del logaritmo, por ejemplo b, siempre es posible encontrar un n_0 de tal manera que se cumpla la definición. Bastaría escoger $n_0 = b$, ya que $(\log_b(n) - 1) \geq 0$ para todo $n \geq b$. Por tanto, podemos afirmar que $5n \in \mathcal{O}(n \log(n))$.

Ejercicio 2 [2.5 puntos]

La fórmula para considerar todas las operaciones que se llevan a cabo en un bucle de tipo FOR o WHILE es:

$$T_{\text{bucle}} = 1_{\text{inicialización}} + \sum^{n}(1_{\text{comparación}} + T_{\text{cuerpo}} + 1_{\text{incremento}}) + 1_{\text{última comparación}}$$

Utilízala para hallar el número de operaciones ($T(n)$, simplificada) del siguiente código:

```
1  int i=0;
2  while (i<n){
3      int j=i;
4      while (j<=n){
5          procesa(i,j);  // una operación
6          j++;
7      }
8      i++;
9  }
```

Se considera que las inicializaciones, comparaciones, e incrementos siempre necesitan una sola operación.

Solución:

El código consta de 2 bucles, que podemos descomponer de la siguiente manera:

$$T(n) = 1 + \sum_{i=0}^{n-1}(1 + T_{\text{While interno}} + 1) + 1$$

$$T_{\text{While interno}} = 1 + \sum_{j=i}^{n}(1+1+1) + 1 = 2 + 3(n-i+1)$$

Sustituyendo en $T(n)$ tenemos:

$$T(n) = 2 + \sum_{i=0}^{n-1}\big(2 + 2 + 3(n-i+1)\big) = 2 + \sum_{i=0}^{n-1}(7 + 3n - 3i)$$

$$= 2 + 7n + 3n^2 - 3\sum_{i=0}^{n-1} i = 2 + 7n + 3n^2 - 3\frac{n(n-1)}{2}$$

Finalmente:

$$T(n) = 2 + 7n + 3n^2 - \frac{3}{2}n^2 + \frac{3}{2}n = \frac{3}{2}n^2 + \frac{17}{2}n + 2$$

Ejercicio 3 [3 puntos]

Halla una expresión no recursiva para la siguiente recurrencia por el método de **expansión de recurrencias**:

$$T(n) = 4T(n/2) + n$$

donde n es una potencia de 2 ($n = 2^k$, para $k = 1, 2, \ldots$), y donde $T(1) = 1$. Indica además el orden de $T(n)$.

Solución:

Procedemos a expandir la recurrencia:

$$\begin{aligned}
T(n) &= 4T(n/2) + n \\
&= 4\left[4T(n/2^2) + \frac{n}{2}\right] + n = 4^2T(n/2^2) + 2n + n \\
&= 4^2\left[4T(n/2^3) + \frac{n}{2^2}\right] + 2n + n = 4^3T(n/2^3) + 4n + 2n + n \\
&= 4^3\left[4T(n/2^4) + \frac{n}{2^3}\right] + 4n + 2n + n = 4^4T(n/2^4) + 8n + 4n + 2n + n \\
&\vdots \\
&= 4^iT(n/2^i) + n\sum_{j=0}^{i-1} 2^j = 4^iT(n/2^i) + (2^i - 1)n
\end{aligned}$$

Se llega al caso base cuando $n/2^i = 1$. Es decir, cuando $n = 2^i$. En ese caso, $n^2 = (2^i)^2 = (2^2)^i = 4^i$. Sustituyendo:

$$T(n) = n^2 + n^2 - n = 2n^2 - n$$

Ejercicio 4 [3 puntos]

Halla una expresión no recursiva para la siguiente recurrencia por el método **general de resolución recurrencias**:

$$T(n) = T(n/2) + \log_2 n$$

donde n es una potencia de 2 ($n = 2^k$, para $k = 1, 2, \ldots$), y donde $T(1) = 1$. Indica además el orden de $T(n)$.

Solución:

Para poder aplicar el método primero tenemos que hacer el cambio de variable $n = 2^k$ (donde $k = \log_2 n$):

$$T(n) = T(2^k) = T(2^k/2) + \log_2 2^k = T(2^{k-1}) + k$$

A continuación hacemos un cambio de función $t(k) = T(2^k)$:

$$T(n) = T(2^k) = t(k) = t(k-1) + k$$

El polinomio característico de la recurrencia es:

$$(x-1)(x-1)^2 = (x-1)^3$$

Por tanto, la recurrencia tiene la siguiente forma:

$$t(k) = C_1 1^k + C_2 k 1^k + C_3 k^2 1^k = C_1 + C_2 k + C_3 k^2$$

Para hallar las constantes necesitamos dos casos base adicionales. Procedemos a calcular $T(2) = T(1) + \log_2 2 = 2$, y $T(4) = T(2) + \log_2 4 = 4$. Los correspondientes casos base para $t(k)$ son:

$$\begin{array}{rcl} T(1) = T(2^0) = t(0) & = & 1 \\ T(2) = T(2^1) = t(1) & = & 2 \\ T(4) = T(2^2) = t(2) & = & 4 \end{array}$$

Por tanto, el sistema de ecuaciones a resolver es:

$$\left.\begin{array}{rcrcl} t(0) & = & C_1 & = & 1 \\ t(1) & = & C_1 + C_2 + C_3 & = & 2 \\ t(2) & = & C_1 + 2C_2 + 4C_3 & = & 4 \end{array}\right\}$$

Las soluciones son $C_1 = 1$ y $C_2 = C_3 = 1/2$. Por tanto, $t(k) = 1 + k/2 + k^2/2$. Deshaciendo el cambio de variable, donde $k = \log_2 n$, obtenemos:

$$t(k) = T(2^k) = T(n) = 1 + \frac{1}{2}\log_2 n + \frac{1}{2}(\log_2 n)^2 \in \theta\big((\log n)^2\big)$$

Universidad
Rey Juan Carlos

Prueba de evaluación

Soluciones examen diseño – mayo 2016-2017

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez

Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2016-2017

Prueba de Diseño de Algoritmos

Valor: 35 % de la nota final. Duración: **1 hora y 45 minutos**

Soluciones

Ejercicio 1 [1 punto]

Dado un laberinto, se desea encontrar un camino que no solo sea una solución, sino que además sea de longitud más corta. Este problema se puede resolver con la técnica de *backtracking*, pero en este ejercicio se pide una solución mediante un algoritmo **voraz**. La solución consiste en aplicar una estrategia de "transforma y vencerás", para poder aplicar uno de los algoritmos voraces vistos en clase.

Se pide describir cómo se puede resolver el problema concreto (cómo se transformaría el problema, y qué algoritmo voraz se aplicaría). No hay que escribir código. La descripción será muy breve: 5 líneas como máximo.

Solución:

Primero se genera un grafo asociado al laberinto. En concreto, tendrá un nodo por cada celda vacía en el laberinto. En cuanto a las aristas, habrá una entre dos nodos si sus respectivas celdas son adyacentes. Finalmente, el camino más corto a través del laberinto es el camino más corto a través del grafo. Por tanto, el problema se puede resuelver aplicando el algoritmo de **Dijkstra**.

Ejercicio 2 [2 puntos]

Considera un número entero no negativo a cuyos dígitos aparecen ordenados orden creciente desde el más significativo hasta el menos (por ejemplo, $a = 245778$). Dado un dígito x, implementa una función que devuelva un nuevo número resultante de insertar x en a, de manera que los dígitos también queden ordenados en orden creciente. Ejemplos:

- $a = 245778$, $x = 0 \longrightarrow 245778$
- $a = 245778$, $x = 1 \longrightarrow 1245778$
- $a = 245778$, $x = 6 \longrightarrow 2456778$
- $a = 245778$, $x = 9 \longrightarrow 2457789$

Posible solución:

```
def inserta_en_entero_creciente(x,a):
    if x>=a%10:  # caso base
        return a*10 + x
    else:  # caso recursivo
        return a%10 + 10*inserta_en_entero_creciente(x,a//10)
```

Ejercicio 3 [3 puntos]

Sea **a** una lista ordenada de n enteros (pueden ser negativos) todos distintos. Se pide diseñar un algoritmo de complejidad $\mathcal{O}(\log n)$ en el peor caso, capaz de encontrar un índice i tal que $a[i] = i$, suponiendo que tal índice i exista (donde $0 \leq i \leq n-1$). El algoritmo devolverá el valor i en caso de que exista, y -1 en caso contrario.

Posible solución:

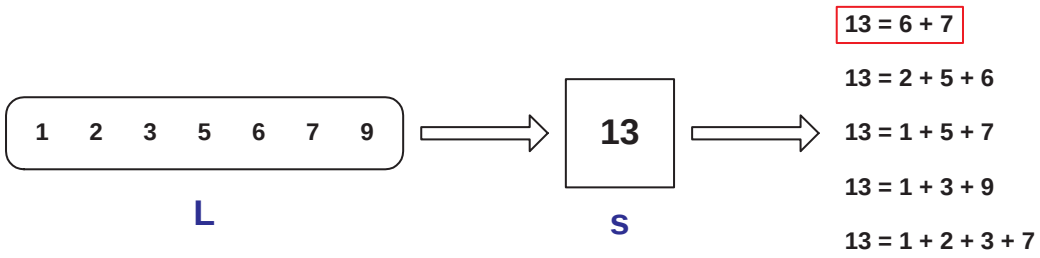
```
def elemento_en_posicion(a, inf, sup):
    mitad = (inf+sup)//2

    if mitad==a[mitad]:  # Caso base 1
        return mitad
    elif inf>=sup:  # Caso base 2
        return -1
    elif mitad<a[mitad]:  # Casos recursivos
        return elemento_en_posicion(a, inf, mitad-1)
    else:
        return elemento_en_posicion(a, mitad+1, sup)
```

Ejercicio 4 [4 puntos]

Se pide implementar un algoritmo basado en la técnica de *backtracking* para resolver el siguiente problema.

Dada una lista L de números enteros no negativos (almacenada en un simple array/lista), y otro número entero no negativo s, se desea hallar un conjunto de números de L tal que su suma sea igual a s, y que además dicho conjunto sea el de menor cardinalidad de entre todos los posibles que cumplan la restricción asociada a la suma. El conjunto se especificará mediante un vector de booleanos de la misma longitud que L.

En el siguiente ejemplo hay varios conjuntos de elementos de L tal que su suma sea igual a 13 (s). El algoritmo devolvería el conjunto $\{6, 7\}$ al ser el de menor cardinalidad (es el que tiene el menor número de elementos):



NOTA: la solución no tiene por qué ser única. Si hay varios conjuntos que cumplan que la suma de sus elementos es s, y además tienen la misma mínima cardinalidad, basta con devolver cualquiera de esos conjuntos.

Posible solución:

```
def busca_subconj_aux(L,s):
    sol_parcial = [False] * (len(L))
    sol_opt = [False] * (len(L))

    card_opt = busca_subconj(0,0,L,s,sol_parcial,sol_opt,len(L)+1,0)

    print(card_opt)
    print(sol_opt)
```

```
def busca_subconj(i,suma_parcial,L,s,sol_parcial,sol_opt,card_opt,card):

    # genera candidatos (esquema para búsqueda de SUBCONJUNTOS)
    for k in range(0,2):

        nueva_suma_parcial = suma_parcial + k*L[i]

        # Comprueba si se puede podar el árbol de recursión
        if nueva_suma_parcial<=s:

            # Actualiza solución parcial
            sol_parcial[i] = (k==1)

            # Incrementa cardinalidad si se incluye el candidato
            if k==1:
                card = card+1

            # Comprueba si se ha alcanzado el valor en s
            if nueva_suma_parcial==s:
                # Actualiza solución y valor óptimos si menor cardinalidad
                if card<card_opt:
                    card_opt = card
                    for j in range(0,len(L)):
                        sol_opt[j] = sol_parcial[j]

            elif i<len(L)-1 and card<card_opt-1:
                card_opt = busca_subconj(i+1,nueva_suma_parcial,L,s,
                    sol_parcial,sol_opt,card_opt,card)

    sol_parcial[i] = False

    return card_opt
```

Universidad
Rey Juan Carlos

Prueba de evaluación

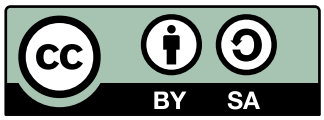
Soluciones examenes – mayo 2018-2019 Móstoles

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2018-2019 – 10 de mayo de 2019

Duración: **2 horas y 45 minutos**

Soluciones

1. Análisis de algoritmos

Ejercicio 1 [1 punto]

Ordena de menor a mayor orden de complejidad $\mathcal{O}$ las siguientes funciones (algunas pueden tener la misma complejidad):

$$10n, \quad n\log_{10} n, \quad n^2\log n, \quad 5\log n, \quad \log n^2, \quad n^2, \quad n^{1+a} \text{ con } 0<a<1$$

Solución:

$$\log n^2 \;=\; 5\log n \;<\; 10n \;<\; n\log_{10} n \;<\; n^{1+a}\ (\text{con } 0<a<1) \;<\; n^2 \;<\; n^2\log n$$

Ejercicio 2 [3 puntos]

Calcula el número de operaciones ($T(n)$) que realiza el siguiente código:

```
1  for (int i=0; i<n; i++)
2    for (int j=i+1; j<=n; j++)
3      if (condicion(i))    // una operación
4        for (int k=0; k<j; k++)
5          procesa(i,j,k);  // dos operaciones
```

Se considera que las inicializaciones, comparaciones, e incrementos siempre necesitan una sola operación.

Solución:

El código consta de 3 bucles, que podemos descomponer de la siguiente manera:

$$T(n) = 1 + \sum_{i=0}^{n-1}(1 + T_{\text{For 2}} + 1) + 1$$

$$T_{\text{For 2}} = 1 + \sum_{j=i+1}^{n}(1 + T_{\text{If}} + 1) + 1$$

$$T_{\text{If}} = \begin{cases} 1 & \text{en el mejor caso} \\ 1 + 1 + \sum_{k=0}^{j-1}(1+2+1) + 1 = 3 + 4j & \text{en el peor caso} \end{cases}$$

Veamos primero el coste en el mejor caso. Si $T_{\text{If}} = 1$, entonces sustituyendo en $T_{\text{For 2}}$ tenemos:

$$T_{\text{For 2}} = 2 + \sum_{j=i+1}^{n} 3 = 2 + 3(n-i) = 3n + 2 - 3i.$$

Sustituyendo en $T(n)$ obtenemos:

$$T(n) = 2 + \sum_{i=0}^{n-1}(3n + 4 - 3i) = 2 + 3n^2 + 4n - 3\sum_{i=0}^{n-1} i$$

$$= 2 + 3n^2 + 4n - 3\frac{(n-1)n}{2} = \frac{4 + 6n^2 + 8n - 3n^2 + 3n}{2} = \frac{3n^2 + 11n + 4}{2} \in \Theta(n^2)$$

En el peor caso, sustituyendo T_{If} en $T_{\text{For 2}}$ tenemos:

$$T_{\text{For 2}} = 2 + \sum_{j=i+1}^{n} (5 + 4j) = 2 + 5(n-i) + 4\sum_{j=i+1}^{n} j$$

$$= 2 + 5n - 5i + 4\left[\sum_{j=1}^{n} j - \sum_{j=1}^{i} j\right]$$

$$= 2 + 5n - 5i + 4\left[\frac{n(n+1)}{2} - \frac{i(i+1)}{2}\right]$$

Simplificando obtenemos:

$$T_{\text{For 2}} = 2 + 5n - 5i + 2n^2 + 2n - 2i^2 - 2i = 2n^2 + 7n + 2 - 7i - 2i^2$$

Sustituyendo en la expresión para el primer bucle tenemos:

$$T(n) = 2 + \sum_{i=0}^{n-1}(2n^2 + 7n + 4 - 7i - 2i^2) = 2 + 2n^3 + 7n^2 + 4n - 7\sum_{i=0}^{n-1} i - 2\sum_{i=0}^{n-1} i^2$$

$$= 2 + 2n^3 + 7n^2 + 4n - 7\frac{(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6}$$

$$= 2 + 2n^3 + 7n^2 + 4n - 7\frac{n^2}{2} + 7\frac{n}{2} - 2\frac{2n^3 - 3n^2 + n}{6}$$

$$= \frac{12 + 12n^3 + 42n^2 + 24n - 21n^2 + 21n - 4n^3 + 6n^2 - 2n}{6}$$

Finalmente:

$$T(n) = \frac{8n^3 + 27n^2 + 43n + 12}{6} \in \theta(n^3)$$

Ejercicio 3 [3 puntos]

Resolved la siguiente relación de recurrencia por el **método general de resolución de recurrencias**:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/4) + n^2 & \text{si } n > 1. \end{cases}$$

Solución:

Para poder aplicar el método primero tenemos que hacer el cambio de variable $n = 4^k$. Además, en ese caso:

$$n^2 = (4^k)^2 = 4^{2k} = (4^2)^k = 16^k.$$

Podemos reescribir la expresión para $T(n)$ de la siguiente manera:

$$T(n) = T(4^k) = T(4^k/4) + 16^k = T(4^{k-1}) + 16^k$$

A continuación hacemos un cambio de función $t(k) = T(4^k)$:

$$T(n) = T(4^k) = t(k) = t(k-1) + 16^k$$

El polinomio característico de la recurrencia $t(k) = t(k-1) + 16^k$ es:

$$(x-1)(x-16)$$

Por tanto, la recurrencia tiene la siguiente forma:

$$t(k) = C_1 1^k + C_2 16^k = C_1 + C_2 16^k$$

En este momento podemos deshacer el cambio de variable fácilmente:

$$T(n) = C_1 + C_2 n^2$$

Para hallar las constantes necesitamos dos casos base. Procedemos a calcular $T(4) = T(1) + 4^2 = 17$. Por tanto, el sistema de ecuaciones a resolver es:

$$\left.\begin{array}{rcrcr} T(1) & = & C_1 + C_2 & = & 1 \\ T(4) & = & C_1 + 16C_2 & = & 17 \end{array}\right\}$$

Las soluciones son $C_1 = -1/15$ y $C_2 = 16/15$. Por tanto:

$$T(n) = \frac{16}{15}n^2 - \frac{1}{15} \in \Theta(n^2)$$

Ejercicio 4 [3 puntos]

Resolved la siguiente relación de recurrencia por el **método de expansión de recurrencias**:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/4) + n^2 & \text{si } n > 1. \end{cases}$$

Solución:

Procedemos a expandir la recurrencia:

$$\begin{aligned} T(n) &= T(n/4) + n^2 \\ &= \left[T(n/4^2) + \left(\frac{n}{4}\right)^2\right] + n^2 = T(n/4^2) + n^2\left(1 + \frac{1}{16}\right) \\ &= \left[T(n/4^3) + \left(\frac{n}{4^2}\right)^2\right] + n^2\left(1 + \frac{1}{16}\right) = T(n/4^3) + n^2\left(1 + \frac{1}{16} + \frac{1}{16^2}\right) \\ &\vdots \\ &= T(n/4^i) + n^2\left(1 + \frac{1}{16} + \frac{1}{16^2} + \cdots + \frac{1}{16^{i-1}}\right) = T(n/4^i) + n^2 \sum_{j=0}^{i-1} \frac{1}{16^j} \end{aligned}$$

El último sumatorio es una serie geométrica:

$$\sum_{j=0}^{i-1} \frac{1}{16^j} = \frac{\frac{1}{16^i} - 1}{\frac{1}{16} - 1}.$$

Por tanto, tenemos:

$$T(n) = T(n/4^i) + n^2\left[\frac{\frac{1}{16^i} - 1}{\frac{1}{16} - 1}\right] = T(n/4^i) + n^2\left[\frac{\frac{1}{16^i} - 1}{-\frac{15}{16}}\right] T(n/4^i) + n^2\left[-\frac{16}{15} \cdot \frac{1}{16^i} + \frac{16}{15}\right]$$

Se llega al caso base cuando $n/4^i = 1$. Es decir, cuando $n = 4^i$. En ese caso $n^2 = 16^i$. Sustituyendo:

$$T(n) = T(1) + n^2\left[-\frac{16}{15}\frac{1}{n^2} + \frac{16}{15}\right] = 1 - \frac{16}{15} + \frac{16}{15}n^2 = \frac{16}{15}n^2 - \frac{1}{15} \in \Theta(n^2)$$

2. Diseño de algoritmos

Ejercicio 5 [5 puntos]

Se pide implementar un algoritmo basado en la estrategia de divide y vencerás para resolver el problema de la sublista de máxima suma. Dada una lista de números el objetivo es hallar una sublista de elementos contiguos cuya suma sea máxima. Por ejemplo, dada la lista $[-1, -4, 5, 2, -3, 4, 2, -5]$, la lista óptima es $[5, 2, -3, 4, 2]$, cuyos elementos suman 10. Se asumirá que la lista inicial no es vacía, y está compuesta de números enteros.

Solución:

El algoritmo debe descomponer la lista dada en dos mitades. La sublista de suma máxima estará en alguna de esas mitades (esto se calcula mediante dos llamadas recursivas), o será la concatenación de dos sublistas de ambas mitades, que sean contiguas en la original. Por ejemplo, la sublista de suma máxima de la mitad izquierda contendrá el elemento en la posición (mitad-1), mientras que la sublista de suma máxima de la mitad derecha contendrá el elemento en la posición (mitad). Ambas sublistas máximas se pueden hallar mediante simples bucles, y la suma máxima de su concatenación es la suma sus respectivas sumas máximas. El siguiente código describe el algoritmo:

Java:

```java
import java.io.*;

public class MaxSumList {

    public static int max(int a, int b) {
        if (a>b)
            return a;
        else
            return b;
    }
```

```java
    public static int maxima_suma_sublista(int a[], int ini, int fin) {
        int n = a.length;
        if (ini==fin)
            return a[ini];
        else {
            int mitad = (ini+fin)/2;

            // Calculas la suma máxima de ambas mitades de la lista a
            int max_mitad_izq = maxima_suma_sublista(a,ini,mitad);
            int max_mitad_dcha = maxima_suma_sublista(a,mitad+1,fin);

            // suma máxima de las listas de la mitad izquierda
            // que contienen el índice (mitad-1)
            int max_izq = -Integer.MAX_VALUE;
            int aux_suma = 0;
            for (int i=mitad; i>=0; i--) {
                aux_suma = aux_suma + a[i];
                max_izq = max(max_izq,aux_suma);
            }

            // suma máxima de las listas de la mitad derecha
            // que contienen el índice (mitad)
            int max_dcha = -Integer.MAX_VALUE;
            aux_suma = 0;
            for (int i=mitad+1; i<n; i++) {
                aux_suma = aux_suma + a[i];
                max_dcha = max(max_dcha,aux_suma);
            }

            return max(max(max_mitad_izq, max_mitad_dcha), max_izq+max_dcha);
        }
    }

    // función no pedida en el enunciado:
    public static void main(String args[]) throws Exception {
        //lectura:
        BufferedReader cin =
            new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(cin.readLine());

        int a[] = new int[n];

        //lectura:
        String[] listaEnteros = cin.readLine().split(" ");
        for (int i=0; i<n; i++) {
            int m = Integer.parseInt(listaEnteros[i]);
            a[i] = m;
        }

        //escritura:
        System.out.println(maxima_suma_sublista(a,0,n-1));
    }
}
```

Python:

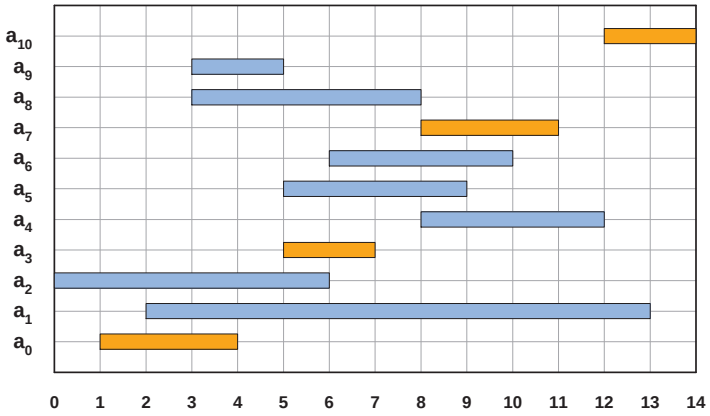
```python
import math

def maxima_suma_sublista(a):
    n = len(a)
    if n==1:
        return a[0]
    else:
        mitad = n//2

        # Calculas la suma máxima de ambas mitades de la lista a
        max_mitad_izq = maxima_suma_sublista(a[0:mitad])
        max_mitad_dcha = maxima_suma_sublista(a[mitad:])

        # suma máxima de las listas de la mitad izquierda
        # que contienen el índice (mitad-1)
        max_izq = -math.inf
        aux_suma = 0
        for i in range(mitad-1,-1,-1):
            aux_suma = aux_suma + a[i]
            max_izq = max(max_izq,aux_suma)

        # suma máxima de las listas de la mitad derecha
        # que contienen el índice (mitad)
        max_dcha = -math.inf
        aux_suma = 0
        for i in range(mitad,n):
            aux_suma = aux_suma + a[i]
            max_dcha = max(max_dcha,aux_suma)

        return max(max_mitad_izq, max_mitad_dcha, max_izq+max_dcha)

# No pedido en el enunciado:
a = [int(x) for x in input().split()]
print(maxima_suma_sublista(a))
```

Ejercicio 6 [5 puntos]

Se pide implementar un algoritmo basado en la técnica de *backtracking* para resolver el problema de selección de actividades. Sea un conjunto A de n actividades $\{a_0, a_1, \ldots, a_{n-1}\}$ que necesitan utilizar un recurso común, por ejemplo, una sala de reuniones. El recurso solo puede ser usado por una actividad en cada momento. Cada actividad tiene un instante de comienzo c_i y un instante de finalización f_i, donde $0 \leq c_i < f_i < \infty$. Si se selecciona la actividad a_i, se desarrolla en el intervalo semiabierto de tiempo $[c_i, f_i)$. Las actividades a_i y a_j son compatibles si sus intervalos $[c_i, f_i)$ y $[c_j, f_j)$ no se solapan, es decir, si $c_i \geq f_j$ o $c_j \geq f_i$.

El objetivo de este ejercicio es determinar un subconjunto de actividades compatibles cuya cardinalidad sea máxima. Por ejemplo, sea el siguiente conjunto de actividades:

i	0	1	2	3	4	5	6	7	8	9	10
c_i	1	2	0	5	8	5	6	8	3	3	12
f_i	4	13	6	7	12	9	10	11	8	5	14



Un subconjunto S de actividades compatibles es $\{a_2, a_4, a_{10}\}$. Sin embargo, no es un subconjunto de cardinalidad máxima, como lo son $\{a_0, a_3, a_7, a_{10}\}$ (en naranja en la figura) y $\{a_9, a_3, a_4, a_{10}\}$ (ya que tienen 4 elementos).

Aunque este problema se puede resolver con un algoritmo voraz, el método desarrollado deberá seguir una estrategia basada en *backtracking*. Se asumirá que las tareas no están ordenadas. Además, el algoritmo desarrollado no ordenará las tareas de ninguna manera. El algoritmo podrá usar funciones auxiliares para verificar la validez de los candidatos a introducir en la solución parcial.

Solución:

La solución consiste en analizar todos los subconjuntos posibles de tareas. Para ello el siguiente código usa el esquema para buscar subconjuntos basado en un árbol binario. Para podar el árbol de recursión se usa una función que comprueba si hay solapamiento entre la tarea i y las contenidas en la solución parcial (desde el índice 0 hasta el $i-1$). En caso de haber analizado todos los posibles candidatos (tareas), se actualiza la mejor solución y su cardinalidad si la cardinalidad del conjunto definido por solución parcial es mayor que la encontrada previamente por el algoritmo de *backtracking*.

Java:

```java
import java.io.*;

public class Tareas {

    public static boolean es_candidato_valido(int i, int sol[],
                                              int c[], int f[]) {

        boolean es_valido = true;

        int j=0;
        while ((j<i) && es_valido) {
            if (sol[j]==1)
                es_valido = (c[i]>=f[j] || c[j]>=f[i]);

            j = j + 1;
        }

        return es_valido;
    }


    public static int busca_maxima_cardinalidad(int i, int card_actual,
                                                int sol[], int opt_sol[],
                                                int card_opt,
                                                int c[], int f[]) {

        int n = sol.length;

        if (i==n) {
            if (card_actual>card_opt) {
                card_opt = card_actual;
                for (int k=0; k<n; k++)
                    opt_sol[k] = sol[k];
            }
        }
        else {
            for (int k=0; k<=1; k++) {

                if ((k==0) || es_candidato_valido(i,sol,c,f)) {

                    sol[i] = k;

                    card_actual = card_actual + k;

                    card_opt = busca_maxima_cardinalidad(i+1,card_actual,
                                                         sol,opt_sol,card_opt,c,f);
                }
            }
        }

        return card_opt;
    }
```

```java
    public static int planificacion_tareas(int c[], int f[]) {

        int n = c.length;

        int sol[] = new int[n];
        int opt_sol[] = new int[n];

        int card_opt =  busca_maxima_cardinalidad(0,0,sol,opt_sol,-1,c,f);

        // opcional
        for (int i=0; i<n-1; i++)
            System.out.print(opt_sol[i] + " ");
        System.out.println(opt_sol[n-1]);

        return card_opt;
    }

    // función no pedida en el enunciado:
    public static void main(String args[]) throws Exception {
        //lectura:
        BufferedReader cin =
            new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(cin.readLine());

        int c[] = new int[n];
        int f[] = new int[n];

        //lectura:
        String[] listaEnteros = cin.readLine().split(" ");
        for (int i=0; i<n; i++) {
            int m = Integer.parseInt(listaEnteros[i]);
            c[i] = m;
        }

        listaEnteros = cin.readLine().split(" ");
        for (int i=0; i<n; i++) {
            int m = Integer.parseInt(listaEnteros[i]);
            f[i] = m;
        }

        //escritura:
            System.out.println(planificacion_tareas(c,f));
    }
}
```

Python:

```python
def es_candidato_valido(i,sol,c,f):

    es_valido = True

    j=0
    while j<i and es_valido:
       # ver si la tarea i se solapa con las anteriores,
       # previamente incluidas en la solución parcial
        if sol[j]==1:
            es_valido = (c[i]>=f[j] or c[j]>=f[i])

        j = j + 1

    return es_valido

def busca_maxima_cardinalidad(i,card_actual,sol,opt_sol,card_opt,c,f):
    n = len(sol)

    if i==n:
        if card_actual>card_opt:  # actualizar si se encuentra una solución
            mejor
            card_opt = card_actual
            for k in range(n):
                opt_sol[k] = sol[k]
    else:
        for k in range(0,2):  # árbol de recursión binario

            if k==0 or es_candidato_valido(i,sol,c,f):

                sol[i]=k

                card_actual = card_actual + k

                card_opt = busca_maxima_cardinalidad(i+1,card_actual,sol,
                    opt_sol,card_opt,c,f)

    return card_opt

def planificacion_tareas(c,f):
    n = len(c)

    sol = [None]*n
    opt_sol = [None]*n

    card_opt =  busca_maxima_cardinalidad(0,0,sol,opt_sol,-1,c,f)

    print(opt_sol)  # opcional
    return card_opt

# No pedido en el enunciado:
c = [int(x) for x in input().split()]
f = [int(x) for x in input().split()]
print(planificacion_tareas(c,f))
```

Universidad
Rey Juan Carlos

Prueba de evaluación

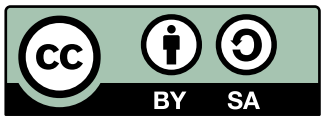
Soluciones examenes – mayo 2018-2019 Vicálvaro

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2018-2019 – 14 de mayo de 2019

Duración: **2 horas y 45 minutos**

Soluciones

1. Análisis de algoritmos

Ejercicio 1 [2 puntos]

Demostrar **mediante la definición** de complejidad asintótica $\mathcal{O}$, si se verifican:

a) ¿ $2^{n+1} \in \mathcal{O}(2^n)$? **[1 punto]**

b) ¿ $2^{2n} \in \mathcal{O}(2^n)$? **[1 punto]**

Solución:

En primer lugar, $2^{n+1} \in \mathcal{O}(2^n)$ es cierto. Aplicando la definición debemos buscar una constante $c > 0$ y otra $n_0 > 0$ tal que:

$$2^{n+1} \leq c \cdot 2^n$$

en un intervalo $[n_0, \infty)$. Como $2^{n+1} = 2 \cdot 2^n$, tenemos que encontrar una c que cumpla:

$$2 \cdot 2^n \leq c \cdot 2^n$$

Escogiendo $c = 2$, la desigualdad anterior naturalmente se cumple para todo n. Por tanto, se cumple en un intervalo $[n_0, \infty)$ para cualquier valor de de n_0. Como al aplicar la definición $n_0 > 0$, podemos escoger $n_0 = 1$. De esta manera, hemos hallado una pareja de constantes que hacen que $2^{n+1} \in \mathcal{O}(2^n)$ sea cierto.

En segundo lugar, $2^{2n} \in \mathcal{O}(2^n)$ no es cierto. Por eso, en vez de intentar encontrar las constantes c y n_0, tenemos que demostrar que no va a ser posible hallarlas. Inicialmente, al intentar aplicar la definición deberíamos encontrar una constante $c > 0$ y otra $n_0 > 0$ tal que:

$$2^{2n} \leq c \cdot 2^n$$

en un intervalo $[n_0, \infty)$. Aplicamos logaritmos (en base 2) a ambos lados:

$$\log_2(2^{2n}) \leq \log_2 c \cdot 2^n$$

$$2n \leq \log_2 c + n$$

$$n \leq \log_2 c$$

La última expresión indica que la desigualdad se cumple para todo n menor o igual que la constante $\log_2 c$. Por tanto, sea cual sea el valor de c, la desigualdad se cumple en el intervalo $(-\infty, \log_2 c]$, pero nunca en un intervalo $[n_0, \infty)$. Por tanto, es imposible hallar la pareja de constantes c y n_0 que necesitamos para verificar $2^{2n} \in \mathcal{O}(2^n)$.

Ejercicio 2 [2 puntos]

Calcula el número de operaciones ($T(n)$) que realiza el siguiente código:

```
1  int i=0;
2  while (i<=n){
3      int j=i;
4      while (j<=n){
5          procesa(i,j);   // dos operaciones
6          j++;
7      }
8      i++;
9  }
```

Se considera que las inicializaciones, comparaciones, e incrementos siempre necesitan una sola operación.

Solución:

El código consta de 2 bucles, que podemos descomponer de la siguiente manera:

$$T(n) = 1 + \sum_{i=0}^{n}(1 + T_{\text{For 2}} + 1) + 1$$

$$T_{\text{For 2}} = 1 + \sum_{j=i}^{n}(1 + 2 + 1) + 1 = 2 + 4(n - i + 1) = 4n + 6 - 4i$$

Sustituyendo $T_{\text{For 2}}$ en $T(n)$ obtenemos:

$$T(n) = 2 + \sum_{i=0}^{n}(2 + 4n + 6 - 4i) = 2 + \sum_{i=0}^{n}(4n + 8 - 4i)$$

$$= 2 + 4n(n+1) + 8(n+1) - 4\sum_{i=0}^{n} i$$

$$= 2 + 4n^2 + 4n + 8n + 8 - 4\frac{n(n+1)}{2}$$

$$= 2n^2 + 10n + 10 \in \Theta(n^2)$$

Ejercicio 3 [3 puntos]

Resolved la siguiente relación de recurrencia por el **método general de resolución de recurrencias**:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/10) + log_{10}(n^2) & \text{si } n > 1. \end{cases}$$

Solución:

En primer lugar podemos expresar $log_{10}(n^2)$ como $2\log_{10}(n)$. Luego, para poder aplicar el método, tenemos que hacer el cambio de variable $n = 10^k$. Además, en ese caso $k = \log_{10}(n)$. Por tanto, podemos reescribir la expresión para $T(n)$ de la siguiente manera:

$$T(n) = T(10^k) = T(10^k/10) + 2k = T(10^{k-1}) + 2k$$

A continuación hacemos un cambio de función $t(k) = T(10^k)$:

$$T(n) = T(10^k) = t(k) = t(k-1) + 2k \cdot 1^k$$

El polinomio característico de la recurrencia $t(k) = t(k-1) + 2k \cdot 1^k$ es:

$$(x-1)^3$$

Por tanto, la recurrencia tiene la siguiente forma:

$$t(k) = C_1 1^k + C_2 k \cdot 1^k + C_2 k^2 \cdot 1^k = C_1 + C_2 k + C_3 k^2$$

Para hallar las constantes necesitamos tres casos base. En primer lugar conocemos $T(1) = t(0) = 1$. Aplicando $t(k) = t(k-1) + 2k$ vemos que:

$$t(1) = t(0) + 2 \cdot 1 = 1 + 2 = 3$$

$$t(2) = t(1) + 2 \cdot 2 = 3 + 4 = 7$$

Por tanto, el sistema de ecuaciones a resolver es:

$$\left.\begin{array}{rcrcrcrcl} t(0) & = & C_1 & + & & & & = & 1 \\ t(1) & = & C_1 & + & C_2 & + & C_3 & = & 3 \\ t(2) & = & C_1 & + & 2C_2 & + & 4C_3 & = & 7 \end{array}\right\}$$

Las soluciones son $C_1 = 1$, $C_2 = 1$ y $C_3 = 1$. Por tanto:

$$t(k) = 1 + k + k^2$$

Finalmente, deshaciendo el cambio de variable y función obtenemos:

$$T(n) = 1 + \log_{10}(n) + [\log_{10}(n)]^2 \quad \in \quad \Theta[(\log n)^2]$$

Ejercicio 4 [3 puntos]

Resolved la siguiente relación de recurrencia por el **método de expansión de recurrencias**:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/10) + log_{10}(n^2) & \text{si } n > 1. \end{cases}$$

Solución:

En primer lugar podemos expresar $log_{10}(n^2)$ como $2\log_{10}(n)$, quedando:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/10) + 2\log_{10}(n) & \text{si } n > 1. \end{cases}$$

Procedemos a expandir la recurrencia:

$$\begin{aligned}
T(n) &= T(n/10) + 2\log_{10} n \\
&= \left[T(n/10^2) + 2\log_{10}(n/10)\right] + 2\log_{10} n \\
&= T(n/10^2) + 2\log_{10} n - 2\log_{10} 10 + 2\log_{10} n \\
&= \left[T(n/10^3) + 2\log_{10}(n/100)\right] + 2\log_{10} n - 2\log_{10} 10 + 2\log_{10} n \\
&= T(n/10^3) + 2\log_{10} n - 2\log_{10} 100 + 2\log_{10} n - 2\log_{10} 10 + 2\log_{10} n \\
&= T(n/10^3) + 3 \cdot 2\log_{10} n - 2(1+2) \\
&\vdots \\
&= T(n/10^4) + 4 \cdot 2\log_{10} n - 2(1+2+3) \\
&\vdots \\
&= T(n/10^i) + 2i\log_{10} n - 2\sum_{j=1}^{i-1} j \\
&= T(n/10^i) + 2i\log_{10} n - (i-1)i
\end{aligned}$$

Se llega al caso base cuando $n/10^i = 1$. Es decir, cuando $n = 10^i$, o de manera equivalente, cuando $i = \log_{10} n$. Sustituyendo:

$$T(n) = T(1) + 2(\log_{10} n)(\log_{10} n) - (\log_{10} n - 1)(\log_{10} n)$$

$$T(n) = 1 + \log_{10}(n) + [\log_{10}(n)]^2 \quad \in \quad \Theta[(\log n)^2]$$

2. Diseño de algoritmos

Ejercicio 5 [5 puntos]

Se pide implementar un algoritmo basado en la estrategia de divide y vencerás para determinar si una matriz **A** de números enteros está "ordenada" por filas y columnas independientemente. Para ello, los elementos de cada fila y cada columna deben aparecer en orden no-decreciente. La matriz **A** no es necesariamente cuadrada. La siguiente matriz estaría ordenada según el criterio descrito:

$$\mathbf{A} = \begin{pmatrix} 1 & 3 & 6 & 8 & 10 \\ 2 & 4 & 7 & 9 & 10 \\ 4 & 8 & 11 & 14 & 14 \\ 5 & 10 & 12 & 15 & 16 \end{pmatrix}$$

Solución:

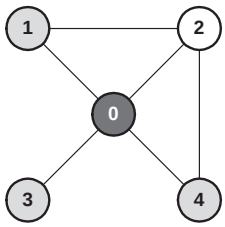
Una posible solución consiste en dividir la matriz en cuatro submatrices (partimos la matriz por la fila central y por la columna central). En el caso recursivo cada una de las submatrices debe estar "ordenada" según especifica el enunciado. Para ello se realizarían cuatro llamadas recursivas. Además, los elementos de las dos columnas centrales, y de las dos filas centrales deben estar ordenados de menor a mayor. Esto se puede ver empleando un bucle en cada caso. El siguiente código ilustra una posible implementación de la función.

```
def esta_ordenada_matriz(A):
    (n,m) = A.shape

    if n==1 and m==1:
        return True
    elif n==0 or m==0:
        return True
    else:

        mitad_n = n//2
        mitad_m = m//2

        elementos_centrales_ordenados = True;
        i = 0
        while i<n and elementos_centrales_ordenados:
            elementos_centrales_ordenados = A[i,mitad_m-1]<=A[i,mitad_m]
            i = i+1

        j = 0
        while j<m and elementos_centrales_ordenados:
            elementos_centrales_ordenados = A[mitad_n-1,j]<=A[mitad_n,j]
            j = j+1

        return (elementos_centrales_ordenados and
                esta_ordenada_matriz(A[0:mitad_n,0:mitad_m]) and
                esta_ordenada_matriz(A[0:mitad_n,mitad_m:m]) and
                esta_ordenada_matriz(A[mitad_n:n,0:mitad_m]) and
                esta_ordenada_matriz(A[mitad_n:n,mitad_m:m]))
```

Ejercicio 6 [5 puntos]

Sea un grafo G no dirigido, y no ponderado, compuesto por n vértices. El grafo G estará definido mediante una matriz de adyacencia simétrica y cuadrada $\mathbf{A}$ de tamaño $n \times n$, cuyos elementos serán 0 o 1. Si el elemento $a_{i,j} = 0$, no habrá una arista conectando a los vértices i y j, mientras que si $a_{i,j} = 1$ sí que habrá una arista conectándolos y serán adyacentes. Se asumirá que no habrá aristas desde un vértice a sí mismo (es decir, $a_{i,i} = 0$ para todo i).

Dado un número entero $m \leq n$, se pide implementar un algoritmo basado en la técnica de *backtracking* que determine si es posible colorear los vértices del grafo con m colores diferentes, de manera que no se asigne el mismo color a dos vértices adyacentes. El siguiente grafo se puede colorear usando 3 colores, pero no es posible colorearlo usando menos colores:



$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

El algoritmo podrá usar funciones auxiliares para verificar la validez de los candidatos a introducir en la solución parcial.

Solución:

En este ejercicio la solución parcial será una lista de n enteros, correspondientes a los n vértices del grafo. Los elementos de la solución parcial podrán tomar m valores diferentes (por ejemplo, desde 0 hasta $m-1$), donde cada uno representará un color diferente. Por tanto, el algoritmo de *backtracking* tendrá un bucle para generar los m posibles candidatos (colores). El bucle será de tipo While, para para en cuanto se haya encontrado una coloración válida. A continuación, se analiza si el color para el vértice i-ésimo es válido. Para ello se puede llamar a una función que compruebe si el color que pretendemos usar para el vértice i-ésimo ya aparece en uno de sus vértices adyacentes. Si el candidato es válido se actualiza la solución parcial y se llama a la función recursiva para expandir la solución parcial a partir del vértice $i+1$. El caso recursivo termina devolviendo si ha encontrado una coloración válida. Por último, el caso base del algoritmo simplemente comprueba si la solución parcial se ha completado, en cuyo caso la coloración sería válida, y la función puede devolver `True`. El siguiente código ilustra una posible implementación.

```python
import numpy as np

def es_candidato_valido(k,i,sol,A):

    es_valido = True

    j=0
    while j<i and es_valido:
        if A[j,i]==1 and sol[j]==k:
            es_valido = False
        j = j + 1

    return es_valido


def colorear_backtracking(i,m,sol,A):
    n = A.shape[0]

    if i==n:
        print(sol)  # opcional
        return True
    else:

        exito = False
        k = 0
        while k<m and not exito:

            if es_candidato_valido(k,i,sol,A):

                sol[i] = k

                exito = colorear_backtracking(i+1,m,sol,A)

            k = k + 1

    return exito


def puede_colorear(A,m):
    n = A.shape[0]

    sol = np.empty(n)

    return colorear_backtracking(0,m,sol,A)
```

Universidad
Rey Juan Carlos

Prueba de evaluación

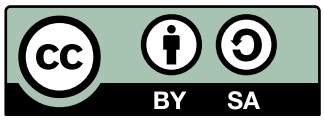
Soluciones examen análisis – junio 2019-2020

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2019-2020 – 18 de mayo de 2020

Soluciones – Análisis de algoritmos

Ejercicio 1 [**1 punto**] [Duración: **10 minutos**]

Demuestra usando la definición de Θ que:

$$\log_4(n) \in \Theta \log_2(n)$$

Solución:

Como nos piden demostrar una expresión que involucra a Θ (cota ajustada), debemos demostrar tanto $\log_4(n) \in \mathcal{O}(\log_2(n))$, como $\log_4(n) \in \Omega(\log_2(n))$. Además, podemos transformar el logaritmo en base 4 en un logaritmo en base 2 para facilitar las operaciones:

$$\log_4(n) = \frac{\log_2(n)}{\log_2(4)} = \frac{1}{2}\log_2(n)$$

Para la cota inferior Ω tenemos que encontrar dos constantes constante C_1 y n_1 tal que:

$$\frac{1}{2}\log_2(n) \geq C_1 \log_2(n)$$

se cumpla par todo $n \geq n_1$. Por ejemplo, podemos elegir $C_1 = 1/4$. En ese caso la expresión queda:

$$2\log_2(n) \geq \log_2(n)$$

La cual se verifica para todo $n \geq 1$ (si $n < 1$ los logaritmos serían negativos, y por tanto sería mayor el miembro derecho de la inecuación). Finalmente, podemos asignar $n_1 = 1$, y habríamos encontrado una pareja de constantes que verifica la definición de Ω.

Para $\mathcal{O}$ el procedimiento es similar. Debemos encontrar dos constantes constante C_2 y n_2 tal que:

$$\frac{1}{2}\log_2(n) \leq C_2 \log_2(n)$$

se cumpla par todo $n \geq n_2$. Por ejemplo, eligiendo $C_2 = 1$ tenemos:

$$\frac{1}{2}\log_2(n) \leq \log_2(n)$$

la cual también se cumple para todo $n \geq 1$. Por tanto, podemos escoger $n_2 = 1$, y habríamos encontrado una pareja de constantes que verifican la definición de $\mathcal{O}$. De esta manera habríamos terminado la demostración y por tanto el ejercicio.

NOTA: la definición formal de Θ pide hallar tres constantes: C_1, C_2 y n_0. Como hemos hecho la demostración para Ω y $\mathcal{O}$ de manera independiente hemos obtenido C_1 y C_2, pero dos constantes n_1 y n_2 (que no siempre son iguales). En general, n_0 simplemente sería el máximo de n_1 y n_2.

Ejercicio 2 [**2 puntos**] [Duración: **10 minutos**]

Simplifica el siguiente sumatorio (la expresión final no debe contener ningún sumatorio):

$$\sum_{i=1}^{j} \left(1 - j - 2i + 6i^2 + nj + 8ni + 3 \cdot 4^i\right)$$

Solución:

El sumatorio consta de siete términos, con lo cual podemos interpretar que tenemos siete sumatorios independientes:

$$\sum_{i=1}^{j} 1 \; - \; \sum_{i=1}^{j} j \; - \; \sum_{i=1}^{j} 2i \; + \; \sum_{i=1}^{j} 6i^2 \; + \; \sum_{i=1}^{j} nj \; + \; \sum_{i=1}^{j} 8ni \; + \; \sum_{i=1}^{j} 3 \cdot 4^i$$

Las expresiones dentro del primer, segundo y quinto sumatorio no dependen de la variable i, por tanto solo tenemos que sumarlas j veces, lo que es equivalente a multiplicarlas por j. Del resto de sumatorios podemos extraer fuera todos los términos multiplicativos que no involucren a la variable i. De esta manera obtenemos:

$$j \; - \; j^2 \; - \; 2\sum_{i=1}^{j} i \; + \; 6\sum_{i=1}^{j} i^2 \; + \; nj^2 \; + \; 8n\sum_{i=1}^{j} i \; + \; 3\sum_{i=1}^{j} 4^i$$

Aplicando las fórmulas para la suma de los primeros naturales y sus cuadrados, y la fórmula para series geométricas (último sumatorio), obtenemos:

$$j \; - \; j^2 \; - \; 2\frac{j(j+1)}{2} \; + \; 6\frac{(2j+1)j(j+1)}{6} \; + \; nj^2 \; + \; 8n\frac{j(j+1)}{2} \; + \; 3\frac{4^{j+1}-4}{4-1}$$

Opcionalmente podemos cancelar denominadores:

$$j \; - \; j^2 \; - \; j(j+1) \; + \; (2j+1)j(j+1) \; + \; nj^2 \; + \; 4nj(j+1) \; + \; 4^{j+1} \; - \; 4$$

Ejercicio 3 [**1 punto**] [Duración: **10 minutos**]

Halla una cota inferior y una superior del siguiente sumatorio:

$$\sum_{j=0}^{n-1} e^j$$

Solución:

Este ejercicio se puede resolver de dos maneras teniendo en cuenta el contenido de la asignatura. En primer lugar, se puede usar la técnica de aproximación por integrales. En ese caso la función dentro del sumatorio, que es $f(x) = e^x$, es monótona creciente. Por tanto, de manera general las cotas vendrían dadas por:

$$\int_{m-1}^{n} f(x)dx \quad \leq \quad \sum_{j=m}^{n} f(j) \quad \leq \quad \int_{m}^{n+1} f(x)dx$$

En nuestro caso el límite inferior es 0 ($m = 0$), mientras que el superior es $n - 1$ (en vez de n). Así que tendríamos:

$$\int_{-1}^{n-1} e^x dx \quad \leq \quad \sum_{j=0}^{n-1} e^j \quad \leq \quad \int_{0}^{n} e^x dx$$

Además,

$$\int e^x dx = e^x$$

Por tanto:

$$\left[e^x\right]_{-1}^{n-1} \quad \leq \quad \sum_{j=0}^{n-1} e^j \quad \leq \quad \left[e^x\right]_{0}^{n}$$

Quedando:

$$e^{n-1} - e^{-1} = \frac{e^n - 1}{e} \quad \leq \quad \sum_{j=0}^{n-1} e^j \quad \leq \quad e^n - 1$$

La segunda forma de resolver este ejercicio simplemente consiste en darse cuenta de que el sumatorio del enunciado es una serie geométrica, para la cual podemos hallar una expresión exacta:

$$\sum_{j=0}^{n-1} e^j = \frac{e^n - 1}{e - 1}$$

Esta expresión sería tanto cota superior como inferior (considerando desigualdades del tipo $\leq$ y $\geq$).

Ejercicio 4 [**3 puntos**] [Duración: **15 minutos**]

Resolved la siguiente relación de recurrencia por el **método general de resolución de recurrencias**:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ 2T(n/4) + \log_2(n) & \text{si } n > 1. \end{cases}$$

Para poder aplicar el método tenemos que hacer el cambio de variable $n = 4^k$. En ese caso podemos reescribir la expresión para $T(n)$ de la siguiente manera:

$$T(n) = T(4^k) = 2T(4^k/4) + \log_2(4^k) = 2T(4^k/4) + k\log_2(4) = 2T(4^k/4) + 2k$$

A continuación hacemos un cambio de función $t(k) = T(4^k)$:

$$T(n) = T(4^k) = t(k) = 2t(k-1) + 2k \cdot 1^k$$

El polinomio característico de la recurrencia $t(k) = 2t(k-1) + 2k \cdot 1^k$ es:

$$(x-2)(x-1)^2$$

Por tanto, la recurrencia tiene la siguiente forma:

$$t(k) = C_1 2^k + C_2 \cdot 1^k + C_3 \cdot k \cdot 1^k = C_1 2^k + C_2 + C_3 k$$

Para hallar las constantes necesitamos tres casos base. En primer lugar conocemos $T(1) = t(0) = 1$. Aplicando $t(k) = 2t(k-1) + 2k$ vemos que:

$$t(1) = 2t(0) + 2 \cdot 1 = 2 + 2 = 4$$

$$t(2) = 2t(1) + 2 \cdot 2 = 8 + 4 = 12$$

Por tanto, el sistema de ecuaciones a resolver es:

$$\left.\begin{array}{rcrcrcrcr} t(0) & = & C_1 & + & C_2 & & & = & 1 \\ t(1) & = & 2C_1 & + & C_2 & + & C_3 & = & 4 \\ t(2) & = & 4C_1 & + & C_2 & + & 2C_3 & = & 12 \end{array}\right\}$$

Las soluciones son $C_1 = 5$, $C_2 = -4$ y $C_3 = -2$. Por tanto:

$$t(k) = 5 \cdot 2^k - 4 - 2k$$

Finalmente, debemos deshacer el cambio de variable. Por un lado, dado el cambio de variable aplicado, $k = \log_4(n)$. Además,

$$n = 4^k = (2^2)^k = 2^{2k} = (2^k)^2$$

Por tanto, $2^k = \sqrt{n}$. De esta manera la expresión final para $T(n)$ es:

$$T(n) = 5\sqrt{n} - 4 - 2log_4(n) \quad \in \quad \Theta[\ \sqrt{n}\ ]$$

Ejercicio 5 [**3 puntos**] [Duración: **15 minutos**]

Resolved la siguiente relación de recurrencia por el **método de expansión de recurrencias**:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ 2T(n/4) + \sqrt{n} & \text{si } n > 1. \end{cases}$$

Procedemos a expandir la recurrencia:

$$\begin{aligned} T(n) &= 2T(n/4) + \sqrt{n} \\ &= 2\left[\, 2T(n/4^2) + \sqrt{\frac{n}{4}} \,\right] + \sqrt{n} \\ &= 4T(n/4^2) + 2\sqrt{n} \\ &= 4\left[\, 2T(n/4^3) + \sqrt{\frac{n}{4^2}} \,\right] + 2\sqrt{n} \\ &= 8T(n/4^3) + 3\sqrt{n} \\ &\vdots \\ &= 2^i T(n/4^i) + i\sqrt{n} \end{aligned}$$

Se llega al caso base cuando $n/4^i = 1$. Es decir, cuando $n = 4^i$, o de manera equivalente, cuando $i = \log_4(n)$. Además, como

$$n = 4^i = (2^2)^i = 2^{2i} = (2^i)^2$$

entonces $2^i = \sqrt{n}$. Sustituyendo:

$$T(n) = \sqrt{n}T(1) + \log_4(n) \cdot \sqrt{n} = \sqrt{n} + \sqrt{n} \cdot \log_4(n) \quad \in \quad \Theta[\, \sqrt{n}\log(n) \,]$$

Universidad
Rey Juan Carlos

Prueba de evaluación

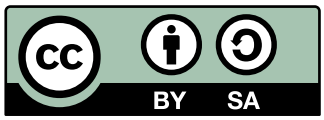
Soluciones examen diseño – junio 2019-2020 Móstoles

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos – Móstoles

Curso 2019-2020 – 4 de junio de 2020

Diseño de algoritmos - Soluciones

Ejercicio 1 [**2 puntos**] [Duración: **20 minutos**]

Implementa un algoritmo recursivo que reciba una lista de elementos (se pueden considerar números reales) y genere dos listas. La primera contendrá los elementos que se encuentren en índices pares, mientras que la segunda contendrá los elementos que se encuentran en índices impares. Suponed que el primer índice de una lista es el 0. Por ejemplo, dada la lista $\mathbf{a} = [3, 6, 5, 7, 0, 9, 2]$, el algoritmo generará las listas:

$$\mathbf{a}_{\text{par}} = [3, 5, 0, 2] \qquad \text{y} \qquad \mathbf{a}_{\text{impar}} = [6, 7, 9]$$

Posible solución:

En Python se pueden devolver varias salidas simultáneamente. El siguiente código devuelve las dos listas mediante una "tupla" (solo hay que especificar las salidas entre paréntesis y separadas por comas). Si la lista de entrada es vacía se devuelven dos listas vacías. Si tiene un elemento la lista de índices pares será la propia lista, mientras que la de impares será vacía. En el caso recursivo se extraen las listas de índices pares e impares de toda la lista exceptuando al último elemento (mediante una llamada recursiva). Después, lo único que falta es concatenar el último elemento a la lista de índices pares o impares, dependiendo de la paridad de la longitud de la lista original.

```python
def extrae_indices_pares_impares(a):
    n = len(a)
    if n==0:
        return ([],[])
    elif n==1:
        return (a,[])
    else:
        (lista_pares, lista_impares) = extrae_indices_pares_impares(a[:n-1])

        if n%2==0:
            return (lista_pares, lista_impares + [a[n-1]])
        else:
            return (lista_pares + [a[n-1]], lista_impares)
```

Ejercicio 2 [3 puntos] [Duración: **30 minutos**]

Implementa un algoritmo basado en la estrategia de **DIVIDE Y VENCERÁS** para evaluar un polinomio P de grado $n-1$:

$$P = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$$

en un determinado valor real x. Evaluar un polinomio simplemente consiste en determinar qué valor toma el polinomio para un valor de x concreto. Por ejemplo, si $P = 2x^3 - x + 2$, evaluarlo en $x = 1$ sería calcular $P(1) = 2 \cdot (1)^3 - 1 + 2 = 3$.

El polinomio se definirá mediante una lista $\mathbf{c} = [c_0, c_1, \ldots, c_{n-1}]$ de longitud n, que contiene los coeficientes (reales) del polinomio. Además, el algoritmo de divide y vencerás se basará necesariamente en el método descrito en el **Ejercicio 1** para dividir la lista **c** en dos:

$$\mathbf{a}_{\text{par}} = [c_0, c_2, \ldots] \qquad \text{y} \qquad \mathbf{a}_{\text{impar}} = [c_1, c_3, \ldots]$$

Se puede asumir que habéis implementado correctamente el método del Ejercicio 1 (podéis hacer llamadas al método como si estuviese implementado en alguna librería).

Posible solución:

Si el polinomio es una constante **c** tendrá longitud 1, y el resultado siempre será c_0. En caso contrario el polinomio se descompone en dos según los índices pares e impares. Por ejemplo, supongamos que deseamos evaluar $P(x) = 2x^6 + 9x^5 + 0x^4 + 7x^3 + 5x^2 + 6x + 3$ en un valor de x. La lista asociada al polinomio $\mathbf{c} = [3, 6, 5, 7, 0, 9, 2]$ primero se divide en:

$$\mathbf{a}_{\text{pares}} = [3, 5, 0, 2] \qquad \text{y} \qquad \mathbf{a}_{\text{impares}} = [6, 7, 9]$$

Esta descomposición daría lugar a dos subproblemas asociados a la evaluación de los polinomios:

$$P_{\text{pares}}(x) = 2x^3 + 0x^2 + 5x^1 + 3 \qquad \text{y} \qquad P_{\text{impares}}(x) = 9x^2 + 7x^1 + 6$$

Para obtener el algoritmo de divide y vencerás hay que pensar en cómo podemos modificar y combinar las soluciones a estos subproblemas (que obtenemos mediante llamadas recursivas). Para este problema si evaluamos P_{pares} y P_{impares} en x^2 obtendríamos:

$$P_{\text{pares}}(x^2) = 2x^6 + 0x^4 + 5x^2 + 3 \qquad \text{y} \qquad P_{\text{impares}}(x^2) = 9x^4 + 7x^2 + 6$$

cuya suma es casi $P(x)$. Lo único que faltaría sería multiplicar el polinomio de índices impares por x. Por tanto, podemos recuperar $P(x)$ de la siguiente manera, lo cual nos indica la regla recursiva:

$$P(x) = P_{\text{pares}}(x^2) + x \cdot P_{\text{impares}}(x^2).$$

```
1 def evalua_polinomio(c, x):
2     if len(c)==1:
3         return c[0]
4     else:
5         (cp,ci) = extrae_indices_pares_impares(c)
6 
7         return evalua_polinomio(cp, x*x) + x*evalua_polinomio(ci, x*x)
```

Ejercicio 3 [**5 puntos**] [Duración: **50 minutos**]

Implementa un algoritmo basado en la técnica de **BACKTRACKING** para generar permutaciones *con repetición*. Los datos de entrada son una lista $\mathbf{a} = [a_0, a_1, \ldots, a_{m-1}]$ de m elementos distintos, y otra lista $\mathbf{r} = [r_0, r_1, \ldots, r_{m-1}]$, también de longitud m, que indica el número de veces que un elemento de $\mathbf{a}$ debe aparecer en cada permutación. En concreto, r_i indica el número de repeticiones de a_i en una permutación. Esto implica que las permutaciones tendrán $n = r_0 + r_1 + \cdots + r_{m-1}$ elementos.

Por ejemplo, para $\mathbf{a}$ = ['a','b','c'] y $\mathbf{r} = [1, 2, 1]$, generaremos permutaciones de $n = 4$ elementos en las que 'a' y 'c' aparecerán una vez, y 'b' dos veces. En este ejemplo las permutaciones con repetición son:

c b b a
c b a b
c a b b
b c b a
b c a b
b b c a
b b a c
b a c b
b a b c
a c b b
a b c b
a b b c

El método a desarrollar debe imprimir todas las permutaciones con repetición. Además, debe calcular el número P de permutaciones generadas, a medida que las va creando. Podéis verificar si el algoritmo está generando P correctamente ya que:

$$P = \frac{n!}{r_0! \cdot r_1! \cdot \ldots \cdot r_{m-1}!}$$

En el ejemplo, $P = 4!/(1! \cdot 2! \cdot 1!) = 12$.

Se valorará el uso de estructuras de datos para acelerar la comprobación de validez de candidatos/soluciones parciales.

Posible solución:

La solución es muy parecida al algoritmo para generar permutaciones (sin repetición) visto en teoría (transparencias/libro). En ese método se usaba una lista de valores Booleanos que indicaban si un elemento todavía no se había utilizado en la permutación (es decir, si estaba libre). Esa lista también se puede interpretar como una lista binaria, donde un 0 indica que ya no puedes usar un elemento, mientras que un 1 indica que se puede utilizar una vez. Es decir, podemos interpretar que la lista indica el número de veces que puedes usar un elemento. Pues bien, para generar permutaciones con repetición el algoritmo es prácticamente idéntico, pero esa lista podrá tener valores mayores que 1. En concreto, inicialmente contendrá el número de repeticiones de los elementos. Es decir, la lista será precisamente $\mathbf{r}$. Si la vamos modificando a medida que avance el algoritmo la condición de validez es simplemente $r_k > 0$ para el k-ésimo elemento. Además, debemos decrementar r_k si incorporamos el elemento k a la solución parcial, y debemos incrementar r_k al retornar de la llamada recursiva. Por último, en el caso base se imprime una permutación y se devuelve un 1. En el caso recursivo se acumulan los resultados de las llamadas en la variable s, por lo que el método acaba devolviendo el número de permutaciones halladas.

```python
def genera_permutaciones_repeticion(i,sol,r,a):
    n = len(sol)
    m = len(a)

    # Caso base
    if i==n:
        for i in range(0,n):
            print(sol[i],' ',end='')
        print()

        return 1
    else:
        s = 0  # Número de permutaciones halladas

        # Genera candidatos (todos los posibles elementos)
        for k in range(m):

            # Comprueba su validez
            if r[k]>0:

                # Incluye candidato en solución parcial
                sol[i] = a[k]

                # Decrementa el número de apariciones del candidato k
                r[k] = r[k]-1

                # Expande la solución parcial a partir de la posición i+1
                s = s + genera_permutaciones_repeticion(i+1,sol,r,a)

                # Incrementa el número de apariciones del candidato k
                r[k] = r[k]+1

        return s  # Devolvemos el valor acumulado en s

def genera_permutaciones_repeticion_wrapper(a,r):
        n = 0
        for i in range(len(r)):
            n = n + r[i]

        sol = [None] * n
        nsols = genera_permutaciones_repeticion(0,sol,r,a)
        print('Permutaciones con repetición halladas = ',nsols)

        # Comprobación opcional
        m = len(r)
        p = 1
        for i in range(m):
            p = p * factorial(r[i])
        print('Permutaciones con repetición correctas = ',
              int(factorial(n)/p))

# Ejemplo
genera_permutaciones_repeticion_wrapper(['a','b','c'],[1,2,1])
```

Universidad
Rey Juan Carlos

Prueba de evaluación

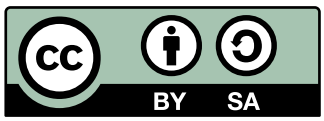
Soluciones examen diseño – junio 2019-2020 Vicálvaro

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos – Vicálvaro

Curso 2019-2020 – 8 de junio de 2020

Diseño de algoritmos - Soluciones

Ejercicio 1 [**3 puntos**] [Duración: **30 minutos**]

Implementa un algoritmo recursivo que, dada una lista $\mathbf{l} = [l_0, l_1, \ldots, l_{n-1}]$ de longitud n (podéis considerar que contiene caracteres) cuyos elementos se pueden repetir, calcule el número de apariciones de cada elemento distinto de la lista. Suponiendo que $\mathbf{l}$ contiene m elementos distintos, generará dos listas de longitud m. La primera ($\mathbf{d} = [d_0, d_1, \ldots, d_{m-1}]$) contendrá los elementos distintos de $\mathbf{l}$, mientras que la segunda ($\mathbf{a} = [a_0, a_1, \ldots, a_{m-1}]$) contendrá el número de apariciones de los elementos de $\mathbf{d}$ en la lista $\mathbf{l}$. Es decir, el elemento d_i aparecerá a_i veces en $\mathbf{l}$.

Por ejemplo, para $\mathbf{l} =$ ['b','a','c','b','a','a','d','a','c','a'], el algoritmo debe generar:

$$\mathbf{d} = \text{['a', 'c', 'd', 'b']} \qquad \text{y} \qquad \mathbf{a} = [5, 2, 1, 2]$$

ya que 'a' aparece 5 veces en $\mathbf{l}$, 'c' y 'b' aparecen 2 veces, y solo hay una 'd'.

El algoritmo podrá llamar a una función para buscar un elemento concreto en una lista. Esta función de búsqueda devolverá el índice en el que se encuentre el elemento, o -1 si éste no aparece en la lista. No es necesario implementar esta función.

Posible solución:

En Python se pueden devolver varias salidas simultáneamente. El siguiente código devuelve las dos listas mediante una "tupla" (solo hay que especificar las salidas entre paréntesis y separadas por comas). Si la lista de entrada es vacía se devuelven dos listas vacías. En el caso recursivo primero se resuelve el subproblema más sencillo (mediante una llamada recursiva), que en este caso contempla la lista de entrada menos su primer elemento (l_0). Con esta operación obtenemos una lista de elementos y otra de apariciones, y casi tenemos la solución completa, pero falta por tratar l_0. Primero, hacemos una búsqueda de l_0 en la lista de elementos. Si no lo encontramos entonces debemos aumentar la lista de elementos con l_0 (en este caso lo concatenamos al final) y debemos concatenar también un 1 a la de apariciones. En cambio, si l_0 está en la lista de elementos, en una determinada posición, entonces tendremos que incrementar el valor de la lista de apariciones en esa posición.

```python
# No es necesario implementar esta función
def busqueda_lineal(a, x):
    n = len(a)
    if a==[]:
        return -1
    elif a[n-1]==x:
        return n-1
    else:
        return busqueda_lineal(a[:n-1], x)

def elementos_y_apariciones(a):
    if a==[]:
        return ([], [])
    else:
        (elems, aps) = elementos_y_apariciones(a[1:])

        pos = busqueda_lineal(elems, a[0])

        if pos==-1:
            return (elems + [a[0]], aps + [1])
        else:
            aps[pos] = aps[pos]+1
            return (elems, aps)

# Ejemplo
print(elementos_y_apariciones(
        ['b','a','c','b','a','a','d','a','c','a']))
```

Ejercicio 2 [**2 puntos**] [Duración: **20 minutos**]

Implementa un algoritmo recursivo basado en la estrategia de **DIVIDE Y VENCERÁS** para evaluar un polinomio P de grado $n-1$:

$$P = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$$

en un determinado valor real x. Evaluar un polinomio simplemente consiste en determinar qué valor toma el polinomio para un valor de x concreto. Por ejemplo, si $P = 2x^3 - x + 2$, evaluarlo en $x = 1$ sería calcular $P(1) = 2 \cdot (1)^3 - 1 + 2 = 3$.

El polinomio se definirá mediante una lista $\mathbf{c} = [c_0, c_1, \ldots, c_{n-1}]$ de longitud n, que contiene los coeficientes (reales) del polinomio. El método podrá llamar a una función para calcular una potencia x^t, donde t es un entero (en python se puede usar la expresión `x**t`).

Posible solución:

Si el polinomio es una constante $\mathbf{c}$ tendrá longitud 1, y el resultado siempre será c_0. En el caso recursivo realizamos dos llamadas recursivas con cada una de las mitades de $\mathbf{c}$. El resultado será la suma de estos dos valores, pero debemos multiplicar uno de ellos por x^m, donde m es la longitud de la primera mitad de la lista. Por ejemplo, supongamos que deseamos evaluar $P(x) = 2x^6 + 9x^5 + 0x^4 + 7x^3 + 5x^2 + 6x + 3$ en un valor de x. La lista asociada al polinomio $\mathbf{c} = [3, 6, 5, 7, 0, 9, 2]$ primero se divide en:

$$\mathbf{c}_1 = [3, 6, 5] \qquad \text{y} \qquad \mathbf{c}_2 = [7, 0, 9, 2]$$

las cuales definen:

$$P_1(x) = 5x^2 + 6x + 3 \qquad \text{y} \qquad P_2(x) = 2x^3 + 9x^2 + 0x + 7$$

Finalmente, podemos recuperar $P(x)$ sumando los dos polinomios, pero multiplicando $P_2(x)$ por x^m para $m = 3$, que es la longitud de $\mathbf{c}_1$:

$$P(x) = P_2(x) \cdot x^m + P_1(x).$$

```python
def eval_dyv_alt(c, x):
    n = len(c)
    if n==1:
        return c[0]
    else:
        mitad = n//2

        val_1 = eval_dyv_alt(c[:mitad], x)
        val_2 = eval_dyv_alt(c[mitad:], x)

        return val_2*(x**(mitad)) + val_1

# Ejemplo
c = [2, -3, -1, 0, 1]  # x^4 - x^2 - 3x + 2
x = 2
print(eval_dyv(c, 0, len(c)-1, x))
print(eval_dyv_alt(c, x))
```

Ejercicio 3 [**5 puntos**] [Duración: **50 minutos**]

Implementa un algoritmo basado en la técnica de **BACKTRACKING** para generar combinaciones con repetición de n elementos tomados de m en m. Los datos de entrada son una lista $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ de n elementos distintos, y un entero m ($1 \leq m \leq n$). Las combinaciones con repetición son secuencias de m elementos de $\mathbf{a}$, en las que se pueden repetir los elementos, y el orden en el que aparecen los elementos no importa.

Por ejemplo, para $\mathbf{a}$ = ['a','b','c','d'] y $m = 3$ las combinaciones con repetición son:

a a a	b b b
a a b	b b c
a a c	b b d
a a d	b c c
a b b	b c d
a b c	b d d
a b d	c c c
a c c	c c d
a c d	c d d
a d d	d d d

Observad que la combinación (a a b) indica que escoges dos 'a' y una 'b', sin importar el orden. Por eso, sería un error generar además (a b a) o (b a a).

El método a desarrollar debe imprimir todas las combinaciones con repetición. Además, debe calcular el número C de permutaciones generadas, a medida que las va creando. Podéis verificar si el algoritmo está generando C correctamente ya que:

$$C = \frac{(n+m-1)!}{m! \cdot (n-1)!}$$

En el ejemplo, $C = 6!/(3! \cdot 3!) = 20$.

Se valorará el uso de estructuras de datos para acelerar la comprobación de validez de candidatos/soluciones parciales.

Posible solución:

La solución es muy parecida al segundo esquema para generar subconjuntos, en el que se obtenía una solución de tamaño m en el nivel m del árbol recursivo. La propiedad clave en este problema está relacionada con el orden de los elementos en la lista $\mathbf{a}$ (notad que los caracteres en el ejemplo siempre aparecen en orden ascendente en las combinaciones). En concreto, si en la posición i-ésima de la solución parcial aparece el elemento a_j, entonces en la $i+1$ solo podremos ubicar los elementos a_j, a_{j+1},...,a_{n-1}. Por tanto, debemos pasarle al método un parámetro con el valor de j. Otra forma (menos eficiente y más confusa) de implementar el algoritmo sería considerar los n candidatos de la lista $\mathbf{a}$, mediante un bucle donde k va desde 1 hasta n, pero usar la condición de validez $k \geq j$. Por último, en el caso base se imprime una combinación y se devuelve un 1. En el caso recursivo se acumulan los resultados de las llamadas en la variable s, por lo que el método acaba devolviendo el número de combinaciones halladas.

```
def genera_combinaciones_repeticion(i,j,sol,a):
    n = len(a)
    m = len(sol)

    # Caso base
    if i==m:
        for i in range(0,m):
            print(sol[i],' ',end='')
        print()

        return 1
    else:
        s = 0

        # Genera candidatos
        for k in range(j,n):

            # Incluye candidato en solución parcial
            sol[i] = a[k]

            # Expande la solución parcial en la posición i+1
            s = s + genera_combinaciones_repeticion(i+1,k,sol,a)

        return s


def genera_combinaciones_repeticion_wrapper(a,m):
    sol = [None] * m
    nsols = genera_combinaciones_repeticion(0,0,sol,a)
    print('Combinaciones con repetición halladas = ',nsols)

    # Opcional
    n = len(a)
    print('Combinaciones con repetición correctas = ',
          int(factorial(n+m-1)/factorial(m)/factorial(n-1)))


# Ejemplo
genera_combinaciones_repeticion_wrapper(['a','b','c','d'],3);
```

Universidad
Rey Juan Carlos

Prueba de evaluación

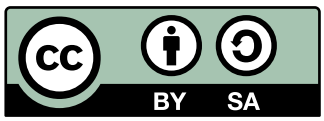
Soluciones examen diseño – enero 2021-2022

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2021-2022 – 18 de enero de 2022

Prueba de Diseño de Algoritmos

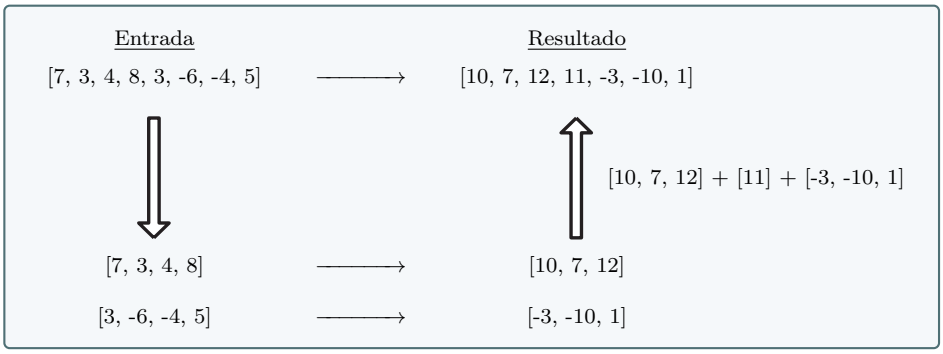
Soluciones

Ejercicio 1 [2 puntos]

Diseña un método **recursivo** basado en la técnica de **divide y vencerás** que, dada una lista **a** de números, devuelva una lista cuyos elementos sean la suma de dos elementos consecutivos de **a**. Por ejemplo, si $\mathbf{a} = [3, 5, -2, 8]$, el método debe devolver $[8, 3, 6]$. Se asumirá que la lista **a** no es vacía.

Posible solución:

Al aplicar divide y vencerás sobre una lista la estrategia debe basarse en las soluciones a los subproblemas que consideran las dos mitades de la lista de entrada. Para este problema el diagrama recursivo (como los vistos en clase) podría ser:

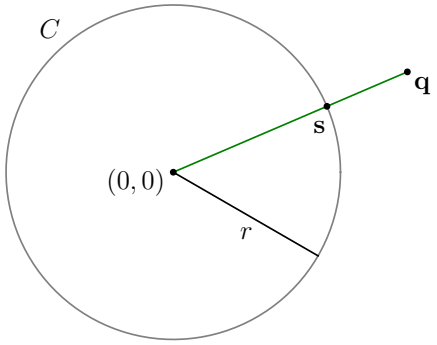


La regla recursiva debe concatenar el resultado del primer subproblema con la (lista de la) suma de los elementos centrales, y luego con el resultado del segundo subproblema. El código resultante podría ser el siguiente, donde se retorna una lista vacía en el caso base (cuando la lista de entrada solo tiene un elemento):

```
def f(a):
    if len(a)==1:
        return []
    else:
        mitad = len(a)//2

        return f(a[:mitad]) + [a[mitad-1]+a[mitad]] + f(a[mitad:])
```

Ejercicio 2 [3 puntos]

Dada una circunferencia C de radio r centrada en el origen, y un punto $\mathbf{q}$ en el plano, se pide implementar una variante **recursiva** del algoritmo de bipartición (también denominado "bisección") para hallar el punto de corte $\mathbf{s}$ entre C y el segmento cuyos extremos son el origen y $\mathbf{q}$. La siguiente figura ilustra el objetivo del ejercicio:



Se recuerda que la función que define la circunferencia C es: $x^2 + y^2 = r^2$. Además, un punto $\mathbf{p} = (p_x, p_y)$ se encontrará:

- Dentro de C si $p_x^2 + p_y^2 < r^2$,
- fuera de C si $p_x^2 + p_y^2 > r^2$,
- exactamente sobre C si $p_x^2 + p_y^2 = r^2$.

Se asumirá que el punto $\mathbf{q}$ inicial estará fuera de la circunferencia.

Por último, el método debe parar cuando la distancia entre el punto hallado por el algoritmo y el punto $\mathbf{s}$ sea menor o igual a un valor ϵ muy pequeño. Este valor se pasará como parámetro a la función a implementar.

Posible solución:

El algoritmo debe considerar como posible aproximación a **s** el punto medio (llamémosle **z**) de un segmento $\overline{\mathbf{tq}}$, donde inicialmente **t** es el origen $(0, 0)$. Posteriormente hará varias llamadas recursivas, dividiendo el segmento por dos, y analizando solo la mitad que pueda contener a **s**. El algoritmo irá dividiendo el segmento progresivamente hasta encontrar la solución exacta o hasta que su longitud sea lo suficientemente pequeña. A continuación se muestra una posible implementación:

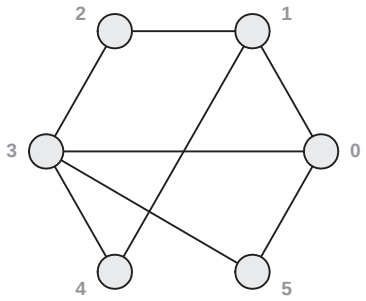
```
def modulo(v):
    return math.sqrt(v[0]**2 + v[1]**2)

def biparticion_listas(r, t, q, epsilon):
    # Punto medio entre t y q
    z = [(t[0]+q[0])/2, (t[1]+q[1])/2]

    # Vector q-t. Su módulo es la distancia entre t y q
    q_t = [q[0]-t[0], q[1]-t[1]]

    if modulo(z)==r or modulo(q_t)<=2*epsilon:
        return z
    elif modulo(z) > r:
        return biparticion_listas(r, t, z, epsilon)
    else:
        return biparticion_listas(r, z, q, epsilon)
```

El caso base corresponde a la situación en la que se halla la solución exacta (porque $\|\mathbf{z}\|$=r), o si la aproximación es lo suficientemente buena. Este segundo caso se da cuando la distancia entre **t** y **q** es menor que 2ϵ, ya que asegura que la distancia entre **z** y **s** sea menor que ϵ. En ambos casos el algoritmo devuelve **z**.

En los casos recursivos resolvemos el mismo problema pero usando un segmento cuya longitud es la mitad de la del original. Si el punto medio **z** se encuentra fuera de la circunferencia debemos usar $\overline{\mathbf{tz}}$, mientras que si está dentro debemos usar $\overline{\mathbf{zq}}$. Esto garantiza que la solución (**s**) esté contenida en el segmento considerado.

Ejercicio 3 [5 puntos]

Sea un grafo G no dirigido compuesto por n vértices, indexados desde 0 hasta $n-1$. El grafo G estará definido mediante una matriz de adyacencia simétrica y cuadrada $\mathbf{A}$ de tamaño $n \times n$, cuyos elementos serán 0 o 1. Si el elemento $a_{i,j} = 0$ no habrá una arista conectando a los vértices i y j, mientras que si $a_{i,j} = 1$ sí que habrá una arista conectándolos y serán adyacentes. Se asumirá que no habrá aristas desde un vértice a sí mismo (es decir, $a_{i,i} = 0$ para todo i). A continuación se muestra un ejemplo de un grafo con $n = 6$ nodos, donde los números en el grafo representan índices de los nodos:

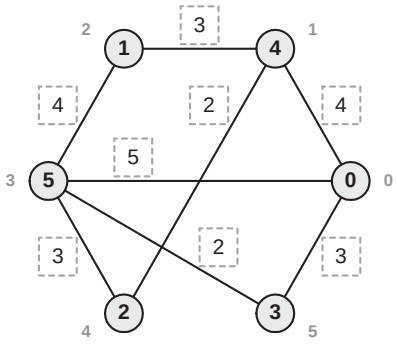


$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Se pide implementar un algoritmo basado en la técnica de **backtracking** para etiquetar los nodos del grafo con números diferentes desde 0 hasta $n-1$ con las siguientes condiciones:

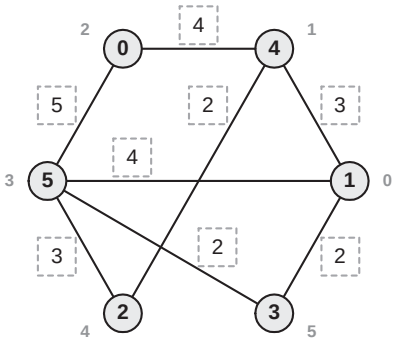
- Las etiquetas de dos nodos adyacentes (conectados mediante una arista) no pueden ser números consecutivos.
- Por otro lado, considerad que ponderamos las aristas con pesos correspondientes a los valores absolutos de las diferencias entre las etiquetas de nodos adyacentes. El algoritmo debe asignar etiquetas a los nodos de manera que se minimice la suma total de estos pesos.

La siguiente figura ilustra dos ejemplos donde los números dentro de los nodos circulares son las etiquetas, y los números en los cuadrados son los pesos de las aristas. En este caso la solución óptima corresponde al grafo etiquetado de la derecha.



Etiquetas: `[0, 4, 1, 5, 2, 3]`

Suma de pesos asociados a las aristas:
$2+2+3+3+3+4+4+5=26$



Etiquetas: `[1, 4, 0, 5, 2, 3]`

Suma de pesos asociados a las aristas:
$2+2+2+3+3+4+4+5=25$

Posible solución:

Como el grafo tiene n nodos y debemos asignarles una etiqueta diferente la solución consiste en hallar una *permutación*. Por tanto, podemos usar y adaptar el esquema de generación de permutaciones. Además, debemos usar parámetros para almacenar la solución (permutación) óptima y el valor óptimo que se pide minimizar. Para acelerar el algoritmo también es conveniente usar un parámetro que contenga la suma o "coste" (de los valores absolutos de las diferencias entre las etiquetas de nodos adyacentes) asociada a la solución parcial. Este parámetro evitará que tengamos que calcular toda la suma cada vez que se ejecute el método. Además, podremos usar su valor para podar el árbol recursivo si éste es mayor que el de la mejor solución hallada hasta el momento. Finalmente, la solución que se describe a continuación usa una función auxiliar para asegurar que no haya etiquetas consecutivas entre nodos adyacentes:

```
def condicion_consecutivos(k,i,sol,A):
    for j in range(i):
        if A[i][j]==1 and abs(k-sol[j])<=1:
            return False

    return True


def genera_etiquetas(i,coste,libres,sol,A,opt_val,opt_sol):
    n = len(sol)

    # Caso base
    if i==n:
        if coste<opt_val:  # Actualizamos opt_val y opt_sol
            opt_val = coste
            for j in range(n):
                opt_sol[j] = sol[j]

    else:
        # Genera candidatos
        for k in range(n):

            # Comprueba validez del candidato
            if libres[k] and condicion_consecutivos(k,i,sol,A):

                sol[i] = k   # Incluye candidato en solución parcial
                libres[k] = False   # Candidate k-ésimo ya no libre

                # Calculamos el coste asociado a añadir la etiqueta k
                coste_extra = 0
                for j in range(i):
                    if A[i][j]==1:
                        coste_extra = coste_extra + abs(k-sol[j])

                # Expande solución parcial si es posible mejorar la mejor solución
                    hallada
                if coste+coste_extra<opt_val:
                    opt_val = genera_etiquetas(i+1,coste+coste_extra,libres,sol,A,
                        opt_val,opt_sol)

                libres[k] = True   # Candidate k-ésimo vuelve a estar libre

    return opt_val


def genera_etiquetas_wrapper(A):
    n = len(A)
    libres = [True] * n
    sol = [0] * n
    opt_sol = [0] * n
    opt_val = genera_etiquetas(0,0,libres,sol,A,n**3,opt_sol)
```

Universidad
Rey Juan Carlos

Prueba de evaluación

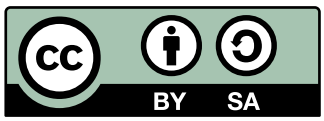
Soluciones examen análisis – mayo 2021-2022

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2021-2022 – 25 de mayo de 2022

Prueba de Análisis de Algoritmos

50 minutos

Soluciones

Ejercicio 1 [1.5 puntos]

Analiza aplicando la definición formal de $\mathcal{O}$ si se verifica:

$$n^3 + 6n^2 - 11n + 6 \ \in \ \mathcal{O}(n^3)$$

Solución:

Como nos piden demostrar una expresión que involucra a $\mathcal{O}$ debemos encontrar una pareja de constantes $c > 0$ y $n_0 > 0$ tal que se verifique:

$$n^3 + 6n^2 - 11n + 6 \ \leq \ c \cdot n^3,$$

para todo $n \geq n_0$. Como queremos que el miembro derecho sea mayor o igual que el izquierdo debemos escoger una $c > 1$. Por ejemplo, $c = 2$. En ese caso tenemos:

$$n^3 + 6n^2 - 11n + 6 \ \leq \ 2n^3,$$

$$0 \ \leq \ n^3 - 6n^2 + 11n - 6.$$

Factorizando el polinomio (por ejemplo, por Ruffini) tenemos:

$$0 \ \leq \ (n-1)(n-2)(n-3),$$

lo cual es cierto para todo $n \geq 3$. Por tanto, para $c = 2$ podemos escoger $n_0 = 3$, y hemos encontrado una pareja de constantes que verifica la definición de $\mathcal{O}$ y podemos afirmar que $n^3 + 6n^2 - 11n + 6 \ \in \ \mathcal{O}(n^3)$.

Ejercicio 2 [2.5 puntos]

La fórmula para considerar todas las operaciones que se llevan a cabo en un bucle de tipo FOR o WHILE es:

$$T_{\text{bucle}} = 1_{\text{inicialización}} + \sum^{n}(1_{\text{comparación}} + T_{\text{cuerpo}} + 1_{\text{incremento}}) + 1_{\text{última comparación}}$$

Utilízala para hallar el número de operaciones ($T(n)$, simplificada) del siguiente código:

```
1  for (int i=1; i<n; i++)
2    for (int j=0; j<2i; j++)
3      procesa(i,j);   // dos operaciones
```

Solución:

En este caso, la función T que mide el número de operaciones (en función de n) será igual a la función asociada al bucle externo:

$$T(n) = 1 + \sum_{i=1}^{n-1}(1 + T_{\text{bucle\_interno}} + 1) + 1,$$

mientras que

$$T_{\text{bucle\_interno}} = 1 + \sum_{j=0}^{2i-1}(1 + T_{\text{cuerpo}} + 1) + 1.$$

Como $T_{\text{cuerpo}} = 2$, por la instrucción `procesa(i,j)` tenemos:

$$T_{\text{bucle\_interno}} = 1 + \sum_{j=0}^{2i-1}(1+2+1) + 1 = 2 + \sum_{j=0}^{2i-1} 4 = 2 + 4(2i) = 2 + 8i.$$

Sustituyendo en $T(n)$ obtenemos:

$$T(n) = 2 + \sum_{i=1}^{n-1}(2+2+8i) = 2 + \sum_{i=1}^{n-1} 4 + 8\sum_{i=1}^{n-1} i.$$

Resolviendo los sumatorios obtenemos:

$$T(n) = 2 + 4(n-1) + 8\frac{(n-1)n}{2}.$$

Finalmente, simplificando:

$$T(n) = 2 + 4n - 4 + 4n^2 - 4n = 4n^2 - 2.$$

Ejercicio 3 [3 puntos]

Resuelve la siguiente relación de recurrencia por el método de **EXPANSIÓN** de recurrencias, e indica su orden:

$$T(n) = \begin{cases} 0 & \text{si } n = 1, \\ 3T(n/3) + n^2 & \text{si } n > 1. \end{cases}$$

Solución:

Procedemos a expandir la recurrencia:

$$\begin{aligned} T(n) &= 3T(n/3) + n^2 \\ &= 3\left[3T(n/9) + \left(\frac{n}{3}\right)^2\right] + n^2 = 9T(n/9) + \frac{n^2}{3} + n^2 \\ &= 9\left[3T(n/27) + \left(\frac{n}{9}\right)^2\right] + \frac{n^2}{3} + n^2 = 27T(n/27) + \frac{n^2}{9} + \frac{n^2}{3} + n^2 \\ &\vdots \\ &= 3^i T(n/3^i) + n^2\left[1 + \frac{1}{3} + \frac{1}{3^2} + \cdots + \frac{1}{3^{i-1}}\right] = 3^i T(n/3^i) + n^2 \sum_{j=0}^{i-1} \left(\frac{1}{3}\right)^j \end{aligned}$$

El sumatorio es una simple serie geométrica:

$$\sum_{j=0}^{i-1} \left(\frac{1}{3}\right)^j = \frac{\frac{1}{3^i} - 1}{\frac{1}{3} - 1} = \frac{\frac{1}{3^i} - \frac{3^i}{3^i}}{-\frac{2}{3}} = \frac{\frac{3^i - 1}{3^i}}{\frac{2}{3}} = \frac{3}{2} \cdot \frac{3^i - 1}{3^i}.$$

Se llega al caso base cuando $n/3^i = 1$. Es decir, cuando $n = 3^i$. Sustituyendo tenemos:

$$T(n) = nT(1) + n^2 \cdot \frac{3}{2} \cdot \frac{n-1}{n} = n \cdot 0 + \frac{3}{2} n(n-1) = \frac{3}{2} n^2 - \frac{3}{2} n \ \in \ \Theta(n^2).$$

Ejercicio 4 [3 puntos]

Resuelve la siguiente relación de recurrencia (incluyendo las constantes) por el método **GENERAL** de resolución de recurrencias, e indica su orden:

$$T(n) = \begin{cases} 0 & \text{si } n = 1, \\ 3T(n/3) + n^2 & \text{si } n > 1. \end{cases}$$

Solución:

Para poder aplicar el método tenemos que hacer el cambio de variable $n = 3^k$. En ese caso podemos reescribir la expresión para $T(n)$ de la siguiente manera:

$$T(n) = T(3^k) = 3T(3^k/3) + (3^k)^2 = 3T(3^{k-1}) + 3^{2k} = 3T(3^{k-1}) + 9^k.$$

A continuación hacemos un cambio de función $t(k) = T(3^k)$:

$$T(n) = T(3^k) = t(k) = 3t(k-1) + 9^k.$$

El polinomio característico de la recurrencia $t(k)$ es:

$$(x-3)(x-9).$$

Por tanto, la recurrencia tiene la siguiente forma:

$$t(k) = C_1 3^k + C_2 9^k.$$

Deshaciendo los cambios tenemos:

$$T(n) = C_1 n + C_2 n^2.$$

Para hallar las constantes necesitamos dos casos base. En primer lugar conocemos $T(1) = 0$ del enunciado. Por otro lado,

$$T(3) = 3T(1) + 3^2 = 9.$$

Por tanto, el sistema de ecuaciones a resolver es:

$$\left.\begin{array}{rcrcrcl} T(1) & = & C_1 & + & C_2 & = & 0 \\ T(3) & = & 3C_1 & + & 9C_2 & = & 9 \end{array}\right\}$$

Las soluciones son $C_1 = -3/2$ y $C_2 = 3/2$. Por tanto:

$$T(n) = \frac{3}{2}n^2 - \frac{3}{2}n \ \in \ \Theta(n^2).$$

Universidad
Rey Juan Carlos

Prueba de evaluación

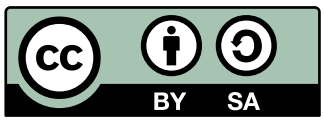
Soluciones examen diseño – mayo 2021-2022

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2021-2022 – 25 de mayo de 2022

Prueba de Diseño de Algoritmos

Duración: 2 horas

Soluciones

Ejercicio 1 [1 punto]

Se pide diseñar un algoritmo **recursivo** que, dada una lista $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ de n números calcule la suma/resta alternante $s = a_0 - a_1 + a_2 - a_3 \ \cdots$.

Posibles soluciones:

Una solución muy concisa consiste en devolver a_0 menos la suma alternante de la lista `a[1:]` en el caso recursivo. Esto es correcto ya que el signo menos afecta a toda la suma de la llamada recursiva.

```
def suma_resta_alternante1(a):
    if len(a)==0:
        return 0
    else:
        return a[0] - suma_resta_alternante1(a[1:])
```

Otra solución consiste en considerar el subproblema de tamaño $n-2$ que calcula la suma alternante de los elementos $a_2, \ldots, a_{n-1}$. El caso recursivo simplemente calcula la resta $a_0 - a_1$ y le suma el resultado del subproblema. Por otro lado, en esta solución se necesitan dos casos base triviales:

```
def suma_resta_alternante2(a):
    if len(a)==0:
        return 0
    elif len(a)==1:
        return a[0]
    else:
        return a[0] - a[1] + suma_resta_alternante2(a[2:])
```

En la siguiente solución consideramos el subproblema con la lista inicial a la que le quitamos el último elemento (`a[:-1]`). Si la longitud de la lista inicial es par el método debe devolver el resultado del subproblema menos el último elemento de la lista (`a[-1]`). En cambio, si es impar debemos sumar el último elemento. El método se puede implementar de la siguiente manera:

```
def suma_resta_alternante3(a):
    if len(a)==0:
        return 0
    elif len(a)%2==0:
        return suma_resta_alternante3(a[:-1]) - a[-1]
    else:
        return suma_resta_alternante3(a[:-1]) + a[-1]
```

Ejercicio 2 [4 puntos]

En este ejercicio se pide diseñar un algoritmo **recursivo** basado en la técnica de **divide y vencerás** que, dada una lista **a** de 2^p números (donde p es un entero no negativo), devuelva una lista que sea una permutación concreta de los elementos de **a**. En particular, la permutación se define invirtiendo los bits de los índices de los elementos, asumiendo que los índices son numeros binarios de p bits.

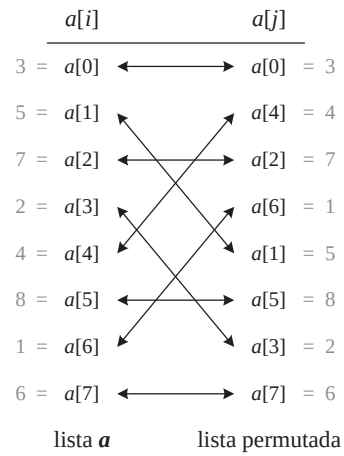
Por ejemplo, para $p = 5$, considérese que un índice i es el número binario $b_4b_3b_2b_1b_0$. Entonces el índice $j = b_0b_1b_2b_3b_4$ sería el correspondiente tras haber invertido el orden de los bits de i. La permutación pedida tendrá que intercambiar los elementos ubicados en los todos los posibles índices i y j. Por ejemplo, para $p = 5$ el elemento $a[26]$ se intercambiará con el elemento $a[11]$ ya que 26 en binario es 11010, mientras que 11 es 01011. Se asumirá por tanto que los índices tomarán los valores desde 0 hasta $2^p - 1$.

El siguiente ejemplo presenta un caso concreto para $p = 3$, con la lista $\mathbf{a} = [3, 5, 7, 2, 4, 8, 1, 6]$:

relación entre *i* y *j*, para *p* = 3

i	b_2	b_1	b_0		b_0	b_1	b_2	*j*
0	0	0	0		0	0	0	0
1	0	0	1		1	0	0	4
2	0	1	0		0	1	0	2
3	0	1	1	⟺	1	1	0	6
4	1	0	0		0	0	1	1
5	1	0	1		1	0	1	5
6	1	1	0		0	1	1	3
7	1	1	1		1	1	1	7

permutación (intercambios) para *p* = 3



Ejemplo: lista ***a*** = [3 , 5 , 7 , 2 , 4 , 8 , 1 , 6] lista permutada = [3 , 4 , 7 , 1 , 5 , 8 , 2 , 6]

- Se podrán usar funciones auxiliares. Todas deberán ser **recursivas**.

Posibles soluciones:

Como en todas las soluciones recursivas, los algoritmos se construyen asumiendo que se conocen las soluciones a subproblemas (instancias más pequeñas del mismo problema). Considérese la lista del enunciado [3,5,7,2,4,8,1,6], para la cual la solución es [3,4,7,1,5,8,2,6]. Dividiéndola por la mitad tenemos las sublistas $[3, 5, 7, 2]$ y $[4, 8, 1, 6]$. Por otro lado, la permutación para una lista genérica $\mathbf{g}$ de longitud 4 ($p = 2$) es $[g_0, g_2, g_1, g_3]$. Por tanto, aplicando la función recursiva a ambas sublistas obtendríamos $\mathbf{a}_1 = [3, 7, 5, 2]$ y $\mathbf{a}_2 = [4, 1, 8, 6]$. Para obtener el resultado final hay que mezclar $\mathbf{a}_1$ con $\mathbf{a}_2$ tomando elementos de ambas listas de manera alternante. El siguiente código muestra la función recursiva basada en divide y vencerás (`invbitperm`) y la función recursiva para mezclar las dos listas:

```
def mezcla(a1,a2):
    if len(a1)==1:
        return [a1[0],a2[0]]
    else:
        return [a1[0]] + [a2[0]] + mezcla(a1[1:],a2[1:])

def invbitperm(a):
    if len(a)<=2:
        return a
    else:
        mitad = len(a)//2
        a1 = invbitperm(a[:mitad])
        a2 = invbitperm(a[mitad:])
        return mezcla(a1,a2)
```

Otra solución consiste en dividir la lista original en dos que contengan los elementos en índices pares e impares. Por ejemplo, podemos descomponer [3,5,7,2,4,8,1,6] en las listas $\mathbf{a}_1 = [3, 7, 4, 1]$ y $\mathbf{a}_2 = [5, 2, 8, 6]$. Aplicando el método recursivo a estas listas obtendríamos $\mathbf{t}_1 = [3, 4, 7, 1]$ y $\mathbf{t}_2 = [5, 8, 2, 6]$. En este caso el resultado final es la simple concatenación de $\mathbf{t}_1$ y $\mathbf{t}_2$: [3,4,7,1,5,8,2,6]. A continuación se muestra esta solución:

```
def pares_impares(a):
    if a==[]:
        return ([],[])
    else:
        (a1,a2) = pares_impares(a[2:])
        return ([a[0]] + a1, [a[1]] + a2)

def invbitperm2(a):
    if len(a)<=2:
        return a
    else:
        (a1,a2) = pares_impares(a)
        return invbitperm2(a1) + invbitperm2(a2)
```

Ejercicio 3 [4 puntos]

Considérese un conjunto C de cardinalidad n (es decir, de n elementos). Por otro lado, sea $\mathbf{a} = [n_1, n_2, \ldots, n_m]$ una lista de m enteros positivos de manera que $n_1 + n_2 + \cdots + n_m = n$. Se pide implementar un algoritmo basado en la técnica de **backtracking** para calcular el número total de particiones de C en m subconjuntos tal que el primero tenga n_1 elementos, el segundo tenga n_2 elementos, etc. En otras palabras, se pide hallar el número de formas de generar m subconjuntos de C, tal que sus cardinalidades sean precisamente $n_1, n_2, \ldots, n_m$.

Por ejemplo, si $C = \{0, 1, 2, 3, 4\}$, donde $n = 5$, y tenemos $\mathbf{a} = [2, 1, 2]$, hay un total de 30 posibles particiones:

$\{0,1\}_1\ \{2\}_2\ \{3,4\}_3$	$\{2,3\}_1\ \{0\}_2\ \{1,4\}_3$
$\{0,1\}_1\ \{3\}_2\ \{2,4\}_3$	$\{2,4\}_1\ \{0\}_2\ \{1,3\}_3$
$\{0,1\}_1\ \{4\}_2\ \{2,3\}_3$	$\{3,4\}_1\ \{0\}_2\ \{1,2\}_3$
$\{0,2\}_1\ \{1\}_2\ \{3,4\}_3$	$\{1,2\}_1\ \{3\}_2\ \{0,4\}_3$
$\{0,3\}_1\ \{1\}_2\ \{2,4\}_3$	$\{1,2\}_1\ \{4\}_2\ \{0,3\}_3$
$\{0,4\}_1\ \{1\}_2\ \{2,3\}_3$	$\{1,3\}_1\ \{2\}_2\ \{0,4\}_3$
$\{0,2\}_1\ \{3\}_2\ \{1,4\}_3$	$\{1,4\}_1\ \{2\}_2\ \{0,3\}_3$
$\{0,2\}_1\ \{4\}_2\ \{1,3\}_3$	$\{1,3\}_1\ \{4\}_2\ \{0,2\}_3$
$\{0,3\}_1\ \{2\}_2\ \{1,4\}_3$	$\{1,4\}_1\ \{3\}_2\ \{0,2\}_3$
$\{0,4\}_1\ \{2\}_2\ \{1,3\}_3$	$\{2,3\}_1\ \{1\}_2\ \{0,4\}_3$
$\{0,3\}_1\ \{4\}_2\ \{1,2\}_3$	$\{2,4\}_1\ \{1\}_2\ \{0,3\}_3$
$\{0,4\}_1\ \{3\}_2\ \{1,2\}_3$	$\{3,4\}_1\ \{1\}_2\ \{0,2\}_3$
$\{1,2\}_1\ \{0\}_2\ \{3,4\}_3$	$\{2,3\}_1\ \{4\}_2\ \{0,1\}_3$
$\{1,3\}_1\ \{0\}_2\ \{2,4\}_3$	$\{2,4\}_1\ \{3\}_2\ \{0,1\}_3$
$\{1,4\}_1\ \{0\}_2\ \{2,3\}_3$	$\{3,4\}_1\ \{2\}_2\ \{0,1\}_3$

Un detalle importante para simplificar el ejercicio es que la partición $\{0,1\}_1\ \{2\}_2\ \{3,4\}_3$ se considera distinta de $\{3,4\}_1\ \{2\}_2\ \{0,1\}_3$.

Por otro lado $\{0,1\}_1\ \{2\}_2\ \{3,4\}_3$ y $\{1,0\}_1\ \{2\}_2\ \{3,4\}_3$ son la misma partición (no importa el orden de los elementos en un subconjunto).

Implementad también una función "*wrapper*" que llame a la función recursiva con los parámetros adecuados.

Finalmente, el método recursivo devolverá un valor numérico y no imprimirá nada.

Posible solución:

En este ejercicio no debemos buscar permutaciones de n elementos ya que $\{0,1\}_1\ \{2\}_2\ \{3,4\}_3$ es equivalente a $\{1,0\}_1\ \{2\}_2\ \{4,3\}_3$. En otras palabras, las permutaciones [0,1,2,3,4], [1,0,2,3,4], [0,1,2,4,3] y [1,0,2,4,3] realmente corresponden a la misma partición. Tampoco debemos buscar subconjuntos. Lo importante en este ejercicio es comprender que cada uno de los n elementos puede incorporarse a uno de los m conjuntos. Por tanto, cuando formemos la solución parcial tenemos que considerar los m candidatos para cada uno de los n elementos. De esta manera, si $\mathbf{a} = [2, 1, 2]$ (y $C = \{0, 1, 2, 3, 4\}$ con $n = 5$), la solución parcial será una lista de n valores entre 0 y $m - 1$. Por ejemplo, una solución válida podría ser $\mathbf{s} = [0, 2, 0, 1, 2]$, que correspondería con la partición $\{0,2\}_1\ \{3\}_2\ \{1,4\}_3$. Como $s_0 = 0$ y $s_2 = 0$ entonces el 0 y el 2 deben ir en el primer conjunto de la partición. El 3 iría en el segundo conjunto, y el 1 y el 4 irían en el tercer conjunto.

Una vez definida la estructura de la solución parcial y los candidatos, debemos considerar restricciones para no incluir un elemento en un conjunto si éste ya está lleno. Para hacer esto de manera eficiente se puede usar la lista $\mathbf{a}$, que indica el número de elementos que se podrían incluir en los m conjuntos. Si incluimos un elemento en el conjunto k-ésimo (donde k toma valores de 0 a $m - 1$), tendremos que decrementar a_k (este cambio tendrá que deshacerse

después de realizar la llamada recursiva). A continuación se muestra esta solución, que devuelve 1 cuando se completa una solución, y retorna la suma de los resultados de las diversas llamadas recursivas:

```
def particiones(i,a,sol):
    n = len(sol)
    m = len(a)
    if i==n:
        return 1
    else:
        nsols = 0
        for k in range(m):
            if a[k]>0:
                sol[i] = k
                a[k] = a[k]-1
                nsols = nsols + particiones(i+1,a,sol)
                a[k] = a[k]+1

        return nsols


a = [2,1,2]
n = sum(a)
m = len(a)
sol = [None]*n
print(particiones(0,a,sol))
```

Un detalle adicional es que como no se pide imprimir o procesar las particiones (solo se pide contarlas), no es necesario usar la solución parcial `sol`. Por tanto, la siguiente solución también sería válida, donde podemos aprovechar que la profundidad del árbol recursivo será n, la cual podemos decrementar hasta que sea 0 (no sería necesario usar un índice i para saber la profundidad de un nodo del árbol recursivo):

```
def particiones_simple(n,a):
    m = len(a)
    if n==0:
        return 1
    else:
        nsols = 0
        for k in range(m):
            if a[k]>0:
                a[k] = a[k]-1
                nsols = nsols + particiones_simple(n-1,a)
                a[k] = a[k]+1

        return nsols

a = [2,1,2]
n = sum(a)
print(particiones_simple(n,a))
```

Ejercicio 4 [1 punto]

Se pide diseñar un (solo) método **recursivo** que implemente la estrategia voraz óptima para el problema del cambio de monedas. Dada una cierta cantidad de dinero x, el algoritmo debe imprimir por pantalla los valores de las monedas tal que su suma sea igual a x, teniendo en cuenta que:

- Los posibles valores de las monedas son: $\{2,\ 1,\ 0{,}5,\ 0{,}2,\ 0{,}1,\ 0{,}05,\ 0{,}02,\ 0{,}01\}$
- Se usará el mínimo número de monedas posible
- Se asumirá que $x \geq 0{,}01$ y que x estará redondeado a dos cifras decimales

Por ejemplo, para $x = 5{,}34$ el método debe imprimir la secuencia:

$$2\ \ 2\ \ 1\ \ 0{,}2\ \ 0{,}1\ \ 0{,}02\ \ 0{,}02$$

El método será recursivo, pero podrá usar bucles.

Posibles soluciones:

El método debe recibir la cantidad x y una lista de monedas que en este caso asumiremos que está ordenada de mayor a menor. El caso base corresponde a la situación en la que $x = 0$, en cuyo caso no habría que imprimir nada. En el caso recursivo primero determinamos qué moneda debemos "devolver" (es decir, imprimir), que será la de mayor valor que no sea inferior a x. Esto lo podemos calcular mediante un simple bucle (también se puede usar una función recursiva alternativa). Asumiendo que se ha imprimido la moneda i-ésima (llamémosla m_i) el algoritmo debe continuar resolviendo el problema para $x - m_i$ haciendo una nueva llamada recursiva. En el siguiente código se redondea la cantidad $x - m_i$ por cuestiones de precisión a la hora de trabajar con números reales (esto no se tendrá en cuenta a la hora de corregir el ejercicio):

```
1  def devolucion_cambio(x,monedas):
2      if x>0:
3          i = 0
4          while x<monedas[i]:
5              i = i+1
6
7          print(monedas[i],' ',end='')
8
9          c = round(x-monedas[i],2)
10         devolucion_cambio(c,monedas)
11
12 monedas = [2,1,0.5,0.2,0.1,0.05,0.02,0.01]
13 x = 5.34
14 devolucion_cambio(round(x,2),monedas)
```

Se puede sustituir el código de las líneas 3–5 por la llamada `i = indice_moneda(x,monedas)`, donde:

```
1  # Se asume que x no puede ser menor que el valor más pequeño de la lista de
2  # monedas, por lo que la lista de monedas nunca será vacía
3  def indice_moneda(x,monedas):
4      if x>=monedas[0]:
5          return 0
6      else:
7          return 1 + indice_moneda(x,monedas[1:])
```

Otra solución consiste en usar un bucle para imprimir la moneda de mayor valor todas las veces que haga falta usando un bucle o una función auxiliar. Posteriormente se realiza una llamada recursiva para resolver el problema con la cantidad que quede por devolver, pero eliminando la moneda de mayor valor de la lista de monedas:

```
def cambio_monedas(x,monedas):
    if x>0:
        while x>=monedas[0]:
            print(monedas[0],' ',end='')
            x = round(x-monedas[0],2)

        cambio_monedas(x,monedas[1:])
```

Universidad
Rey Juan Carlos

Prueba de evaluación

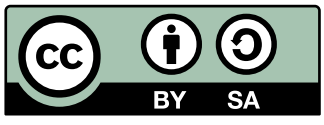
Soluciones examen diseño – mayo 2022-2023

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

17 de diciembre de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos – Grado en Ingeniería Informática

Curso 2022-2023 – 25 de mayo de 2023

Prueba de Diseño de Algoritmos

2 horas

Soluciones

Ejercicio 1 [1 punto]

Se pide diseñar una función **recursiva** que, dado un número entero n no negativo, devuelva el número de dígitos impares de n. Por ejemplo, para $n = 727368238$ la función devolverá 4.

Posible solución:

```
def digitos_impares(n):
    if n<10:
        return n%2
    else:
        return digitos_impares(n//10) + n%2
```

El tamaño del problema es el número de dígitos de n. Por tanto, el caso base ocurre cuando n solo contiene un dígito ($n < 10$). Si el dígito es impar se devolverá un 1 y si es par un 0. Por tanto, simplemente se puede devolver `n%2`. En el caso recursivo la función devolverá el número de dígitos impares de n ignorando su dígito menos significativo (a través de la llamada recursiva `digitos_impares(n//10)`), más 1 en caso de que el dígito menos significativo sea impar (a través de `n%2`).

Ejercicio 2 [2 puntos]

Se pide diseñar un algoritmo **recursivo** basado en la técnica de **divide y vencerás** que, dada una lista **a** de 2^p números (donde p es un entero no negativo), devuelva una lista de la misma longitud, que contenga en su primera mitad los elementos de **a** ubicados en los índices pares, y en su segunda mitad los ubicados en índices impares. Además se preservará el orden de los elementos de **a** en ambas mitades. Por ejemplo, para $\mathbf{a} = [3, 8, 1, 2, 4, 5, 7, 6]$ la salida debe ser $[3, 1, 4, 7, 8, 2, 5, 6]$.

Posible solución:

```
def particiona(a):
    if len(a)<=2:
        return a
    else:
        mitad = len(a)//2
        b = particiona(a[:mitad])
        c = particiona(a[mitad:])
        m = mitad//2
        return b[:m] + c[:m] + b[m:] + c[m:]
```

El tamaño de este problema es la longitud de la lista (determinada por p). Si esta longitud es 1 o 2 la lista a devolver sería la misma que la de entrada. Este sería el caso base. En el caso recursivo se puede dividir la lista de entrada por la mitad. Para el ejemplo del enunciado tendríamos las listas [3,8,1,2] y [4,5,7,6]. Resolviendo el problema para ambas (es decir, realizando llamadas recursivas para estas sublistas) tendríamos $\mathbf{b} = [3, 1, 8, 2]$ para la primera mitad, y $\mathbf{c} = [4, 7, 5, 6]$ para la segunda. Para obtener la solución $[3, 1, 4, 7, 8, 2, 5, 6]$ debemos concatenar la primera mitad de **b** con la primera mitad de **c**, y a continuación la segunda mitad de **b** con la segunda mitad de **c**.

Ejercicio 3 [**4 puntos**]

Considérese una expresión matemática con n parejas de paréntesis – uno abierto y otro cerrado. Se pide implementar un algoritmo basado en la técnica de **backtracking** que imprima por pantalla todas las secuencias válidas que se puedan formar con n parejas de paréntesis. Además la función deberá retornar el número total de estas secuencias válidas. Por ejemplo, para $n = 3$, hay 5 secuencias válidas:

((()))
(()())
(())()
()(())
()()()

Posible solución:

```
def parentesis(i,n_abiertos,sol,n):
    if i==2*n:  # Solución completa y válida
        # Imprimir la solución
        for j in range(2*n):
            if sol[j]==0:
                print('(',end='')
            else:
                print(')',end='')
        print()
        return 1
    else:
        nsols = 0
        # Genera candidatos (0: paréntesis abierto, 1: paréntesis cerrado)
        for k in range(0,2):

            # número de paréntesis abiertos
            n_abiertos = n_abiertos + (1-k)

            # Comprueba validez (i-n_abiertos+1 es el número de paréntesis
                cerrados)
            if n_abiertos>=i-n_abiertos+1 and n_abiertos<=n:

                # Expande la solución parcial
                sol[i]=k

                # Intenta seguir expandiendo la solución parcial
                nsols = nsols + parentesis(i+1,n_abiertos,sol,n)

            n_abiertos = n_abiertos - (1-k)

        return nsols


def parentesis_wrapper(n):
    sol = [None] * (2*n)
    return parentesis(0,0,sol,n)
```

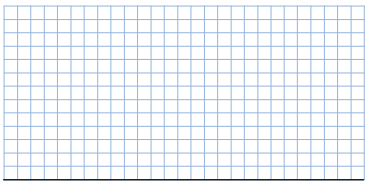
La solución parcial en este problema es una lista de tamaño $2n$, que debe contener dos tipos de candidatos para indicar un paréntesis abierto o cerrado. En esta solución un 0 indicará un paréntesis abierto y un 1 uno cerrado. El parámetro `i` indicará el índice de la solución parcial donde ubicaremos un nuevo candidato. Por tanto, si

`i` es $2n$ la solución parcial estará completa y podremos proceder a imprimirla y a retornar un 1, indicando que hemos encontrado una solución válida.

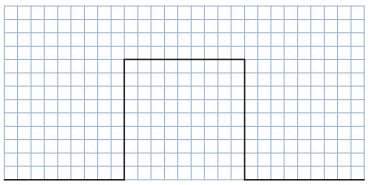
En el caso recursivo usamos un bucle para generar los dos candidatos. Además, la función puede llevar la cuenta del número de paréntesis abiertos (`n_abiertos`) y cerrados (o su diferencia), para poder determinar la validez de un candidato en $\mathcal{O}(1)$. Habiendo actualizado `n_abiertos`, el número de paréntesis cerrados es `i-n_abiertos+1` (también se puede usar un parámetro que lleve la cuenta del número de paréntesis cerrados). La condición de validez es que el número de paréntesis abiertos sea mayor o igual que el número de paréntesis cerrados. Además, el número de paréntesis abiertos no puede ser mayor que n. Si el candidato es válido se actualiza la solución parcial y se realiza una nueva llamada recursiva para intentar expandir la solución parcial (si estuviese completa se llegaría al caso base). Por último, se deshace la modificación realizada sobre el número de paréntesis abiertos, y se retorna el número de soluciones halladas, que se actualiza al hacer la llamada recursiva.

Ejercicio 4 [3 puntos]

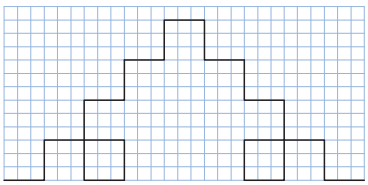
Se pide implementar una función **recursiva** para dibujar una variante del fractal conocido como la curva de Koch, de orden n. La curva de orden 0 es un simple segmento, mientras que la de orden 1 está compuesto por cinco segmentos de un tercio de la longitud del segmento de orden 0, como se ilustra en la siguiente figura.



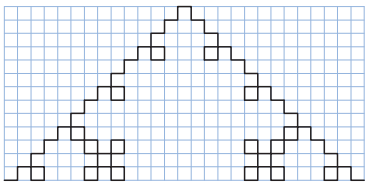
Orden 0



Orden 1



Orden 2



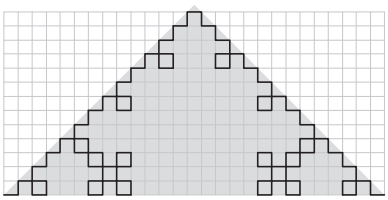
Orden 3

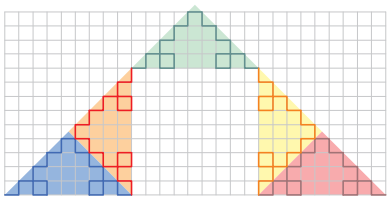
Para obtener sucesivos fractales de orden superior n se descompondría cada segmento del fractal de orden $n-1$ en otros 5 segmentos de manera análoga.

Para dibujar un segmento se usará una función `dibuja(p,q)` que tomará como parámetros los extremos (puntos en el plano) del segmento `p` y `q`. No hay que implementar esta función. Se recomienda usar vectores y matrices, como los que se pueden crear con la librería `numpy` de Python.

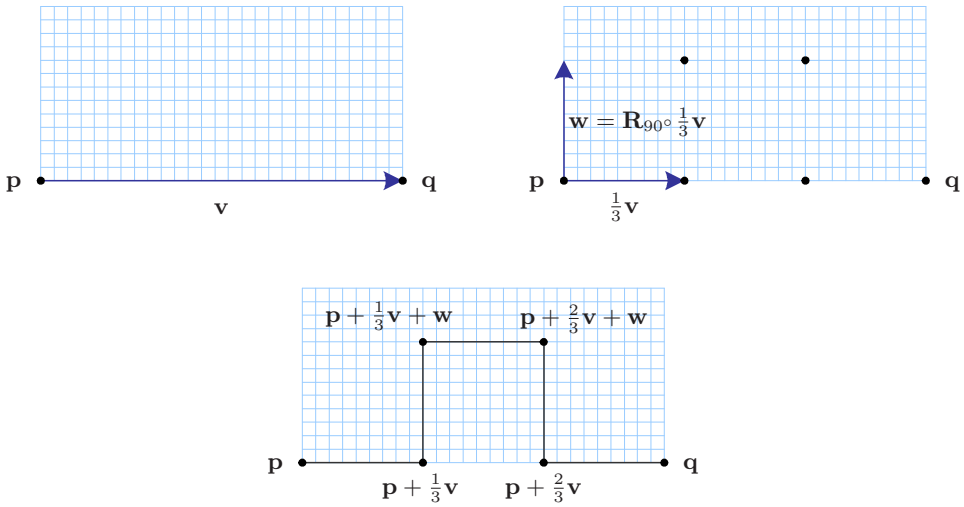
Posible solución:

Este problema es análogo al de la curva de Koch. La clave es comprender que un fractal de orden n se compone de 5 fractales de orden $n-1$, como ilustra la siguiente figura:





Por tanto, en el caso recursivo la función tendrá que hacer 5 llamadas recursivas a fractales de orden $n-1$, donde tendremos que elegir los puntos de los extremos de cada subfractal adecuadamente. La siguiente figura ilustra estos puntos, donde se hace uso de una matriz de rotación $\mathbf{R}_{90^\circ}$ para calcularlos (matriz que, si se multiplica por un vector, lo rota 90 grados en sentido contrario a las agujas del reloj), junto con los extremos iniciales $\mathbf{p}$ y $\mathbf{q}$.



En concreto:

$$\mathbf{R}_{90^\circ} = \begin{bmatrix} \cos(90^\circ) & -\sin(90^\circ) \\ \sin(90^\circ) & \cos(90^\circ) \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Como en el caso base (si el orden es 0) solo es necesario dibujar un segmento entre **p** y **q**, el fractal se puede implementar de la siguiente manera (el ejercicio solo pide implementar la función `fractal`):

```python
import matplotlib.pyplot as plt
import numpy as np

def dibuja(p,q):
    plt.plot([p[0,0],q[0,0]],[p[1,0],q[1,0]],'k-')

def fractal(p, q, n, R_90):
    if n == 0:   # Caso base
        dibuja(p,q)
    else:
        v = q-p
        fractal(p, p+v/3, n-1, R_90)
        x = p+v/3+R_90*v/3
        fractal(p+v/3, x, n-1, R_90)
        fractal(x, x+v/3, n-1, R_90)
        fractal(x+v/3, p+2*v/3, n-1, R_90)
        fractal(p+2*v/3, q, n-1, R_90)


# Matriz de rotación (90 grados en sentido contrario a las agujas del reloj)
R_90 = np.matrix([[0, -1], [1, 0]])

# Dos puntos iniciales
p = np.array([[0], [0]])
q = np.array([[1], [0]])

# Orden
n = 4

fig1 = plt.figure()
fractal(p, q, n, R_90)
plt.axis('equal')
plt.axis('off')
plt.show()
```

Exámenes sin soluciones

Universidad
Rey Juan Carlos

Prueba de evaluación

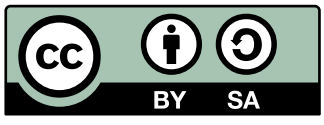
Examen diseño – junio 2014-2015

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

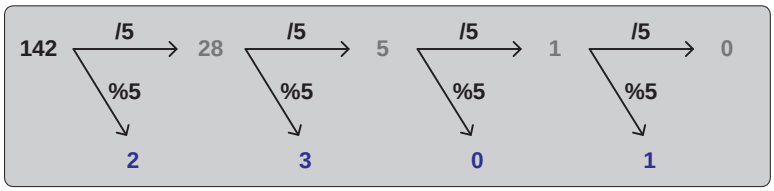
Curso 2014-2015 – 15 de junio de 2015

Prueba de Diseño de Algoritmos

Valor: 35 % de la nota final. Duración: **2 horas**

Ejercicio 1 [3 puntos]

Un mismo número se puede representar en diferentes "bases". Nosotros solemos usar números decimales (es decir, en base 10), donde los dígitos representan unidades, decenas, centenas, etc. Por ejemplo, el número 1473 se puede descomponer en $1 \cdot 10^3 + 4 \cdot 10^2 + 7 \cdot 10^1 + 3 \cdot 10^0$. De esta manera, los dígitos del número se multiplican por potencias de 10. Ahora bien, podemos representar ese número (1473) en otra base, obteniendo otros dígitos multiplicados por potencias de esa base. Por ejemplo, en base 5, $1473 = 2 \cdot 5^4 + 1 \cdot 5^3 + 3 \cdot 5^2 + 4 \cdot 5^1 + 3 \cdot 5^0$. En ese caso podemos escribir: $1473_{10} = 21343_5$, donde el subíndice denota la base.



La figura de arriba ilustra los pasos de un algoritmo que transformaría 142_{10} a 1032_5, donde '/' es división entera y '%' es el resto de la división entera. Se pide implementar una función recursiva que implemente el algoritmo, el cual calcula el cambio de base de un cierto número decimal n (entero), a su representación en una base entera $b \in [2, 9]$. Se trata de calcular el nuevo entero (no se pide imprimirlo por pantalla). Es decir, la función devolvería el entero 1032 para las entradas $n = 142$ y $b = 5$.

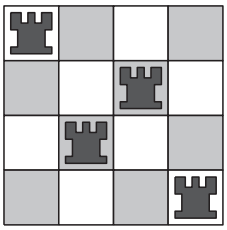
Ejercicio 2 [3.5 puntos]

Se pide implementar una función recursiva que calcule el producto de dos matrices $A \cdot B$ utilizando la técnica de **divide y vencerás**. La descomposición debe realizarse partiendo el problema en los siguientes bloques:

$$A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Se deben implementar las funciones auxiliares que se empleen (por ejemplo, la suma de matrices). Las matrices no tienen por qué ser cuadradas.

Ejercicio 3 [3.5 puntos]



Uno de los problemas más conocidos que se resuelven mediante la técnica de *backtracking* es el de las n reinas de ajedrez. En este ejercicio se pide resolver el problema de las n torres, también usando *backtracking*. Las torres se pueden mover en horizontal y vertical, con lo cual se trata de ubicar n torres en un tablero cuadrado de $n \times n$ casillas, de manera que solo haya una en cada fila y en cada columna. En concreto se pide escribir un programa que calcule todas las posibles soluciones y las imprima por pantalla.

Universidad
Rey Juan Carlos

Prueba de evaluación

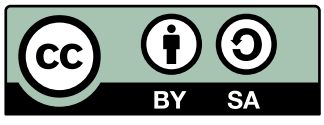
Examen diseño – mayo 2015-2016

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2015-2016 – 4 de mayo de 2016

Prueba de Diseño de Algoritmos

Valor: 35 % de la nota final. Duración: **2 horas**

Ejercicio 1 ¿Dígito impar? [2 puntos]

Se pide diseñar e implementar un algoritmo recursivo en java para determinar si un número entero no negativo contiene un dígito impar. Además, se debe especificar:

(a) El tamaño del problema

(b) El/los casos base

(c) El/los subproblemas de menor tamaño similares al original que se usarán para construir los casos recursivos

(d) El/los casos recursivos, junto con un diagrama que indique el razonamiento seguido para hallar dichos casos recursivos

---

Ejercicio 2 Punto de inflexión [3 puntos]

Sea $\mathbf{v}$ un vector de n números enteros, el cual está organizado de manera que los números pares aparecen antes que los impares. Es decir, el índice de cualquier número par será menor que el índice de cualquier número impar. Por ejemplo:

$$\mathbf{v} = [2, -4, 10, 8, 0, \mathbf{12}, 9, 3, -15, 3, 1].$$

Se pide implementar un algoritmo recursivo eficiente en java, que se ejecute en $\Theta(\log n)$, para determinar el mayor de los índices de números pares. Es decir, aquel índice i para el cual v_i es par, pero donde todos los elementos v_j serán impares para todo $j > i$. Se considera que el índice del primer elemento es 0. Por tanto, en el ejemplo el resultado sería $i = 5$, ya que $v_5 = 12$ (par), mientras que $v_6 = 9$ (impar). Si el vector $\mathbf{v}$ no contiene números pares entonces la función recursiva devolverá -1.

Ejercicio 3 Camino de longitud mínima por un laberinto [1 punto]

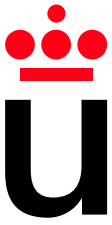
Dado un laberinto, se desea encontrar un camino que no solo sea una solución, sino que además sea de longitud más corta. Este problema se puede resolver con la técnica de *backtracking*, pero en este ejercicio se pide una solución mediante un algoritmo **voraz**. La solución consiste en aplicar una estrategia de "transforma y vencerás", para poder aplicar uno de los algoritmos voraces vistos en clase.

Se pide describir cómo se puede resolver el problema concreto (cómo se transformaría el problema, y qué algoritmo voraz se aplicaría). No hay que escribir código. La descripción será muy breve: 5 líneas como máximo.

Ejercicio 4 Permutación optima [4 puntos]

En este problema se trata de implementar una función recursiva en java, basada en la técnica de *backtracking*, para hallar una permutación óptima de n elementos distintos (por ejemplo, enteros desde 0 hasta $n-1$), de acuerdo a una determinada función objetivo, que hay que minimizar. Dicha función no solo es capaz de devolver un valor para una permutación completa de n elementos, sino que también puede retornar el valor asociado a una permutación (solución) parcial de $m < n$ elementos. En concreto, la función es $f(m, \mathbf{p}, \mathbf{w})$, donde $m \leq n$, y evalúa únicamente los primeros m elementos de la permutación $\mathbf{p}$. Además, usa un vector de n pesos $\mathbf{w}$, que está fijado de antemano, pero se pasa igualmente como parámetro a la función. En este problema $f(m, \mathbf{p}, \mathbf{w}) \leq f(k, \mathbf{p}, \mathbf{w})$ si $m < k$ (es decir, la función objetivo no puede disminuir al ampliar una solución parcial).

La búsqueda debe tener en cuenta el mejor valor hallado hasta el momento para no proseguir en caso de que no pueda mejorarse una solución parcial con $m \leq n$ elementos. Se asume que $f(m, \mathbf{p}, \mathbf{w})$ ya está implementada, y se puede invocar mediante la instrucción `funcionObjetivo(m,p,w)`. Aunque varias permutaciones diferentes podrían conseguir el menor valor para f, solo se pide hallar una (cualquiera) de ellas. El valor máximo de $f(m, \mathbf{p}, \mathbf{w})$ será m^3. Finalmente, la función recursiva deberá recibir como parámetros de entrada toda la información necesaria para hallar la mejor permutación (no podrán usarse variables globales o no locales a la función).



Universidad
Rey Juan Carlos

Prueba de evaluación

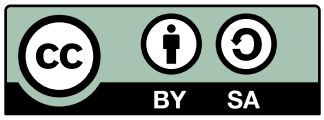
Examen análisis – junio 2016-2017

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2016-2017 – Junio

Prueba de Análisis de Algoritmos

Valor: 30 % de la nota final. Duración: **1 hora**

Ejercicio 1 [2 puntos]

Determina **mediante la definición** de $\mathcal{O}$ si:

$$2^{\ln(n)} \in \mathcal{O}(n)$$

Ejercicio 2 [4 puntos]

La fórmula para considerar todas las operaciones que se llevan a cabo en un bucle de tipo FOR o WHILE es:

$$T_{\text{bucle}} = 1_{\text{inicialización}} + \sum^{n}(1_{\text{comparación}} + T_{\text{cuerpo}} + 1_{\text{incremento}}) + 1_{\text{última comparación}}$$

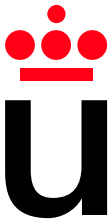
Utilízala para hallar el número de operaciones ($T(n)$, simplificada) del siguiente código en el **PEOR** caso:

```
1  for (int i=0; i<n; i++)
2    if (condicion(i))    // una operación
3      for (int j=i; j<=n; j++)
4        for (int k=0; k<j; k++)
5          procesa(i,j,k);  // dos operaciones
```

Ejercicio 3 [4 puntos]

Resuelve la siguiente relación de recurrencia (se puede resolver mediante cualquiera de los dos métodos vistos en clase):

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ T(n-1) + 2^n + 1 & \text{si } n > 0 \end{cases}$$



Universidad
Rey Juan Carlos

Prueba de evaluación

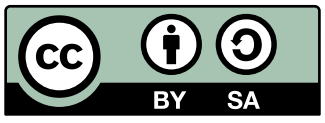
Examen diseño – junio 2016-2017

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2016-2017 – Junio

Prueba de Diseño de Algoritmos

Valor: 35 % de la nota final. Duración: **1 hora y 45 minutos**

Ejercicio 1 Dígitos comunes [4 puntos]

Sea **a** un array o lista de números naturales. Se quiere hallar el conjunto de dígitos comunes a todos los elementos de **a**. Por ejemplo, para $\mathbf{a} = [2348, 1349, 7523, 3215]$, la solución es $\{3\}$.

Para representar el conjunto de dígitos del resultado se usará un array o lista de booleanos de tamaño 10, que expresará si un elemento es común o no. De esta manera, la solución del ejemplo anterior se representará con el array o lista [false, false, false, true, false, false, false, false, false, false].

Se pide implementar un algoritmo basado en la técnica de **divide y vencerás** que resuelva el problema.

Ejercicio 2 [2 puntos]

Sea un vector **v** formado por un número n par de enteros. Hay que agrupar a los elementos de **v** en $n/2$ parejas de forma que si se halla la suma de los dos miembros de cada pareja y se toma el máximo de estas sumas, el número resultante sea el menor posible.

Por ejemplo, dado el vector $\mathbf{v} = [-1, 5, 12, 7, 0, 2, -3, 9]$, una partición óptima de **v** es el conjunto de pares {(0,7),(-1,9),(2,5),(-3,12)}, donde la suma máxima de sus pares es 9, del par $(-3, 12)$. Puede comprobarse que cualquier otra partición produce una suma máxima mayor o igual que 9.

Se pide describir una **estrategia voraz** que obtenga una solución óptima al problema propuesto. En este ejercicio no se pide código.

Ejercicio 3 [4 puntos]

Sea una lista S de números enteros. Se desea hallar todas las formas de dividir S en dos listas A y B, tal que las sumas de sus elementos sean iguales. Por ejemplo, dado $S = [1, 2, 3, 4, 5, 9]$, dos posibles particiones son $(A = [1, 2, 4, 5], B = [3, 9])$ y $(A = [1, 2, 9], B = [3, 4, 5])$. Se pide implementar un algoritmo **eficiente** (que realice podas en el árbol de recursión) para resolver el problema basado en la estrategia de **backtracking**. Para ello se deberá partir de alguno de los esquemas explicados en las transparencias para generar subconjuntos. El algoritmo imprimirá por pantalla las parejas de listas cada vez que las encuentre.

Universidad
Rey Juan Carlos

Prueba de evaluación

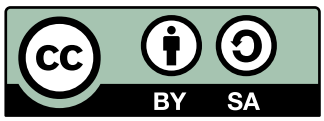
Examen análisis – mayo 2016-2017

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2016-2017 – 3 de marzo de 2017

Prueba de Análisis de Algoritmos

Valor: 30 % de la nota final. Duración: **1 hora y media**

Ejercicio 1 [2 puntos]

Demostrar **mediante las definiciones** de complejidad asintótica si se verifican:

a) ¿ $(n+1)! \in \mathcal{O}(n!)$? **[1 punto]**

b) ¿ $2^n \in \mathcal{O}(3^n)$? **[1 punto]**

Ejercicio 2 [3 puntos]

Expresa los siguientes sumatorios mediante fórmulas que no incluyan los sumatorios ($\sum$). Ambas expresiones resultan ser iguales:

a) **[1 punto]**

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

b) **[1 punto]**

$$\sum_{i=0}^{n-1}(n-i)2^i$$

Ejercicio 3 [3 puntos]

Halla una expresión no recursiva para la siguiente recurrencia por el método de **expansión de recurrencias**:

$$T(n) = 2T(n/2) + n\log_2 n$$

donde n es una potencia de 2 ($n = 2^k$, para $k = 1, 2, \ldots$), y donde $T(1) = 1$. Indica además el orden de $T(n)$.

Ejercicio 4 [3 puntos]

Halla una expresión no recursiva para la siguiente recurrencia por el método **general de resolución recurrencias**:

$$T(n) = 2T(n/2) + n\log_2 n$$

donde n es una potencia de 2 ($n = 2^k$, para $k = 1, 2, \ldots$), y donde $T(1) = 1$. Indica además el orden de $T(n)$.

Universidad
Rey Juan Carlos

Prueba de evaluación

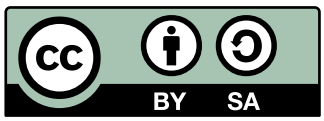
Examen análisis – enero 2020-2021

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2020-2021 – 23 de noviembre de 2020

Prueba de Análisis de Algoritmos

Ejercicio 1 [2 puntos]

Demuestra utilizando la definición formal de Θ si se verifica:

$$2^{\log_3(n)} \in \Theta(n)$$

Ejercicio 2 [2 puntos]

La fórmula para considerar todas las operaciones que se llevan a cabo en un bucle de tipo FOR o WHILE es:

$$T_{\text{bucle}} = 1_{\text{inicialización}} + \sum^{n}(1_{\text{comparación}} + T_{\text{cuerpo}} + 1_{\text{incremento}}) + 1_{\text{última comparación}}$$

Utilízala para hallar el número de operaciones ($T(n)$, simplificada) del siguiente código:

```
1  for (int i=0; i<n; i++)
2      for (int j=n; j>i; j--)
3          procesa(i,j);  // dos operaciones
```

Ejercicio 3 [6 puntos]

Resuelve la siguiente relación de recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ 2T(n/4) + \sqrt{n} & \text{si } n > 1, \end{cases}$$

por el:

(a) Método general de resolución de recurrencias [**3 puntos**]
(b) Método de expansión de recurrencias [**3 puntos**]

Además, indica el orden de complejidad de T.

Universidad
Rey Juan Carlos

Prueba de evaluación

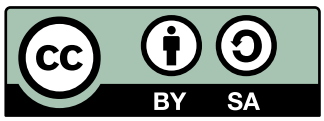
Examen diseño – enero 2020-2021

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2020-2021 – 25 de enero de 2021

Prueba de Diseño de Algoritmos

Ejercicio 1 [2 puntos]

Se pide implementar un método recursivo que, dada una lista **a** de números, devuelva otra cuyos elementos sean la suma de elementos consecutivos de **a**. Por ejemplo, para $\mathbf{a} = [1, 5, 10, -2, 1]$ el método devolverá la lista [6,15,8,-1]. Si la lista de entrada tiene menos de dos elementos el método deberá devolver una lista vacía.

Ejercicio 2 [3 puntos]

Sea **a** un vector ordenado de n enteros (pueden ser negativos) todos *distintos*. Se pide escribir un método recursivo de complejidad $\Theta(\log n)$, capaz de encontrar un índice i tal que $a[i] = i$, suponiendo que tal índice i exista (donde $0 \leq i \leq n-1$). El método devolverá un valor i en caso de que exista, y -1 en caso contrario. Por ejemplo, para $\mathbf{a} = [-3, -1, 0, 2, 4, 5, 13]$, el método deberá devolver 4 o 5 (en este ejemplo $a[4] = 4$ y $a[5] = 5$, así que el método podrá devolver cualquiera de los dos valores).

Ejercicio 3 [5 puntos]

Se pide implementar un algoritmo empleando la técnica de **backtracking** para hallar una solución a una variante del problema de la mochila, así como el método que llamaría a dicha función recursiva.

En particular, se tiene un conjunto de n objetos, cada uno con un peso p_i, para $i = 0, \ldots, n-1$, y una mochila con capacidad C (el peso máximo que puede soportar la mochila). El algoritmo debe buscar un subconjunto de objetos que pueden introducirse en la mochila sin sobrepasar su capacidad C, y de manera que se maximice el peso de los objetos que se introducen en ella. Para obtener la máxima nota el algoritmo debe ser eficiente (usad variables auxiliares para evitar repetir cálculos, y podad el árbol recursivo en cuanto se pase de la capacidad de la mochila o no se pueda mejorar una solución hallada en un paso anterior).

Universidad
Rey Juan Carlos

Prueba de evaluación

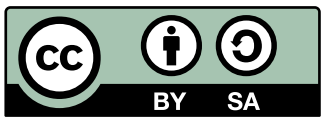
Examen análisis – junio 2020-2021

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2020-2021

Prueba de Análisis de Algoritmos - Convocatoria de junio

30 de junio de 2021

Duración: 40 minutos

Ejercicio 1 [2 puntos]

Demuestra utilizando la definición formal de $\mathcal{O}$ si se verifica:

$$n^{1,5} \in \mathcal{O}(n^2)$$

Ejercicio 2 [2 puntos]

Usando la técnica de aproximación por integrales, halla la cota superior de:

$$\sum_{i=1}^{n} i \cdot \ln(i)$$

sabiendo que:

$$\int x \ln(x)\, dx = \frac{x^2}{2}\left(\ln(x) - \frac{1}{2}\right)$$

Ejercicio 3 [6 puntos]

Resuelve la siguiente relación de recurrencia:

$$T(n) = \begin{cases} 0 & \text{si } n = 0, \\ T(n-1) + n^2 & \text{si } n > 0, \end{cases}$$

por el método que consideréis oportuno (general o expansión de recurrencias).

Universidad
Rey Juan Carlos

Prueba de evaluación

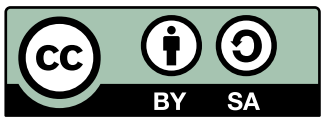
Examen diseño – junio 2020-2021

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

10 de enero de 2023

Autor: Manuel Rubio Sánchez



Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos

Curso 2020-2021

Prueba de Diseño de Algoritmos - Convocatoria de junio

30 de junio de 2021

Duración: 1 hora y 15 minutos

Ejercicio 1 [3 puntos]

Se pide implementar un algoritmo **recursivo** para calcular el determinante de una matriz cuadrada $\mathbf{A}$ de dimensiones $n \times n$, cuyos elementos son $a_{i,j}$, por el método de los adjuntos.

El determinante a calcular se puede definir como:

$$|\mathbf{A}| = \sum_{i=0}^{n-1} (-1)^i a_{i,0} \cdot |\mathbf{M}_{i,0}|$$

donde $\mathbf{M}_{i,j}$ es la matriz complementaria de $a_{i,j}$, es decir, la resultante de eliminar la fila i y la columna j de la matriz $\mathbf{A}$. Nota: en esta fórmula se consideran los matrices complementarias de la primera columna ($j = 0$). También se puede hallar el determinante considerando las de otras filas o columnas, pero implementad el algoritmo según la fórmula descrita.

El algoritmo usará matrices de la librería `numpy`. Para implementar el algoritmo solo necesitaréis lo siguiente:

- Se asume que habéis importado `numpy` mediante: `import numpy as np`
- `A[i][j]` representa al elemento $a_{i,j}$
- `len(A)` es el número de filas de `A`
- `B = np.copy(A)` realiza una copia de `A` y la guarda en `B`
- `B = np.delete(B,f,0)` elimina la fila `f` de la matriz `B`
- `B = np.delete(B,0,1)` elimina la columna `0` de la matriz `B`

Finalmente, el método puede contener bucles, pero debe llamarse a sí mismo necesariamente ya que debe ser recursivo.

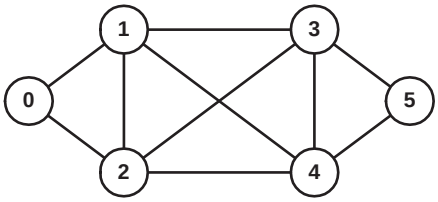
Ejercicio 2 [7 puntos]

Sea un grafo G no dirigido, y no ponderado, compuesto por n vértices. El grafo G estará definido mediante una matriz de adyacencia simétrica y cuadrada $\mathbf{A}$ de tamaño $n \times n$, cuyos elementos serán 0 o 1. Si el elemento $a_{i,j} = 0$ no habrá una arista conectando a los vértices i y j, mientras que si $a_{i,j} = 1$ sí que habrá una arista conectándolos y serán adyacentes. Se asumirá que no habrá aristas desde un vértice a sí mismo (es decir, $a_{i,i} = 0$ para todo i).

Se pide implementar un algoritmo recursivo basado en la técnica de *backtracking* para encontrar un ciclo Euleriano en un grafo. Un ciclo Euleriano es un camino que pasa por cada arista una y solo una vez, y comienza y termina en el mismo vértice. Se puede entender como una secuencia de los vértices del grafo, en la que vértices contiguos en la secuencia deben estar conectados mediante una arista en el grafo, y donde el primer vértice debe ser igual al último.

La siguiente figura ilustra un ejemplo en el que una posible solución es la secuencia:

$$\langle 0, 1, 2, 3, 1, 4, 3, 5, 4, 2, 0 \rangle$$



$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

El algoritmo debe parar en cuanto encuentre un ciclo Euleriano, e imprimir la secuencia de vértices que lo compone, en caso de existir. Se puede asumir que la secuencia empezará en el vértice 0. Se valorará la eficiencia del algoritmo desarrollado.

Ejercicios adicionales

1. Ejercicios de complejidad computacional

Ejercicio 1.1 Demuestra la siguiente identidad:

$$\left(\frac{a}{b^k}\right)^{\log_b n} = \frac{n^{\log_b a}}{n^k}$$

Ejercicio 1.2 Muestra que $\log n! \in \Theta(n \log n)$ usando límites. Pista: cuando n tiende a infinito $n!$ se puede sustituir por la "aproximación de Stirling":

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Ejercicio 1.3 Demuestra que $n \log n \in \mathcal{O}(n^{1+a})$, donde $a > 0$. Usa límites y la regla de L'Hopital.

Ejercicio 1.4 Sean m y n un par de números enteros. Determina el valor del siguiente sumatorio:

$$\sum_{i=m}^{n} 1.$$

Ejercicio 1.5 Escribe la suma de los primeros n enteros impares usando un sumatorio, y simplifícala (es decir, indica una expresión equivalente que no contenga un sumatorio). El resultado es un polinomio sencillo.

Ejercicio 1.6 Usa la siguiente identidad:

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

para hallar una expresión de los primeros n enteros al cuadrado ($1^2 + 2^2 + \cdots + n^2$).

Ejercicio 1.7 Demuestra que una suma parcial general de n términos de una secuencia aritmética ($s_i = s_{i-1} + d$, para algún valor inicial s_0) sigue:

$$\sum_{i=0}^{n-1} s_i = \frac{n}{2}\big(s_0 + s_{n-1}\big).$$

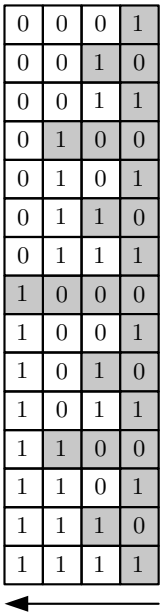
Ejercicio 1.8 Resuelve las siguientes recurrencias, sin calcular las constantes multiplicativas (C_i). Esto implica que los casos base no son necesarios en este ejercicio.

a) $T(n) = 4T(n-1) - 5T(n-2) + 2T(n-3) + n - 3 + 5n^2 \cdot 2^n$

b) $T(n) = T(n-1) + 3n - 3 + n^3 \cdot 3^n$

c) $T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3) + 3 + n^2 + n2^n$

Ejercicio 1.9 Un algoritmo procesa algunos bits de la representación binaria de los números desde 1 a $2^n - 1$, donde n es el número de bits de cada número. En concreto, el algoritmo procesa los bits menos significativos de cada número (de derecha a izquierda), hasta que encuentra el primer bit a 1. Dado n, determina, usando sumatorios, el número total de bits que procesa el algoritmo. Por ejemplo, para $n = 4$ el algoritmo procesa los 26 bits sombreados de la siguiente figura:



Ejercicio 1.10 Especifica relaciones de recurrencia que describan el coste computacional en tiempo de algoritmos que implementen las siguientes funciones:

$$f(n) = \begin{cases} 1 & \text{si } n = 1 \text{ o } n = 2, \\ \left\lfloor \Phi \cdot f(n-1) + \dfrac{1}{2} \right\rfloor & \text{si } n > 2, \end{cases} \tag{1}$$

donde $\Phi = (1 + \sqrt{5})/2$ es la "proporción aurea" y $\lfloor \cdot \rfloor$ es la función suelo.

$$f(n, a, b) = \begin{cases} b & \text{si } n = 1, \\ f(n-1, a+b, a) & \text{si } n > 1. \end{cases} \tag{2}$$

$$f(n) = \begin{cases} 1 & \text{si } n = 1 \text{ o } n = 2, \\ \left[f\left(\frac{n}{2} + 1\right)\right]^2 - \left[f\left(\frac{n}{2} - 1\right)\right]^2 & \text{si } n > 2 \text{ y } n \text{ par}, \\ \left[f\left(\frac{n+1}{2}\right)\right]^2 + \left[f\left(\frac{n-1}{2}\right)\right]^2 & \text{si } n > 2 \text{ y } n \text{ impar}. \end{cases} \tag{3}$$

$$f(n, s) = \begin{cases} 1 + s & \text{si } n = 1 \text{ o } n = 2, \\ f(n-1, s + f(n-2, 0)) & \text{si } n > 2. \end{cases} \tag{4}$$

Ejercicio 1.11 Define una relación de recurrencia para el coste computacional en tiempo del siguiente código:

```
def es_par(n):
    if n == 0:
        return True
    elif n == 1:
        return False
    else:
        return es_par(n - 2)
```

Resuelve la relación de recurrencia por el método de expansión de recurrencias y el método general de resolución de recurrencias. Por último, indica su orden de crecimiento.

Ejercicio 1.12 Resuelve la siguiente recurrencia por el método de expansión de recurrencias:

$$T(n) = \begin{cases} 1 & \text{si } n = 0, \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{si } n > 0. \end{cases}$$

Ejercicio 1.13 Resuelve la siguiente recurrencia por el método de expansión de recurrencias:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ 2T(n/4) + \log_2 n & \text{si } n > 1. \end{cases}$$

Ejercicio 1.14 Resuelve la siguiente recurrencia por el método general de resolución de recurrencias:

$$T(n) = \begin{cases} a & \text{si } n = 1, \\ T(n-1) + b & \text{si } n > 1, \end{cases}$$

Ejercicio 1.15 Define una relación de recurrencia para el coste computacional en tiempo de la siguiente función:

```
def suma_naturales(n):
    if n == 1:
        return 1
    elif n % 2 == 0:
        return 2 * suma_naturales(n / 2) + (n / 2)**2
    else:
        return (2 * suma_naturales((n - 1) / 2) + ((n + 1) / 2)**2)
```

Resuélvela usando el teorema maestro, el método de expansión de recurrencias y el método general de resolución de recurrencias. Por último, indica su orden de crecimiento.

Ejercicio 1.16 Resuelve la siguiente recurrencia usando el teorema maestro, el método de expansión de recurrencias y el método general de resolución de recurrencias. Por último, indica su orden de crecimiento.

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ 3T(n/2) + n & \text{si } n > 1, \end{cases}$$

donde n es una potencia de 2.

Ejercicio 1.17 Resuelve la siguiente recurrencia por el método general de resolución de recurrencias:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/10) + 1 & \text{si } n > 1. \end{cases}$$

Ejercicio 1.18 Resuelve las siguientes recurrencias:

a) $T(n) = 2T(n-1) + 3n - 2$, donde $T(0) = 0$.

b) $T(n) = T(n/2) + n$, donde $T(1) = 1$, y n es una potencia de 2.

c) $T(n) = T(n/\alpha) + n$, donde $T(1) = 0$, and donde $\alpha \geq 2$ es un entero.

d) $T(n) = T(n/3) + n^2$, donde $T(1) = 1$, y n es una potencia de 3.

e) $T(n) = 3T(n/3) + n^2$, donde $T(1) = 0$, y n es una potencia de 3.

f) $T(n) = 2T(n/4) + n$, donde $T(1) = 1$, y n es una potencia de 4.

g) $T(n) = T(n/2) + \log_2 n$, donde $T(1) = 1$, y n es una potencia de 2.

h) $T(n) = 4T(n/2) + n$, donde $T(1) = 1$, y n es una potencia de 2.

i) $T(n) = 2T(n/2) + n \log_2 n$, donde $T(1) = 1$, y n es una potencia de 2.

j) $T(n) = \frac{3}{2}T(n/2) - \frac{1}{2}T(n/4) - \frac{1}{n}$, donde $T(1) = 1$, $T(2) = 3/2$, y n es una potencia de 2.

2. Ejercicios resueltos de complejidad computacional

Ejercicio 2.1 Demostrar: $n \ln n = \mathcal{O}(n^{1+a})$, donde $0 < a < 1$.

Solución:
Podemos demostrar que n^{1+a} $(0 < a < 1)$ es cota superior de $n \ln n$ si se verifica:

$$\lim_{n\to\infty} \frac{n \ln n}{n^{1+a}} = 0$$

Procedemos a calcular el límite:

$$\lim_{n\to\infty} \frac{n \ln n}{n^{1+a}} = \lim_{n\to\infty} \frac{\ln n}{n^a} = \frac{\infty}{\infty}$$

Aplicando L'Hopital (derivando numerador y denominador) se obtiene:

$$\lim_{n\to\infty} \frac{1/n}{an^{a-1}} = \lim_{n\to\infty} \frac{1}{an^a} = 0$$

NOTA: se ha utilizado un logaritmo neperiano, pero el resultado es válido independientemente de la base del logaritmo.

Ejercicio 2.2 Simplificar: $\mathcal{O}(3m^3 + 2mn^2 + n^2 + 10m + m^2)$

Solución:
Para simplificar la expresión, en primer lugar, podemos eliminar las constantes, quedando:

$$\mathcal{O}(m^3 + mn^2 + n^2 + m + m^2)$$

En segundo lugar, resulta trivial ver que los términos m y m^2 son de menor orden que m^3, por tanto también se pueden cancelar:

$$\mathcal{O}(m^3 + mn^2 + n^2)$$

Finalmente n^2 también se puede eliminar dado que el término mn^2 es de mayor orden. Esto puede comprobarse hallando los siguientes límites:

$$\lim_{n\to\infty} \frac{n^2}{mn^2} = \frac{1}{m} \neq \infty \qquad \lim_{m\to\infty} \frac{n^2}{mn^2} = 0$$

Como uno es igual a 0 y otro distinto de 0 se verifica que mn^2 es de mayor orden n^2. Finalmente la expresión no puede simplificarse más, quedando:

$$\boxed{\mathcal{O}(m^3 + mn^2)}$$

NOTA: los dos últimos términos no se pueden eliminar. No se puede afirmar que m^3 sea de mayor orden que mn^2 ya que:

$$\lim_{n\to\infty} \frac{mn^2}{m^3} = \infty \qquad \lim_{m\to\infty} \frac{mn^2}{m^3} = 0$$

Es decir, un límite es 0 y el otro infinito. Análogamente, tampoco se puede afirmar que mn^2 sea de mayor orden que m^3 dado que:

$$\lim_{n\to\infty} \frac{m^3}{mn^2} = 0 \qquad \lim_{m\to\infty} \frac{m^3}{mn^2} = \infty$$

Ejercicio 2.3 Simplificar la siguiente expresión:

$$\mathcal{O}(m^2 + m\log n + n\log m^2 + mn)$$

Solución:

En primer lugar eliminamos las constantes:

$$\mathcal{O}(m^2 + m\log n + n\log m^2 + mn)$$
$$= \mathcal{O}(m^2 + m\log n + 2n\log m + mn)$$
$$= \mathcal{O}(m^2 + m\log n + n\log m + mn)$$

A continuación comprobamos término a término si uno tiene mayor orden que otro. De esta manera podemos ver que podemos eliminar dos términos, ya que $mn > m\log n$:

$$\lim_{n\to\infty} \frac{m\log n}{mn} = 0 \qquad \lim_{m\to\infty} \frac{m\log n}{mn} = \frac{\log n}{n} < \infty$$

y $mn > n\log m$:

$$\lim_{n\to\infty} \frac{n\log m}{mn} = \frac{\log m}{m} < \infty \qquad \lim_{m\to\infty} \frac{n\log m}{mn} = 0$$

Por tanto, la expresión simplificada final es:

$$\boxed{\mathcal{O}(m^2 + mn)}$$

Para calcular los límites se ha usado:

$$\lim_{x\to\infty} \frac{\log x}{x} = \frac{\infty}{\infty}$$

Donde aplicando la regla de L'Hopital:

$$\lim_{x\to\infty} \frac{\log x}{x} = \lim_{x\to\infty} \frac{1}{x} = 0$$

Ejercicio 2.4 Demostrar utilizando la definición de Θ: $n^2/4 + 3n - 1 \in \Theta(n^2)$

Solución:
Para hacer la demostración debemos comprobar que n^2 es tanto cota inferior como superior. Es decir, $n^2/4 + 3n - 1 \in \Omega(n^2)$ y $n^2/4 + 3n - 1 \in \mathcal{O}(n^2)$.

Ω :
Debemos encontrar un par de constantes positivas c_1 y n_1 tal que se cumpla:

$$n^2/4 + 3n - 1 \geq c_1 n^2$$

para todo $n \geq n_1$. Por ejemplo, podemos elegir $c_1 = 1/4$. En ese caso tenemos

$$n^2/4 + 3n - 1 \geq n^2/4$$

$$3n - 1 \geq 0$$

Lo cual se cumple para todo $n \geq 1$. Por tanto, simplemente escogemos $n_1 = 1$. Finalmente, hemos visto que existen esas constantes y queda demostrado.

$\mathcal{O}$:
Debemos encontrar un par de constantes positivas c_2 y n_2 tal que se cumpla:

$$n^2/4 + 3n - 1 \leq c_2 n^2$$

para todo $n \geq n_2$. Por ejemplo, podemos elegir $c_2 = 1$. En ese caso tenemos

$$n^2/4 + 3n - 1 \leq n^2$$

$$3n - 1 \leq 3n^2/4$$

El orden del termino cuadrático naturalmente es superior al lineal, con lo cual la desigualdad se va a cumplir a partir de un n dado. En ese caso, se cumple a partir de $n \geq 4$, y podemos elegir $n_2 = 4$. Una vez más, hemos visto que existen esas constantes y queda demostrado.

NOTA: podíamos haber elegido otras constantes. Lo importante es que existan, no sus valores concretos.

Ejercicio 2.5 Demostrar mediante la definición de $\mathcal{O}$:

$$n^e \in \mathcal{O}(e^n)$$

Solución:

Para hacer la demostración debemos encontrar un par de constantes positivas c y n_0 tal que se cumpla:

$$n^e \leq ce^n$$

para todo $n \geq n_0$. Podemos escoger $c = 1$. En ese caso tenemos:

$$n^e \leq e^n$$

Aplicando logaritmos a ambos lados:

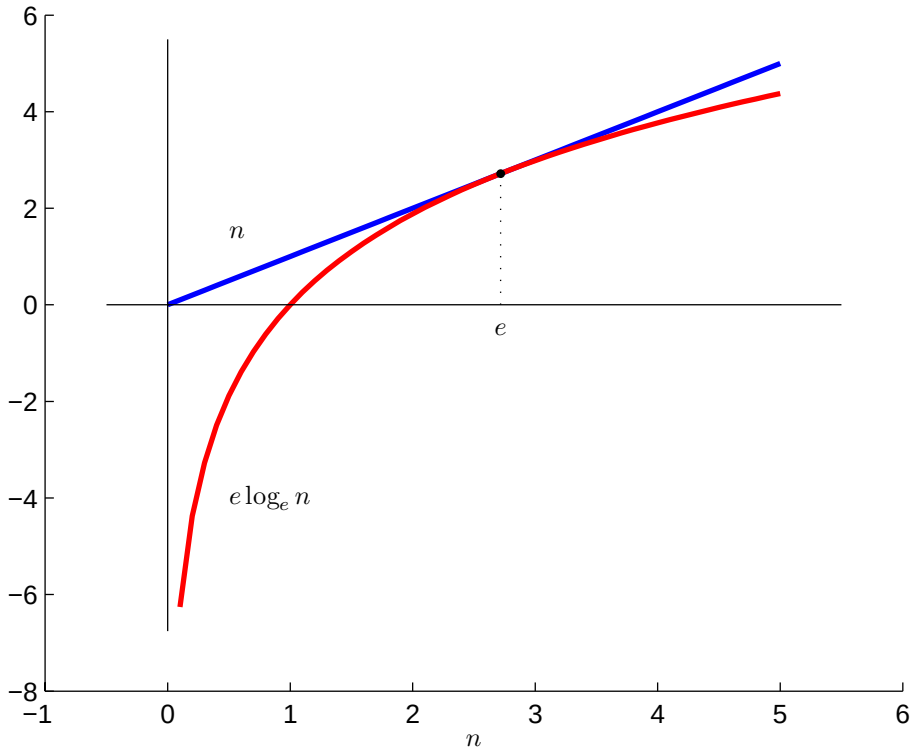
$$e \log_e n \leq log_e e^n = n$$

Esta operación se puede realizar sin que afecte a la desigualdad ya que el logaritmo es una función monótona creciente, por lo que "preserva el orden" (es decir, si $a < b$, entonces $\log a < \log b$).

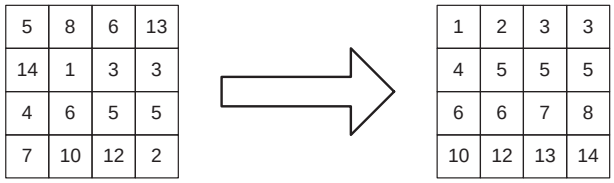
Ahora necesitamos encontrar un n_0 para el que se cumpla la expresión para todo $n \geq n_0$. En este ejercicio no podemos despejar n, por lo que seguiremos otra estrategia (hay varias estrategias válidas). Nos fijamos en los valores de las funciones y sus derivadas (pendientes) con respecto a n. Para $e \log_e n$ la derivada es e/n, mientras que para n la derivada es 1. Por tanto, las dos son funciones crecientes para $n > 0$, pero $e \log_e n$ crece igual o más lentamente que n para $n \geq e$ (ya que su pendiente sería menor o igual a 1). Por tanto, si encontramos un $n_0 \geq e$, de tal forma que la función n sea mayor que $e \log_e n$ en n_0 ($e \log_e n_0 \leq n_0$), la fórmula se verificará para todo $n \geq n_0$. En otras palabras, la función n no solamente será mayor o igual que $e \log_e n$ en n_0, sino que lo será también para todo $n \geq n_0$.

En este caso, tomamos $n_0 = e = 2{,}718\ldots$ Como ambas funciones toman el mismo valor en e, que vale justamente e, entonces $e \log_e n \leq n$ para todo $n \geq e = n_0$, terminando la demostración.

Aunque no es necesario para la demostración, la siguiente figura muestra las dos funciones $e \log_e n$, y n, donde se puede ver que se verifica $e \log_e n \leq n$ incluso para todo $n > 0$:


6
4
2
0
−2
−4
−6
−8
−1
0
1
2
3
4
5
6
n
n
e
e log_e n

Ejercicio 2.6 Determinar la complejidad θ del siguiente algoritmo: se trata de ordenar los elementos de una matriz cuadrada, por filas, utilizando una



Solución:
La variante del algoritmo podría consistir en transformar la matriz en un vector auxiliar, ordenarlo, y volver a convertir el vector ordenado en una matriz. Otra posibilidad sería realizar la ordenación sin el vector auxiliar. En cualquier caso, la solución a este ejercicio depende de la medida que usemos del tamaño de la matriz cuadrada. De esta manera podemos considerar dos posibilidades:

- Si suponemos que la matriz cuadrada tiene n elementos (tendría $\sqrt{n}$ filas y columnas), entonces el algoritmo tendría complejidad cuadrática:

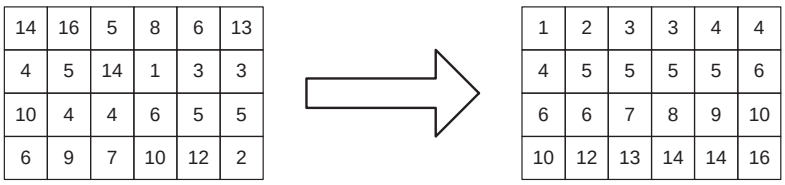
  $$\boxed{\theta(n^2)}$$

  ya que el algoritmo de ordenación Bubble-sort tiene complejidad $\theta(n^2)$, tanto en el peor como el mejor caso, para un vector de tamaño n.

- Si suponemos que la matriz cuadrada tiene n filas y columnas entonces el número de elementos de la matriz es n^2, y el algoritmo de ordenación sobre n^2 elementos tiene complejidad:

  $$\boxed{\theta(n^4)}$$

Ejercicio 2.7 Determinar la complejidad θ del siguiente algoritmo: se trata de ordenar los elementos de una matriz rectangular, por filas, utilizando una variante del algoritmo de ordenación Bubble-sort (Burbuja).



Solución:
Este ejercicio es idéntico al anterior, salvo que en este caso es necesario considerar dos parámetros a la hora de definir el tamaño del problema: el número de filas (n) y columnas (m). De esta manera, la matriz tiene nm elementos, y el algoritmo de ordenación Bubble-sort siempre necesita del orden de $(nm)^2$ operaciones. Por tanto, la medida de complejidad θ para este algoritmo es:

$$\boxed{\theta(m^2n^2)}$$

Ejercicio 2.8 Considérese el siguiente algoritmo, que determina si una matriz cuadrada A de tamaño $N \times N$ es simétrica ($A = A^T$):

```
1  es_traspuesta = TRUE;
2  columna = 1;
3  while (columna<=N) AND es_traspuesta
4      fila = N;
5      while (fila>columna) AND es_traspuesta
6          if A(fila,columna)!=A(columna,fila)
7              es_traspuesta = FALSE;
8          end
9          fila = fila-1;
10     end
11     columna = columna+1;
12 end
13 return es_traspuesta;
```

Se pide hallar la complejidad Θ asumiendo que se realiza **una** operación por cada línea de código (se valorará el uso de sumatorios en la respuesta).

Solución:
Analizamos la complejidad del algoritmo en el mejor y peor caso:

- **Mejor caso:** Si los dos primeros elementos comparados de la matriz son diferentes entonces el algoritmo devuelve el valor FALSE en un número constante de pasos, independientemente del tamaño del problema. Por tanto la complejidad en este caso es:

$$\boxed{\Theta(1)}$$

- **Peor caso:** El peor caso ocurre cuando la matriz es simétrica, y la variable `es_traspuesta` siempre toma el valor verdadero. Para facilitar los cálculos dividimos el código en los siguientes bloques:

```
es_traspuesta = TRUE;
columna = 1;
while (columna<=N) AND es_traspuesta
  fila = N;
  while (fila>columna) AND es_traspuesta
    if A(fila,columna)!=A(columna,fila)
      es_traspuesta = FALSE;
    end
    fila = fila-1;
  end
  columna = columna+1;
end
return es_traspuesta;
```

Considerando que se realiza una operación por cada línea de código (sin contar los `end`, ni la instrucción en el cuerpo del `if`, ya que no se ejecuta en el peor caso), la fórmula para el tiempo de ejecución suponiendo que la matriz es de tamaño N es:

$$\mathbf{T(N)} = \mathbf{1} + \mathbf{1} + \sum_{\mathbf{c=1}}^{\mathbf{N}} \left[\mathbf{1} + \mathbf{1} + \sum_{\mathbf{f=N}}^{\mathbf{c+1}} (\mathbf{1} + \mathbf{1} + \mathbf{1}) + \mathbf{1} + \mathbf{1} \right] + \mathbf{1} + \mathbf{1}$$

Simplificando y resolviendo los sumatorios se obtiene:

$$T(N) = 4 + \sum_{c=1}^{N} \left[4 + \sum_{f=N}^{c+1} 3 \right] = T(N) = 4 + \sum_{c=1}^{N} \left[4 + 3(N - c) \right]$$

$$T(N) = 4 + 4N + 3N^2 - 3 \sum_{c=1}^{N} c = 4 + 4N + 3N^2 - 3\frac{N(N+1)}{2}$$

$$\boxed{T(N) = \frac{3N^2}{2} + \frac{5N}{2} + 4 \quad \in \Theta(N^2)}$$

Ejercicio 2.9 Determina el coste del siguiente algoritmo de ordenación:

```
1  FOR i:=1 TO n-1 DO
2  BEGIN
3    valMenor := v[i];
4    posMenor := i;
5    FOR j:=i+1 TO n DO
6    BEGIN
7      IF v[j] < valMenor THEN
8      BEGIN
9        valMenor:= v[j];
10       posMenor := j
11     END;
12   END;
13   IF posMenor <> i THEN
14   BEGIN
15     v[posMenor] := v[i];
16     v[i] := valMenor;
17   END;
18 END;
```

Solución:

Se trata del algoritmo de ordenación por selección directa (*select-sort*). Para analizarlo usaremos la fórmula asociada al coste de un bucle:

$$T_{\text{bucle}} = 1_{\text{inic.}} + \sum^{n}(1_{\text{comp.}} + T_{\text{cuerpo}} + 1_{\text{incr.}}) + 1_{\text{última comp.}}$$

Además consideraremos los siguientes bloques de código:

```
FOR i:=1 TO n-1 DO
BEGIN
  valMenor := v[i];
  posMenor := i;
  FOR j:=i+1 TO n DO
  BEGIN
    IF v[j] <valMenor THEN
    BEGIN
      valMenor:= v[j];
      posMenor := j
    END;
  END;
  IF posMenor <>i THEN
  BEGIN
    v[posMenor] := v[i];
    v[i] := valMenor;
  END;
END;
```

Considerando el primer bucle la función del coste se puede expresar como:

$$T(n) = 1 + \sum_{i=1}^{n-1}(1 + T_{\text{cuerpo FOR externo}} + 1) + 1$$

donde

$$T_{\text{cuerpo FOR externo}} = 2 + 1 + \sum_{j=i+1}^{n}(1 + T_{\text{cuerpo FOR interno}} + 1) + 1 + 1 + C_1$$

donde C_1 es coste del interior del último IF, y:

$$T_{\text{cuerpo FOR interno}} = 1 + C_2$$

donde C_2 es el coste del interior del primer IF.

Se han introducido las constantes C_1 y C_2 simplemente para diferenciar los costes en el caso mejor y peor. El mejor caso se da cuando el vector está previamente ordenado, en cuyo caso las comparaciones de los IF siempre son falsas, por lo que $C_1 = C_2 = 0$. En cambio, en el peor caso siempre son verdaderas por lo que $C_1 = C_2 = 2$. De esta manera, una constante $C = C_1 = C_2$ es suficiente en este ejercicio.

El coste total viene dado por:

$$T(n) = 1 + \sum_{i=1}^{n-1}\Big(1 + 2 + 1 + \sum_{j=i+1}^{n}(1 + 1 + C + 1) + 1 + 1 + C + 1\Big) + 1$$

$$= 2 + \sum_{i=1}^{n-1}\Big(7 + C + \sum_{j=i+1}^{n}(3 + C)\Big)$$

Como el segundo sumatorio realiza $(n - i)$ operaciones:

$$= 2 + \sum_{i=1}^{n-1}\Big(7 + C + (3 + C)(n - i)\Big)$$

El sumatorio que queda realiza (n-1) operaciones:

$$= 2 + (7 + C)(n - 1) + (3 + C)n(n - 1) - (3 + C)n(n - 1)/2$$

$$= 2 + (7 + C)(n - 1) + (3 + C)n(n - 1)/2$$

Simplificando:

$$= \frac{3 + C}{2}n^2 + \frac{11 + C}{2}n - (C + 5) \quad \in \quad \theta(n^2)$$

donde

$$\boxed{T_{\text{mejor}}(n) = \frac{3}{2}n^2 + \frac{11}{2}n - 5 \qquad T_{\text{peor}}(n) = \frac{5}{2}n^2 + \frac{13}{2}n - 7}$$

Ejercicio 2.10 Analizar la complejidad Θ del problema de las Torres de Hanoi para n discos. Es decir, hallar la expresión no recursiva del número de movimientos de discos T_n, para n discos, y describir su cota ajustada Θ . (Ayuda: ver el enunciado del siguiente ejercicio). El pseudocódigo del algoritmo es:

```
1 hanoi(int n, int destino, int origen, int auxiliar)
2   if (n > 0)
3      hanoi(n-1, auxiliar, origen, destino);
4      << Mover disco n desde origen a destino >>
5      hanoi(n-1, destino, auxiliar, origen);
```

Solución:
La expresión recursiva del algoritmo es la siguiente:

$$T_n = 2T_{n-1} + 1 \qquad T_0 = 0$$

Para resolver la recurrencia y hallar una expresión no recursiva de T_n se puede proceder de dos maneras:

1. Expansión de recurrencias:

$$\begin{aligned} T_n &= 2T_{n-1} + 1 \\ &= 2(2T_{n-2} + 1) + 1 = 2^2T_{n-2} + 2 + 1 \\ &= 2(2(2T_{n-3} + 1) + 1) + 1 = 2^3T_{n-3} + 4 + 2 + 1 \\ &= 2(2(2(2T_{n-4} + 1) + 1) + 1) + 1 = 2^4T_{n-4} + 8 + 4 + 2 + 1 \end{aligned}$$

   Se puede ver, por tanto, que la expresión general es:

$$T_n = 2^iT_{n-i} + \sum_{j=0}^{i-1} 2^j$$

   Se llega al caso base cuando el parámetro de la T es cero. Es decir, cuando $n - i = 0$. Despejando la i, se obtiene $i = n$. Sustituyendo en la expresión obtenemos:

$$T_n = 2^nT_0 + \sum_{j=0}^{n-1} 2^j$$

   Viendo que $T(0) = 0$ y resolviendo la serie geométrica queda finalmente:

$$\boxed{T_n = 2^n - 1 \in \Theta(2^n)}$$

2. Método general de resolución de recurrencias: La recurrencia se puede expresar como:

$$T_n - 2T_{n-1} = 1 \cdot n^0 \cdot 1^n$$

   La cual es una recurrencia no homogénea, cuya ecuación característica es:

$$(x - 2)(x - 1) = 0$$

Por tanto, la solución final tiene la forma

$$T_n = c_1 2^n + c_2 1^n = c_1 2^n + c_2$$

Teniendo en cuenta dos casos base o condiciones iniciales podemos hallar el valor de las constantes resolviendo el siguiente sistema:

$$\left.\begin{array}{r} c_1 + c_2 = 0 = T_0 \\ 2c_1 + c_2 = 1 = T_1 \end{array}\right\}$$

De donde obtenemos $c_1 = 1$ y $c_2 = -1$. Finalmente, concluimos que la solución es:

$$\boxed{T_n = 2^n - 1 \in \Theta(2^n)}$$

NOTA: No es necesario resolver el ejercicio usando los dos métodos. Uno es suficiente. Por otro lado, también se podía haber escogido el caso base $T_1 = 1$ en el método de expansión de recurrencias.

Ejercicio 2.11 La siguiente función recursiva calcula el máximo de un vector $\mathbf{v}$ de n números reales utilizando el paradigma de divide y vencerás:

$$f(\mathbf{v}) = \begin{cases} v_1 & \text{si } n = 1 \\ \text{máx}\{f(v_1, \ldots, v_{n/2}), f(v_{n/2+1}, \ldots, v_n)\} & \text{si } n > 1 \end{cases}$$

A partir de la expresión recursiva T del número de operaciones que realiza la función f para un determinado tamaño de entrada, se pide obtener la expresión no recursiva de T:

a) Por el método de expansión de recurrencias (indicando además su orden de complejidad)

b) Por el método general de resolución de recurrencias (pista: usad un cambio de variable)

Solución:

En primer lugar es necesario definir correctamente la función del número de operaciones que realiza el algoritmo para un vector de tamaño n:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 1 + 2T(n/2) & \text{si } n > 1 \end{cases}$$

a) Expansión de recurrencias:

$$\begin{aligned} T(n) &= 1 + 2T(n/2) \\ &= 1 + 2[1 + 2T(n/4)] = 1 + 2 + 2^2T(n/2^2) \\ &= 1 + 2 + 4[1 + 2T(n/8)] = 1 + 2 + 4 + 2^3T(n/2^3) \\ &= 1 + 2 + 4 + 8[1 + 2T(n/16)] = 1 + 2 + 4 + 8 + 2^4T(n/2^4) \\ &= \sum_{j=0}^{i-1} 2^j + 2^iT(n/2^i) = 2^i - 1 + 2^iT(n/2^i) \end{aligned}$$

Se alcanza el caso base cuando $n/2^i = 1$. Es decir, cuando $2^i = n$. Sustituyendo:

$$\boxed{T(n) = n - 1 + nT(1) = 2n - 1 \in \theta(n)}$$

b) Método general de resolución de recurrencias:

Partimos de la expresión recursiva:

$$T(n) = 1 + 2T(n/2)$$

Como no podemos aplicar el método directamente con argumentos de T que sean fracciones de n, realizamos un cambio de variable:

$$n = 2^k$$

De esta forma tenemos:

$$T(2^k) = 1 + 2T(2^k/2) = 1 + 2T(2^{k-1})$$

que se puede expresar mediante otra función t que dependa de k:

$$t(k) = 1 + 2t(k-1)$$

donde $t(k) = T(2^k)$. En este momento podemos aplicar el método general de resolución de recurrencias con t. Para ello pasamos todos los términos de t a la izquierda, y expresamos el resto como productos de polinomios por constantes elevadas a k:

$$t(k) - 2t(k-1) = 1 \cdot 1^k$$

Su polinomio característico es:

$$(x-2)(x-1)$$

por lo que su solución es:

$$t(k) = C_1 2^k + C_2$$

Como $t(k) = T(2^k)$, los casos base son $t(0) = T(1) = 1$, y $t(1) = T(2) = 3$. Con esta información podemos formar un sistema de ecuaciones para hallar las constantes C_1 y C_2:

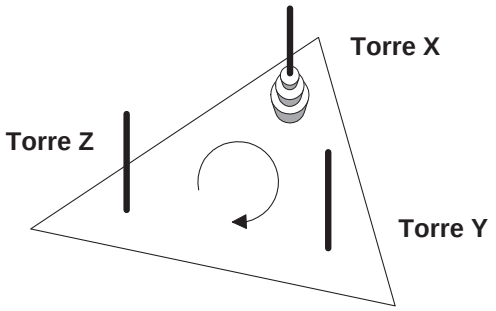
$$\left.\begin{array}{rcrcl} t(0) & = & C_1 + C_2 & = & 1 \\ t(1) & = & 2C_1 + C_2 & = & 3 \end{array}\right\}$$

Resolviendo, $C_1 = 2$ y $C_2 = -1$, por lo que $t(k) = T(2^k) = 2 \cdot 2^k - 1$. Finalmente, deshaciendo el cambio $2^k = n$ obtenemos:

$$\boxed{T(n) = 2n - 1 \in \theta(n)}$$

que naturalmente tiene la misma expresión que la hallada por el método de expansión de recurrencias.

Ejercicio 2.12 En este ejercicio se pide analizar la complejidad Θ del problema de las Torres de Hanoi Cíclicas para n



Si a la función `AntiClock` la denominamos A, y `Clock` C, entonces tenemos las siguientes definiciones recursivas del número de operaciones (movimientos de discos), para n discos:

$$A_n = \begin{cases} 0 & \text{si } n = 0 \\ 2A_{n-1} + C_{n-1} + 2 & \text{si } n > 0 \end{cases} \quad (5)$$

$$C_n = \begin{cases} 0 & \text{si } n = 0 \\ 2A_{n-1} + 1 & \text{si } n > 0 \end{cases} \quad (6)$$

Se pide demostrar:

$$A_n = \frac{1}{4\sqrt{3}} \left[(1+\sqrt{3})^{n+2} - (1-\sqrt{3})^{n+2} \right] - 1$$

$$C_n = \frac{1}{2\sqrt{3}} \left[(1+\sqrt{3})^{n+1} - (1-\sqrt{3})^{n+1} \right] - 1$$

Solución:
En este ejercicio el primer paso consiste en eliminar la recursividad mutua, y expresar las funciones A_n y C_n en términos de sí mismas.

Dado que $C_n = 2A_{n-1} + 1$, decrementando el índice una unidad obtenemos $C_{n-1} = 2A_{n-2} + 1$. Sustituyendo en (5) logramos una definición recursiva que sólo depende de A, a la que añadimos los dos casos base necesarios:

$$A_n = \begin{cases} 0 & \text{si } n = 0 \\ 2 & \text{si } n = 1 \\ 2A_{n-1} + 2A_{n-2} + 3 & \text{si } n \geq 2 \end{cases}$$

Se puede observar que $A_1 = C_0 + 2 = 2$. Además $A_2 = 7$.

También podemos hallar una expresión recursiva de C solamente en términos de sí misma. Por ejemplo, de (6) obtenemos las igualdades $2A_{n-1} = C_n - 1$, y $A_n = (C_{n+1} - 1)/2$, que podemos sustituir en (5) para obtener:

$$\frac{C_{n+1} - 1}{2} = C_n - 1 + C_{n-1} + 2$$

$$C_{n+1} = 2C_n + 2C_{n-1} + 3$$

Finalmente, decrementando el índice y añadiendo los casos base, se obtiene:

$$C_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ 2C_{n-1} + 2C_{n-2} + 3 & \text{si } n \geq 2 \end{cases}$$

Se puede observar que $C_1 = A_0 + 1 = 1$. Además $C_2 = 5$.

En primer lugar, desarrollamos la fórmula no recursiva para A resolviendo la recurrencia no homogénea:

$$A_n - 2A_{n-1} - 2A_{n-2} = 3$$

La ecuación característica es:

$$(x^2 - 2x - 2)(x - 1) = 0$$

Las raíces del primer polinomio son:

$$x = \frac{2 \pm \sqrt{4+8}}{2} = 1 \pm \sqrt{3}$$

Por tanto, la ecuación característica es:

$$(1 + \sqrt{3})(1 - \sqrt{3})(x - 1) = 0$$

Y la solución tiene la siguiente forma:

$$A_n = c_1(1 + \sqrt{3})^n + c_2(1 - \sqrt{3})^n + c_3 1^n$$

Para hallar los valores de las constantes resolvemos el siguiente sistema de ecuaciones, usando 3 casos base:

$$\left.\begin{aligned} c_1 + c_2 + c_3 = 0 = A_0 \\ c_1(1 + \sqrt{3}) + c_2(1 - \sqrt{3}) + c_3 = 2 = A_1 \\ c_1(1 + \sqrt{3})^2 + c_2(1 - \sqrt{3})^2 + c_3 = 7 = A_1 \end{aligned}\right\}$$

De la primera ecuación obtenemos

$$c_3 = -c_1 - c_2 \tag{7}$$

que sustituimos en las dos restantes ecuaciones para obtener:

$$\left.\begin{aligned} c_1(1 + \sqrt{3}) + c_2(1 - \sqrt{3}) - c_1 - c_2 = 2 \\ c_1(1 + \sqrt{3})^2 + c_2(1 - \sqrt{3})^2 - c_1 - c_2 = 7 \end{aligned}\right\}$$

De la primera ecuación se obtiene la expresión:

$$c_1 = \frac{2}{\sqrt{3}} + c_2 \tag{8}$$

que sustituyendo en la segunda se puede verificar que el valor de la constante c_2 es:

$$c_2 = \frac{\sqrt{3}-2}{2\sqrt{3}} = -\frac{2-\sqrt{3}}{2\sqrt{3}} = -\frac{4-2\sqrt{3}}{4\sqrt{3}} = -\frac{(1-\sqrt{3})^2}{4\sqrt{3}}$$

Sustituyendo el valor de c_2 en (8) se obtiene:

$$c_1 = \frac{2}{\sqrt{3}} + \frac{\sqrt{3}-2}{2\sqrt{3}} = \frac{2+\sqrt{3}}{2\sqrt{3}} = \frac{4+2\sqrt{3}}{4\sqrt{3}} = \frac{(1+\sqrt{3})^2}{4\sqrt{3}}$$

Finalmente, sustituyendo c_1 y c_2 en (7) se obtiene $c_3 = -1$, con lo cual la definición de A resulta ser:

$$A_n = \frac{1}{4\sqrt{3}}\left[(1+\sqrt{3})^{n+2} - (1-\sqrt{3})^{n+2}\right] - 1 \quad \in \Theta\big((1+\sqrt{3})^n\big) \tag{9}$$

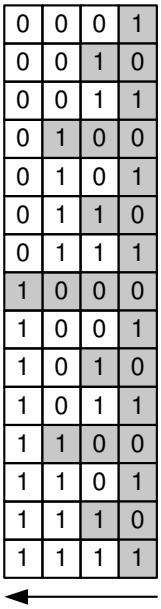
tal y como se había pedido.

La expresión no recursiva de C_n se puede hallar de forma análoga, resolviendo un sistema de ecuaciones lineales. Sin embargo, dado que ya se ha calculado una expresión no recursiva para A_n, podemos combinar (6) y (9) para hallar C_n:

$$C_n = 2A_{n-1} + 1 = 2\left[\frac{1}{4\sqrt{3}}\left[(1+\sqrt{3})^{n+1} - (1-\sqrt{3})^{n+1}\right] - 1\right] + 1$$

$$C_n = \frac{1}{2\sqrt{3}}\left[(1+\sqrt{3})^{n+1} - (1-\sqrt{3})^{n+1}\right] - 1 \quad \in \Theta\big((1+\sqrt{3})^n\big)$$

Ejercicio 2.13 Considérese un algoritmo que procesa una serie de números en base 2 distintos del 0. Para cada número empieza a procesar cada bit (realiza una operación por bit), empezando por el bit menos significativo, hasta el bit más significativo, pero para en el momento que encuentra el primer 1. La siguiente figura ilustra las operaciones (sombreadas) que haría con todos los números de 4 bits distintos de 0, es decir, procesa desde el 1 hasta el 15.



Si se procesan todos los números el algoritmo habrá realizado 26 operaciones (que es el número de bits sombreados).

Se pide hallar la complejidad θ para este algoritmo si procesa todos los números:

a) Contando operaciones

b) Resolviendo la recurrencia no homogénea mediante la técnica de expansión de recurrencias

c) Resolviendo la recurrencia no homogénea mediante el método de resolución de relaciones en recurrencias

Solución:

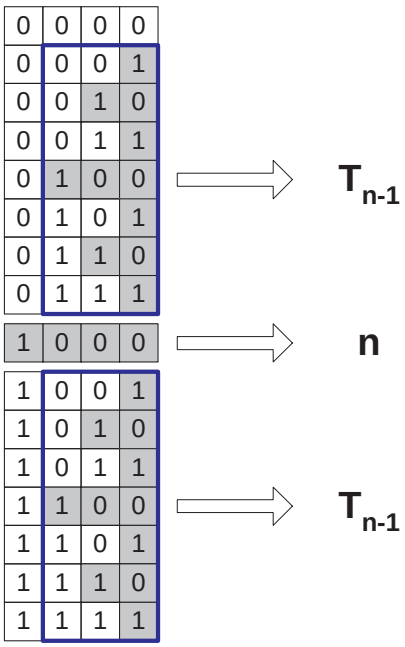
a) Analizando las operaciones que se realizan por cada columna, empezando desde el bit más significativo al menos, podemos ver que la secuencia es 1,3,7,15... Es decir, para n bits el número de operaciones que se realizan se puede expresar como:

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

Por tanto, resolviendo la serie geométrica:

$$\boxed{T_n = 2^{n+1} - 2 - n \quad \in \theta(2^n)}$$

b) En primer lugar, es necesario obtener una fórmula recursiva del número de operaciones que se realizan. Podemos descomponer el número de operaciones de la siguiente manera:



Para n bits contamos n operaciones (correspondientes a la palabra central 2^{n-1}), además de resolver dos veces el mismo problema para $n-1$ bits, la fórmula recursiva es:

$$T_n = 2T_{n-1} + n \qquad T_0 = 0 \tag{10}$$

Una vez hallada la fórmula recursiva procedemos a realizar la expansión de recurrencias:

$$\begin{aligned} T_n &= 2T_{n-1} + n \\ &= 2(2T_{n-2} + n - 1) + n = 2^2 T_{n-2} + 2(n-1) + n \\ &= 2(2(2T_{n-3} + n - 2) + n - 1) + n \\ &\quad = 2^3 T_{n-3} + 2^2(n-2) + 2(n-1) + n \\ &= 2(2(2(2T_{n-4} + n - 3) + n - 2) + n - 1) + n \\ &\quad = 2^4 T_{n-4} + 2^3(n-3) + 2^2(n-2) + 2(n-1) + n \end{aligned}$$

Se puede ver, por tanto, que la expresión general es:

$$T_n = 2^i T_{n-i} + \sum_{j=0}^{i-1} 2^j (n-j) = 2^i T_{n-i} + \sum_{j=0}^{i-1} n 2^j - \sum_{j=0}^{i-1} j 2^j$$

El caso base se alcanza en $T_0 = 0$. Es decir, cuando $i = n$. Realizando las correspondientes sustituciones obtenemos

$$T_n = \sum_{j=0}^{n-1} n2^j - \sum_{j=0}^{n-1} j2^j = n(2^n - 1) - \sum_{j=0}^{n-1} j2^j \tag{11}$$

La dificultad en este ejercicio radica en hallar una expresión para el sumatorio de la fórmula anterior:

$$\sum_{j=0}^{n-1} j2^j = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + (n-1) \cdot 2^{n-1} \tag{12}$$

Para hallarla partimos de una serie geométrica general:

$$S = 1 + x + x^2 + \cdots + x^{m-1} = \frac{x^m - 1}{x - 1}$$

derivando S con respecto a x obtenemos la siguiente igualdad:

$$1 + 2x + 3x^2 + \cdots + (m-1)x^{m-2} = \frac{mx^{m-1}(x-1) - (x^m - 1)}{(x-1)^2} =$$

$$= \frac{mx^m - mx^{m-1} - x^m + 1}{(x-1)^2}$$

Multiplicando por x a ambos lados obtenemos:

$$x^1 + 2x^2 + 3x^3 + \cdots + (m-1)x^{m-1} = \frac{mx^{m+1} - mx^m - x^{m+1} + x}{(x-1)^2} \tag{13}$$

Por tanto, podemos ver que el sumatorio que queremos calcular en (12) es idéntico al descrito en (13) realizando los

$$\sum_{j=0}^{n-1} j2^j = \frac{n2^{n+1} - n2^n - 2^{n+1} + 2}{(2-1)^2} = n2^{n+1} - n2^n - 2^{n+1} + 2$$

Finalmente, sustituyendo en (11) obtenemos la expresión final de T_n:

$$\boxed{T_n = n2^n - n - n2^{n+1} + n2^n + 2^{n+1} - 2 = 2^{n+1} - n - 2 \quad \in \theta(2^n)}$$

c) La ecuación característica de asociada a (10) es:

$$(x-2)(x-1)^2 = 0$$

Por tanto, la solución tiene la siguiente forma:

$$T_n = c_1 2^n + c_2 1^n + c_3 n 1^n \tag{14}$$

Para hallar el valor de las constantes debemos resolver el siguiente sistema de ecuaciones (teniendo en cuenta que $T_0 = 0$, $T_1 = 1$, y $T_2 = 4$):

$$\left.\begin{aligned} c_1 + c_2 &= 0 = T_0 \\ 2c_1 + c_2 + c_3 &= 1 = T_1 \\ 4c_1 + c_2 + 2c_3 &= 4 = T_2 \end{aligned}\right\}$$

Resolviendo el sistema se obtiene $c_1 = 2$, $c_2 = -2$, y $c_3 = -1$. Por tanto, sustituyendo en (14), la expresión final del número de operaciones en función del número de bits n es:

$$\boxed{T_n = 2^{n+1} - 2 - n \quad \in \theta(2^n)}$$

NOTA: El algoritmo tiene un coste exponencial dado el número de bits de las palabras. También podríamos haber calculado la complejidad en función del número total de palabras procesadas (m). En ese caso, dado que $m = 2^n - 1$, bastaría con realizar la sustitución ($n = \log_2(m+1)$) en la fórmula del número de operaciones $T_n = 2^{n+1} - 2 - n$:

$$T'(m) = 2 \cdot 2^{\log_2(m+1)} - \log_2(m+1) - 2 = 2m - \log_2(m+1) \quad \in \theta(m)$$

Ejercicio 2.14 Demostrar que $\log(n!) \in \Omega(n \log n)$, de dos formas diferentes:

a) Usando la aproximación de Stirling:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

para valores muy grandes de n.

b) Utilizando la definición de Ω. Ayuda: usad la aproximación por integrales

Solución:

a) Podemos demostrar que $n \log n$ es cota inferior de $\log(n!)$ si se verifica:

$$\lim_{n\to\infty} \frac{\log(n!)}{n \log n} > 0$$

Procedemos a calcular el límite:

$$\lim_{n\to\infty} \frac{\log(n!)}{n \log n} = \frac{\infty}{\infty}$$

Aplicamos la aproximación de Stirling (ya que estamos calculando el límite cuando n toma valores muy grandes):

$$\lim_{n\to\infty} \frac{\log(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n)}{n \log n} = \lim_{n\to\infty} \frac{\log\sqrt{2\pi} + \log\sqrt{n} + \log\left(\frac{n}{e}\right)^n}{n \log n} =$$

$$\lim_{n\to\infty} \frac{\log\sqrt{2\pi} + \frac{1}{2}\log n + n\log\frac{n}{e}}{n \log n} =$$

$$\lim_{n\to\infty} \frac{\log\sqrt{2\pi} + \frac{1}{2}\log n + n\log n - n\log e}{n \log n} =$$

$$0 + 0 + 1 + 0 = 1 > 0$$

En efecto, como el límite es una constante, no solo $\log(n!) \in \Omega(n \log n)$, sino que además $\log(n!) \in \Theta(n \log n)$.

b) Utilizando la definición de Ω tenemos que demostrar que existe una constante $c > 0$ y un $n_0 > 0$ tal que $\log(n!) \geq cn \log n$ para tono $n \geq n_0$. Comenzamos transformando la parte izquierda en un sumatorio:

$$\log(n!) = \sum_{i=1}^{n} \log i = \sum_{i=2}^{n} \log i$$

El logaritmo es una función monótona creciente, por lo que se puede acotar utilizando la aproximación por integrales. En este caso nos interesa encontrar una cota inferior:

$$\sum_{i=2}^{n} \log i \geq \int_1^n \log x \; dx$$

Resolviendo la integral por partes:

$$\int_1^n \log x \; dx = \Big[x \log x - x\Big]_1^n = n \log n - n - 0 + 1$$

Por tanto, combinando estos resultados:

$$\log(n!) = \sum_{i=2}^{n} \log i \geq \int_1^n \log x \; dx = n \log n - n + 1 \geq cn \log n$$

Lo cual implica que si podemos demostrar que $n \log n$ es cota inferior de $n \log n - n + 1$, entonces necesariamente será cota inferior de $\log(n!)$. De esta manera, solo queda encontrar las constantes c y n_0 para:

$$n \log n - n + 1 \geq cn \log n$$

Escojamos $c = 1/2$:

$$n \log n - n + 1 \geq \frac{1}{2} n \log n \quad \Rightarrow \quad \frac{1}{2} n \log n - n + 1 \geq 0 \quad \Rightarrow$$

$$n\big(\frac{1}{2} \log n - 1\big) + 1 \geq 0$$

Lo cual es cierto para:

$$\frac{1}{2} \log n - 1 \geq 0 \quad \Rightarrow \quad \log n \geq 2$$

Finalmente, para cualquier base del logaritmo b, la desigualdad se cumple para todo $n \geq b^2$, y podemos escoger un valore de $n_0 = b^2$ finito y positivo, terminando la demostración.

Ejercicio 2.15 Indicar el orden de la siguiente recurrencia, a partir de su expresión no recursiva en la que no es necesario calcular constantes:

$$T(n) = T(n-1) + 3n - 3 + n^3 \cdot 3^n$$

Solución:

En primer lugar pasamos todos los términos con T al lado izquierdo, y expresamos cada término del derecho como un producto de un polinomio por una constante elevada a n:

$$T(n) - T(n-1) = (3n-3) \cdot 1^n + n^3 \cdot 3^n$$

El siguiente paso consiste en hallar el polinomio característico. De la parte izquierda obtenemos el factor $(x-1)$. De la parte derecha obtenemos los factores $(x-1)^2$ y $(x-3)^4$. Combinando los factores el polinomio característico final es:

$$(x-1)^3(x-3)^4$$

Por tanto la solución es:

$$T(n) = C_1 1^n + C_2 n 1^n + C_3 n^2 1^n + C_4 3^n + C_5 n 3^n + C_6 n^2 3^n + C_7 n^3 3^n$$

$$\boxed{T(n) \quad \in \quad \theta(n^3 3^n)}$$

3. Ejercicios básicos de recursividad

Ejercicio 3.1 ¿Qué calcula la siguiente función?

$$f(n) = \begin{cases} 1 & \text{si } n = 0, \\ f(n-1) \times n & \text{si } n > 0. \end{cases}$$

Ejercicio 3.2 Considera una secuencia definida por la siguiente definición recursiva: $s_n = s_{n-1} + 3$. Calcula sus cuatro primeros términos teniendo en cuenta que: (a) $s_0 = 0$, y (b) $s_0 = 4$. Indica una definición no recursiva de s_n en ambos casos.

Ejercicio 3.3 Considera una secuencia definida por la siguiente definición recursiva: $s_n = s_{n-1} + s_{n-2}$. Si tuviéramos los dos valores iniciales s_1 y s_2 tendríamos la información suficiente como para construir la secuencia entera. Demuestra que también se puede construir la secuencia dados dos valores cualquiera s_i y s_j, donde $i < j$. Finalmente, encuentra los elementos de la secuencia entre $s_1 = 1$ y $s_5 = 17$.

Ejercicio 3.4 El conjunto "descendientes de una persona" se puede definir recursivamente como los hijos de la persona, junto con los descendientes de esos hijos. Indica una definición matemática de ese concepto empleando notación de conjuntos. En concreto, define una función $D(p)$, donde D denota descendientes, y el parámetro p se refiere a una persona concreta. Además, considera que se puede usar una función $H(p)$, que retornaría el conjunto de hijos de p.

Ejercicio 3.5 Sea $F(n) = F(n-1)+F(n-2)$ una función recursiva, donde n es un entero positivo, and donde $F(1)$ y $F(2)$ son valores iniciales arbitrarios. Demuestra que $F(n) = F(2) + \sum_{i=1}^{n-2} F(i)$, para $n \geq 2$.

Ejercicio 3.6 Implementa en Python la función factorial: $n! = (n-1)! \times n$ si $n > 0$, donde $0! = 1$.

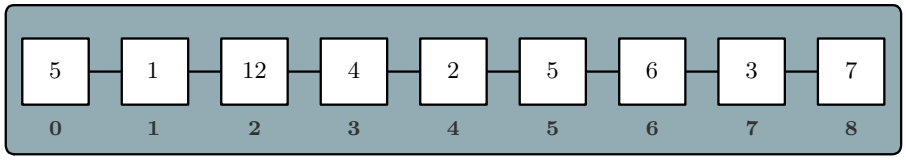
Ejercicio 3.7 Al considerar tipos abstractos de datos, una lista puede ser vacía, o puede consistir en un solo elemento (la "cabeza" de la lista, es decir, su primer elemento), junto con otra lista (la "cola"), que podría estar vacía. Sea `a` una lista en Python. Hay varias formas de comprobar si está vacía. Por ejemplo, la condición `a==[]` devuelve `True` si la lista está vacía, y `False` en caso contrario. Además, la cabeza de la lista es simplemente `a[0]`, mientras que la cola se puede especificar mediante `a[1:]`. Escribe una versión alternativa de la siguiente función:

```
1 def sum_list_length_2(a):
2     if len(a) == 0:
3         return 0
4     else:
5         return a[0] + sum_list_length_2(a[1:len(a)])
```

usando los elementos mencionados, pero que no use la longitud de la lista (`len`).

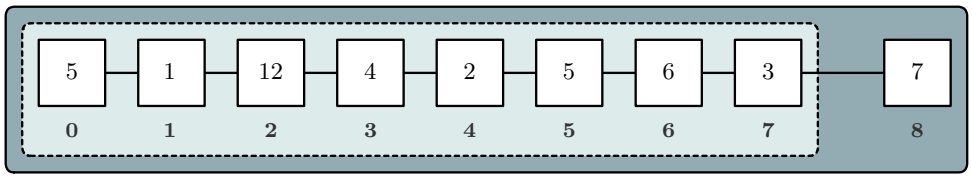
Ejercicio 3.8 Implementa tres funciones que calculen la suma de los elementos de una lista usando las tres descomposiciones indicadas en la siguiente figura:

$$s(\mathbf{a}) = \mathbf{a}[0] + \mathbf{a}[1] + \cdots + \mathbf{a}[n-1]$$



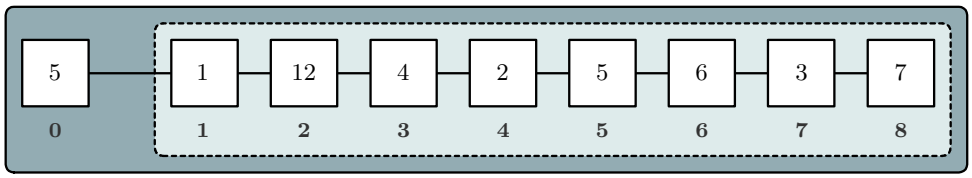
(a) Problema original

$$s(\mathbf{a}) = s(\mathbf{a}[0:n-1]) + \mathbf{a}[n-1]$$



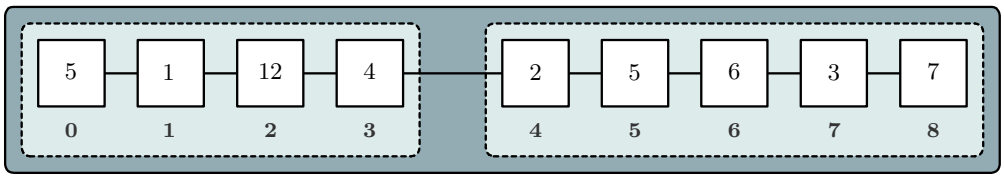
(b) Primera descomposición

$$s(\mathbf{a}) = \mathbf{a}[0] + s(\mathbf{a}[1:n])$$



(c) Segunda descomposición

$$s(\mathbf{a}) = s(\mathbf{a}[0:n//2]) + s(\mathbf{a}[n//2:n])$$



(d) Tercera descomposición

Las funciones recibirán dos parámetros de entrada: una lista, y su tamaño (es decir, su longitud). Además, indica ejemplos de llamadas a estas funciones, e imprime sus resultados.

Ejercicio 3.9 Demuestra por inducción matemática la siguiente identidad asociada a series geométricas, donde n es un entero positivo, y $x \neq 1$ es un número real:

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}.$$

Ejercicio 3.10 Codifica las siguientes funciones que calculan números de Fibonacci $F(n)$ descritas en el Ejercicio 1.10. Además codifica las siguientes funciones, que también sirven para hallar números de Fibonacci:

$$A(n) = \begin{cases} 0 & \text{si } n = 1, \\ A(n-1) + B(n-1) & \text{si } n > 1, \end{cases}$$

$$B(n) = \begin{cases} 1 & \text{si } n = 1, \\ A(n-1) & \text{si } n > 1. \end{cases}$$

Como algunas de estas funciones requieren parámetros adcionales, además de n, o no calculan un número de Fibonacci directamente, implementa funciones auxiliares ("*wrapper*") adicionales que solo reciban el parámetro n, y llamen a las funciones implementadas para calcular los números de Fibonacci. Por último, comprueba que producen las salidas correctas para $n = 1, \ldots, 10$.

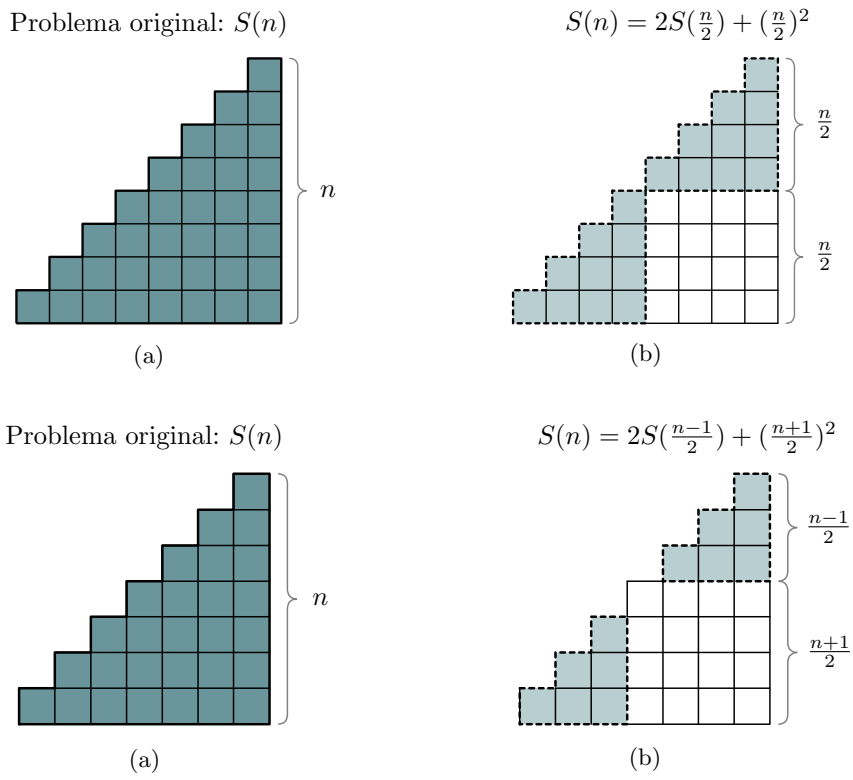
Ejercicio 3.11 Sea n un entero positivo. Considera el problema de determinar el número de bits puestos a 1 en la representación binaria de n (es decir, n expresado en base 2). Por ejemplo, para $n = 25_{10} = 11001_2$ (el subíndice indica la base en la que se expresa el número), el resultado es 3 bits. Indica el tamaño del problema (con palabras) y una expresión matemática para definirlo.

Ejercicio 3.12 Considera la función que suma los primeros n enteros positivos en:

$$S(n) = \begin{cases} 1 & \text{si } n = 1, \\ S(n-1) + n & \text{si } n > 1. \end{cases}$$

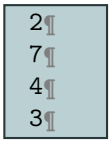
Define una función más general que pueda aplicarse también a números no negativos. En otras palabras, modifica la función para considerar que n puede ser 0. Por último, codifica la función.

Ejercicio 3.13 Usa diagramas similares a los siguientes:



para derivar definiciones recursivas de la suma de los primeros n enteros positivos ($S(n)$), donde el caso recursivo sume los resultados de cuatro subproblemas de aproximadamente la mitad del tamaño del problema original. Finalmente, define y codifica la función recursiva completa (caso base y casos cuando n es par e impar).

Ejercicio 3.14 Considera el problema de imprimir los dígitos de un número entero no negativo n en una consola, uno a uno, en vertical, y en orden "normal", donde el dígito más significativo aparezca en la primera línea, el segundo más significativo en la segunda línea, etc. Por ejemplo, si $n = 2743$ el programa debe imprimir las siguientes líneas en la consola:



Indica el tamaño del problema y su caso base. Dibuja un diagrama para un entero no negativo general $n = d_{m-1} \cdots d_1 d_0$, donde m es el número de dígitos de n, para poder ilustrarla descomposición del problema, y como recuperar la solución al problema original dada la solución de un subproblema. Por último, deriva el caso recursivo e implementa el método.

Ejercicio 3.15 Define un diagrama general que use un esquema de divide y vencerás para calcular el máximo de una lista **a** de n elementos. Usa notación general en lugar de ejemplos concretos.

Ejercicio 3.16 Define una función recursiva que calcule el máximo de una lista **a** de n elementos, en la que la descomposición simplemente reduzca el tamaño del problema en una unidad.

4. Ejercicios de recursividad lineal

Ejercicio 4.1 El siguiente código es una función booleana que determina si un número entero no negativo n es par:

```
def es_par(n):
    if n == 0:
        return True
    elif n == 1:
        return False
    else:
        return es_par(n - 2)
```

La descomposición empleada reduce el tamaño (n) en dos unidades. Define y codifica una versión alternativa donde la descomposición reduzca el tamaño del problema en una sola unidad.

Ejercicio 4.2 Implementa una función recursiva que calcule la potencia b^n en tiempo logarítmico, para una base real b, y un entero n, el cual puede ser negativo.

Ejercicio 4.3 Implementa una función recursiva que calcule la n-ésima potencia de una matriz cuadrada, que será otra matriz cuadrada de las mismas dimensiones. Usa el paquete de Python NumPy y los métodos :

- `identity`: devuelve la matriz identidad.
- `shape`: indica las dimensiones de una matriz.
- `dot`: multiplica matrices.

Por último, calcula las matrices:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

para $n = 1, \ldots, 10$, e imprime el elemento de la primera fila y segunda columna. ¿Reconoces estos números?

Ejercicio 4.4 En Python y otros lenguajes de programación las funciones pueden ser también parámetros de otras funciones. Define e implementa una función general recursiva que calcule el siguiente sumatorio::

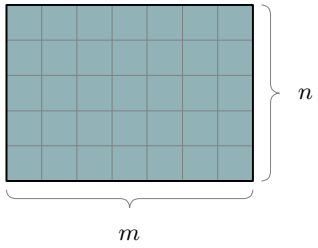
$$g(m, n, f) = \sum_{i=m}^{n} f(i) = f(m) + f(m+1) + \cdots + f(n-1) + f(n),$$

donde m y n son enteros, y f es una función. Úsala para calcular e imprimir:

$$\sum_{i=1}^{n} i^3,$$

para $n = 0, \ldots, 4$.

Ejercicio 4.5 Implementa funciones recursivas para calcular un "producto lento" entre dos números enteros no negativos m y n. Los algoritmos pueden sumar y restar números, pero no los pueden multiplicar (no puede usar el operador $*$). Usa descomposiciones que reduzcan uno o ambos parámetros de entrada en una unidad. Además, crea diagramas en los que se aprecie el proceso de diseño recursivo (descomposición y generación de la regla recursiva). Pueden ser diagramas donde el producto de m y n sea el área de un rectángulo de base m y altura n (el rectángulo tendrá mn bloques cuadrados de área 1 (cuadrados de 1×1), como se ilustra en la siguiente figura:



Ejercicio 4.6 Implementa una versión recursiva más eficiente del "producto lento" empleando una descomposición que divida a ambos parámetros de entrada por dos. Emplea diagramas rectangulares como el del ejercicio 4.5 para hallar los casos recursivos.

Ejercicio 4.7 Define e implementa una función recursiva que calcule el número de dígitos de un número entero n no negativo.

Ejercicio 4.8 Define e implementa una función recursiva que, dado un número decimal n, cuyos dígitos son ceros o unos, retorne el número cuya representación en binario fuera precisamente la secuencia de unos y ceros de n. Por ejemplo, si $n = 10110_{10}$, la función devolverá 22, ya que $10110_2 = 22$.

Ejercicio 4.9 Implementa un algoritmo que realice el cambio de base de decimal a una base b ($2 \leq b \leq 9$). Usa un diagrama general que ilustre el proceso mental de diseño de algoritmos recursivos.

Ejercicio 4.10 Considérese el problema del ejercicio 1.9. Define e implementa un algoritmo que, dado un número x de n bits, determine la posición de su bit menos significativo puesto a 1. Considera además que la posición del bit menos significativo es la 1. Por ejemplo, para $x = 01110\underline{1}00$ el bit menos significativo (más a la derecha) puesto a 1 está en la posición 3.

Ejercicio 4.11 Escribe un método recursivo que emplee la función desarrollada en el ejercicio 4.10 para resolver el ejercicio 1.9 de manera computacional. Imprime las soluciones para números expresados en $n = 1, \ldots, 5$ bits.

Ejercicio 4.12 Implementa una función recursiva que retorne el número de vocales en una cadena de caracteres.

Ejercicio 4.13 Considera el siguiente código para generar la n-ésima fila del triángulo de Pascal:

```
def pascal(n):
    if n == 0:
        return [1]
    else:
        fila = [1]

        fila_previa = pascal(n - 1)

        for i in range(len(fila_previa) - 1):
            row.append(fila_previa[i] + fila_previa[i + 1])

        fila.append(1)
        return fila
```

Reemplaza el bucle por una función recursiva. Deberá recibir una fila del triángulo de Pascal mediante una lista, y devolverá la lista con las sumas de sus elementos consecutivos. Por ejemplo, para la entrada $[1, 3, 3, 1]$ el resultado deberá ser $[4, 6, 4]$.

Ejercicio 4.14 Considera un número entero no negativo n cuyos dígitos siempre aparecen en orden ascendente de izquierda a derecha, como el 24667. En otras palabras, si $d_{m-1} \cdots d_1 d_0$ representa la secuencia de (m) dígitos de n, entonces $d_i \leq d_j$ para $i < j$. Dado un dígito adicional $0 \leq x \leq 9$, implementa una función que devuelva el entero que resulta de insertar x en n, de manera que los dígitos sigan apareciendo ordenados de izquierda a derecha. Por ejemplo, si $n = 24667$ y $x = 5$, la función deberá devolver 245667. Evita incluir casos base redundantes.

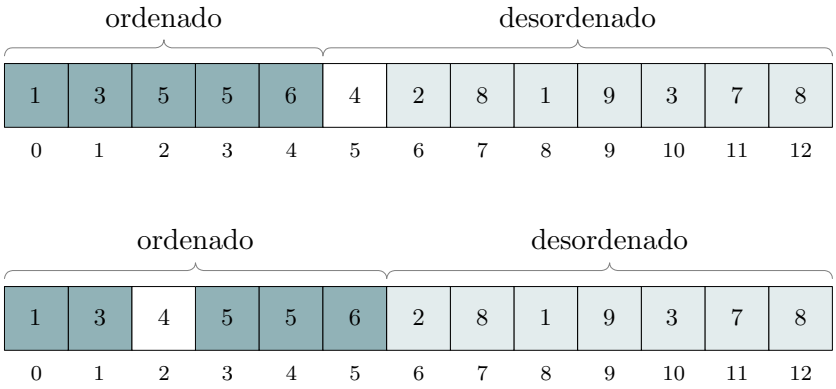
Ejercicio 4.15 Este ejercicio es similar al 4.14. Dada una lista ordenada de números en orden ascendente, y un número x cualquiera, implementa una función que inserte x en la lista de manera que ésta siga estando ordenada.

Ejercicio 4.16 Un coeficiente binomial $\binom{n}{m}$ es una función de dos parámetros enteros n y m, con $n \geq m \geq 0$. Define tres funciones recursivas lineales para calcular coeficientes binomiales basadas en descomposiciones que: (i) decrementen n en una unidad, (ii) decrementen m en una unidad, y (iii) decrementen tanto n como m en una unidad. Usa las siguientes definiciones para desarrollar los casos recursivos:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \text{ o } n = m, \\ \dfrac{n!}{m!(n-m)!} & \text{en otro caso,} \end{cases}$$

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \text{ o } n = m, \\ \dbinom{n-1}{m-1} + \dbinom{n-1}{m} & \text{en otro caso.} \end{cases}$$

Ejercicio 4.17 El algoritmo "*insertion sort*" ordena una lista aplicando repetidamente el procedimiento que se ilustra en la siguiente figura:



En la etapa i-ésima los elementos desde el índice 0 hasta el $i-1$ estarán ordenados correctamente, mientras el resto de la lista no estará (necesariamente) ordenada. Para proceder, el método inserta el elemento correspondiente al índice i en la sublista ordenada, de manera que la sublista desde el índice 0 al i quede ordenada. Si esta operación se realiza desde el índice (paso) 1 (o 0) hasta el $n-1$, la lista quedará ordenada al final del proceso. Implementa una variante recursiva del método para ordenar una lista, que use la función desarrollada en el ejercicio 4.15.

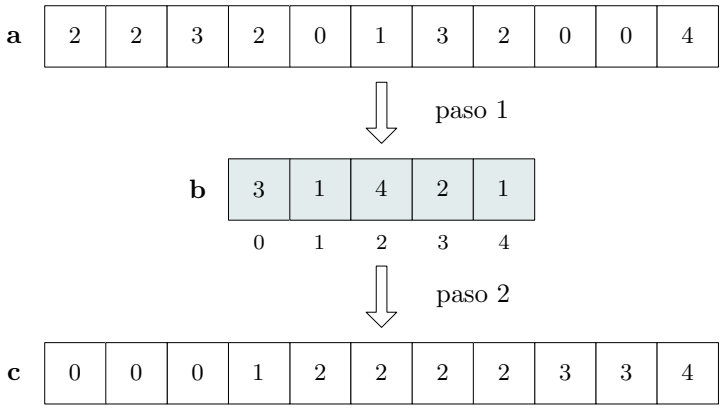
5. Ejercicios de recursividad por cola

Ejercicio 5.1 Define e implementa funciones booleanas, basadas en recursividad lineal y recursividad por cola, que determinen si un número entero no negativo contiene un dígito impar.

Ejercicio 5.2 Es más agradable mostrar polinomios en un estilo formateado que mediante una lista de coeficientes. Escribe un método que reciba una lista de longitud n que represente un polinomio de grado $n-1$, y lo imprima de manera formateada. Por ejemplo, dada la lista de entrada $\mathbf{p} = [3, -5, 0, 1]$, asociada al polinomio $x^3 - 5x^1 + 3$ (es decir, p_i es el coeficiente asociado al término x^i), el método deberá imprimir una línea similar a: `+ 1x^3 - 5x^1 + 3`. Asume que $p_{n-1} \neq 0$, salvo si el polinomio es la constante 0. Finalmente, indica su coste computacional en tiempo.

Ejercicio 5.3 El algoritmo "*counting sort*" es un método para ordenar una lista **a** de n enteros pertenecientes al intervalo $[0, k]$, donde k suele ser pequeño. Su coste en tiempo es $\mathcal{O}(n + k)$, lo cual implica que corre en tiempo $\mathcal{O}(n)$ si $k \in \mathcal{O}(n)$.

Dada una lista **a**, el método crea una lista **b** que contiene el número de apariciones de cada entero en **a**, como se muestra en el paso 1 de la siguiente figura:



Por ejemplo, $b_2 = 4$, ya que el entero 2 aparece cuatro veces en **a**. Implementa un procedimiento recursivo por cola que reciba las listas **a** and **b** (ésta inicializada con ceros), y rellene **b** con las apariciones de los enteros en **a**. Además, implementa una función recursiva lineal que reciba la lista de apariciones **b**, y devuelva una nueva lista (**c**) que sea la versión ordenada (de menor a mayor) de **a**, como se mostraba en el paso 2 de la figura anterior. Finalmente, implementa una función que llame a las dos anteriores para codificar esta versión recursiva del algoritmo *counting sort*, y especifica su coste computacional en espacio y en tiempo.

Ejercicio 5.4 Implementa una función que busque al elemento con la clave más pequeña en un árbol de búsqueda binario T, el cual se define como una lista de cuatro componentes: clave, elemento, subárbol izquierdo y subárbol derecho.

Ejercicio 5.5 Implementa una versión alternativa del siguiente código asociado a una búsqueda binaria:

```
def busqueda_binaria(a, x, inf, sup):
    if inf > sup:  # lista vacia
        return -1
    else:
        mitad = (inf + sup) // 2

        if x == a[mitad]:
            return mitad
        elif x < a[mitad]:
            return busqueda_binaria(a, x, inf, mitad - 1)
        else:
            return busqueda_binaria(a, x, mitad + 1, sup)

def busqueda_binaria_wrapper(a, x):
    return busqueda_binaria(a, x, 0, len(a) - 1)
```

que no use el parámetro `sup`. Especifica también su coste computacional en tiempo.

Ejercicio 5.6 Implementa una función booleana que realice una búsqueda binaria de un elemento x en una lista ordenada $\mathbf{a}$. La función retornará `True` si x está presente en $\mathbf{a}$. La función solo debe tener dos parámetros de entrada: x y $\mathbf{a}$. Además, descompondrá el problema en otro de la mitad de su tamaño. Especifica también el coste computacional en tiempo de la función.

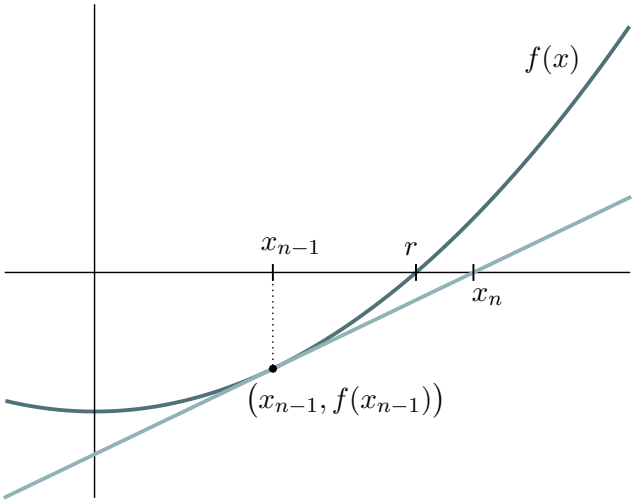
Ejercicio 5.7 Sea $\mathbf{a}$ una lista ordenada en orden creciente de enteros diferentes. Se pide desarrollar un algoritmo eficiente para buscar un elemento en una lista $\mathbf{a}$ que tenga el mismo valor que el índice en el que se encuentra. Es decir, queremos encontrar un elemento i para el que $a_i = i$. Por ejemplo, si $\mathbf{a} = [-3, -1, 2, 5, 6, 7, 9]$, entonces la función devolverá un 2, ya que $a_2 = 2$. Observa que el primer elemento se localiza en la posición 0 de la lista. Por simplicidad, asume que tendrá como mucho un elemento que satisfaga la propiedad $a_i = i$. Si la lista no contiene un elemento que la satisfaga la función deberá retornar un -1. Finalmente, especifica el coste computacional en tiempo de la función.

Ejercicio 5.8 Sea $\mathbf{a}$ una lista de n enteros, organizados de manera que los pares aparezcan antes que los impares (es decir, el índice asociado a cualquier elemento par será menor que el índice de un elemento impar). Por ejemplo: la lista $\mathbf{a} = [2, -4, 10, 8, 0, \mathbf{12}, 9, 3, -15, 3, 1]$ cumple esa condición. Se pide desarrollar un algoritmo recursivo eficiente para calcular el mayor índice asociado a elementos pares. En otras palabras, se busca el índice i para el que a_i es par, pero donde a_j será impar para todo $j > i$. En el ejemplo el método retornaría 5, ya que $a_5 = 12$ es par, pero a_6, a_7, a_8, etc., son impares. Si $\mathbf{a}$ no contuviera números pares la función deberá retornar -1. Por otro lado, si la lista solo contiene números pares deberá devolver $n - 1$. Finalmente, especifica el coste computacional en tiempo de la función.

Ejercicio 5.9 El método de Newton-Raphson es una estrategia para encontrar una aproximación a la raíz de una función real $f(x)$ que sea derivable, es decir, un valor $\hat{z}$ donde $f(\hat{z})$ sea cero o tome un valor muy pequeño. Está basada en la siguiente regla recursiva:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})},$$

explicada geométricamente en la siguiente figura:



Considera una aproximación inicial x_{n-1} de la raíz r de la función $f(x)$. El procedimiento encuentra la recta tangente a $f(x)$ en x_{n-1}, y calcula una nueva aproximación (x_n) de r que es el valor de$x$ en el que la recta corta al eje de abscisas (observa que x_n se encuentra más cerca de r que x_{n-1}). Por tanto, dado un valor inicial x_0, $\hat{z} = x_n$ será la aproximación de la raíz de $f(x)$ tras aplica la regla recursiva n veces.

Este procedimiento se puede usar, por ejemplo, para hallar aproximaciones muy precisas de la raíz cuadrada de un número, en solo unos pocos pasos. Supóngase que se desea calcular $\sqrt{a}$, que será algún valor x *a priori* desconocido. En ese caso tenemos las identidades: $x = \sqrt{a}$, donde $x^2 = a$, lo cual implica que $x^2 - a = 0$. Por tanto, la raíz de la parábola $x^2 - a$ será la raíz cuadrada de a. En ese caso, la fórmula recursiva asociada al método es:

$$x_n = x_{n-1} - \frac{x_{n-1}^2 - a}{2x_{n-1}}.$$

Implementa una función recursiva lineal y otra recursiva por cola que reciban el valor de a, una estimación inicial de su raíz cuadrada x_0 y un cierto número de pasos n, y retorne la estimación $\hat{z} = x_n$ de $\sqrt{a}$ aplicando la regla recursiva anterior n veces. Finalmente, especifica su coste computacional en tiempo.

6. Ejercicios de divide y vencerás

Ejercicio 6.1 Implementa una función booleana basada en divide y vencerás que determine si una lista **a** contiene un elemento concreto x.

Ejercicio 6.2 Sea **a** una lista de n enteros no negativos. Escribe una función basada en divide y vencerás que retorne el conjunto de dígitos que aparezcan en todos los elementos de **a**, y determina su coste computacional en tiempo. Por ejemplo, para $\mathbf{a} = [2348, 1349, 7523, 3215]$, la solución es $\{3\}$. La función a implementar deberá llamar a otra que devuelva el conjunto de dígitos de un entero no negativo. Implementa esta función también.

Ejercicio 6.3 El problema de la máxima sublista consiste en encontrar la sublista de elementos contiguos de una lista tal que la suma de sus elementos sea la mayor de todas. Por ejemplo, para la lista $[-1, -4, 5, 2, -3, 4, 2, -5]$, la sublista óptima es $[5, 2, -3, 4, 2]$, cuyos elementos suman 10. Dada una lista no vacía **a** de números, implementa una función de divide y vencerás que retorne la suma de los elementos de su máxima sublista.

Ejercicio 6.4 Diseña un algoritmo recursivo rápido para multiplicar polinomios basada en la técnica de divide y vencerás, que sea análoga al enfoque empleado por el algoritmo de Karatsuba. Usa listas para codificar los polinomios. Por ejemplo, el polinomio de grado $n-1$: $P = u_{n-1}x^{n-1} + \cdots + u_1 x + u_0$ se indicará mediante una lista **a** de longitud n, donde el elemento $a[i]$ contendrá el valor del coeficiente u_i. Por último, el método necesitará métodos para sumar y restar polinomios. Implementa estas funciones también.

Ejercicio 6.5 Implementa una función recursiva que reciba una matriz ($\mathbf{A}$) de dimensiones $n \times m$ y devuelva su traspuesta ($\mathbf{A}^\intercal$), de dimensiones $m \times n$. Utiliza el paquete NumPy, y realiza una descomposición basada en divide y vencerás que considere que $\mathbf{A}$ está formada por cuatro submatrices "bloque", resultado de dividir m y n por dos:

$$\mathbf{A} = \left[\begin{array}{c|c} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \hline \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{array}\right].$$

En ese caso, la traspuesta de $\mathbf{A}$ se define como:

$$\mathbf{A}^\intercal = \left[\begin{array}{c|c} \mathbf{A}_{1,1}^\intercal & \mathbf{A}_{2,1}^\intercal \\ \hline \mathbf{A}_{1,2}^\intercal & \mathbf{A}_{2,2}^\intercal \end{array}\right].$$

Ejercicio 6.6 Implementa el algoritmo de multiplicación de matrices de Strassen. El código deberá contener una función auxiliar (*wrapper*) que permita multiplicar una matriz de dimensiones $p \times q$ por otra de dimensiones $q \times r$, rellenando las matrices de entrada con ceros.

Ejercicio 6.7 Implementa una función de divide y vencerás para multiplicar matrices basada en la siguiente descomposición de las matrices de entrada:

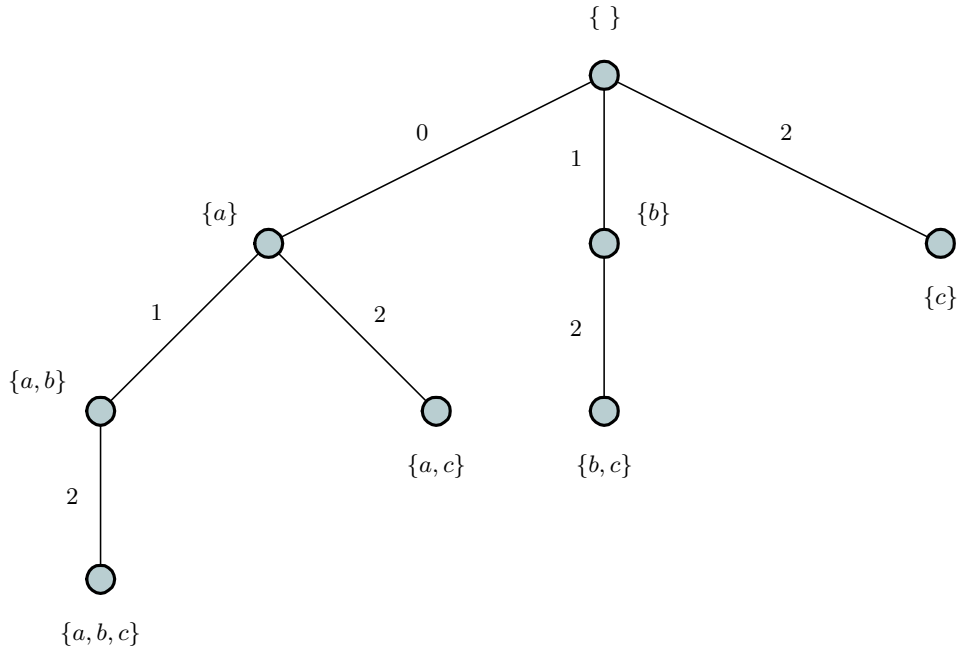
$$\mathbf{A} \cdot \mathbf{B} = \left[\begin{array}{c|c} \mathbf{A}_1 & \mathbf{A}_2 \end{array}\right] \cdot \left[\begin{array}{c} \mathbf{B}_1 \\ \hline \mathbf{B}_2 \end{array}\right] = \left[\begin{array}{c} \mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_2\mathbf{B}_2 \end{array}\right].$$

Ejercicio 6.8 Implementa una función de divide y vencerás para multiplicar matrices basada en la siguiente descomposición de las matrices de entrada:

$$\mathbf{A} \cdot \mathbf{B} = \left[\begin{array}{c} \mathbf{A}_1 \\ \hline \mathbf{A}_2 \end{array}\right] \cdot \left[\begin{array}{c|c} \mathbf{B}_1 & \mathbf{B}_2 \end{array}\right] = \left[\begin{array}{cc} \mathbf{A}_1\mathbf{B}_1 & \mathbf{A}_1\mathbf{B}_2 \\ \mathbf{A}_2\mathbf{B}_1 & \mathbf{A}_2\mathbf{B}_2 \end{array}\right].$$

7. Ejercicios de vuelta atrás - *backtracking*

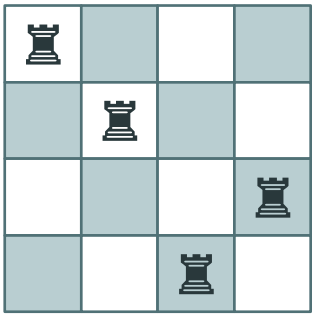
Ejercicio 7.1 Existen varias formas de obtener todos los subconjuntos de un conjunto inicial. Una forma sencilla consiste en generar un árbol binario, donde en cada nodo se decida si se incorpora un elemento del conjunto inicial a un subconjunto. El objetivo en este ejercicio es implementar un procedimiento alternativo, cuyo árbol de recursión sea como el de la siguiente figura:



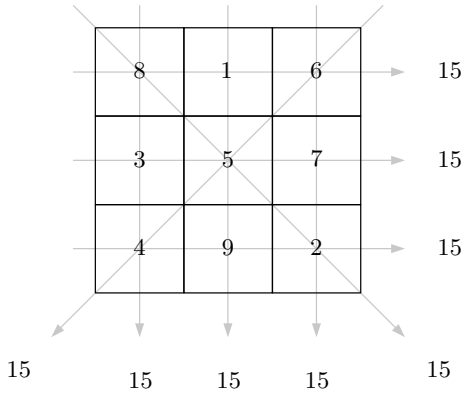
Como puede apreciarse, se genera un subconjunto en cada nodo del árbol. Por tanto, el árbol tiene exactamente 2^n nodos, donde n es la cardinalidad del conjunto inicial. Asumiendo que el conjunto inicial viene especificado en una lista de longitud n, observa que las etiquetas del árbol hacen referencia a los índices de la lista (en la figura la lista de entrada sería $[a, b, c]$). Además, la solución parcial será una lista que contenga los índices de los elementos que forman un subconjunto. En el ejemplo, la lista $[0, 2]$ representará al conjunto $\{a, c\}$. Por tanto, las soluciones parciales también serán soluciones completas. Finalmente, el método desarrollado deberá imprimir los elementos de cada subconjunto.

Ejercicio 7.2 Implementa una función de backtracking que cuente el número de soluciones válidas de un Sudoku que, posiblemente, no esté bien planteado (es decir, que pueda tener 0, 1, o más soluciones).

Ejercicio 7.3 Implementa un algoritmo de backtracking que imprima todas las soluciones del problema de las n torres de ajedrez. Se trata de un problema análogo al de las n reinas, pero usa torres en lugar de reinas. Como las torres solo se pueden mover de manera vertical u horizontal, dos o más torres no pueden estar en la misma columna ni fina. Sin embargo, puede haber torres en la misma diagonal. La siguiente figura muestra una solución al problema para 4 torres:

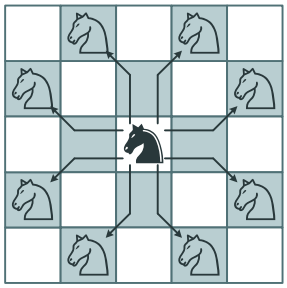


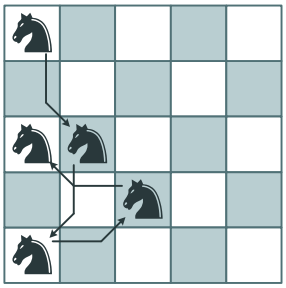
Ejercicio 7.4 Un cuadrado mágico es una rejilla (o matriz) de $n \times n$, que contiene los primeros n^2 enteros positivos tales que la suma de los elementos en cada fila, columna, y diagonales sea la misma. Esa cantidad se denomina "constante mágica" y es $n(n^2+1)/2$ para un n general. La siguiente figura muestra un cuadrado mágico de dimensiones 3×3, donde la constante mágica es 15:



Diseña un algoritmo basado en backtracking para imprimir todos los posibles cuadrados mágicos de dimensiones 3×3. Nota: no es necesario implementar un método para un n general. ¿Qué problema podríamos tener al intentar calcular cuadrados mágicos más grandes?

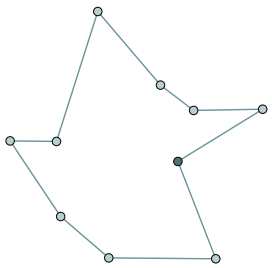
Ejercicio 7.5 Un caballo en ajedrez puede saltar desde una casilla a otra en patrones de tipo L, como se muestra en la siguiente figura (izquierda).





En la derecha se muestran cuatro saltos consecutivos de una caballo que empieza en la celda superior-izquierda. El problema de la "ruta del caballo" consiste en determinar una secuencia de saltos de un caballo, en un tablero de dimensiones $n \times n$, que empiece en una casilla determinada, y consiga visitar el resto de casillas, sin pasar dos o más veces por la misma. Implementa un algoritmo de backtracking que encuentre una ruta del caballo, y prueba el código para $n = 5$ y $n = 6$. Además de la n inicial, el método recibirá las coordenadas de la casilla inicial en el tablero. Por último, no es necesario encontrar una ruta "cerrada", tal que se pueda saltar desde la casilla final a la inicial con unu movimiento legal del caballo.

Ejercicio 7.6 El problema del viajante de comercio ("*traveling salesman problem*" o simplemente TSP), es un problema clásico de optimización discreta. Dadas n ciudades, el objetivo es hallar el camino más corto que visite cada ciudad una sola vez y retorne a la ciudad inicial. La siguiente figura ilustra un ejemplo del camino más corto para 10 ciudades:

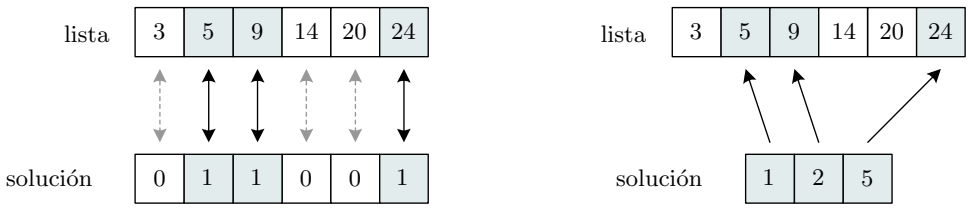


Implementa un algoritmo de backtracking que resuelva el problema. Asume que las posiciones de las ciudades se especifican en un fichero de texto, donde cada línea contiene las coordenadas x e y de una ciudad en un mapa bidimensional. El fichero se puede leer empleando el método `loadtxt` del paquete NumPy. La ciudad inicial será la de la primera línea del fichero. Además, considera distancias euclídeas entre las ciudades (el viajante iría de una ciudad a otra en línea recta). Estas distancias se pueden calcular con el método `pdist` del paquete SciPy. En concreto, el método recursivo de backtracking deberá recibir una matriz de distancias entre las ciudades (las coordenadas x e y son irrelevantes si se dispone de las distancias). Finalmente, prueba el código con ejemplos donde $n \leq 10$.

Ejercicio 7.7 El siguiente problema, denominado "*tug of war*" en inglés, busca dividir un conjunto de elementos en dos lo más "parecidos" como sea posible. Formalmente, dada una lista no vacía de n números (que corresponderían a valoraciones de los elementos del conjunto original), donde n es par, el objetivo consiste en dividirla en dos de $n/2$ elementos, de manera que se minimice la diferencia absoluta entre la suma de los elementos de cada sublista. Por ejemplo, la forma óptima de dividir la lista $[3, 5, 9, 14, 20, 24]$ consiste en considerar las sublistas $[5, 9, 24]$, y $[3, 14, 20]$. La suma de sus elementos es 38 y 37, respectivamente, y la diferencia absoluta entre esas sumas es 1.

Implementa un algoritmo basado en backtracking que genere las sublistas que resuelven el problema. La solución puede ser una lista binaria de longitud n con $n/2$ ceros y $n/2$ unos. Los ceros estarían asociados a una sublista y los unos a la otra. En el ejemplo anterior la solución sería la lista $[0, 1, 1, 0, 0, 1]$ ($[1, 0, 0, 1, 1, 0]$ sería equivalente).

Además, implementa un algoritmo más eficiente donde la solución sea una lista de longitud $n/2$ que contenga los índices (en orden creciente) de la lista original en los que se encuentran los elementos de una de las sublistas. Por ejemplo, la sublista $\{5, 9, 24\}$ se representaría mediante la lista de índices $[1, 2, 5]$. La siguiente figura ilustra las dos formas de representar la solución:



Ejercicio 7.8 El problema de la suma de subconjuntos consiste en determinar los subconjuntos de uno inicial compuesto de números positivos, y especificado a través de una lista **L**, tal que la suma de sus elementos sea igual a una cantidad s. Implementa una función recursiva alternativa basada en backtracking que calcule el subconjunto válido (sus elementos suman s) que además tenga la menor cardinalidad. El método almacenará el conjunto óptimo a través de una lista, y retornará la cardinalidad óptima (menor). Por ejemplo, para el conjunto $\{1, 2, 3, 5, 6, 7, 9\}$ y $s = 13$, el método almacenará el conjunto $\{6, 7\}$, y devolverá el valor 2. Además, la solución no tiene por qué ser única, pero la cardinalidad óptima sí lo es. Por ejemplo, para $\{2, 6, 3, 5\}$ y $s = 8$, tanto $\{2, 6\}$ como $\{3, 5\}$ son subconjuntos óptimos. Codifica la solución parcial (y óptima) como una lista de dígitos binarios. Además, poda el árbol de recursión considerando la validez de una solución y la mejor cardinalidad encontrada hasta el momento. Por último. codifica una función auxiliar *wrapper* que calcule el subconjunto óptimo y lo imprima en caso de existir.

8. Ejercicios de algoritmos voraces

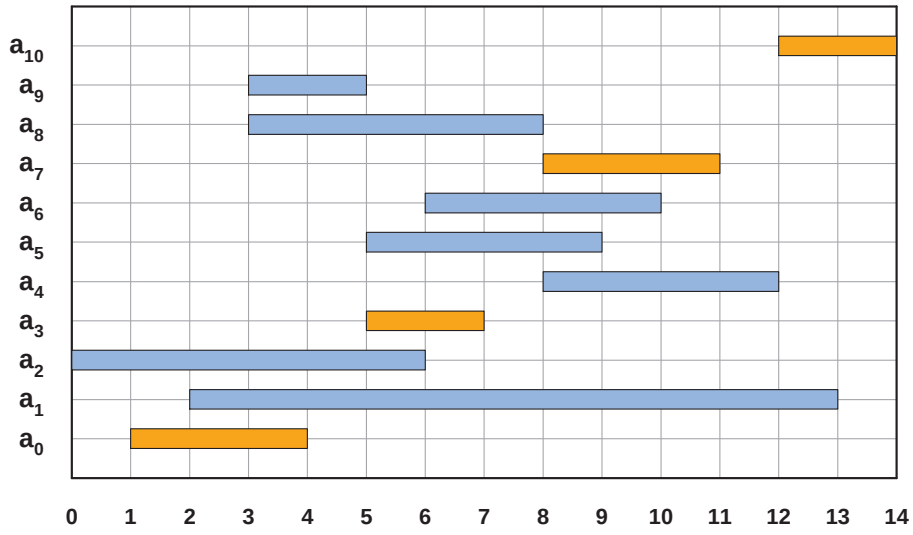
Ejercicio 8.1 Sea un conjunto A de n actividades $\{a_0, a_1, \ldots, a_{n-1}\}$ que necesitan utilizar un recurso común, por ejemplo, una sala de reuniones. El recurso solo puede ser usado por una actividad en cada momento. Cada actividad tiene un instante de comienzo c_i y un instante de finalización f_i, donde $0 \leq c_i < f_i < \infty$.

Si se selecciona la actividad a_i, se desarrolla en el intervalo semiabierto de tiempo $[c_i, f_i)$. Las actividades a_i y a_j son compatibles si sus intervalos $[c_i, f_i)$ y $[c_j, f_j)$ no se solapan, es decir, si $c_i \geq f_j$ o $c_j \geq f_i$.

El objetivo de este ejercicio es elaborar e implementar una estrategia voraz para seleccionar un subconjunto de actividades compatibles cuya cardinalidad sea máxima.

Por ejemplo, sea el siguiente conjunto de actividades:

i	0	1	2	3	4	5	6	7	8	9	10
c_i	1	2	0	5	8	5	6	8	3	3	12
f_i	4	13	6	7	12	9	10	11	8	5	14



Un subconjunto S de actividades compatibles es $\{a_2, a_4, a_{10}\}$. Sin embargo, no es un subconjunto de cardinalidad máxima, como lo son $\{a_0, a_3, a_7, a_{10}\}$ (en naranja en la figura) y $\{a_9, a_3, a_4, a_{10}\}$ (ya que tienen 4 elementos).

Ejercicio 8.2 Propón varias estrategias voraces para el problema de la mochila-01. Impleméntalas y realiza una simulación con 1000 instancias del problema, donde el número de objetos n varíe entre 10 y 100, y los pesos y valores estén en el intervalo $(0, 10)$. Tanto n como los pesos y valores se escogerán aleatoriamente de distribuciones uniformes. ¿Hay alguna estrategia óptima o que ofrezca mejores resultados en general?

Ejercicio 8.3 Se pide diseñar un método **recursivo** que implemente la estrategia voraz óptima para el problema del cambio de monedas. Dada una cierta cantidad de dinero x, el algoritmo debe imprimir por pantalla los valores de las monedas tal que su suma sea igual a x, teniendo en cuenta que:

- Los posibles valores de las monedas son: {2, 1, 0,5, 0,2, 0,1, 0,05, 0,02, 0,01}
- Se usará el mínimo número de monedas posible
- Se asumirá que $x \geq 0{,}01$ y que x estará redondeado a dos cifras decimales

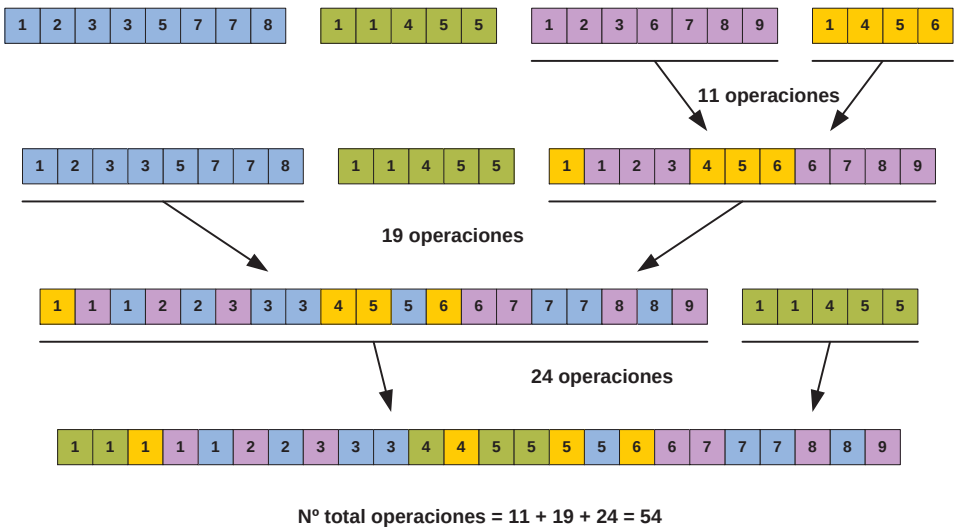
Por ejemplo, para $x = 5{,}34$ el método debe imprimir la secuencia:

2 2 1 0,2 0,1 0,02 0,02

El método será recursivo, pero podrá usar bucles.

Ejercicio 8.4 Este problema está relacionado con la eficiencia de una generalización del algoritmo de ordenación *mergesort*, en el que la lista original a ordenar se descompone en varias sublistas de tamaños diferentes. El algoritmo emplea un método para combinar las sublistas mezclando parejas de éstas progresivamente hasta ordenar la lista original.

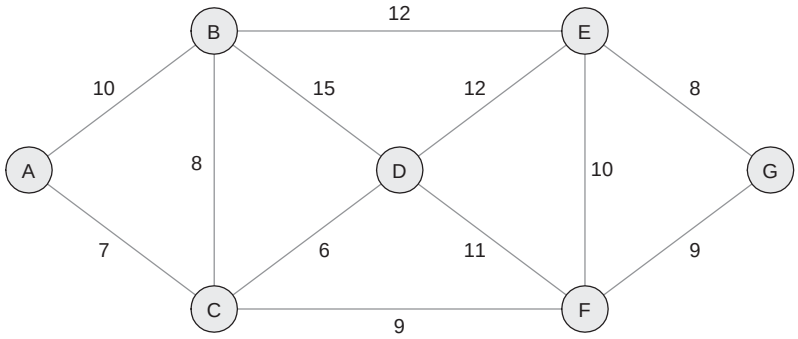
Este ejercicio consiste implementar un método para hallar el número mínimo de operaciones que necesitaría una estrategia (voraz) óptima para ordenar la lista original, cuando se desea mezclar n listas ordenadas, las cuales pueden tener tamaños diferentes. Se supondrá que se realiza una operación por cada elemento a ordenar de una pareja de sublistas.



En el ejemplo de la figura partimos de cuatro sublistas de tamaños 8, 5, 7 y 4. Al mezclar por parejas de la forma ilustrada el número total de operaciones realizadas es 54. Sin embargo, esa estrategia no es óptima. Hay una forma más eficiente de mezclar que lleva a cabo solo 48 operaciones.

Ejercicio 8.5 Sea una lista **a** que contenga las frecuencias de aparición de n caracteres en un documento. Implementa un programa que calcule los códigos de Huffman asociados a los caracteres a partir de las frecuencias en **a**.

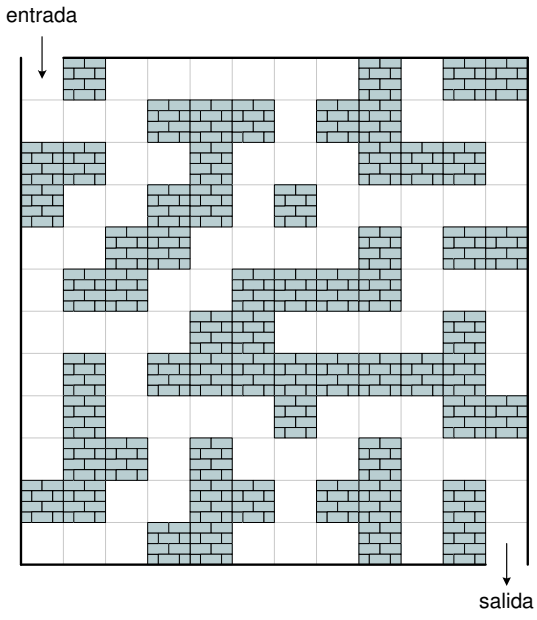
Ejercicio 8.6 Halla un árbol de recubrimiento de coste mínimo para el siguiente grafo ponderado empleando las estrategias de Prim y Kruskal.



Ejercicio 8.7 Implementa el algoritmo de Prim para hallar el árbol de recubrimiento de coste mínimo (*minimum spanning tree*).

Ejercicio 8.8 Implementa el algoritmo de Kruskal para hallar el árbol de recubrimiento de coste mínimo (*minimum spanning tree*).

Ejercicio 8.9 Implementa el algoritmo de Dijkstra y úsalo para hallar el camino más corto desde la entrada a un laberinto (definido sobre una rejilla rectangular como el de la figura de abajo) hasta la correspondiente casilla de salida.



9. Ejercicios adicionales de recursividad

Ejercicio 9.1 Se pide implementar un método recursivo que, dado un determinado entero no negativo n, calcule la siguiente suma:

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

Ejercicio 9.2 En este ejercicio se pide implementar una versión recursiva del **algoritmo de Horner**, que sirve para evaluar polinomios de manera eficiente. En concreto, evalúa un polinomio de grado n utilizando solamente n multiplicaciones. La clave detrás del algoritmo de Horner es la descomposición de un polinomio $p(x)$ de grado n de la siguiente manera:

$$p(x, \mathbf{d}) = d_0 + x(d_1 + x(d_2 + \cdots + x(d_{n-1} + d_n x) \cdots))$$

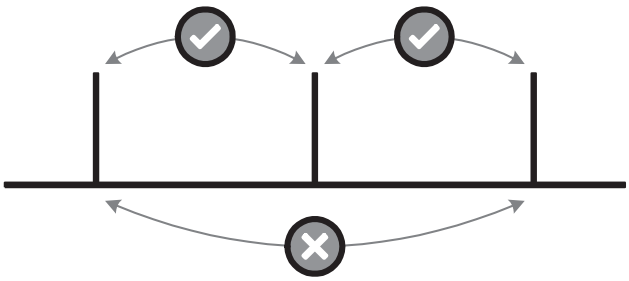
donde d representa los coeficientes del polinomio de grado n. En concreto, d_0 es la constante, mientras que d_n es el coeficiente asociado al término x^n. El método recibirá la lista de coeficientes $\mathbf{d}$ que definen el polinomio, y el número real x.

Ejercicio 9.3 Se pide implementar la siguiente función mediante métodos recursivos, para un entero positivo n:

$$f(n) = \begin{cases} 1 & \text{si } n = 1, 2 \\ 1 + \sum_{i=1}^{n-2} f(i) & \text{si } n \geq 3 \end{cases}$$

El sumatorio en el caso recursivo se puede implementar de dos maneras: (1) usando un bucle, y (2) mediante una segunda función recursiva. Implementad ambos casos.

Ejercicio 9.4 Asuma que las tres torres están dispuestas de izquierda a derecha. En esta variante las reglas del problema son las mismas que las del original, pero NO se permite mover discos directamente desde la torre de la izquierda a la de la derecha, ni viceversa, como ilustra la siguiente figura:



Dados n discos ubicados en la torre de la izquierda, implementa un procedimiento que resuelva el problema, y mueva los n discos a la torre de la derecha.

Ejercicio 9.5 Implementa un método recursivo que corra en tiempo logarítmico para buscar un elemento x en una lista o array ordenado **a**. El método devolverá el índice en el que se encuentre x dentro de la lista. Es decir, si $\mathbf{a}[i] = x$ el método devolverá i. Si x aparece varias veces en la lista basta con devolver cualquier índice i que satisfaga $\mathbf{a}[i] = x$. Si x no se encuentra en la lista el resultado será -1.

Ejercicio 9.6 Implementa un método recursivo que, dado un entero positivo n, imprima por pantalla una pirámide de n filas con la siguiente estructura (para $n = 4$):

```
   X
  XXX
 XXXXX
XXXXXXX
```

El método puede usar funciones auxiliares recursivas.

Ejercicio 9.7 Implementa el método recursivo de bisección para hallar una raíz de una función continua $f(x)$ en un intervalo (a, b). Para aplicar el método sabemos que $f(a) \cdot f(b) < 0$. Es decir, tiene diferentes signos en los extremos del intervalo, lo cual implica que debe tener al menos un cero en (a, b). El método debe retornar un valor z en cuanto encuentre un valor para el cual $|b - a| < 2\varepsilon$, donde ε es un valor pequeño que se pasará al método como parámetro.

Ejercicio 9.8 Implementa el método recursivo que, dada una cadena **s**, encuentre la subcadena más larga contenida en **s** que constituya un palíndromo (es decir, que sea "capicúa"). El método podrá llamar a otro auxiliar Booleano que determine si una cadena es un palíndromo.