

Apuntes de Análisis y Diseño de Algoritmos (ADA)

Juan Llinares Mauri

jlml09@alu.ua.es

Contents

| | | |
|----------|--|----------|
| 1 | Introducción | 4 |
| 2 | Eficiencia | 5 |
| 2.1 | Noción de complejidad | 5 |
| 2.1.1 | Tamaño de un problema | 5 |
| 2.2 | Tiempo de ejecución | 5 |
| 2.2.1 | Análisis empírico | 6 |
| 2.2.2 | Análisis teórico | 6 |
| 2.2.3 | Ejemplos de complejidades | 6 |
| 2.3 | Cotas de complejidad | 8 |
| 2.3.1 | Ejemplos cota superior e inferior | 8 |
| 2.4 | Análisis asintótico | 8 |
| 2.5 | O - Cota superior. Peor caso. | 10 |
| 2.5.1 | Propiedades de las cotas superiores (Notación O) | 10 |
| 2.6 | Ω - Cota inferior. Mejor caso. | 11 |
| 2.6.1 | Propiedades de las cotas inferiores (Notación Ω) | 11 |
| 2.7 | Θ - Coste promedio o exacto. Caso promedio. | 12 |
| 2.7.1 | Propiedades de las cotas promedio (Notación Θ) | 13 |
| 2.7.2 | Jerarquías de Funciones | 13 |
| 2.8 | ¿Verdadero o falso? | 14 |
| 2.9 | Cálculo de complejidades | 14 |
| 2.10 | Quicksort - Ordenación por partición | 16 |
| 2.10.1 | QuickSort mediana | 18 |
| 2.11 | HeapSort - Ordenación por montículo | 18 |
| 2.11.1 | El <i>heap</i> | 19 |

| | |
|---|-----------|
| 3 Divide y vencerás | 22 |
| 3.1 Análisis de eficiencia | 22 |
| 3.2 Teorema Maestro | 23 |
| 3.3 Mergesort | 23 |
| 3.4 Torres de Hanoi | 24 |
| 3.5 Selección del k-ésimo mínimo | 25 |
| 3.6 Búsqueda binaria | 25 |
| 4 Programación dinámica | 26 |
| 4.1 Problema de la mochila | 26 |
| 4.1.1 Subestructura óptima | 26 |
| 4.1.2 Complejidades | 28 |
| 4.2 Corte de tubos | 28 |
| 4.2.1 Complejidades | 28 |
| 4.3 Divide y Vencerás a Programación Dinámica | 29 |
| 4.4 PD recursiva vs PD iterativa | 30 |
| 4.5 Cálculo del coeficiente binomial | 30 |
| 5 Algoritmos voraces | 31 |
| 5.1 Ejemplos | 31 |
| 5.1.1 Problema de la mochila continua: | 31 |
| 5.1.2 Problema de la mochila discreta (aproximación voraz): | 32 |
| 5.1.3 Problema del cambio: | 32 |
| 5.1.4 Algoritmo de Prim | 32 |
| 5.1.5 Algoritmo de Kruskal | 32 |
| 5.1.6 Fontanero diligente: | 33 |
| 5.1.7 Asignación de tareas: | 33 |
| 6 Vuelta atrás | 34 |
| 6.1 Ajuste de podas | 34 |
| 6.2 Definición | 34 |
| 6.3 Podas | 35 |
| 6.3.1 Cota optimista | 35 |
| 6.3.2 Cota pesimista | 35 |
| 6.4 Mochila discreta | 35 |
| 6.5 Problemas | 36 |
| 6.5.1 Permutaciones | 36 |
| 6.5.2 El viajante de comercio | 36 |
| 6.6 N Reinas | 37 |
| 6.7 Función compuesta mínima | 37 |
| 7 Ramificación y poda | 39 |
| 7.1 Aspectos en común con vuelta atrás | 39 |
| 7.2 Definiciones | 39 |
| 7.3 Funcionamiento | 39 |
| 7.4 Esquema de ramificación y poda | 40 |
| 7.5 Viajero de comercio | 41 |

| | | |
|----------|---|-----------|
| 7.6 | Función compuesta mínima | 41 |
| 7.7 | Problemas | 42 |
| 7.7.1 | El Puzzle | 42 |
| 8 | Prácticas | 43 |
| 8.1 | Práctica 2 - Complejidad temporal: Análisis empírico (II) | 43 |
| 8.2 | Práctica 3 - Complejidad temporal: Cálculo analítico | 44 |
| 8.2.1 | Ejercicio 1 (resuelto) | 44 |
| 8.2.2 | Ejercicio 2 | 45 |
| 8.2.3 | Ejercicio 3 | 46 |
| 8.2.4 | Soluciones | 47 |
| 8.3 | Práctica 4 - Complejidad temporal: Cálculo analítico (II) | 52 |
| 8.3.1 | Ejercicio 1 (resuelto) | 52 |
| 8.3.2 | Ejercicio 2 | 53 |
| 8.3.3 | Ejercicio 3 | 53 |
| 8.3.4 | Soluciones | 54 |
| 8.4 | Práctica 5 - Divide y vencerás | 59 |
| 8.5 | Práctica 6 - El problema del laberinto | 61 |
| 8.5.1 | 6.1 - El problema del laberinto I | 61 |
| 8.5.2 | 6.2 - El problema del laberinto II | 61 |
| 8.5.3 | Naive | 61 |
| 8.5.4 | Memoización | 62 |
| 8.5.5 | Iterativo con PD (Matriz) | 64 |
| 8.5.6 | Iterativo con PD (Vector) | 65 |
| 8.6 | Práctica 7 - El problema del laberinto II | 68 |
| 8.7 | Práctica 8 - El problema del laberinto III | 69 |
| 8.8 | Práctica Final - Camino de coste mínimo y IV | 73 |
| 9 | Repaso | 76 |
| 9.1 | Primer parcial | 76 |
| 9.2 | Segundo parcial | 77 |
| 9.3 | Final | 79 |
| 9.4 | Exámenes (míos) | 107 |
| 9.4.1 | Primer parcial (2024/25) | 108 |
| 9.4.2 | Segundo parcial (2024/25) | 109 |
| 9.5 | Preguntas de otros años | 110 |

1. Introducción

La **algoritmia** es la ciencia que trata el estudio de los algoritmos.



Es un proceso circular en el que cada etapa alimenta a la siguiente y a su vez retroalimenta a las anteriores.

Un **algoritmo** se define como una serie **finita** de instrucciones no ambiguas que expresa un método de resolución de un problema.

Para realizar el análisis de algoritmos se ha de tener claro qué recursos se van a analizar:

1. **Tiempo** que un algoritmo necesita para su ejecución.
2. **Espacio** que un algoritmo consume.

La finalidad de estos análisis es comparar y valorar los algoritmos.

2. Eficiencia

Para poder hablar de eficiencia es importante tener en cuenta la **máquina** donde se ejecutará el algoritmo, los **recursos** que el algoritmo necesitará en cada paso y que el algoritmo debe **terminar**, es decir, que tenga un número de pasos finitos.

2.1. Noción de complejidad

La **complejidad algorítmica** es una medida de los recursos que necesita un algoritmo para resolver un problema. Los más usuales son el **tiempo**¹ la **memoria**².

2.1.1. Tamaño de un problema

Se suele expresar en función de la dificultad del problema algo representativo de la misma. El tamaño del problema será lo que ocupa su representación. También puede haber un parámetro representativo del mismo. Ejemplos en la tabla 1.

| Problema | Tamaño del problema |
|--|---------------------|
| Sumar 1 a un entero ³ | 32 |
| Elegir el mayor de 2 enteros | $2 * 32$ |
| Ordenar un vector de n enteros | $32n$ |
| Multiplicar dos matrices de enteros de $m * n$ y $n * l$ | $32(mn + nl)$ |

Table 1. Ejemplos de tamaños de problema.

La complejidad depende de cómo se codifique el problema.

2.2. Tiempo de ejecución

El tiempo de ejecución depende de los siguientes factores:

- Externos:
 - Máquina donde se ejecuta.
 - Compilador.
 - Datos de entrada.
- Internos:
 - Número de instrucciones.
 - Duración de las instrucciones.

Para estudiar el tiempo de ejecución podemos realizar dos tipos de análisis.

¹Complejidad temporal.

²Complejidad espacial.

2.2.1. Análisis empírico

También llamado *a posteriori*, se ejecuta el algoritmo para distintos valores de entrada y se cronometra el tiempo de ejecución. Se trata de una medida del comportamiento del algoritmo en su **entorno**, pero el resultado depende de los **factores externos e internos**.

2.2.2. Análisis teórico

También llamado *a priori*, obtiene una función que represente el tiempo de ejecución en operaciones elementales⁴ del algoritmo para cualquier valor de entrada. El resultado sólo depende de factores internos y no es necesario implementar ni ejecutar el algoritmo, pero no obtiene una medida real del comportamiento del algoritmo en el entorno.

Entonces, el tiempo de ejecución de un algoritmo es una función $T(n)$ que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de problema n .

2.2.3. Ejemplos de complejidades

Suma de los elementos de un vector.

Ejemplo (sintaxis de la STL)

```

1 int sumar( const vector<int> &v) {
2     int s = 0;
3
4     for(int i = 0; i < v.size(); i++)
5         s += v[i];
6
7     return s;
8 }
```

Si estudiamos el bucle ($n = v.size()$):

| n | asign. | comp. | inc. | total |
|-----|--------|---------|------|----------|
| 0 | 1 | 1 | 0 | 2 |
| 1 | 1 | 2 | 1 | 4 |
| 2 | 1 | 3 | 2 | 6 |
| : | : | : | : | : |
| n | 1 | $n + 1$ | n | $2 + 2n$ |

La complejidad del algoritmo será:

$$T(n) = \underbrace{1}_{\text{primera asignación}} + \underbrace{2 + 2n}_{\text{bucle}} + \underbrace{n}_{\text{interior del bucle}} = 3 + 3n$$

Figure 1. Suma de los elementos de un vector.

Traspuesta de una matriz cuadrada

Se puede simplificar mediante la cuenta de pasos de programa. Esta métrica permite obtener expresiones menos farragosas. Es una agrupación de instrucciones cuyo tiempo de ejecución es constante con el tamaño del problema, es decir, suponer que un conjunto de instrucciones actúan como una sola.

⁴Las operaciones elementales son aquellas que realiza el ordenador en un tiempo acotado por una constante. Para simplificar, se suele considerar que el coste de estas operaciones elementales es unitario (1).

Traspuesta de una matriz $d \times d$

```

1 void traspuesta( mat& A){ // supongo que A.n_rows == A.n_cols
2   for( int i = 1; i < A.n_rows; i++ )
3     for( int j = 0; j < i ; j++ )
4       swap( A(i,j), A(j,i) );
5 }
```

Como la complejidad del bucle interior es: $2 + 3i$ veces

$$T_d(d) = \underbrace{2(d - 1) + 2}_{\text{bucle exterior}} + \underbrace{\sum_{i=1}^{d-1} (2 + 3i)}_{\text{interior}} = \dots = O(d^2)$$

Si queremos la complejidad con respecto al tamaño del problema ($n = d^2$):

$$T_n(n) = T_d(d) = O(d^2) = O(n)$$

Figure 2. Traspuesta de una matriz cuadrada

Traspuesta de una matriz $d \times d$

```

1 void traspuesta( mat& A){ // supongo que A.n_rows == A.n_cols
2   for( int i = 1; i < A.n_rows; i++ )
3     for( int j = 0; j < i ; j++ )
4       swap( A(i,j), A(j,i) );
5 }
```

Volviendo al ejemplo anterior, calculamos la complejidad con respecto a d mediante cuenta de pasos:

$$T_d(d) = \sum_{i=1}^{d-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{d-1} i = (d - 1) \frac{1 + d - 1}{2} \in O(d^2)$$

Figure 3. Traspuesta de una matriz cuadrada con pasos de programa.

Producto de dos matrices

Cuenta de pasos:

$$T_d(d) = \sum_{i=1}^{d-1} \sum_{j=0}^{d-1} \left(1 + \sum_{k=0}^{d-1} 1\right) \in O(d^3)$$

La complejidad con respecto al tamaño de este problema es $d = \sqrt{n/2}$ porque el tamaño del problema es $n = 2d^2$. Es decir: $T_n(n) = T_d(d) = O(d^3) = O((\sqrt{n/2})^3) = O(n^{3/2})$

Producto de dos matrices $d \times d$ (Sintaxis de la librería armadillo)

```

1 mat producto( const mat &A, const mat &B ) {
2     mat R(A.n_rows, B.n_cols);
3     for( int i = 0; i < A.n_rows; i++ )
4         for( int j = 0; j < B.n_cols; j++ ) {
5             double acc = 0.0;
6             for( int k = 0; k < A.n_cols; k++ )
7                 acc += A(i,k) * B(k,j);
8             R(i,j) = acc;
9         }
10    }
11    return R;
12 }
```

- La complejidad de las líneas 6-7 es $O(d)$
- La complejidad de las líneas 4-9 es $O(d) + d \cdot O(d) = O(d^2)$
- La complejidad de las líneas 3-10 es $O(d) + d \cdot O(d^2) = O(d^3)$

La complejidad del algoritmo será: $T_d(d) = O(d^3)$

Figure 4. Producto de dos matrices.

Luego, en cuanto a la complejidad espacial, no se tiene en cuenta lo que ocupa la codificación del problema sino lo que es imputable al algoritmo, es decir:

1. Con respecto a d : $M_d(d) = O(d^2)$
2. Con respecto al tamaño del problema n : $M_n(n) = M_d(d) = O(d^2) = O((\sqrt{n}/2)^2) = O(n)$

2.3. Cotas de complejidad

Cuando aparecen distintos casos para una misma talla n se introducen las siguientes medidas de la complejidad, las cuales son todas funciones del **tamaño** del problema:

- $C_s(n)$: Peor caso o **cota superior**.
- $C_i(n)$: Mejor caso o **cota inferior**.
- $C_m(n)$: Caso promedio o **coste promedio**.

El coste promedio es difícil de evaluar a priori pues es necesario conocer la distribución de probabilidad de la entrada y **NO** es la media de la cota inferior y superior.

2.3.1. Ejemplos cota superior e inferior

2.4. Análisis asintótico

El estudio de la complejidad resulta realmente interesante para tamaños grandes de problema por los siguientes motivos:

Buscar elemento

```

1 int buscar( const vector<int> &v, int z ) {
2     for( int i = 0; i < v.size(); i++ )
3         if( v[i] == z )
4             return i;
5     return -1;
6 }

```

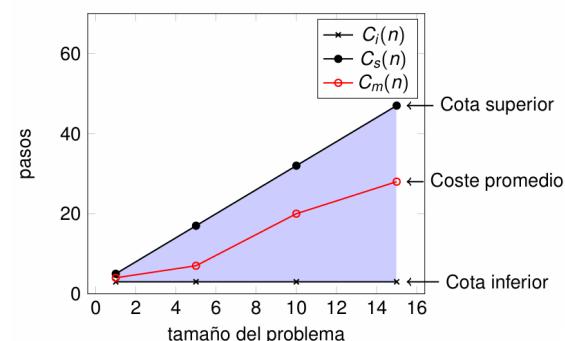
- En este caso el tamaño del problema es $n = v.size()$

| Mejor caso | Peor caso |
|-------------|--------------|
| $1 + 1 + 1$ | $1 + 3n + 1$ |
| Suma | $3n + 2$ |

Cotas:

$$C_s(n) = 3n + 2 \in O(n)$$

$$C_l(n) = 3 \in \Omega(1)$$

Coste de la función buscar

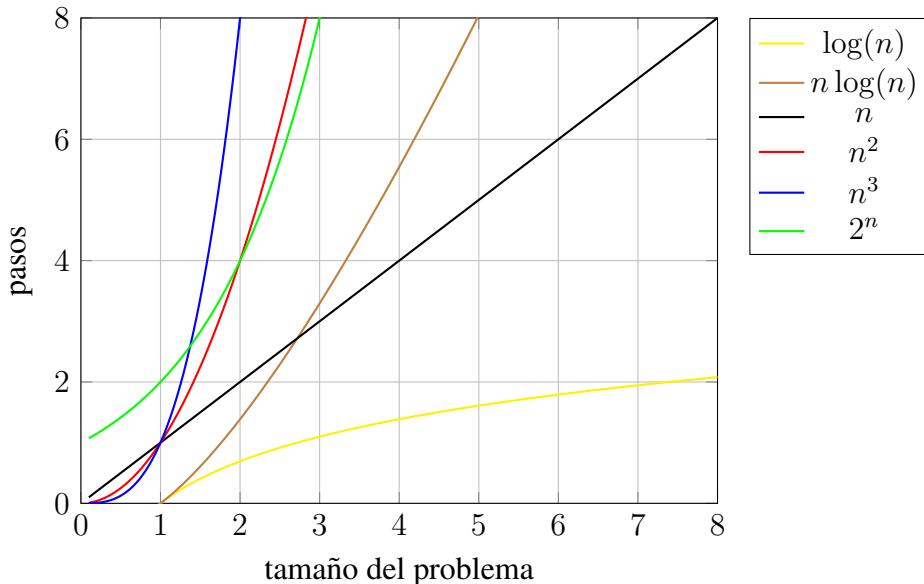
1. Las diferencias de tiempo de ejecución en algoritmos pequeños no suelen ser muy significativas.
2. Es lógico invertir tiempo en el desarrollo de un buen algoritmo.

Este estudio de la complejidad es llamado análisis asintótico. Permite clasificar las funciones de complejidad de forma que podamos compararlas entre si fácilmente y se definen clases de equivalencia que engloban a las funciones que crecen de la misma forma. Se emplea la notación asintótica para ello.

Notación asintótica Notación matemática utilizada para representar la complejidad cuando el tamaño del problema crece. Se definen tres tipos:

- O : Ómicron, **cota superior**.
- Ω : Omega, **cota inferior**.
- Θ : Zeta, **coste exacto**.

Permiten agrupar funciones con el mismo crecimiento.



2.5. O - Cota superior. Peor caso.

Si decimos que $f(n)$ es $O(g(n))$, significa que a medida que n crece, $f(n)$ no crece más rápido que un múltiplo constante de $g(n)$. Dicho de otro modo, $g(n)$ actúa como un "techo" que $f(n)$ no puede superar para valores grandes de n .

Decimos que $f(n)$ es $O(g(n))$ si existe una constante positiva c y un número n_0 tal que para todos los n mayores o iguales a n_0 , se cumple que: $f(n) \leq c * g(n)$.

2.5.1. Propiedades de las cotas superiores (Notación O)

Las propiedades de las cotas superiores son las siguientes.

Identidad:

- Toda función es una cota superior de sí misma. Ejemplo: $n^2 \in O(n^2)$.

Transitividad:

- Si $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$, entonces $f(n) \in O(h(n))$.
- Esto significa que si una función crece más rápido o igual que otra, y esta última crece más rápido o igual que una tercera, entonces la primera crece más rápido o igual que la tercera. Ejemplo: Si $n \in O(n^2)$ y $n^2 \in O(n^3)$, entonces $n \in O(n^3)$.

Adición:

- Si $f(n) \in O(g(n))$ y $h(n) \in O(k(n))$, entonces $f(n)+h(n) \in O(\max(g(n), k(n)))$.
- Al sumar dos funciones, la complejidad resultante está acotada superiormente por la que crezca más rápido. Ejemplo: Si $2n+3 \in O(n)$ y $5n^2+7 \in O(n^2)$, entonces $2n+3+5n^2+7 \in O(n^2)$.

Multiplicación:

- Si $f(n) \in O(g(n))$ y $h(n) \in O(k(n))$, entonces $f(n) \cdot h(n) \in O(g(n) \cdot k(n))$.
- Al multiplicar dos funciones, la complejidad resultante está acotada superiormente por el producto de las cotas superiores individuales. Ejemplo: Si $3n \in O(n)$ y $4n^2 \in O(n^2)$, entonces $3n \cdot 4n^2 \in O(n^3)$

Constante y Polinomio:

- Para cualquier polinomio $p(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_0$ donde $a_m > 0$, se tiene que $p(n) \in O(n^m)$.
- Significa que un polinomio está acotado superiormente por su término de mayor grado. Ejemplo: $5n^3 + 2n^2 + 4n + 6 \in O(n^3)$

$$\begin{aligned}
& f \in O(f) \\
& f \in O(g) \Rightarrow O(f) \subseteq O(g) \\
& O(f) = O(g) \Leftrightarrow f \in O(g) \wedge g \in O(f) \\
& f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h) \\
& f \in O(g) \wedge f \in O(h) \Rightarrow f \in O(\min\{g, h\}) \\
& f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(\max\{g_1, g_2\}) \\
& f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2) \\
& f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2) \\
& \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \\
& f(n) = a_m n^m + \dots + a_1 n + a_0 \text{ con } a_m > 0 \Rightarrow f(n) \in O(n^m) \\
& O(f) \subset O(g) \Rightarrow f \in O(g) \wedge g \notin O(f)
\end{aligned}$$

Figure 5. Propiedades de la cota superior.

2.6. Ω - Cota inferior. Mejor caso.

Si decimos que $f(n)$ es $\Omega(g(n))$, significa que a medida que n crece, $f(n)$ no puede ser menor que un múltiplo constante de $g(n)$. Dicho de otro modo, $g(n)$ actúa como un "piso" que $f(n)$ no puede descender para valores grandes de n .

Decimos que $f(n)$ es $\Omega(g(n))$ si existe una constante positiva c y un número n_0 tal que para todos los n mayores o iguales a n_0 , se cumple que: $f(n) \geq c * g(n)$.

2.6.1. Propiedades de las cotas inferiores (Notación Ω)

Las propiedades de las cotas inferiores son las siguientes.

Identidad:

- Toda función es una cota inferior de sí misma. Ejemplo: $n^2 \in \Omega(n^2)$.

Transitividad:

- Si $f(n) \in \Omega(g(n))$ y $g(n) \in \Omega(h(n))$, entonces $f(n) \in \Omega(h(n))$.
- Esto significa que si una función crece más lento o igual que otra, y esta última crece más lento o igual que una tercera, entonces la primera crece más lento o igual que la tercera. Ejemplo: Si $n^2 \in \Omega(n)$ y $n \in \Omega(\log n)$, entonces $n^2 \in \Omega(\log n)$.

Adición:

- Si $f(n) \in \Omega(g(n))$ y $h(n) \in \Omega(k(n))$, entonces $f(n)+h(n) \in \Omega(\max(g(n), k(n)))$.

- Al sumar dos funciones, la complejidad resultante está acotada inferiormente por la que crezca más lento. Ejemplo: Si $2n + 3 \in \Omega(n)$ y $5n^2 + 7 \in \Omega(n^2)$, entonces $2n + 3 + 5n^2 + 7 \in \Omega(n^2)$.

Multiplicación:

- Si $f(n) \in \Omega(g(n))$ y $h(n) \in \Omega(k(n))$, entonces $f(n) \cdot h(n) \in \Omega(g(n) \cdot k(n))$.
- Al multiplicar dos funciones, la complejidad resultante está acotada inferiormente por el producto de las cotas inferiores individuales. Ejemplo: Si $3n \in \Omega(n)$ y $4n^2 \in \Omega(n^2)$, entonces $3n \cdot 4n^2 \in \Omega(n^3)$.

Constante y Polinomio:

- Para cualquier polinomio $p(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_0$ donde $a_m > 0$, se tiene que $p(n) \in \Omega(n^m)$.
- Significa que un polinomio está acotado inferiormente por su término de mayor grado. Ejemplo: $5n^3 + 2n^2 + 4n + 6 \in \Omega(n^3)$.

$$\begin{aligned}
& f \in \Omega(f) \\
& f \in \Omega(g) \Rightarrow \Omega(f) \subseteq \Omega(g) \\
& \Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g) \wedge g \in \Omega(f) \\
& f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h) \\
& f \in \Omega(g) \wedge f \in \Omega(h) \Rightarrow f \in \Omega(\max\{g, h\}) \\
& f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 + f_2 \in \Omega(\max\{g_1, g_2\}) \\
& f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 + f_2 \in \Omega(g_1 + g_2) \\
& f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 f_2 \in \Omega(g_1 g_2) \\
& \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow g \in \Omega(f) \\
& f(n) = a_m n^m + \dots + a_1 n + a_0 \text{ con } a_m > 0 \\
& \Rightarrow f(n) \in \Omega(n^m)
\end{aligned}$$

Figure 6. Propiedades de la cota inferior.

2.7. Θ - Coste promedio o exacto. Caso promedio.

Si decimos que $f(n)$ es $\Theta(g(n))$, significa que el tiempo promedio de ejecución del algoritmo está acotado superior e inferiormente por $g(n)$ multiplicado por constantes, para valores grandes de n .

Decimos que $f(n)$ es $\Theta(g(n))$ si existen constantes positivas c_1, c_2 y un número n_0 tal que para todos los n mayores o iguales a n_0 , se cumple que: $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$.

2.7.1. Propiedades de las cotas promedio (Notación Θ)

Las propiedades de las cotas promedio son las siguientes.

Identidad:

- Toda función es una cota promedio de sí misma. Ejemplo: $n^2 \in \Theta(n^2)$.

Transitividad:

- Si $f(n) \in \Theta(g(n))$ y $g(n) \in \Theta(h(n))$, entonces $f(n) \in \Theta(h(n))$.
- Esto significa que si una función crece al mismo ritmo que otra, y esta última crece al mismo ritmo que una tercera, entonces la primera crece al mismo ritmo que la tercera. Ejemplo: Si $n \in \Theta(n^2)$ y $n^2 \in \Theta(n^3)$, entonces $n \in \Theta(n^3)$.

Adición:

- Si $f(n) \in \Theta(g(n))$ y $h(n) \in \Theta(k(n))$, entonces $f(n)+h(n) \in \Theta(\max(g(n), k(n)))$.
- Al sumar dos funciones, la complejidad resultante está acotada por la que crezca más rápido. Ejemplo: Si $2n + 3 \in \Theta(n)$ y $5n^2 + 7 \in \Theta(n^2)$, entonces $2n + 3 + 5n^2 + 7 \in \Theta(n^2)$.

Multiplicación:

- Si $f(n) \in \Theta(g(n))$ y $h(n) \in \Theta(k(n))$, entonces $f(n) \cdot h(n) \in \Theta(g(n) \cdot k(n))$.
- Al multiplicar dos funciones, la complejidad resultante está acotada por el producto de las cotas promedio individuales. Ejemplo: Si $3n \in \Theta(n)$ y $4n^2 \in \Theta(n^2)$, entonces $3n \cdot 4n^2 \in \Theta(n^3)$.

Constante y Polinomio:

- Para cualquier polinomio $p(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_0$ donde $a_m > 0$, se tiene que $p(n) \in \Theta(n^m)$.
- Significa que un polinomio está acotado por su término de mayor grado. Ejemplo: $5n^3 + 2n^2 + 4n + 6 \in \Theta(n^3)$.

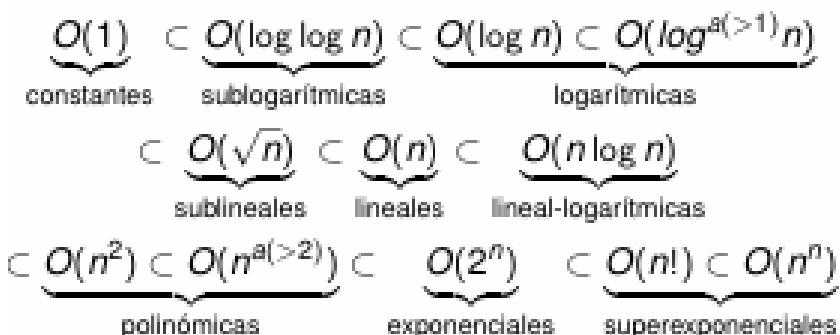
2.7.2. Jerarquías de Funciones

Los conjuntos de funciones están incluidos unos en otros generando una ordenación de las diferentes funciones.

$$\mathcal{O}(f_1(n)) \subseteq \mathcal{O}(f_2(n)) \subseteq \mathcal{O}(f_3(n))$$

Las clases más utilizadas en la expresión de complejidad son:

$$\begin{aligned}
& f \in \Theta(f) \\
& f \in \Theta(g) \Rightarrow \Theta(g) = \Theta(f) \\
& \Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g) \wedge g \in \Theta(f) \\
& f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h) \\
& f \in \Theta(g) \wedge f \in \Theta(h) \Rightarrow f \in \Theta(\max\{g, h\}) \\
& f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 + f_2 \in \Theta(g_1 + g_2) \\
& \quad \Theta(f_1 + f_2) = \Theta(\max(f_1, f_2)) \\
& f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 f_2 \in \Theta(g_1 g_2) \\
& \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k, k \neq 0, k \neq \infty \Rightarrow \Theta(f) = \Theta(g) \\
& f(n) = a_m n^m + \dots + a_1 n + a_0 \text{ con } a_m > 0 \\
& \quad \Rightarrow f(n) \in \Theta(n^m)
\end{aligned}$$

Figure 7. Propiedades de la cota promedio

2.8. ¿Verdadero o falso?

2.9. Cálculo de complejidades

Pasos para calcular las complejidades:

1. Determinar la talla del problema. Será la variable de la función de complejidad que se pretende encontrar.
2. Determinar si hay mejor o peor caso.
3. Obtener las cotas para cada caso.

Para el paso 3 podemos diferenciar entre algoritmos iterativos y recursivos.

Ejemplo: Búsqueda en un vector ordenado Este ejemplo se ve a partir de capturas de las transparencias.

Sumar elementos

```

1 int sumar( const vector<int> &v ) {
2     int s = 0;
3     for( int i = 0; i < v.size(); i++ )
4         s += v[i];
5     return s;
6 }

```

| Línea | Pasos | C. Asintótica |
|-------|-------|---------------|
| 2 | 1 | $\Theta(1)$ |
| 3,4 | n | $\Theta(n)$ |
| 5 | 1 | $\Theta(1)$ |
| Suma | $n+2$ | $\Theta(n)$ |

Buscar elemento

```

1 int buscar( const vector<int> &v, int z ) {
2     for( int i = 0; i < v.size(); i++ )
3         if( v[i] == z )
4             return i;
5     return -1;
6 }

```

| Línea | Cuenta Pasos | | C. Asintótica | |
|-------|--------------|-----------|---------------|-----------|
| | Mejor caso | Peor caso | Mejor caso | Peor caso |
| 2 | 1 | n | $\Omega(1)$ | $O(n)$ |
| 3 | 1 | n | $\Omega(1)$ | $O(n)$ |
| 4 | 1 | 0 | $\Omega(1)$ | — |
| 5 | 0 | 1 | — | $O(1)$ |
| Suma | 3 | $2n+1$ | $\Omega(1)$ | $O(n)$ |

$$C_s(n) = 2n + 1$$

$$C_i(n) = 3$$

$$C_s(n) \in O(n)$$

$$C_i(n) \in \Omega(1)$$

Figure 8. Algoritmo iterativo.

Figure 9. Algoritmo recursivo.

Búsqueda binaria

```

1 int buscar( const vector<int> &v,
2             int x
3             ) {
4     int pri = 0;
5     int ult = v.size() - 1;
6     while( pri < ult ) {
7         int m = ( pri + ult ) / 2;
8         if( v[m] < x )
9             pri = m + 1;
10        else
11            ult = m;
12    }
13    if( v[pri] == x )
14        return pri;
15    else
16        return -1;
17 }

```

- Tamaño del problema: $n = \text{ult} - \text{pri} + 1$
- No existe caso mejor y peor.
- Coste exacto:

| iteración | Tamaño | Pasos |
|-----------|-------------------|-------|
| 1 | n | 1 |
| 2 | $n/2$ | 1 |
| 3 | $n/4$ | 1 |
| ... | ... | ... |
| k | $n/2^{k-1} = 2^*$ | 1 |

* Despejar k para obtener el máximo valor que puede tomar.

$$k = \lfloor \log_2 n \rfloor$$

$$c_e(n) = 1 + \sum_{k=1}^{\lfloor \log_2 n \rfloor} 1 \in \Theta(\log n)$$

Figure 10. Búsqueda en vector ordenado.

```

1 int ejemplo(int n){
2     int k=0;
3     for( int i = 1; i < n; i*=2 )
4         for( int j = 0; j < i; j++ )
5             k++;
6     return k;
7 }

```

Errores habituales:

1. $C_e(n) = \frac{1}{2} \sum_{i=1}^{n-1} i \in \Theta(n^2)$
2. $C_e(n) = \sum_{i=1}^{\log_2 n} i \in \Theta(\log^2 n)$
3. $C_e(n) = \sum_{i=1}^{\log_2 n} \frac{i}{2} \in \Theta(i \log n)$
4. $C_e(n) = \sum_{i=1}^{n/2} 2^i \in \Theta(2^n)$
5. $C_e(n) = \sum_{i=1}^{n-1} \log_2 i \in \Theta(n \log n)$
6. $C_e(n) = \sum_{k=1}^i \log_2 n \in \Theta(i \log n)$

¿Porqué están todos mal?

Figure 11. Errores habituales.

Dado un algoritmo recursivo, el coste depende de las llamadas recursivas, y por tanto, debe definirse recursivamente:

Ordenación por selección (recursivo)

```

1 void ordenar( vector<int> &v, int pri) {
2     if( pri == v.size() )
3         return;
4     int m = pri;
5     for( int i = pri + 1; i < v.size(); i++ )
6         if( v[i] < v[m] )
7             m = i;
8     swap( v[m], v[pri]);
9     ordenar(v,pri + 1);
10 }
```

- Obtener ecuación de recurrencia a partir del algoritmo:

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

Figure 12. Eficiencia de ordenación por selección.

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ \Theta(1) + T\left(\frac{n}{2}\right) & n > 1 \end{cases} \quad (n = \text{ult} - \text{pri} + 1)$$

Esto es una relación de recurrencia. Se trata de una expresión que relaciona el valor de una función f definida para un entero n con uno o más valores de la misma función para valores menores que n . Se usan para algoritmos recursivos principalmente, pero se pueden aplicar a iterativos también.

Volviendo al método de obtener la complejidad de un algoritmo, usaremos el método de sustitución, pues es el más sencillo para funciones lineales. Consiste en sustituir cada $f(n)$ por su valor al aplicarle de nuevo la función hasta obtener un término general.

Ejemplo: Ordenar un vector a partir del elemento pri Tendremos en cuenta que $n = v.size() - pri$.

Por sustitución, la recurrencia quedaría como:

$$\begin{aligned} T(n) &= n + T(n - 1) = n + (n - 1) + T(n - 2) = n + (n - 1) + (n - 2) + T(n - 3) = \\ &n + (n - 1) + (n - 2) + (n - 3) + \dots + 3 + 2 + T(1) = n + (n - 1) + (n - 2) + (n - 3) + \dots + 3 + 2 + 1 = \\ &\sum_{j=1}^n j = \frac{n(n + 1)}{2} \end{aligned}$$

Por lo tanto, $T(n) \in \Theta(n^2)$.

2.10. Quicksort - Ordenación por partición

Contiene un elemento pivote que sirve para dividir en dos partes el vector. Su elección define variantes del algoritmo (al azar, primero, central, mediana...).

Sigue los siguientes pasos:

1. Elección del pivote.
2. División del vector en dos partes:
 - Izquierda: Elementos menores.
 - Derecha: Elementos mayores.
3. Doble llamada recursiva en cada parte del vector.

El código del quicksort se puede consultar en la imagen 13.

Quicksort

```

1 void quicksort( int v[], int pri, int ult ) {
2     if( ult <= pri )
3         return;
4     int p = pri;
5     int j = ult;
6     while(p < j) {
7         if( v[p+1] < v[p] ) {
8             swap( v[p+1], v[p] );
9             p++;
10        } else {
11            swap( v[p+1], v[j] );
12            j--;
13        }
14    }
15    quicksort(v, pri, p-1);
16    quicksort(v, p+1, ult);
17 }
```

Figure 13. Código QuickSort.

Para un tamaño de problema n en quicksort tenemos:

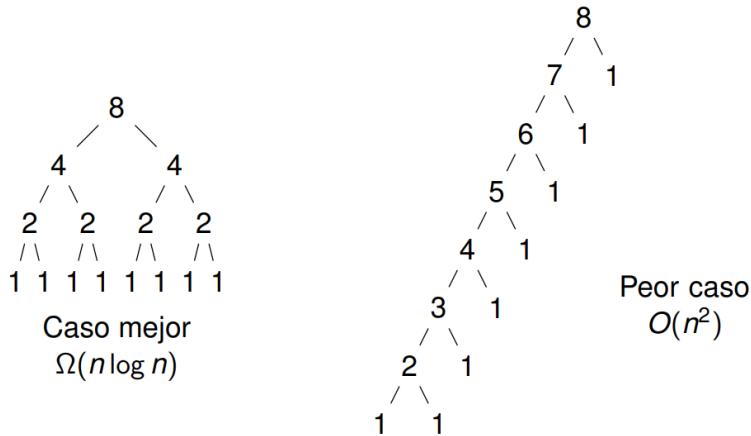
Mejor caso Los subproblemas son ambos $\frac{n}{2}$. Es decir, el pivote está justo en el centro del vector.

$$T(n) \in \begin{cases} \Theta(1) & n \leq 1 \\ \Theta(n) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

Mediante sustitución:

$$\begin{aligned}
 f(n) &= n + 2T\left(\frac{n}{2}\right) = n + 2\left(\frac{n}{2} + 2f\left(\frac{n}{2 * 2}\right)\right) = n + 2\left(\frac{n}{2} + 2f\left(\frac{n}{2^2}\right)\right) = n + \frac{2n}{2} + 2 * 2T\left(\frac{n}{2^2}\right) = \\
 &n + n + 2^2T\left(\frac{n}{2^2}\right) = 2n + 2^2T\left(\frac{n}{2^2}\right) = 2n + 2^2f\left(\frac{n}{2^2} + 2T\left(\frac{n}{2^3}\right)\right) = 3n + 2^3T\left(\frac{n}{3}\right) = \dots = \\
 &in + 2^iT\left(\frac{n}{2^i}\right)
 \end{aligned}$$

Como la recursión termina cuando $\frac{n}{2^i} = 1$, habrá $\frac{n}{2^i} = 1 \rightarrow n = 2^i \rightarrow \log_2 n = i$ llamadas recursivas. Por tanto, la complejidad en el mejor de los casos de QuickSort es $T(n) \in \Omega(n \log n)$.

**Figure 14.** Comparativa visual QuickSort.

Peor caso Los subproblemas son $0, n - 1$ y $(n - 1, 0)$. Es decir, el pivote se encuentra al final.

$$T(n) \in \begin{cases} \Theta(1) & n \leq 1 \\ \Theta(n) + T(0) + T(n - 1) & n > 1 \end{cases}$$

Mediante sustitución:

$$\begin{aligned} f(n) &= n + T(n - 1) = n + f(n - 1 + T(n - 2)) = n + (n - 1) + T(n - 2) = \dots \\ &= n + (n - 1) + (n - 2) + \dots + T(n - i) \end{aligned}$$

Como la recursión termina cuando $n - i = 1$, habrá $i = n - 1$ llamadas recursivas.

$$n + (n - 1) + (n - 2) + \dots + 3 + 2 + T(1) = \sum_{j=2}^n j + 1 = \frac{n(n + 1)}{2}$$

Por tanto, el peor caso de QuickSort tiene complejidad de $T(n) \in \Theta(n^2)$.

2.10.1. QuickSort mediana

Forzamos el mejor caso: $O(n \log n)$.

2.11. HeapSort - Ordenación por montículo

Basado en el algoritmo de ordenación por selección directa⁵. Mejora la eficiencia de la selección repetida del máximo construyendo un *heap* (montículo).

Sigue los siguientes pasos:

⁵Seleccionar sucesivamente el máximo y colocarlo en la posición correcta ($\Theta(n^2)$).

1. Creación del *heap* (montículo) sobre el mismo vector a ordenar.
2. Aprovecha el *heap* para ordenar el vector mediante extracción sucesiva de la cima del montículo para colocarla en la posición correcta del vector.

2.11.1. El *heap*

El montículo es una estructura de tipo árbol binario con las dos siguientes características:

- Es esencialmente completo, es decir, todas las hojas están desplazadas lo más a la izquierda posible y todos los niveles están completados salvo, en todo caso, el último.
- Todo nodo es mayor que sus dos hijos (*max heap*).

Contiene las operaciones:

- top: Consultar el valor de la cima. Tiene complejidad de $O(1)$.
- pop: Borrar la cima. Reorganiza para que siga siendo un montículo. Tiene complejidades $\Omega(1)$ y $\Theta(\log n)$.
- push: Insertar un elemento. Se coloca al final y se reorganiza el árbol. Tiene complejidad de $\Theta(\log n)$ y $\Omega(1)$.

Entonces, HeapSort lo que hace es usar el propio vector como árbol binario esencialmente completo (todo vector lo es) y va hundiendo uno a uno los elementos del vector recorriéndolo de derecha a izquierda desde la posición $\frac{n}{2-1}$ (último nodo con hojas) hasta la 0 (nodo raíz). Esto tiene complejidad de $\Theta(n)$. Es eficiente.

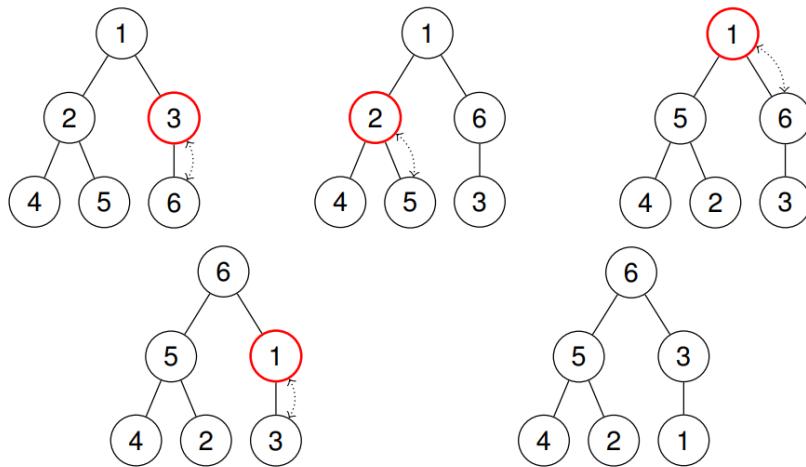


Figure 15. Ejemplo de montículo para $v = [1, 2, 3, 4, 5, 6]$

La construcción del montículo tiene:

Mejor caso El vector de entrada ya es un montículo y sólo se visitan los nodos internos, $\Omega(n)$.

Peor caso Todos los nodos se hunden hasta las hojas.

| Nivel en el árbol | Nodos/nivel | Pasos/nodo | Total pasos en el nivel |
|-------------------|-------------|------------------------------------|---|
| 0 (raíz) | 1 | $\log_2 \frac{n+1}{2}$ | $1 \times \log_2 \frac{n+1}{2}$ |
| 1 | 2 | $\log_2 \frac{n+1}{4}$ | $2 \times \log_2 \frac{n+1}{4}$ |
| 2 | 4 | $\log_2 \frac{n+1}{8}$ | $4 \times \log_2 \frac{n+1}{8}$ |
| ... | ... | ... | ... |
| k (hojas) | 2^k | $\log_2 \frac{n+1}{2^{k+1}} = 0^*$ | $2^k \times \log_2 \frac{n+1}{2^{k+1}}$ |

(*) El máximo valor que puede tomar k es $\log_2(n+1) - 1$. Se tiene:

$$c_s(n) = \sum_{k=0}^{\log_2(n+1)-1} 2^k \log_2 \left(\frac{n+1}{2^{k+1}} \right) = n - \log_2(n+1) \in O(n)$$

Heapsort

```

1 void heapSort(int *v, size_t n) {
2     // Build a MAX-HEAP with the input array
3     for (size_t i = n / 2 - 1; true ; i--) {
4         sink(v, n, i);
5         if (i==0) break; // note that size_t is unsigned type
6     }
7
8     for (size_t i=n-1; i>0; i--) {
9         swap(v[0], v[i]); // Move current root to the end.
10        sink(v, i, 0);
11    }
12 }
```

¿Complejidad temporal? Líneas 3–6, $\Theta(n)$; Líneas 8–11, $O(n \log(n))$.

¿Complejidad espacial? $\Theta(1)$

| Enunciado | V/F | Explicación |
|---|---------------------------------------|--|
| $4n^3 - 2n^2 + 8 \in O(n^3)$ | <input checked="" type="checkbox"/> V | Es verdadero porque el término de mayor grado domina la función. |
| $n^3 \in O(4n^3 - 2n^2 + 8)$ | <input checked="" type="checkbox"/> V | Es verdadero porque n^3 domina los otros términos para valores grandes de n . |
| $n + n\sqrt{n} \in O(n)$ | <input type="checkbox"/> F | Es falso porque $n\sqrt{n}$ crece más rápido que n . |
| $n + n \log n \in \Theta(n)$ | <input type="checkbox"/> F | Es falso porque $n \log n$ crece más rápido que n . |
| $n + n \log n \in \Omega(n)$ | <input checked="" type="checkbox"/> V | Es verdadero porque $n + n \log n$ es al menos lineal. |
| Si $f \notin O(g)$ y $\exists \lim_{n \rightarrow \infty} \frac{f}{g}: f \in \Omega(g)$ | <input type="checkbox"/> F | Es falso porque la existencia del límite no garantiza la cota inferior. |
| Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$: $f \in O(g_1 + g_2)$ | <input checked="" type="checkbox"/> V | Es verdadero porque f está acotado superiormente por la suma. |
| Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$: $f \in \Omega(g_1 + g_2)$ | <input checked="" type="checkbox"/> V | Es verdadero porque f está acotado inferiormente por la suma. |
| Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$: $f \in O(g_1 \cdot g_2)$ | <input checked="" type="checkbox"/> V | Es verdadero porque f está acotado superiormente por el producto. |
| Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$: $f \in \Omega(g_1 \cdot g_2)$ | <input checked="" type="checkbox"/> V | Es verdadero porque f está acotado inferiormente por el producto. |
| $O(2^{\log_2(n)}) \subset O(n^2)$ | <input checked="" type="checkbox"/> V | Es verdadero porque $2^{\log_2(n)} = n$. |
| $O(4^{\log_2(n)}) \subset O(n^2)$ | <input checked="" type="checkbox"/> V | Es verdadero porque $4^{\log_2(n)} = n^2$. |
| $\Theta(2^n) = \Theta(3^n)$ | <input type="checkbox"/> F | Es falso porque 2^n y 3^n crecen a ritmos exponenciales diferentes. |
| $\Theta(\log_2(n)) = \Theta(\log_3(n))$ | <input checked="" type="checkbox"/> V | Es verdadero porque los logaritmos en diferentes bases difieren solo por una constante. |
| $\Theta(\log_2(n^3)) = \Theta(\log_2(n^2))$ | <input type="checkbox"/> F | Es falso porque $\log_2(n^3) = 3\log_2(n)$ y $\log_2(n^2) = 2\log_2(n)$. |

Table 2. Solución de los Ejercicios de Verdadero o Falso

3. Divide y vencerás

Técnica de diseño de algoritmos que consiste en descomponer el problema en subproblemas de menor tamaño, resolver estos de manera individual y combinar las soluciones de los subproblemas para obtener la solución del problema original.

Cabe tener en cuenta que no siempre un problema de menor talla es más fácil de resolver y que la solución de estos no implica necesariamente que la solución del problema original se pueda obtener fácilmente. Por tanto, es aplicable si encontramos una forma de:

- descomponer el problema en subproblemas de talla menor
- resolver problemas menores a un tamaño determinado directamente
- combinar las soluciones de los subproblemas que permita obtener la solución del problema original.

Esquema divide y vencerás (DC)

```

1 Solution DC( Problem p ) {
2     if( is_simple(p) )
3         return trivial(p);
4
5     list<Solution> s;
6     for( Problem q : divide(p) )
7         s.push_back( DC(q) );
8
9     return combine(s);
10 }
```

Figure 16. Esquema en código de Divide y Vencerás

QuickSort puede ser visto como divide y vencerás.

3.1. Análisis de eficiencia

Los costes pueden ir de logarítmicos a exponenciales, pues depende de los subproblemas:

- Cantidad (h)
- Tamaño
- Intersección entre ellos

Tendremos entonces que $T(n) = hT(\frac{n}{b}) + g(n)$, donde:

- $g(n)$: tiempos de divide y combine para n sin llamadas recursivas.
- b : constante de división del tamaño de problema.

La solución general es entonces:

3.2. Teorema Maestro

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } h < b^k \\ \Theta(n^k \log_b n) & \text{si } h = b^k \\ \Theta(n^{\log_b h}) & \text{si } h > b^k \end{cases}$$

Aplicando el teorema de reducción, los mejores resultados en cuanto a coste se consiguen cuando los subproblemas son aproximadamente del mismo tamaño. Se cumple cuando $b = h$ entonces $b = h = a$ y $T(n) = aT(\frac{n}{a}) + g(n)$, siendo $g(n) \in \Theta(n^k)$ y $k \in R^{\geq 0}$.

La anterior ecuación quedaría como:

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } k > 1 \\ \Theta(n \log n) & \text{si } k = 1 \\ \Theta(n) & \text{si } k < 1 \end{cases}$$

3.3. Mergesort

Ordenar de forma ascendente un vector v de n elementos. Usa el esquema divide y vencerás.

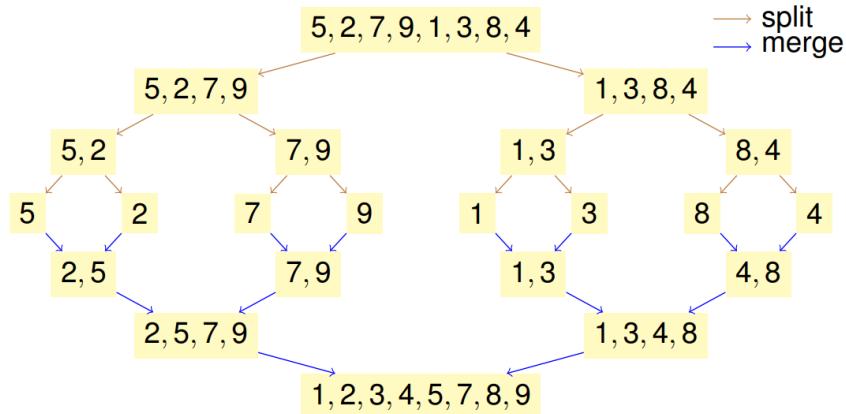
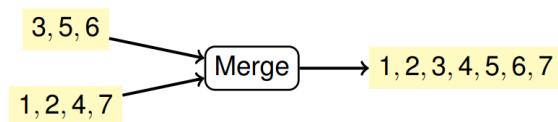


Figure 17. Ejemplo MergeSort

Usa la función `merge` que obtiene un vector ordenado como fusión de dos vectores también ordenados. Tiene complejidad $\Theta(n)$ temporal y espacial donde $n = \text{last} - \text{first}$.



Sigue los siguientes pasos:

1. Dividir la lista en dos sublistas de aproximadamente igual tamaño.

2. Ordenar cada sublistas recursivamente aplicando mergesort.
3. Mezclar las dos sublistas ordenadas en una lista ordenada.
4. Si la longitud de la lista es 0 o 1, termina.

Para una talla n , la complejidad del algoritmo MergeSort es $\Theta(n \log n)$.

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

¿Cuál es la complejidad espacial?

- Sin tener en cuenta la pila de recursión, viene dada por el mayor bloque de memoria auxiliar que necesita `merge`:

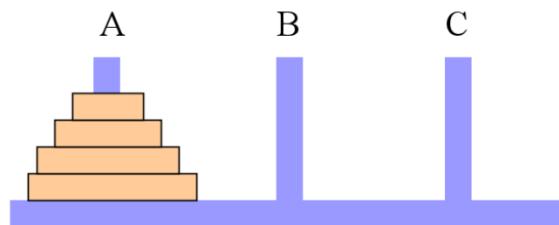
$$\max_{0 \leq i \leq \log_2 \frac{n}{2}} \frac{n}{2^i} = n$$

- Notar que estos bloques de memoria auxiliar nunca coexisten.
- Si añadimos la pila, hay que añadir un término de orden $\log n$.
- **Complejidad espacial:** $\Theta(n)$

Figure 18. Complejidad espacial de Mergesort

3.4. Torres de Hanoi

Se trata de un problema donde se colocan los discos de la torre A en la C empleando como ayuda la torre B. Los discos se mueven de uno a uno y sin colocar nunca un disco sobre otro más pequeño.



La talla del problema es $n - 1$. No se pueden aplicar las fórmulas generales de las transparencias anteriores y el problema tiene una complejidad intrínseca peor que las vistas en mergesort y los ejemplos de divide y vencerás.

Asumiendo que todas las operaciones de 1 disco son de coste $O(1)$:

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

$$\begin{aligned}
T(n) &\stackrel{1}{=} 1 + 2T(n - 1) \\
&\stackrel{2}{=} 1 + 2(1 + 2T(n - 2)) = 1 + 2 + 2^2T(n - 2) \\
&\stackrel{3}{=} 1 + 2 + 2^2(1 + 2T(n - 3)) = 2^0 + 2^1 + 2^2 + 2^3T(n - 3) \\
&\vdots \\
&\stackrel{k}{=} \sum_{i=0}^{k-1} 2^i + 2^k T(n - k) = 2^k - 1 + 2^k T(n - k)
\end{aligned}$$

Cuando $n - k = 1$ parará la relación de recurrencia, es decir, cuando $k = n - 1$. La complejidad será entonces de

$$T(n) = 2^n - 1 \in \Theta(2^n)$$

3.5. Selección del k-ésimo mínimo

Se puede abordar de las siguientes maneras:

1. **Ineficiente.** Seleccionar k veces el elemento más pequeño del vector. $\Theta(k * n)$
2. **Mejorable.** Construir un heap sobre el vector y realizar $k - 1$ operaciones pop y una pop. $O(n + k \log n)$.
3. **Eficiente.** *Quickselect*, basado en QuickSort. $\Omega(n)$ y $O(n^2)$. Coincide con el promedio estadístico $\Theta(n)$.

3.6. Búsqueda binaria

Se puede ver como un divide y vencerás en el que:

- divide es med = first step.
- is_simple corresponde al caso count == 0.
- Sólo se resuelve uno de los dos subproblemas: **reduce y vencerás**.
- No es necesario combine.

El peor caso tiene la siguiente ecuación de recurrencia:

$$\begin{aligned}
T(n) &= \begin{cases} 1 & n = 1 \\ 1 + T(\frac{n}{2}) & n > 1 \end{cases} \\
T(n) &\stackrel{1}{=} 1 + T(\frac{n}{2}) \\
&\stackrel{2}{=} 1 + 1 + T(\frac{n}{2^2}) = 2 + T(\frac{n}{2^2}) \\
&\vdots \\
&\stackrel{k}{=} k + T(\frac{n}{2^k})
\end{aligned}$$

La recursión termina cuando $\frac{n}{2^k} = 1$, entonces $k = \log_2 n$. Tiene una complejidad de $T(n) \in O(\log n)$

4. Programación dinámica

4.1. Problema de la mochila

Sean n objetos con valores $v_i \in R$ y pesos $w_i \in R^{>0}$ conocidos y una mochila con capacidad máxima de carga W , ¿cuál es el valor máximo que puede transportar la mochila sin sobrepasar su capacidad?

Este problema presenta variaciones dependiendo de si los objetos se pueden fraccionar o no.

A la mochila con pesos **no** fraccionables se le llama **mochila discreta**. Es la más difícil y no conoce ningún algoritmo que resuelva este problema en un tiempo razonable, es decir, polinómico, pero tampoco se ha demostrado que ese algoritmo no exista.

Se puede abordar simplificándolo mediante programación dinámica. Como los pesos son cantidades discretas se usan para indexar un almacén de resultados parciales.

Esto se puede ver como una secuencia de decisiones (x_1, x_2, \dots, x_n) donde $x_i \in \{0, 1\}$, $1 \leq i \leq n$. En x_i se almacena la decisión sobre el objeto i y si x_i es escogido entonces será $x_i = 1$. En caso contrario $x_i = 0$. La secuencia óptima de decisiones será la que maximice $\sum_{i=1}^n x_i v_i$. Con las restricciones de la mochila: $\sum_{i=1}^n x_i w_i \leq W$, $\forall i : 1 \leq i \leq n$, $x_i \in \{0, 1\}$.

Las decisiones se toman en orden descendente $(x_n, x_{n-1}, \dots, x_1)$. Ante la decisión x_i hay dos alternativas:

- Aceptar el objeto: $i : x_i = 1$. La ganancia adicional es v_i a costa de reducir la capacidad de la mochila en w_i .
- Rechazar el objeto: $i : x_i = 0$. No hay ganancia pero tampoco se pierde capacidad en la mochila.

Se selecciona la alternativa que mayor ganancia global resulte. Esto implica tener que resolver todos los subproblemas de menor a mayor.

4.1.1. Subestructura óptima

Un problema tiene una subestructura óptima si una solución óptima puede construirse eficientemente a partir de las soluciones óptimas de sus subproblemas. Es decir, un problema tiene subestructura óptima si se puede resolver dividiéndolo en partes más pequeñas (subproblemas), resolviendo cada parte de la mejor manera posible y luego combinando esas soluciones para obtener la mejor solución del problema original. Esto se conoce como el **principio de optimalidad**. Es una **condición necesaria** para que se pueda aplicar programación dinámica a un problema. También lo es para divide y vencerás. Para demostrar que se cumple, basta con probar que las soluciones de los subproblemas han de ser necesariamente óptimas si también lo es la solución del problema en su conjunto.

El problema de la mochila discreta presenta una subestructura óptima.

La solución recursiva al problema de la mochila sigue usando divide y vencerás. Es la transcripción literal de la recurrencia matemática.

En el **mejor** de los **casos** ningún objeto cabrá en la mochila, es decir, $T(n) \in \Omega(n)$.

En el **peor** de los **casos**, en cambio, $T(n) =$

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

, quedando el término general como $T(n) =^k 2^k - 1 + 2^k T(n - k)$. Terminará cuando $n - k = 0$: $T(n) = 2^n - 1 + 2^n \in O(2^n)$.

Esta implementación es ineficiente porque:

- Los problemas se reducen en subproblemas de tamaño similar ($n - 1$).
- Un problema se divide en dos subproblemas, y así sucesivamente, creando complejidades exponenciales.

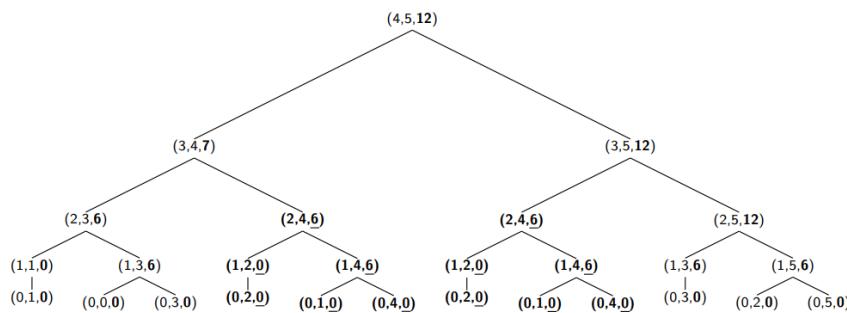
No obstante, sólo hay nW problemas distintos. La solución recursiva está generando y resolviendo el mismo problema muchas veces, pero la ineficiencia no se debe a la recursividad.

En efecto, se resuelven muchos subproblemas repetidos:

$$n = 4, W = 5$$

- Ejemplo: $w = (3, 2, 1, 1)$
 $v = (6, 6, 2, 1)$

Nodos: $(i, W, \text{knapsack}(i, W))$; izquierda, $x_i = 1$; derecha, $x_i = 0$.



Estas repeticiones se pueden evitar siguiendo la siguiente metodología para cada subproblema que se presenta:

- Si no se ha resuelto, se resuelve y se guarda la solución.
- Si ya ha sido resuelto, se devuelve la solución almacenada.

Esta técnica es denominada *memoization*.

Para transformar la versión recursiva a una iterativa tendremos que:

1. Declarar el almacén incorporando espacio para albergar los casos base.
2. Utilizar los casos base de la solución recursiva para llenar el contorno del almacén.
3. A partir del caso general en la función recursiva, diseñar la estructura que permita crear los bucles que completen el almacén a partir de los subproblemas ya resueltos.

La complejidad temporal y espacial de esta versión iterativa será $T(n, W) \in \Theta(nW)$ y $T_S(n, W) \in \Theta(nW)$, respectivamente. Esto se explica con:

$$T(n, W) = 1 + \sum_{i=0}^W 1 + \sum_{i=1}^n \sum_{j=1}^W 1 = 1 + n + W + 1 + W(n+1)$$

La complejidad espacial se puede mejorar si las soluciones del problema se pueden representar mediante subconjuntos. Es decir, si se pueden representar conociendo sólo las filas y no el almacén entero. La complejidad temporal no va a cambiar.

Las decisiones que corresponden al valor óptimo se pueden extraer del almacén tanto de la versión iterativa como de la recursiva.

4.1.2. Complejidades

La **complejidad temporal** de la solución obtenida mediante programación dinámica es de $\Theta(nW)$. Un recorrido descendente a través de la tabla permite obtener también $\Theta(n)$. Con esto, si W es muy grande, entonces la solución obtenida mediante PD puede no ser buena. También, si los pesos w_i o W pertenecen a dominios continuos la solución no sirve (porque no podremos almacenarlos en el almacén). La **complejidad espacial** se puede reducir hasta $\Theta(W)$.

4.2. Corte de tubos

Una empresa compra tubos de longitud n y los corta en tubos más cortos que luego vende. El corte es gratis y el precio de venta de un tubo de longitud i es p_i . ¿Cuál es la forma óptima de cortar un tubo de longitud n maximizando el precio total?

Probar todas las formas de cortar es prohibitivo pues hay 2^{n-1} .

Buscamos una descomposición $n = i_1 + i_2 + \dots + i_k$ por la que se obtenga el precio máximo, siendo este $r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$.

4.2.1. Complejidades

Solución ineficiente recursiva: cortar el tubo de las n formas posibles y buscar el corte que maximiza la suma del precio del trozo cortado y del resto. Esto, como hemos visto antes, es prohibitivo pues es $O(2^n)$.

Solución recursiva con almacén (memoization):

- Complejidad espacial $O(n)$.
- Complejidad temporal $O(n^2)$.

Solución iterativa:

- Complejidad espacial $O(n)$.
- Complejidad temporal $O(n^2)$.

La complejidad espacial no se puede reducir. Con estos ejemplos de corte de tubos y mochila hemos aprendido que:

- **Evitar repeticiones** guardando resultados mejora instantáneamente el coste de las soluciones descendentes.
- **Aprovechar la subestructura óptima** resulta muy eficiente.

Una estrategia de diseño es la siguiente:

- Diseñar la solución recursiva (*top-down*).
- Analizar los subproblemas.
- Verificar si podemos establecer una subestructura óptima.

4.3. Divide y Vencerás a Programación Dinámica

Esquema Divide y Vencerás (DC)

```

1 Solution DC( Problem p ) {
2   if( is_simple(p) ) return trivial(p);
3
4   list<Solution> s;
5   for( Problem q : divide(p) ) s.push_back( DC(q) );
6   return combine(s);
7 }
```

Esquema Programación dinámica (DP, recursiva)

```

1 Solution DP( Problem p ) {
2   if( is_solved(p) ) return M[p];
3   if( is_simple(p) ) return M[p] = trivial(p); // or simply: return trivial(p)
4
5   list<Solution> s;
6   for( Problem q : divide(p) ) s.push_back( DP(q) );
7   M[p] = combine(s);
8   return M[p];
9 }
```

Figure 19. Comparación Esquemas Divide y Vencerás y Programación Dinámica

4.4. PD recursiva vs PD iterativa

La complejidad asintótica suele ser la misma, pero la recursiva puede ser más eficiente. La complejidad espacial puede ser más eficiente en la iterativa. La PD no implica necesariamente una transformación a iterativo, pero suele referirse a la versión iterativa.

4.5. Cálculo del coeficiente binomial

Solución recursiva ineficiente:

- Si $T(n - 1, r) \geq T(n - 1, r - 1)$: $O(2^{n-r})$.
- Si $T(n - 1, r) \leq T(n - 1, r - 1)$: $O(2^r)$.
- Combinando los dos: $T(n, r) \sim g(n, r) \in O(2^{\min(r, n-r)})$.

NO es aceptable: resultados prohibitivos.

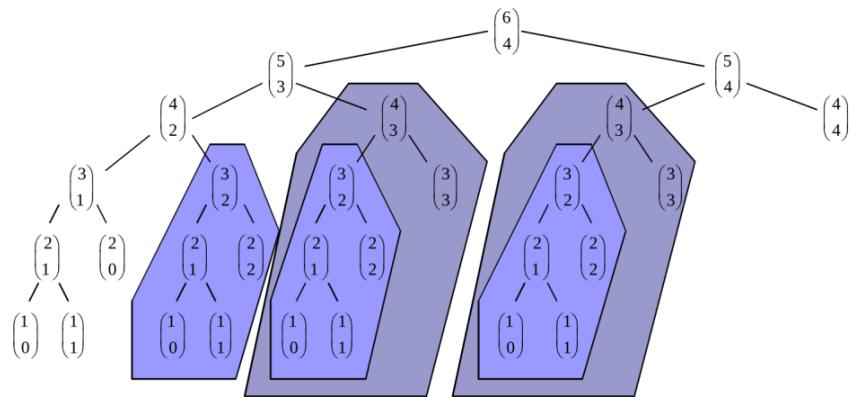


Figure 20. Problemas repetidos que se pueden ahorrar.

Podemos solucionar esto resolviendo los subproblemas en sentido ascendente y almacenando sus soluciones. Tendremos con esto una solución polinómica (mejorable) con $\Theta(rn)$.

5. Algoritmos voraces

El ejemplo introductorio es el problema de la mochila continuo. Es el mismo problema de siempre pero esta vez pudiendo fraccionar objetos. Este problema NO pertenece al grupo de problemas de optimización por selección discreta pues hay selecciones fraccionarias.

El algoritmo voraz siempre encontrará la solución óptima para este problema, siendo tan buena como cualquier otra solución factible.

Los algoritmos voraces implementan una estrategia de resolución para los problemas de optimización por selección discreta. Podemos obtener dos tipos de soluciones:

- **Factible:** Solución que satisface las restricciones del problema.
- **Óptima:** Solución factible que optimiza la función objetivo del problema.

Entonces, un algoritmo voraz toma decisiones siguiendo un criterio de selección voraz con la esperanza de que esta estrategia conduzca a la solución general óptima. Un criterio de selección voraz es aquel que siempre escoge la alternativa localmente óptima.

Esquema voraz

```

1 t_conjuntoElementos VORAZ(t_problema dp)
2 {
3     t_conjuntoElementos y, solucion;
4     elemento decision;
5     y=prepararDatos(dp);           // preparacion de datos para facilitar seleccion
6     while(noVacio(y) || !esSolucion(solucion)) { // quedan datos por seleccionar
7         // y aun no se ha llegado a la solucion
8         decision=selecciona(y);
9         if (esFactible(decision,solucion))
10             solucion=anadeElemento(decision,solucion);
11         y=quitaElemento(decision,y); // descartar en cualquier caso
12     }
13     return solucion;
14 }
15

```

Son fáciles de implementar y eficientes con costes normalmente polinómicos o $n \log(n)$. No obstante, la selección de óptimos locales no siempre conduce a óptimos globales.

5.1. Ejemplos

5.1.1. Problema de la mochila continua:

- **Complejidad:** $O(n \log n)$
- **Explicación:** La ordenación de los objetos según su relación v_i/w_i domina el tiempo total del algoritmo.

5.1.2. Problema de la mochila discreta (aproximación voraz):

- **Complejidad:** $O(n \log n)$
- **Explicación:** Similar al caso continuo, pero no siempre encuentra la solución óptima.
- NO resuelve el problema.

5.1.3. Problema del cambio:

- **Complejidad:** $O(n)$
- **Explicación:** El algoritmo selecciona iterativamente la moneda de mayor valor posible, reduciendo la cantidad restante hasta llegar a cero. Funciona óptimamente en sistemas monetarios adecuados.

5.1.4. Algoritmo de Prim

El algoritmo de Prim se utiliza para encontrar el Árbol de Recubrimiento de Coste Mínimo (MST) en un grafo ponderado y conexo. Su funcionamiento es incremental:

- **Inicialización:** Se parte de un vértice inicial y se marca como visitado.
- **Selección:** En cada paso, se elige la arista de menor peso que conecta un vértice visitado con uno no visitado.
- **Actualización:** La arista seleccionada se añade al árbol y el vértice se marca como visitado.
- **Criterio de parada:** El proceso termina cuando todos los vértices han sido visitados.

Complejidad:

- En su versión básica, tiene una complejidad de $O(V^3)$.
- Optimizando con índices y estructuras como montículos, se reduce a $O(V^2)$.

5.1.5. Algoritmo de Kruskal

El algoritmo de Kruskal también encuentra el Árbol de Recubrimiento de Coste Mínimo, pero sigue un enfoque diferente:

- **Inicialización:** Se ordenan todas las aristas por peso en orden ascendente.
- **Selección:** Cada arista se evalúa en orden y se añade al árbol si no forma ciclos.
- **Unión de componentes:** Cuando se añade una arista, las dos componentes que conecta se fusionan en una.
- **Criterio de parada:** El proceso termina cuando se han añadido $V - 1$ aristas.

Complejidad:

- Ordenar las aristas tiene complejidad $O(E \log E)$, donde E es el número de aristas.
- Las operaciones de conjuntos disjuntos (Union-Find) para manejar componentes tienen complejidad $O(V \log V)$.
- En total, la complejidad es $O(E \log E + V \log V)$.

Comparativa

- **Prim:** Mejor en grafos densos, donde hay muchas aristas.
- **Kruskal:** Más eficiente en grafos dispersos, con pocas aristas.

5.1.6. Fontanero diligente:

- **Complejidad:** $O(n \log n)$
- **Explicación:** Ordena las tareas por tiempo de ejecución para minimizar el tiempo medio de espera.

5.1.7. Asignación de tareas:

- **Complejidad:** $O(n^3)$
- **Explicación:** Utiliza programación dinámica para encontrar la asignación óptima, evaluando todas las combinaciones posibles.

6. Vuelta atrás

Generar todas las combinaciones de la mochila discreta supone un coste temporal del orden de $\Theta(n2^n)$.

Para generar solo soluciones factibles basta con imprimir las soluciones (nodos hoja) que cumplen:

$$\sum_{i=1}^n x_i w_i \leq W$$

La búsqueda del valor óptimo se puede acelerar evitando explorar ramas que no pueden dar soluciones factibles ni soluciones peores que las que ya tenemos.

6.1. Ajuste de podas

Interesa que los mecanismos de poda actúen lo antes posible. Una poda más ajustada se puede obtener usando la solución voraz al problema de la mochila continua.

La solución al problema de la mochila continua siempre será mayor que la solución al problema de la mochila discreta.

La efectividad de la poda puede aumentarse partiendo de una solución factible muy buena.

Puede ser relevante el orden en el que se exploran las soluciones y la manera en la que se despliega el árbol.

6.2. Definición

Vuelta atrás proporciona una forma sistemática de generar todas las posibles soluciones a un problema, generalmente usado en la resolución de problemas de selección u optimización en los que el conjunto de soluciones posibles es finito. Las soluciones que se pretenden encontrar son:

- Factibles: Satisfacen las restricciones del problema.
- Óptimas: Optimizan una cierta función objetivo.

Se trata de un recorrido sobre una estructura arbórea imaginaria. La solución se debe poder expresar mediante una tupla de decisiones:

- Dominios discretizables.
- Número de soluciones finito.

La estrategia puede proporcionar:

- Una solución factible.
- Todas las soluciones óptimas.

- La solución óptima al problema.
- Las n mejores soluciones factibles al problema.

En la mayoría de los casos, las soluciones son prohibitivas.

NO hay estrategia de búsqueda.

6.3. Podas

Cuanto más ajustadas sean las cotas, más podas se producirán.

6.3.1. Cota optimista

Estima, a mejor, el mejor valor que podría alcanzarse al expandir el nodo. Puede que no haya ninguna solución factible que alcance ese valor. Normalmente se obtienen relajando las restricciones del problema.

Si la cota optimista de un nodo es peor que la solución en curso, se puede podar el nodo.

6.3.2. Cota pesimista

Valor seguro (puede no ser el mejor) que puede alcanzarse al expandir el nodo. Debe corresponder con una solución factible (no tiene por qué ser la mejor). Normalmente se obtienen mediante soluciones voraces del problema.

Se puede eliminar un nodo si su cota optimista es peor que la mejor cota pesimista.

6.4. Mochila discreta

Usando Vuelta atrás con el problema de la mochila, las restricciones son:

- Peso límite.
- Objetos no partibles.

Las relajaciones, en cambio:

- No hay límite de peso.
- Podemos partir los objetos.

El esquema general de Vuelta atrás es:

```

1 void backtracking(node n, solution &current_best) {
2     if ( is_leaf(n) ) {
3         visited_leaf_nodes++;
4         if( is_best( solution(n), current_best ) )
5             current_best = solution(n);
6         return;

```

```

7 }
8
9   for( node a : expand(n) )
10    visited_nodes++;
11    if( is_feasible(a) ) {
12      if( is_promising(a) ) {
13        explored_nodes++;
14        backtracking( a, current_best );
15      } else
16        no_promising_discarded_nodes++;
17    } else
18      no_feasible_discarded_nodes++;
19    return;
20  }

```

6.5. Problemas

6.5.1. Permutaciones

Dado un entero positivo n , mostrar todas las combinaciones de la secuencia $(0, \dots, n-1)$. Es decir, para $n = 4$, por ejemplo, $(0, 1, 2, 3)$, $(1, 2, 3, 0)$, etc.

Este problema se resuelve:

1. Separando los elementos ya usados (izquierda) de los sin usar (derecha) en un vector.
2. Estableciendo un pivote que se encuentra a la izquierda de los no usados.
3. Se va intercambiando el pivote con cada uno de los no usados.
4. Para cada uno de los intercambios se integra el pivote en los usados.

Se usa la función `swap` para mejorar el coste temporal. De otra manera, tiene un coste temporal constante.

6.5.2. El viajante de comercio

Dado un grafo ponderado $g = (V, E)$ con pesos no negativos, el problema consiste en encontrar un ciclo hamiltoniano⁶ de mínimo coste. El peso de un ciclo viene dado por la suma de las aristas que lo compone.

Restricciones del problema:

- No se puede visitar dos veces el mismo vértice.
- Que una arista no tenga un peso infinito.
- Que una arista cierre el camino.

⁶Recorrido en el grafo que recorre todos los vértices una sola vez y regresa al de partida.

El viajante de comercio usa la función `swap` también, pero la propuesta con este *approach* es inviable, pues la complejidad temporal es $O(n!)$. Para mejorar este enfoque se puede realizar lo siguiente:*

- Cálculo incremental de la vuelta.
- Cota optimista:
 - Restricciones:
 - * Ha de pasar por todas las ciudades.
 - * Ha de ser continuo.
 - Relajaciones:
 - * No pasar por todas las ciudades.
 - * Ir saltando de una ciudad a otra.
- Cota pesimista:
 - Poda basada en la mejor solución hasta el momento. Buscar una solución subóptima.
 - * Algoritmo voraz.

Este problema tiene una complejidad temporal de $O(n!)$ si se realiza mediante fuerza bruta.

Enfoque de ramificación y poda en la sección [7.5](#).

6.6. *N* Reinas

Consiste en obtener todas las formas de colocar n reinas de forma que no se ataquen mutuamente.

Restricciones:

- No puede haber dos reinas en la misma fila.
- No puede haber dos reinas en la misma columna.
- No puede haber dos reinas en la misma diagonal.

Tiene un coste temporal de $O(n!)$.

6.7. Función compuesta mínima

Dadas dos funciones $f(x)$ y $g(x)$ y dados dos números cualesquiera x e y , encontrar la función compuesta mínima que obtiene el valor y a partir de x tras aplicaciones sucesivas e indistintas de $f(X)$ y $g(x)$.

Se pretende minimizar el tamaño de la composición y se asume un máximo de M composiciones para evitar ramas infinitas.

Restricciones:

- $k < M$ para evitar búsquedas infinitas.
- Siempre se puede calcular $f(X, k, x)$.
- $F(X, k, x) \neq F(X, i, x) \forall i < k$ para evitar recálculos.
- $k < v_b$ tupla prometedora ($v_b = \text{mejor solución actual}$).

Enfoque de ramificación y poda en la sección [7.6](#).

7. Ramificación y poda

Refinamiento de vuelta atrás. Permite definir estrategias de búsqueda o exploración

Selecciona el siguiente nodo a expandir de entre todos los visitados, definiendo así lo que es la búsqueda informada. Útil en IA.

7.1. Aspectos en común con vuelta atrás

Enumeración parcial del espacio de soluciones mediante la generación de un árbol de expansión.

7.2. Definiciones

Nodo vivo (o prometedor) Aquel con posibilidades de ser ramificado.

LNV - Lista de nodos vivos Estructura donde se almacenan los nodos vivos.

Estrategia de búsqueda Forma en la que se recorre la lista de nodos vivos para extraer el siguiente nodo a expandir. Hay:

- **FIFO:** Recorrido en anchura, se implementa con una cola.
- **LIFO:** Recorrido en profundidad, se implementa con una pila.
- **Dirigidas:** Cola de prioridad siguiendo una heurística determinada.

7.3. Funcionamiento

1. Se parte de un nodo inicial y se inserta en la lista de nodos vivos.
2. Se asigna una solución pesimista.
3. Selección:
 - (a) Extracción del nodo de la LNV.
 - (b) Actualización de la mejor solución.
4. Ramificación:
 - (a) Se expande el nodo seleccionado.
5. Poda:
 - (a) Se eliminan nodos que no contribuyen a la solución.
 - (b) El resto de nodos se añaden a la LNV.
6. Cuando la LNV queda vacía, se termina el algoritmo.

7.4. Esquema de ramificación y poda

```

1 solution branch_and_bound(problem p) {
2     node initial = initial_node(p);
3     solution current_best = pessimistic_solution(initial);
4     priority_queue<Node> q;
5     q.push(initial);
6
7     while (!q.empty()) {
8         node n = q.top();
9         q.pop();
10
11         if (!is_promising(n, current_best)) {
12             discarded_promising_nodes++;
13             continue;
14         }
15
16         if (is_leaf(n)) {
17             completed_nodes++;
18             if (is_better(solution(n), current_best)) {
19                 current_best_updates_from_completed_nodes++;
20                 current_best = solution(n);
21             }
22             continue;
23         }
24
25         expanded_nodes++;
26         for (node a : expand(n)) {
27             visited_nodes++;
28
29             if (is_feasible(a)) {
30                 if (is_better(pessimistic_solution(a),
31                               current_best)) {
32                     current_best_updates_from_pessimistic_bounds
33                    ++;
34                     current_best = pessimistic_solution(a);
35
36                     if (is_promising(a, current_best)) {
37                         explored_nodes++;
38                         q.push(a);
39                     } else {
40                         no_promising_discarded_nodes++;
41                     }
42                 }
43             }
44         }
45     }
}

```

```
46     return current_best;
47 }
48 }
```

La estrategia puede proporcionar:

- Todas las soluciones factibles.
- Una solución al problema.
- La solución óptima al problema.
- Las n mejores soluciones.

Desventajas:

- Encontrar una buena cota optimista.
- Encontrar una buena cota pesimista.
- Encontrar una buena estrategia de exploración.
- Mayor requerimiento de memoria que Vuelta atrás.
- Complejidades muy altas en el peor de los casos.

Ventajas:

- Más rápidas.

Si se realizan podas muy agresivas puede que se pierda la solución óptima.

7.5. Viajero de comercio

Se usa como cota:

- **Optimista:** *minimum spanning tree* de las ciudades restantes.
- **Pesimista:** algoritmo voraz basado en la ciudad más cercana.

Se acelera el algoritmo de vuelta atrás casi x2 usando este enfoque de ramificación y poda.

7.6. Función compuesta mínima

Memorizando, se obtiene casi la mitad de iteraciones que vuelta atrás. Sin memorizar, todos los enfoques son bastante parecidos en cuanto a número de iteraciones.

7.7. Problemas

7.7.1. El Puzzle

Tablero de n^2 casillas y $n^2 - 1$ piezas numeradas del 1 al $n^2 - 1$. Objetivo: ordenar la disposición de las fichas dejando libre la última casilla (n,n).

Según se relaje el problema tenemos dos funciones de coste diferentes:

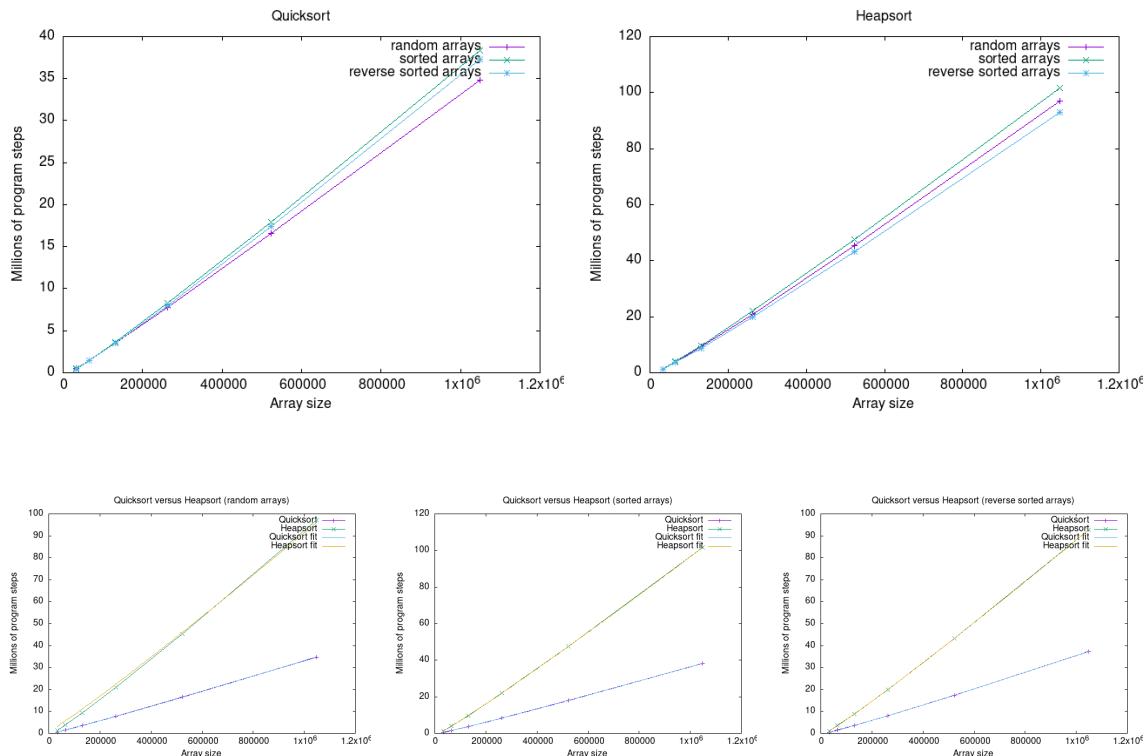
- Calcular el número de piezas que están en una posición distinta de la que les corresponde en la disposición final.
- Calcular la suma de las distancias de Manhattan⁷ desde la posición de cada pieza a su posición en la disposición final.

⁷Distancia entre dos puntos: $|x_1 - x_2| + |y_1 - y_2|$.

8. Prácticas

8.1. Práctica 2 - Complejidad temporal: Análisis empírico (II)

En esta práctica se han implementado los pasos de programa. Son una medida para medir la eficiencia de los algoritmos QuickSort y HeapSort. Las gráficas que (me) han salido han sido las siguientes:



Las funciones *fit* de las tres gráficas *qs-vs-hs* han sido:

- $y_{qs}(x) = a_{qs} * x * \log(x) + b_{qs}$
- $y_{hs}(x) = a_{hs} * x * \log(x) + b_{hs}$

Siendo:

- a : el tamaño del vector⁸.
- b : el número de pasos de programa para el algoritmo dado.

⁸En esta práctica se prueba desde 2^{15} hasta 2^{20}

8.2. Práctica 3 - Complejidad temporal: Cálculo analítico

Buscamos encontrar la complejidad temporal de 3 algoritmos diferentes dados.

8.2.1. Ejercicio 1 (resuelto)

```

1  int exercisel (vector < int > & v) {
2      int i, sum=0, n=v.size();
3      if (n>0) {
4          int j=n;
5          while (j>0 and sum<100) {
6              j=j/2;
7              sum=0;
8              for (i=j; i<n; i++) sum+=v[i];
9          }
10         return j;
11     } else
12         return -1;
13 }
```

Tamaño de problema: $n = v.size()$.

Tiene mejor y peor caso **NO** porque dependa del tamaño del vector, pues no podemos cambiar el tamaño del problema, sino porque para un tamaño n dado igual en dos casos con vectores con igual tamaño y distintos elementos el programa se comportará diferente. La clave está en la condición del while, pues si $sum < 100$ se cumple a la primera iteración, se omitirá el bucle while en seguida, siendo este el mejor caso. Sin embargo, si el $sum < 100$ no se cumple nunca, el bucle while se repetirá hasta que $j > 0$ sea falso.

En la corrección se dice:

Mejor caso: Las instancias que pertenecen al caso más favorable se caracterizan por ser vectores cuyos elementos, desde la mitad en adelante, suman al menos 100. Es decir, en el mejor de los casos están todos los vectores n , de cualquier tamaño n , tal que $\sum_{i=\frac{n}{2}}^{n-1} v_i \geq 100$. La complejidad temporal en este caso viene dada por:

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

Es $n - \frac{n}{2}$ porque el tamaño es n pero se le resta la otra mitad del vector que nunca se evalúa en este caso.

Peor caso: En el caso más desfavorable están (entre otros) todos los vectores cuyos elementos suman menos de 100. Es decir:

$$v \in Z^n \mid \sum_{i=0}^{n-1} v_i < 100$$

Para representar la complejidad, se representa en una tabla los pasos que sigue el programa (21).

| Iteración bucle while | j | Pasos en cada iteración |
|-----------------------|-------------------------|---|
| 1 | n | $\frac{n}{2}$ |
| 2 | $\frac{n}{2}$ | $\frac{n}{2} + \frac{n}{4}$ |
| 3 | $\frac{n}{4}$ | $\frac{n}{2} + \frac{n}{4} + \frac{n}{8}$ |
| ... | ... | ... |
| k | $\frac{n}{2^{k-1}} = 1$ | $\sum_{j=1}^k \frac{n}{2^j}$ |

Figure 21. Pasos ej1 P3.

La complejidad del algoritmo viene dada por la suma de los pasos en todas y cada una de las iteraciones del bucle `while`. Además, de la segunda columna se deduce que realiza $\log_2(n) + 1$ iteraciones. Por lo tanto:

$$c_s(n) = \sum_{k=1}^{\log_2(n)+1} \sum_{j=1}^k \frac{n}{2^j} = n \sum_{k=1}^{\log_2(n)+1} \sum_{j=1}^k \frac{1}{2^j}$$

Teniendo en cuenta que $\sum_{j=1}^k \frac{1}{2^j} \in \Theta(1)$, se tiene que:

$$c_s(n) = n \sum_{k=1}^{\log_2(n)+1} 1 \in O(n \log n)$$

8.2.2. Ejercicio 2

```

1 unsigned exercise2 (unsigned m) {
2     unsigned i=1, p=0;
3     while (i <= m) {
4         unsigned q=0, j=i;
5         while (j > 0) {
6             j--;
7             q++;
8         }
9         p+=q;
10        i*=3;
11    }
12    return p;
13 }
```

El tamaño del problema viene dado por m . No hay mejor ni peor caso, pues se trata de un `unsigned` sin contenido interno como puede ser el vector del ejercicio 1.

Realizamos la tabla (3)

| Iteración while ($i \leq m$) | i | Pasos |
|--------------------------------|-----------------|---------------------|
| 1 | 1 | 1 |
| 2 | 3 | $1 + 3j$ |
| 3 | 9 | $1 + 9j$ |
| 4 | 27 | $1 + 27j$ |
| ... | ... | ... |
| $k = \log_3(m)$ | $i_k = 3^{k-1}$ | $p_i = 1 + i_k * j$ |

Table 3. Pasos ejercicio 2.

Los pasos los podemos interpretar como

$$p_i = 1 + i_k * j = 1 + 3^{k-1} * j = 1 + 3^{\log_3(m)-1} * j = 1 + (m - 1) * j = 1 + jm - j$$

, siendo del orden de $O(m)$. Entonces, el programa queda con complejidad:

$$c_i(n) = O(\log m) + O(m) = O(m)$$

8.2.3. Ejercicio 3

```

1 void exercise3 (vector <int> &v) {
2     int i=1, n=v.size();
3     bool swaped=true;
4     while (swaped) {
5         swaped=false;
6         for (int j=n-1; j>=i; j--) {
7             if (v[j] < v[j-1]) {
8                 int x=v[j];
9                 v[j]=v[j-1];
10                v[j-1]=x;
11                swaped=true;
12            }
13        }
14        i++;
15    }
16 }
```

En este ejercicio contamos con un tamaño de problema $n = v.size()$.

Se trata de un programa que ordena de menor a mayor un vector dado desde el último número al primero. No obstante, si no hay ningún cambio durante la última pasada, el programa parará pues el vector ya estará ordenado. Es un algoritmo basado en *bubble sort* pero con la mejora de la variable *swaped*.

Aquí podemos diferenciar dos casos claros:

- **Peor caso:** Cuando el vector está ordenado a la inversa, es decir, de mayor a menor.
- **Mejor caso:** Cuando el vector ya está ordenado de menor a mayor.

Peor caso En este caso el bucle `for` será recorrido $n*n$ veces, pues la variable `swaped` nunca será false hasta que el bucle ya no pueda realizar más intercambios. La complejidad de este caso será entonces de $\Theta(n^2)$, igual que la de *Bubble Sort* en su peor caso.

Para comprenderlo mejor, miramos el número de iteraciones totales que se realizan en este caso. Estas siguen la siguiente suma aritmética (dentro del bucle `for`):

$$(n - 1) + (n - 2) + (n - 3) + \dots + 1 = \frac{n * (n - 1)}{2} = \frac{n^2 - n}{2}$$

Como todas las demás operaciones fuera del bucle `for` son inmediatas (de coste $O(1)$), podemos confirmar que el peor caso tiene complejidad temporal de $\Theta(n^2)$.

Mejor caso En este caso únicamente se recorrerá el bucle `for` una vez. Como no habrá cambio en la variable `swaped`, saltará al `while` al terminar las $j = n - 1$ veces que se ejecuta y terminará el programa. Por lo tanto, este caso tendrá una complejidad temporal de $\Omega(n)$.

8.2.4. Soluciones

Solución del ejercicio 2 de la práctica 3

Cálculo analítico de la complejidad temporal

Tal y como se especifica en el enunciado, la complejidad ha de calcularse en función del parámetro m (complejidad paramétrica). En cualquier caso, no tendría sentido expresarla en función del tamaño del problema ya que en este algoritmo, el tamaño del problema es constante.

La función no presenta caso mejor y peor dado que solo existe una instancia por cada valor de m . Por lo tanto, la expresión de complejidad que se obtenga corresponderá a un *coste exacto* que vendrá dado por el número total de pasos de programa que realiza el bucle interno, o lo que es lo mismo en este caso, el número total de iteraciones que hace dicho bucle interno (ya que el coste de cada iteración es constante).

La siguiente tabla representa una cuenta de dichas iteraciones mediante un desglose en cada una de las k iteraciones del bucle externo

| Iteración bucle externo | i | número de iteraciones del bucle interno |
|-------------------------|-----------|---|
| 1 ^a | 1 | 1 |
| 2 ^a | 3 | 3 |
| 3 ^a | 9 | 9 |
| ... | ... | ... |
| k -ésima | 3^{k-1} | 3^{k-1} |

En la segunda columna de la tabla se muestran los valores de i al comienzo de cada iteración del bucle externo (puede tomarse cualquier punto del bucle, no necesariamente al comienzo, pero ha de seguirse siempre el mismo criterio). Se ha escogido la variable i porque es la que determina el número de iteraciones que hará dicho bucle externo (dato que se necesita conocer) y, además, de ella depende el número de iteraciones del bucle interno.

El término general de la segunda columna se obtiene siguiendo la progresión numérica y relacionándola con cada iteración k ; es decir, ha de expresar el valor que toma i en función de k .

En la tercera columna se desglosa, para cada iteración del bucle externo, el número de iteraciones que hace el interno para cada valor de i concreto. Claramente, el número de pasos (o de iteraciones) que realiza el último bucle coincide con el valor de i con el que comienza. El caso general de esta columna ha de ponerse también en función de k , para obtener sumatorios coherentes.

Puesto que la complejidad viene dada por la suma de todos los elementos de la tercera columna, para obtenerla se necesita conocer el número de iteraciones

que hace el bucle externo, es decir, el mayor valor de k que se va a alcanzar, que depende en última instancia de m . Para conocerlo, hemos de relacionar k con m , algo que resulta sencillo hacer a través de la última iteración, en la que se cumple la ecuación (2.1) (de no ser así no sería la última iteración):

$$\frac{m}{3} < i \leq m \quad (2.1)$$

Por otra parte, a través de la segunda columna de la tabla, se dedujo que en cualquier iteración k , se cumple:

$$i = 3^{k-1} \quad (2.2)$$

Combinando (2.1) y (2.2) se tiene:

$$\frac{m}{3} < 3^{k-1} \leq m \quad (2.3)$$

Donde, por la forma de obtener (2.1), k corresponde a la última iteración, o lo que es lo mismo, al número de iteraciones que hace el bucle externo.

Operando adecuadamente en (2.3), se tiene:¹

$$m < 3^k \leq 3m$$

$$\lfloor \log_3 m \rfloor < k \leq \lfloor \log_3 m \rfloor + 1$$

Siendo k un número natural, el único que cumple ambas desigualdades es:

$$k = \lfloor \log_3 m \rfloor + 1 \quad (2.4)$$

Conociendo ya el número de iteraciones del bucle externo, la expresión de complejidad temporal de la función algorítmica, viene dada por la suma de todos los elementos de la tercera columna de la tabla (notar que se trata de una serie geométrica de razón 3; con $\lfloor \log_3 m \rfloor + 1$ elementos, siendo 1 el primero):²

$$c_e(n) = \sum_{k=1}^{\lfloor \log_3 m \rfloor + 1} 3^{k-1} = \frac{3^{\lfloor \log_3 m \rfloor + 1} - 1}{3 - 1} = \frac{1}{2}(3m - 1) \in \Theta(m).$$

¹El operador $\lfloor x \rfloor$ devuelve la parte entera de x .

²Por simplicidad y asumiendo que k es un número natural, tomar $k = \lfloor \log_3 m \rfloor$ o simplemente $k = \log_3 m$, también puede ser válido siempre que no afecte al orden de complejidad resultante (como ocurre en este caso).

Solución del ejercicio 3 de la práctica 3

Cálculo analítico de la complejidad temporal

Puesto que no se especifica lo contrario en el enunciado, la complejidad temporal ha de expresarse en función del tamaño del problema, al que se le llamará n . Viene dado por el número de elementos que tiene el vector.

El bucle `for` no presenta casos mejor y peor puesto que el número de iteraciones que realiza (todas con coste constante con n) no depende de valores concretos del vector v . Por el contrario, el número de iteraciones que hace el bucle `while` sí que depende de dichos valores.

Por lo tanto, la complejidad del algoritmo depende del contenido del vector. En estos casos, para expresarla se distingue entre el mejor de los casos (el más favorable) y el peor (el más desfavorable).

Complejidad temporal en el mejor de los casos:

En el mejor de los casos están todos los vectores, de cualquier tamaño, para los cuales el bucle `while` solo hace una iteración. Para que esto ocurra, la variable `swaped` debe quedar con valor `false` en su primera iteración; sucederá siempre que el vector de entrada esté ordenado de manera ascendente.

Por lo tanto, todas las instancias siguientes pertenecen al caso mejor:³

$$v \in \mathbb{Z}^n : v[i] \geq v[i-1], \forall i : 1 \leq i \leq n-1$$

Si asumimos que el bucle externo termina tras la primera iteración, en la que $i = 1$, se tiene:

$$c_i(n) = \sum_{j=1}^{n-1} 1 = n - 1 \in \Omega(n) \quad (3.1)$$

La serie (3.1) expresa la cuenta de pasos realizados por el algoritmo solo en la primera iteración del `while`, donde $i = 1$. Los índices del sumatorio expresan las iteraciones del bucle `for`; cada una de ellas contribuye con un coste constante, de ahí que solo se suma un paso de programa. Al tratarse de un bucle decreciente, se han permutado los límites superior e inferior del sumatorio. Notar que se ha ignorado cualquier otra contribución de pasos de programa cuyo tiempo de ejecución está acotado por una constante (es decir, no depende de n).

³Notar que son instancias de cualquier tamaño n .

Complejidad temporal en el peor de los casos:

En el peor de los casos están todos los vectores, de cualquier tamaño, para los cuales el bucle `while` realiza el máximo número de iteraciones. Esto ocurre cuando la variable `swaped` se queda con valor `false` por el hecho de que no se entra en el `for`. A su vez, esto solo puede ocurrir cuando la variable i alcanza el valor n . Notar que si la variable `swaped` se queda con valor `false` antes de que i alcance el valor máximo, entonces el bucle `while` no realiza el máximo número de iteraciones y por lo tanto, no sería una instancia del caso peor.

Para que i llegue hasta n , el primer elemento del vector debe ser estrictamente mayor que cualquiera de los otros, sin importar el orden que pueda haber entre los elementos del vector.⁴ Por lo tanto, todas las instancias siguientes pertenecen al caso peor:

$$v \in \mathbb{Z}^n : v[0] > v[i], \forall i : 1 \leq i \leq n - 1$$

La complejidad temporal viene dada por el acumulado total de pasos de programa que realiza el `for` para todas y cada una de las iteraciones del `while`. Puesto que el coste de cada iteración del bucle interno es constante, la cantidad de pasos de programa que consume coincide con el número de iteraciones que realiza. La siguiente tabla representa una cuenta de dichas iteraciones mediante un desglose en cada una de las k iteraciones del bucle externo.

| Iteración bucle externo | i | número de iteraciones del bucle interno |
|-------------------------|-----|---|
| 1 ^a | 1 | $n - 1$ |
| 2 ^a | 2 | $n - 2$ |
| 3 ^a | 3 | $n - 3$ |
| ... | ... | ... |
| k -ésima | k | $n - k$ |

Para escoger lo que se representa en cada columna se ha seguido el mismo criterio de el ejercicio anterior. Notar que la primera fila de la tabla corresponde al coste en el mejor de los casos, en el que el bucle externo solo hace una iteración.

Por la segunda columna, tenemos $i = k$ y, según se ha explicado anteriormente, en la última iteración del `while` se cumple $i = n$.

Por lo tanto, el número de iteraciones que hace el `while` en el peor de los casos es n . Sumando los elementos de la tercera columna obtenemos la expresión de complejidad buscada:

$$c_s(n) = \sum_{k=1}^n (n - k) = n \frac{n - 1 + 0}{2} = \frac{1}{2}(n^2 - n) \in O(n^2)$$

⁴Notar que los vectores ordenados de manera descendente son un caso particular del más desfavorable, pero hay muchos más vectores que están en el caso peor.

8.3. Práctica 4 - Complejidad temporal: Cálculo analítico (II)

8.3.1. Ejercicio 1 (resuelto)

```

1 float Mochila(vector<float> &v, vector<unsigned> &p, unsigned P,
2     int i) {
3     float S1, S2;
4     if (i >= 0) {
5         if (p[i] <= P)
6             S1 = v[i] + Mochila(v, p, P - p[i], i - 1);
7         else
8             S1 = 0;
9         S2 = Mochila(v, p, P, i - 1);
10        return max(S1, S2);
11    }
12    return 0;
}

```

El tamaño del problema viene dado por el número de elementos de ambos vectores: n . Asumimos que en la llamada inicial el parámetro i toma valor $n-1$. El algoritmo presenta mejor y peor caso.

Mejor caso Cuando $p_i > P \forall i$. Así, nunca se hará la llamada recursiva de la línea 5. No obstante, la de la línea 8 siempre lo hará, así que tenemos que:

$$T(n) = \begin{cases} 1 & n < 0 \\ 1 + T(n-1) & n \geq 0 \end{cases}$$

Aplicamos el método de sustitución para resolver la recurrencia:

$$f(n) = 1 + T(n-1) = 1 + (1 + f(n-2)) = 2 + T(n-2) = 3 + T(n-3) = \dots = k + T(n-k)$$

Tomando $k = n+1$, tenemos que $T(n) = n+1 + T(n-n+1) = n+1 + T(-1)$. Como $T(-1) = 1$, tenemos que $T(n) = n+1+1 = n+2$, siendo así $\Omega(n)$ su complejidad temporal.

Peor caso Al contrario que el mejor caso, este peor caso ocurrirá cuando $p_i \leq P$.

Resolviendo:

$$T(n) = \begin{cases} 1 & n < 0 \\ 1 + 2T(n-1) & n \geq 0 \end{cases}$$

$$T(n) = 1 + 2T(n-1) = 1 + 2 + 4T(n-2) = 1 + 2 + 4 + 8T(n-3) = \dots = (\sum_{i=0}^{k-1} 2^i) + 2^k T(n-k)$$

Entonces, con $k = n + 1$ tenemos que $T(n) = (\sum_{i=0}^{n+1} 2^i) + 2^{n+1}T(n-n+1) = (\sum_{i=0}^{n+1} 2^i) + 2^{n+1}T(1) = (\sum_{i=0}^{n+1} 2^i) + 2^{n+1}$, siendo así la complejidad temporal de $\Theta(2^n)$.

8.3.2. Ejercicio 2

```

1 void abstracto(unsigned n) {
2     if (n > 1) {
3         for (unsigned i = 1; i < n - 1; i++)
4             cout << "*";
5         cout << endl;
6
7         for (unsigned i = 0; i < 4; i++)
8             abstracto(n / 2);
9     }
10 }
```

Este método no contiene mejor y peor caso. Realiza un printeo de asteriscos hasta $n-1$ y realiza una llamada recursiva con $\frac{n}{2}$ como parámetro. La recursión termina cuando $n \leq 1$. Tenemos entonces que:

$$T(n) = \begin{cases} 1, & n \leq 1 \\ n - 1 + 4T(\frac{n-1}{2}), & n > 1 \end{cases}$$

Usando sustitución, podemos resolver la ecuación de la siguiente manera:

$$\begin{aligned} f(n) &= n - 1 + 4T\left(\frac{n-1}{2}\right) = (n-1) - 1 + 4 * 4T\left(\frac{(n-1)-1}{2}\right) = n - 2 + 16T\left(\frac{n-2}{2}\right) = \dots \\ &= n - k + 4kT\left(\frac{n-k}{2}\right) \end{aligned}$$

Para obtener $T(1)$, la variable k deberá: $1 = \frac{n-k}{2} \rightarrow 2 = n - k \rightarrow k = n - 2$. Entonces: $T(n) = n - (n-2) + 4(n-2)T\left(\frac{n-(n-2)}{2}\right) = 2 + (4n-8)T(1) = 2 + 4n - 8 = 4n - 6$. Tenemos así que la complejidad temporal de este algoritmo es de $O(n)$.

8.3.3. Ejercicio 3

```

1 bool palind(string &pal, int pri, int ult) {
2     if (pri >= ult)
3         return true;
4     else
```

```

5         return (pal[pri] == pal[ult]) && palind(pal, pri + 1,
6             ult - 1);
}

```

El tamaño de problema de este método depende de la palabra que se pase como parámetro. Asumimos que las variables `pri` y `ult` siempre serán `pri<ult` en la primera llamada. Tiene mejor y peor caso los cuales dependen del `and` dentro del `return` del `else`, pues si la primera comparación ya es `false`, no se harán las llamadas recursivas.

Mejor caso Cuando `pal[pri]==pal[ult]` sea `false` en la primera comparación, el algoritmo tendrá una complejidad temporal de $\Omega(1)$, pues sólo realizará un `return`.

Peor caso Cuando `pal[pri]==pal[ult]` sea siempre `true`, el algoritmo realizará todas las llamadas recursivas posibles sobre una palabra n , dejando la relación de recursión siguiente:

$$T(n) \begin{cases} 1 & n \leq 0 \\ 1 + T(n-2) & n > 0 \end{cases}$$

Mediante sustitución:

$$f(n) = 1 + T(n-2) = 2 + T(n-4) = 3 + T(n-6) = \dots = k + T(n - \sum_{i=1}^k 2i)$$

Tendremos entonces que $1 = n - 2k \rightarrow k = \frac{n}{2}$, entonces $T(n) = \frac{n}{2} + T(n - 2(\frac{n}{2})) = \frac{n}{2} + T(n - n) = \frac{n}{2} + 1$, siendo $\Theta(n)$ la complejidad temporal del algoritmo en el peor caso.

8.3.4. Soluciones

Solución del ejercicio 2 de la práctica 4

Cálculo analítico de la complejidad temporal

Se ha de calcular la complejidad en función del parámetro n (complejidad paramétrica) haciendo uso de las relaciones de recurrencia. La función no presenta caso mejor y peor dado que solo existe una instancia por cada valor de n . Por lo tanto, la expresión de complejidad que se obtenga corresponderá a un *coste exacto*.

Sea $T(n)$ una relación de recurrencia que expresa el número de pasos de programa que realiza el algoritmo. Para obtenerla, hemos de identificar el caso base y el caso general, que se define en función de sí misma.

El caso base se obtiene directamente del algoritmo, identificando cuándo termina sin llamada recursiva. En este algoritmo ocurre cuando $n \in \{0, 1\}$. Por otra parte, hemos de calcular los pasos que realiza la función en esta situación. En el caso base de la función algorítmica, se realiza solo 1 paso de programa. Así obtenemos:

$$T(n) = 1 \text{ si } n \in \{0, 1\} \quad (2.1)$$

Para obtener el caso general de la recurrencia, en primer lugar hay que determinar la cantidad de pasos de programa que realiza el algoritmo (también en su caso general) sin tener en cuenta las llamadas recursivas; se trata, por lo tanto, del coste del **for**, que realiza n iteraciones todas ellas de coste constante:

$$c_{\text{for}}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n) \quad (2.2)$$

En segundo lugar, hay que añadir la contribución de las llamadas recursivas formulándolas de manera recurrente según $T(f(n))$, donde $f(n)$ expresa la forma en la que se reduce el problema en función del valor de entrada n . En este caso tenemos cuatro llamadas recursivas; todas ellas reducen n de la misma manera: mediante la división entera $n/2$, es decir, $f(n) = n/2$. Combinando esto con lo obtenido en (2.2) se obtiene el caso general:

$$T(n) = \Theta(n) + 4T(n/2) \text{ si } n > 1 \quad (2.3)$$

Notar que $\Theta(n)$ incluye otras instrucciones cuyo coste temporal está acotado por una constante: el **cout** que está fuera del bucle y las 4 iteraciones del segundo **for**.

Con (2.1) y (2.3) obtenemos la relación de recurrencia completa:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 4T(n/2) & n > 1 \end{cases}$$

Para evitar expresiones farragosas en la resolución de relación de recurrencia, se ha sustituido $\Theta(n)$ por la función más sencilla incluida en ese conjunto, es decir, n .

Ahora hemos de resolver la relación de recurrencia para obtener el orden de complejidad al que pertenece. Resolverla se refiere a expresar la misma función $T(n)$ sin hacer uso de la recursividad. Este tipo de recurrencias se resuelven utilizando el método *sustitución*. Se de trata comenzar en el caso general y realizar sustituciones sucesivas de las llamadas recursivas. Cada una de ellas se reemplaza por lo que resulta de aplicarle de nuevo la recurrencia. Así hasta poder expresar $T(n)$ en función de una profundidad recursiva cualquiera k :

$$\begin{aligned} T(n) &\stackrel{1}{=} n + 4T\left(\frac{n}{2}\right) \\ &\stackrel{2}{=} n + 4\left(\frac{n}{2}\right) + 4^2T\left(\frac{n}{2^2}\right) \\ &\stackrel{3}{=} n + 4\left(\frac{n}{2}\right) + 4^2\left(\frac{n}{2^2}\right) + 4^3T\left(\frac{n}{2^3}\right) \\ &\stackrel{4}{=} n + 4\left(\frac{n}{2}\right) + 4^2\left(\frac{n}{2^2}\right) + 4^3\left(\frac{n}{2^3}\right) + 4^4T\left(\frac{n}{2^4}\right) \\ &\dots \\ &\stackrel{k}{=} n \sum_{i=0}^{k-1} 2^i + 4^kT\left(\frac{n}{2^k}\right) = n(2^k - 1) + 4^kT\left(\frac{n}{2^k}\right) \end{aligned}$$

Por lo tanto, $T(n)$ expresado en función de una profundidad de recursión cualquiera k , es:

$$T(n) = n(2^k - 1) + 4^kT\left(\frac{n}{2^k}\right) \quad (2.4)$$

Recordando que se pretende expresar $T(n)$ de una manera no recurrente, de todos los posibles valores que pueda tomar k , interesa el último pues es el que permite expresar $T\left(\frac{n}{2^k}\right)$ sin hacer uso de la recursividad (puesto que corresponde al caso base). Así pues, tomamos un valor cualquiera de k tal que $\frac{n}{2^k} \leq 1$. $k = \lfloor \log_2 n \rfloor$ lo cumple.

Sustituimos el valor obtenido de k en la expresión (2.4):

$$T(n) = n(2^{\lfloor \log_2 n \rfloor} - 1) + 4^{\lfloor \log_2 n \rfloor}T(1)$$

Sabiendo, a partir de la recurrencia, que $T(1) = 1$ y simplificando la nueva expresión, obtenemos:

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

Ya tenemos expresado $T(n)$ sin hacer uso de la recursividad. Puesto que $T(n)$ expresa el coste temporal exacto del algoritmo, concluimos que este es:

$$c_e(n) \in \Theta(n^2)$$

Solución del ejercicio 3 de la práctica 3

Cálculo analítico de la complejidad temporal

Hemos de expresar la complejidad temporal en función del tamaño del problema, al que llamaremos n . En este algoritmo viene dado por el número de elementos de la cadena que se van a procesar, es decir $n = \text{ult} - \text{pri} + 1$.

Dado que el operador lógico `and`, del lenguaje C/C++, no resuelve el segundo operando si el primero resulta ser `False`, el algoritmo presenta caso mejor y caso peor.

Complejidad temporal en el mejor de los casos:

Para que una instancia del problema a resolver sea del caso mejor, la expresión lógica contenida en el `return` se debe evaluar con `False` en la primera llamada al algoritmo, es decir, no se debe realizar ninguna llamada recursiva. Para que esto ocurra se ha de cumplir $\text{pal}[\text{pri}] \neq \text{pal}[\text{ult}]$, donde `pri` y `ult` son los valores que inicialmente recibe el algoritmo. En este caso, se tiene:

$$c_i(n) = 1 \in \Omega(1)$$

Complejidad temporal en el peor de los casos:

En el peor de los casos están todas las instancias para las que se realiza el máximo número posible de llamadas recursivas, es decir, todas las cadenas `pal` que forman un palíndromo entre las posiciones `pri` y `ult` que inicialmente recibe el algoritmo.

El número de pasos de programa que realiza el algoritmo en el peor de los casos viene dado por la siguiente relación de recurrencia.

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 1 + T(n-2) & n > 1 \end{cases}$$

Notar que en cada llamada recursiva el tamaño del problema se reduce en 2 unidades; el coste del algoritmo sin tener en cuenta la única llamada recursiva que hay es 1.

Resolviendo mediante sustituciones sucesivas:

$$\begin{aligned} T(n) &\stackrel{1}{=} 1 + T(n - 2) \\ &\stackrel{2}{=} 2 + T(n - 4) \\ &\stackrel{3}{=} 3 + T(n - 6) \\ &\dots \\ &\stackrel{k}{=} k + T(n - 2k) \end{aligned}$$

Por lo tanto, $T(n)$ expresado en función de una profundidad de recursión cualquiera k , es:

$$T(n) = k + T(n - 2k) \quad (3.1)$$

Tomando un valor de k que cumpla $n - 2k \leq 1$; por ejemplo, $k = (n - 1)/2$; sabiendo que $T(1) = 1$ y sustituyendo en (3.1), tenemos:

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

Conclusión:

$$c_s(n) \in O(n)$$

La complejidad temporal del algoritmo está entre $\Omega(1)$ y $O(n)$.

8.4. Práctica 5 - Divide y vencerás

En esta práctica se pide implementar tres versiones de la función `pow` con diferente coste temporal.

Se han implementado las siguientes funciones:

- $O(n)$:

```

1 unsigned long pow2_1(unsigned n, long long &pasos)
2 {
3     if (n == 0)
4     {
5         pasos++;
6         return 1;
7     }
8     pasos++;
9     return 2 * pow2_1(n - 1, pasos);
10 }
```

- $O(\log n)$:

```

1 unsigned long pow2_2(unsigned n, long long &pasos)
2 {
3     if (n == 0)
4     {
5         pasos++;
6         return 1;
7     }
8
9     if (n % 2 == 0)
10    {
11        pasos++;
12        int mitad = pow2_2(n / 2, pasos);
13        return mitad * mitad;
14    }
15    else
16    {
17        pasos++;
18        return 2 * pow2_2(n - 1, pasos);
19    }
20 }
```

- $O(n^2)$:

```

1 unsigned long pow2_3(unsigned n, long long &pasos)
2 {
3     pasos++;
```

```
4     int result = 1;
5
6     for (unsigned int i = 0; i < n; i++)
7     {
8         pasos++;
9         int temp = 0;
10        for (int j = 0; j < result; j++)
11        {
12            pasos++;
13            temp += 2;
14        }
15        result = temp;
16    }
17    return result;
18 }
```

8.5. Práctica 6 - El problema del laberinto

Esta práctica se lleva a cabo durante varias semanas y en dos partes.

8.5.1. 6.1 - El problema del laberinto I

En esta primera parte se pide implementar los argumentos, la versión recursiva naive y la recursiva con *memoization*.

8.5.2. 6.2 - El problema del laberinto II

En esta parte se pide terminar la práctica implementando la versión de programación dinámica con matriz y la de con vector.

Vemos la práctica completa a continuación.

8.5.3. Naive

Un enfoque naive es una solución ingenua, directa, simple a un problema. Generalmente no suele estar optimizada y suele ser fácil de implementar.

El código naive del laberinto implementado ha sido el siguiente:

```

1 int maze_naive_helper(int **maze, int n, int m, int i, int j)
2 {
3     if (i >= n || j >= m || maze[i][j] == 0)
4         return -1;
5
6     // Si hemos llegado al final
7     if (i == n - 1 && j == m - 1)
8         return 1;
9
10    // Intentamos movimientos posibles:
11    int right = maze_naive_helper(maze, n, m, i, j + 1);
12        // derecha
13    int down = maze_naive_helper(maze, n, m, i + 1, j);
14        // abajo
15    int diagonal = maze_naive_helper(maze, n, m, i + 1, j + 1);
16        // diagonal
17
18    // Encontramos el minimo camino valido entre los posibles
19    int min_path = -1;
20
21    if (right != -1) // Si el camino a la derecha es valido, lo
22        tomamos como minimo
23        min_path = right;
24
25    if (down != -1)           // Si el camino
26        hacia abajo es valido, lo tomamos como minimo

```

```

22     if (min_path == -1 || down < min_path) // Si no hay
23         camino o el camino hacia abajo es mas corto, lo
24         tomamos como minimo
25         min_path = down;
26
27     if (diagonal != -1)                      // Si el
28         camino en diagonal es valido, lo tomamos como minimo
29         if (min_path == -1 || diagonal < min_path) // Si no hay
30             camino o el camino en diagonal es mas corto, lo
31             tomamos como minimo
32             min_path = diagonal;
33
34     // Si al menos un camino es valido, sumamos 1 al resultado
35     if (min_path != -1)
36         return 1 + min_path;
37
38     return -1;
39 }
40
41 int maze_naive(int **maze, int n, int m)
42 {
43     // Comprobamos si el origen o el final son inaccesibles
44     // antes de hacer nada.
45     if (maze[0][0] == 0 || maze[n - 1][m - 1] == 0)
46         return -1;
47
48     return maze_naive_helper(maze, n, m, 0, 0);
49 }
```

8.5.4. Memoización

Desarrollar un enfoque de memoización implica gastar en espacio para ganar en tiempo. Se guardan instancias de problemas que ya han sido resueltos por el algoritmo en una matriz y se revisa durante el algoritmo si la solución al problema actual lo tenemos guardado en tal matriz.

```

1 int maze_memo_helper(int **maze, int n, int m, int i, int j, int
2     **memo)
3 {
4     // Si estamos fuera de los limites o en una celda bloqueada
5     if (i >= n || j >= m || maze[i][j] == 0)
6         return -1;
7
8     if (memo[i][j] != -2)
9         return memo[i][j];
10
11    // Si hemos llegado al final
12    if (i == n - 1 && j == m - 1)
```

```
12     {
13         memo[i][j] = 1;
14         return 1;
15     }
16
17     // Intentamos movimientos posibles:
18     int right = maze_memo_helper(maze, n, m, i, j + 1, memo);
19         // derecha
20     int down = maze_memo_helper(maze, n, m, i + 1, j, memo);
21         // abajo
22     int diagonal = maze_memo_helper(maze, n, m, i + 1, j + 1,
23         memo); // diagonal
24
25     // Encontramos el minimo camino valido entre los posibles
26     int min_path = -1;
27
28     if (right != -1) // Si el camino a la derecha es valido, lo
29         tomamos como minimo
30         min_path = right;
31
32     if (down != -1) // Si el camino
33         hacia abajo es valido, lo tomamos como minimo
34         if (min_path == -1 || down < min_path) // Si no hay
35             camino o el camino hacia abajo es mas corto, lo
36             tomamos como minimo
37             min_path = down;
38
39     if (diagonal != -1) // Si el
40         camino en diagonal es valido, lo tomamos como minimo
41         if (min_path == -1 || diagonal < min_path) // Si no hay
42             camino o el camino en diagonal es mas corto, lo
43             tomamos como minimo
44             min_path = diagonal;
45
46     // Si al menos un camino es valido, sumamos 1 al resultado
47     if (min_path != -1)
48         memo[i][j] = 1 + min_path;
49     else
50         memo[i][j] = -1;
51
52     return memo[i][j];
53 }
54
55 int maze_memo(int **maze, int n, int m, int **&memo)
56 {
57     // Comprobamos si el origen o el final son inaccesibles
58     // antes de hacer nada.
59     if (maze[0][0] == 0 || maze[n - 1][m - 1] == 0)
60         return -1;
```

```

50
51     // Inicializamos la matriz de memoizacion
52     memo = new int *[n];
53     for (int i = 0; i < n; i++)
54     {
55         memo[i] = new int[m];
56         for (int j = 0; j < m; j++)
57             memo[i][j] = -2;
58     }
59
60     return maze_memo_helper(maze, n, m, 0, 0, memo);
61 }
```

8.5.5. Iterativo con PD (Matriz)

La versión iterativa del problema realiza el mismo trabajo que la recursiva con memoización pero sin realizar la recursividad.

No tiene por qué ser más eficiente.

```

1 int maze_it_matrix(int **maze, int n, int m, int **&dp)
2 {
3     // Comprobamos si el origen o el final son inaccesibles
4     // antes de hacer nada.
5     if (maze[0][0] == 0 || maze[n - 1][m - 1] == 0)
6     {
7         return -1;
8     }
9
10    // Creamos la tabla de programacion dinamica
11    dp = new int *[n];
12    for (int i = 0; i < n; i++)
13    {
14        dp[i] = new int[m];
15        for (int j = 0; j < m; j++)
16        {
17            dp[i][j] = -1; // Inicializamos a -1 (no accesible)
18        }
19
20    // El destino tiene distancia 1 (contamos las celdas)
21    dp[n - 1][m - 1] = 1;
22
23    // Llenamos la tabla desde abajo hacia arriba
24    for (int i = n - 1; i >= 0; i--)
25    {
26        for (int j = m - 1; j >= 0; j--)
27        {
```

```

28         // Si es la celda destino o esta bloqueada, ya esta
29         // procesada
30         if ((i == n - 1 && j == m - 1) || maze[i][j] == 0)
31         {
32             continue;
33         }
34
35         // Posibles movimientos
36         int right = (j + 1 < m) ? dp[i][j + 1] : -1;
37         int down = (i + 1 < n) ? dp[i + 1][j] : -1;
38         int diag = (i + 1 < n && j + 1 < m) ? dp[i + 1][j +
39             1] : -1;
40
41         // Encontramos el minimo valido
42         int min_path = -1;
43         if (right != -1)
44             min_path = right;
45         if (down != -1 && (min_path == -1 || down < min_path
46             ))
47             min_path = down;
48         if (diag != -1 && (min_path == -1 || diag < min_path
49             ))
50             min_path = diag;
51
52         // Si hay al menos un camino valido
53         if (min_path != -1)
54         {
55             dp[i][j] = 1 + min_path;
56         }
57     }
58
59     int result = dp[0][0];
60
61     return result;
62 }
```

8.5.6. Iterativo con PD (Vector)

Esta versión iterativa del problema realiza el mismo trabajo que la iterativa de matriz pero con un vector. Mejora la complejidad espacial de $O(m^2)$ a $O(2m)$, donde $m = \text{columnas}$. Cada vector es $O(m)$ y usamos 2: current_row y next_row.

```

1 int maze_it_vector(int **maze, int n, int m)
2 {
3     // Comprobamos si el origen o el final son inaccesibles
4     // antes de hacer nada.
5     if (maze[0][0] == 0 || maze[n - 1][m - 1] == 0)
```

```
5     return -1;  
6  
7     // Usamos solo dos vectores en lugar de una matriz  
8     vector<int> current_row(m, -1); // Fila actual  
9     vector<int> next_row(m, -1);    // Fila siguiente  
10  
11    // Inicializamos la ultima fila (caso base)  
12    next_row[m - 1] = 1;  
13  
14    // Llenamos la tabla desde abajo hacia arriba  
15    for (int i = n - 1; i >= 0; i--)  
16    {  
17        for (int j = m - 1; j >= 0; j--)  
18        {  
19            // La celda destino ya esta inicializada  
20            if (i == n - 1 && j == m - 1)  
21            {  
22                current_row[j] = 1;  
23                continue;  
24            }  
25  
26            // Si la celda actual esta bloqueada  
27            if (maze[i][j] == 0)  
28            {  
29                current_row[j] = -1;  
30                continue;  
31            }  
32  
33            // Posibles movimientos (derecha usa current_row,  
34            // abajo/diagonal usan next_row)  
35            int right = (j + 1 < m) ? current_row[j + 1] : -1;  
36            int down = (i + 1 < n) ? next_row[j] : -1;  
37            int diag = (i + 1 < n && j + 1 < m) ? next_row[j +  
38            1] : -1;  
39  
39            // Encontramos el minimo valido  
40            int min_path = -1;  
41            if (right != -1)  
42            {  
43                min_path = right;  
44                if (down != -1 && (min_path == -1 || down < min_path  
45                ))  
46                    min_path = down;  
47                if (diag != -1 && (min_path == -1 || diag < min_path  
48                ))  
49                    min_path = diag;  
50  
50            current_row[j] = (min_path != -1) ? 1 + min_path :  
51            -1;  
52        }
```

```
49
50      // Actualizamos next_row para la siguiente iteracion
51      if (i != 0)
52      {
53          next_row = current_row;
54          fill(current_row.begin(), current_row.end(), -1);
55      }
56  }
57
58  return current_row[0];
59 }
```

8.6. Práctica 7 - El problema del laberinto II

Se pide realizar un enfoque *greedy* al problema del laberinto.

No tiene por qué dar una solución. No tiene por qué dar una solución óptima siquiera, con que sea factible⁹ nos vale.

```

1  int maze_greedy(int **maze, int n, int m)
2  {
3      // Si la entrada no es accesible devolvemos 0 directamente
4      if (maze[0][0] == 0)
5          return 0;
6
7      int pasos = 1;      // Contamos con la posicion inicial ya
8          recorrida.
9      int i = 0, j = 0; // Coordenadas para iterar en el laberinto
10
11
12     while (i != n - 1 || j != m - 1)
13     {
14         // Diagonal abajo-derecha
15         if (i + 1 < n && j + 1 < m && maze[i + 1][j + 1] == 1)
16         {
17             i++;
18             j++;
19         }
20         // Abajo
21         else if (i + 1 < n && maze[i + 1][j] == 1)
22             i++;
23         // Derecha
24         else if (j + 1 < m && maze[i][j + 1] == 1)
25             j++;
26         // Si no hay movimientos posibles, no encontramos
27         // solucion
28         else
29             return 0;
30
31         pasos++;
32     }
33
34     return pasos;
35 }
```

⁹Que cumple con las restricciones del problema. En este caso, las restricciones son que el algoritmo siga un camino sin atravesar paredes y que se encuentre dentro de los límites del laberinto.

8.7. Práctica 8 - El problema del laberinto III

Se pide realizar un enfoque de vuelta atrás. Seguimos el esquema visto en las diapositivas para crear el algoritmo.

Vuelta atrás se basa en realizar podas **sin una estrategia de búsqueda**. Se trata de un enfoque que recorre el árbol de soluciones eliminando las que no son factibles ni prometedoras.

Usamos la siguiente estructura para almacenar los nodos:

```

1 struct Node
2 {
3     int x = 0, y = 0;
4     vector<Step> path;
5     int cost = 0;
6 };

```

El algoritmo es el siguiente:

```

1 void maze_bt (vector<vector<int>> maze, int n, int m, const Node
2     &node, Node &currentBest, Stats &stats, bool debug)
3 {
4     stats.nv++;
5
6     if (maze[0][0] == 0)
7         return;
8
9     if (isLeaf(node, n, m))
10    {
11        stats.nhv++;
12        stats.ne++;
13
14        if (isBest(solution(node), solution(currentBest)))
15        {
16            currentBest.cost = solution(node); // Actualizamos
17            el coste
18            currentBest.path = node.path;      // Actualizamos
19            el camino
20        }
21        return;
22    }
23
24    for (Node child : expand(node, n, m))
25    {
26
27        if (isFeasible(child, maze, n, m))
28        {
29            if (isPromising(child, currentBest, n, m))
30            {
31                stats.npv++;
32                maze_bt(maze, n, m, child, currentBest, stats, debug);
33            }
34        }
35    }
36
37    if (currentBest.cost <= 0)
38    {
39        stats.npv++;
40        stats.npr++;
41    }
42
43    if (currentBest.cost > 0)
44    {
45        stats.npv++;
46        stats.npr++;
47    }
48
49    if (currentBest.cost > 0)
50    {
51        stats.npv++;
52        stats.npr++;
53    }
54
55    if (currentBest.cost > 0)
56    {
57        stats.npv++;
58        stats.npr++;
59    }
60
61    if (currentBest.cost > 0)
62    {
63        stats.npv++;
64        stats.npr++;
65    }
66
67    if (currentBest.cost > 0)
68    {
69        stats.npv++;
70        stats.npr++;
71    }
72
73    if (currentBest.cost > 0)
74    {
75        stats.npv++;
76        stats.npr++;
77    }
78
79    if (currentBest.cost > 0)
80    {
81        stats.npv++;
82        stats.npr++;
83    }
84
85    if (currentBest.cost > 0)
86    {
87        stats.npv++;
88        stats.npr++;
89    }
90
91    if (currentBest.cost > 0)
92    {
93        stats.npv++;
94        stats.npr++;
95    }
96
97    if (currentBest.cost > 0)
98    {
99        stats.npv++;
100       stats.npr++;
101    }
102
103    if (currentBest.cost > 0)
104    {
105        stats.npv++;
106        stats.npr++;
107    }
108
109    if (currentBest.cost > 0)
110    {
111        stats.npv++;
112        stats.npr++;
113    }
114
115    if (currentBest.cost > 0)
116    {
117        stats.npv++;
118        stats.npr++;
119    }
120
121    if (currentBest.cost > 0)
122    {
123        stats.npv++;
124        stats.npr++;
125    }
126
127    if (currentBest.cost > 0)
128    {
129        stats.npv++;
130        stats.npr++;
131    }
132
133    if (currentBest.cost > 0)
134    {
135        stats.npv++;
136        stats.npr++;
137    }
138
139    if (currentBest.cost > 0)
140    {
141        stats.npv++;
142        stats.npr++;
143    }
144
145    if (currentBest.cost > 0)
146    {
147        stats.npv++;
148        stats.npr++;
149    }
150
151    if (currentBest.cost > 0)
152    {
153        stats.npv++;
154        stats.npr++;
155    }
156
157    if (currentBest.cost > 0)
158    {
159        stats.npv++;
160        stats.npr++;
161    }
162
163    if (currentBest.cost > 0)
164    {
165        stats.npv++;
166        stats.npr++;
167    }
168
169    if (currentBest.cost > 0)
170    {
171        stats.npv++;
172        stats.npr++;
173    }
174
175    if (currentBest.cost > 0)
176    {
177        stats.npv++;
178        stats.npr++;
179    }
180
181    if (currentBest.cost > 0)
182    {
183        stats.npv++;
184        stats.npr++;
185    }
186
187    if (currentBest.cost > 0)
188    {
189        stats.npv++;
190        stats.npr++;
191    }
192
193    if (currentBest.cost > 0)
194    {
195        stats.npv++;
196        stats.npr++;
197    }
198
199    if (currentBest.cost > 0)
200    {
201        stats.npv++;
202        stats.npr++;
203    }
204
205    if (currentBest.cost > 0)
206    {
207        stats.npv++;
208        stats.npr++;
209    }
210
211    if (currentBest.cost > 0)
212    {
213        stats.npv++;
214        stats.npr++;
215    }
216
217    if (currentBest.cost > 0)
218    {
219        stats.npv++;
220        stats.npr++;
221    }
222
223    if (currentBest.cost > 0)
224    {
225        stats.npv++;
226        stats.npr++;
227    }
228
229    if (currentBest.cost > 0)
230    {
231        stats.npv++;
232        stats.npr++;
233    }
234
235    if (currentBest.cost > 0)
236    {
237        stats.npv++;
238        stats.npr++;
239    }
240
241    if (currentBest.cost > 0)
242    {
243        stats.npv++;
244        stats.npr++;
245    }
246
247    if (currentBest.cost > 0)
248    {
249        stats.npv++;
250        stats.npr++;
251    }
252
253    if (currentBest.cost > 0)
254    {
255        stats.npv++;
256        stats.npr++;
257    }
258
259    if (currentBest.cost > 0)
260    {
261        stats.npv++;
262        stats.npr++;
263    }
264
265    if (currentBest.cost > 0)
266    {
267        stats.npv++;
268        stats.npr++;
269    }
270
271    if (currentBest.cost > 0)
272    {
273        stats.npv++;
274        stats.npr++;
275    }
276
277    if (currentBest.cost > 0)
278    {
279        stats.npv++;
280        stats.npr++;
281    }
282
283    if (currentBest.cost > 0)
284    {
285        stats.npv++;
286        stats.npr++;
287    }
288
289    if (currentBest.cost > 0)
290    {
291        stats.npv++;
292        stats.npr++;
293    }
294
295    if (currentBest.cost > 0)
296    {
297        stats.npv++;
298        stats.npr++;
299    }
299
300    if (currentBest.cost > 0)
301    {
302        stats.npv++;
303        stats.npr++;
304    }
305
306    if (currentBest.cost > 0)
307    {
308        stats.npv++;
309        stats.npr++;
310    }
311
312    if (currentBest.cost > 0)
313    {
314        stats.npv++;
315        stats.npr++;
316    }
317
318    if (currentBest.cost > 0)
319    {
320        stats.npv++;
321        stats.npr++;
322    }
323
324    if (currentBest.cost > 0)
325    {
326        stats.npv++;
327        stats.npr++;
328    }
329
330    if (currentBest.cost > 0)
331    {
332        stats.npv++;
333        stats.npr++;
334    }
335
336    if (currentBest.cost > 0)
337    {
338        stats.npv++;
339        stats.npr++;
340    }
341
342    if (currentBest.cost > 0)
343    {
344        stats.npv++;
345        stats.npr++;
346    }
347
348    if (currentBest.cost > 0)
349    {
350        stats.npv++;
351        stats.npr++;
352    }
353
354    if (currentBest.cost > 0)
355    {
356        stats.npv++;
357        stats.npr++;
358    }
359
360    if (currentBest.cost > 0)
361    {
362        stats.npv++;
363        stats.npr++;
364    }
365
366    if (currentBest.cost > 0)
367    {
368        stats.npv++;
369        stats.npr++;
370    }
371
372    if (currentBest.cost > 0)
373    {
374        stats.npv++;
375        stats.npr++;
376    }
377
378    if (currentBest.cost > 0)
379    {
380        stats.npv++;
381        stats.npr++;
382    }
383
384    if (currentBest.cost > 0)
385    {
386        stats.npv++;
387        stats.npr++;
388    }
389
390    if (currentBest.cost > 0)
391    {
392        stats.npv++;
393        stats.npr++;
394    }
395
396    if (currentBest.cost > 0)
397    {
398        stats.npv++;
399        stats.npr++;
400    }
401
402    if (currentBest.cost > 0)
403    {
404        stats.npv++;
405        stats.npr++;
406    }
407
408    if (currentBest.cost > 0)
409    {
410        stats.npv++;
411        stats.npr++;
412    }
413
414    if (currentBest.cost > 0)
415    {
416        stats.npv++;
417        stats.npr++;
418    }
419
420    if (currentBest.cost > 0)
421    {
422        stats.npv++;
423        stats.npr++;
424    }
425
426    if (currentBest.cost > 0)
427    {
428        stats.npv++;
429        stats.npr++;
430    }
431
432    if (currentBest.cost > 0)
433    {
434        stats.npv++;
435        stats.npr++;
436    }
437
438    if (currentBest.cost > 0)
439    {
440        stats.npv++;
441        stats.npr++;
442    }
443
444    if (currentBest.cost > 0)
445    {
446        stats.npv++;
447        stats.npr++;
448    }
449
450    if (currentBest.cost > 0)
451    {
452        stats.npv++;
453        stats.npr++;
454    }
455
456    if (currentBest.cost > 0)
457    {
458        stats.npv++;
459        stats.npr++;
460    }
461
462    if (currentBest.cost > 0)
463    {
464        stats.npv++;
465        stats.npr++;
466    }
467
468    if (currentBest.cost > 0)
469    {
470        stats.npv++;
471        stats.npr++;
472    }
473
474    if (currentBest.cost > 0)
475    {
476        stats.npv++;
477        stats.npr++;
478    }
479
480    if (currentBest.cost > 0)
481    {
482        stats.npv++;
483        stats.npr++;
484    }
485
486    if (currentBest.cost > 0)
487    {
488        stats.npv++;
489        stats.npr++;
490    }
491
492    if (currentBest.cost > 0)
493    {
494        stats.npv++;
495        stats.npr++;
496    }
497
498    if (currentBest.cost > 0)
499    {
500        stats.npv++;
501        stats.npr++;
502    }
503
504    if (currentBest.cost > 0)
505    {
506        stats.npv++;
507        stats.npr++;
508    }
509
510    if (currentBest.cost > 0)
511    {
512        stats.npv++;
513        stats.npr++;
514    }
515
516    if (currentBest.cost > 0)
517    {
518        stats.npv++;
519        stats.npr++;
520    }
521
522    if (currentBest.cost > 0)
523    {
524        stats.npv++;
525        stats.npr++;
526    }
527
528    if (currentBest.cost > 0)
529    {
530        stats.npv++;
531        stats.npr++;
532    }
533
534    if (currentBest.cost > 0)
535    {
536        stats.npv++;
537        stats.npr++;
538    }
539
540    if (currentBest.cost > 0)
541    {
542        stats.npv++;
543        stats.npr++;
544    }
545
546    if (currentBest.cost > 0)
547    {
548        stats.npv++;
549        stats.npr++;
550    }
551
552    if (currentBest.cost > 0)
553    {
554        stats.npv++;
555        stats.npr++;
556    }
557
558    if (currentBest.cost > 0)
559    {
560        stats.npv++;
561        stats.npr++;
562    }
563
564    if (currentBest.cost > 0)
565    {
566        stats.npv++;
567        stats.npr++;
568    }
569
570    if (currentBest.cost > 0)
571    {
572        stats.npv++;
573        stats.npr++;
574    }
575
576    if (currentBest.cost > 0)
577    {
578        stats.npv++;
579        stats.npr++;
580    }
581
582    if (currentBest.cost > 0)
583    {
584        stats.npv++;
585        stats.npr++;
586    }
587
588    if (currentBest.cost > 0)
589    {
590        stats.npv++;
591        stats.npr++;
592    }
593
594    if (currentBest.cost > 0)
595    {
596        stats.npv++;
597        stats.npr++;
598    }
599
600    if (currentBest.cost > 0)
601    {
602        stats.npv++;
603        stats.npr++;
604    }
605
606    if (currentBest.cost > 0)
607    {
608        stats.npv++;
609        stats.npr++;
610    }
611
612    if (currentBest.cost > 0)
613    {
614        stats.npv++;
615        stats.npr++;
616    }
617
618    if (currentBest.cost > 0)
619    {
620        stats.npv++;
621        stats.npr++;
622    }
623
624    if (currentBest.cost > 0)
625    {
626        stats.npv++;
627        stats.npr++;
628    }
629
630    if (currentBest.cost > 0)
631    {
632        stats.npv++;
633        stats.npr++;
634    }
635
636    if (currentBest.cost > 0)
637    {
638        stats.npv++;
639        stats.npr++;
640    }
641
642    if (currentBest.cost > 0)
643    {
644        stats.npv++;
645        stats.npr++;
646    }
647
648    if (currentBest.cost > 0)
649    {
650        stats.npv++;
651        stats.npr++;
652    }
653
654    if (currentBest.cost > 0)
655    {
656        stats.npv++;
657        stats.npr++;
658    }
659
660    if (currentBest.cost > 0)
661    {
662        stats.npv++;
663        stats.npr++;
664    }
665
666    if (currentBest.cost > 0)
667    {
668        stats.npv++;
669        stats.npr++;
670    }
671
672    if (currentBest.cost > 0)
673    {
674        stats.npv++;
675        stats.npr++;
676    }
677
678    if (currentBest.cost > 0)
679    {
680        stats.npv++;
681        stats.npr++;
682    }
683
684    if (currentBest.cost > 0)
685    {
686        stats.npv++;
687        stats.npr++;
688    }
689
690    if (currentBest.cost > 0)
691    {
692        stats.npv++;
693        stats.npr++;
694    }
695
696    if (currentBest.cost > 0)
697    {
698        stats.npv++;
699        stats.npr++;
700    }
701
702    if (currentBest.cost > 0)
703    {
704        stats.npv++;
705        stats.npr++;
706    }
707
708    if (currentBest.cost > 0)
709    {
710        stats.npv++;
711        stats.npr++;
712    }
713
714    if (currentBest.cost > 0)
715    {
716        stats.npv++;
717        stats.npr++;
718    }
719
720    if (currentBest.cost > 0)
721    {
722        stats.npv++;
723        stats.npr++;
724    }
725
726    if (currentBest.cost > 0)
727    {
728        stats.npv++;
729        stats.npr++;
730    }
731
732    if (currentBest.cost > 0)
733    {
734        stats.npv++;
735        stats.npr++;
736    }
737
738    if (currentBest.cost > 0)
739    {
740        stats.npv++;
741        stats.npr++;
742    }
743
744    if (currentBest.cost > 0)
745    {
746        stats.npv++;
747        stats.npr++;
748    }
749
750    if (currentBest.cost > 0)
751    {
752        stats.npv++;
753        stats.npr++;
754    }
755
756    if (currentBest.cost > 0)
757    {
758        stats.npv++;
759        stats.npr++;
760    }
761
762    if (currentBest.cost > 0)
763    {
764        stats.npv++;
765        stats.npr++;
766    }
767
768    if (currentBest.cost > 0)
769    {
770        stats.npv++;
771        stats.npr++;
772    }
773
774    if (currentBest.cost > 0)
775    {
776        stats.npv++;
777        stats.npr++;
778    }
779
780    if (currentBest.cost > 0)
781    {
782        stats.npv++;
783        stats.npr++;
784    }
785
786    if (currentBest.cost > 0)
787    {
788        stats.npv++;
789        stats.npr++;
790    }
791
792    if (currentBest.cost > 0)
793    {
794        stats.npv++;
795        stats.npr++;
796    }
797
798    if (currentBest.cost > 0)
799    {
800        stats.npv++;
801        stats.npr++;
802    }
803
804    if (currentBest.cost > 0)
805    {
806        stats.npv++;
807        stats.npr++;
808    }
809
810    if (currentBest.cost > 0)
811    {
812        stats.npv++;
813        stats.npr++;
814    }
815
816    if (currentBest.cost > 0)
817    {
818        stats.npv++;
819        stats.npr++;
820    }
821
822    if (currentBest.cost > 0)
823    {
824        stats.npv++;
825        stats.npr++;
826    }
827
828    if (currentBest.cost > 0)
829    {
830        stats.npv++;
831        stats.npr++;
832    }
833
834    if (currentBest.cost > 0)
835    {
836        stats.npv++;
837        stats.npr++;
838    }
839
840    if (currentBest.cost > 0)
841    {
842        stats.npv++;
843        stats.npr++;
844    }
845
846    if (currentBest.cost > 0)
847    {
848        stats.npv++;
849        stats.npr++;
850    }
851
852    if (currentBest.cost > 0)
853    {
854        stats.npv++;
855        stats.npr++;
856    }
857
858    if (currentBest.cost > 0)
859    {
860        stats.npv++;
861        stats.npr++;
862    }
863
864    if (currentBest.cost > 0)
865    {
866        stats.npv++;
867        stats.npr++;
868    }
869
870    if (currentBest.cost > 0)
871    {
872        stats.npv++;
873        stats.npr++;
874    }
875
876    if (currentBest.cost > 0)
877    {
878        stats.npv++;
879        stats.npr++;
880    }
881
882    if (currentBest.cost > 0)
883    {
884        stats.npv++;
885        stats.npr++;
886    }
887
888    if (currentBest.cost > 0)
889    {
890        stats.npv++;
891        stats.npr++;
892    }
893
894    if (currentBest.cost > 0)
895    {
896        stats.npv++;
897        stats.npr++;
898    }
899
900    if (currentBest.cost > 0)
901    {
902        stats.npv++;
903        stats.npr++;
904    }
905
906    if (currentBest.cost > 0)
907    {
908        stats.npv++;
909        stats.npr++;
910    }
911
912    if (currentBest.cost > 0)
913    {
914        stats.npv++;
915        stats.npr++;
916    }
917
918    if (currentBest.cost > 0)
919    {
920        stats.npv++;
921        stats.npr++;
922    }
923
924    if (currentBest.cost > 0)
925    {
926        stats.npv++;
927        stats.npr++;
928    }
929
930    if (currentBest.cost > 0)
931    {
932        stats.npv++;
933        stats.npr++;
934    }
935
936    if (currentBest.cost > 0)
937    {
938        stats.npv++;
939        stats.npr++;
940    }
941
942    if (currentBest.cost > 0)
943    {
944        stats.npv++;
945        stats.npr++;
946    }
947
948    if (currentBest.cost > 0)
949    {
950        stats.npv++;
951        stats.npr++;
952    }
953
954    if (currentBest.cost > 0)
955    {
956        stats.npv++;
957        stats.npr++;
958    }
959
960    if (currentBest.cost > 0)
961    {
962        stats.npv++;
963        stats.npr++;
964    }
965
966    if (currentBest.cost > 0)
967    {
968        stats.npv++;
969        stats.npr++;
970    }
971
972    if (currentBest.cost > 0)
973    {
974        stats.npv++;
975        stats.npr++;
976    }
977
978    if (currentBest.cost > 0)
979    {
980        stats.npv++;
981        stats.npr++;
982    }
983
984    if (currentBest.cost > 0)
985    {
986        stats.npv++;
987        stats.npr++;
988    }
989
990    if (currentBest.cost > 0)
991    {
992        stats.npv++;
993        stats.npr++;
994    }
995
996    if (currentBest.cost > 0)
997    {
998        stats.npv++;
999        stats.npr++;
1000    }
1001
1002    if (currentBest.cost > 0)
1003    {
1004        stats.npv++;
1005        stats.npr++;
1006    }
1007
1008    if (currentBest.cost > 0)
1009    {
1010        stats.npv++;
1011        stats.npr++;
1012    }
1013
1014    if (currentBest.cost > 0)
1015    {
1016        stats.npv++;
1017        stats.npr++;
1018    }
1019
1020    if (currentBest.cost > 0)
1021    {
1022        stats.npv++;
1023        stats.npr++;
1024    }
1025
1026    if (currentBest.cost > 0)
1027    {
1028        stats.npv++;
1029        stats.npr++;
1030    }
1031
1032    if (currentBest.cost > 0)
1033    {
1034        stats.npv++;
1035        stats.npr++;
1036    }
1037
1038    if (currentBest.cost > 0)
1039    {
1040        stats.npv++;
1041        stats.npr++;
1042    }
1043
1044    if (currentBest.cost > 0)
1045    {
1046        stats.npv++;
1047        stats.npr++;
1048    }
1049
1050    if (currentBest.cost > 0)
1051    {
1052        stats.npv++;
1053        stats.npr++;
1054    }
1055
1056    if (currentBest.cost > 0)
1057    {
1058        stats.npv++;
1059        stats.npr++;
1060    }
1061
1062    if (currentBest.cost > 0)
1063    {
1064        stats.npv++;
1065        stats.npr++;
1066    }
1067
1068    if (currentBest.cost > 0)
1069    {
1070        stats.npv++;
1071        stats.npr++;
1072    }
1073
1074    if (currentBest.cost > 0)
1075    {
1076        stats.npv++;
1077        stats.npr++;
1078    }
1079
1080    if (currentBest.cost > 0)
1081    {
1082        stats.npv++;
1083        stats.npr++;
1084    }
1085
1086    if (currentBest.cost > 0)
1087    {
1088        stats.npv++;
1089        stats.npr++;
1090    }
1091
1092    if (currentBest.cost > 0)
1093    {
1094        stats.npv++;
1095        stats.npr++;
1096    }
1097
1098    if (currentBest.cost > 0)
1099    {
1100        stats.npv++;
1101        stats.npr++;
1102    }
1103
1104    if (currentBest.cost > 0)
1105    {
1106        stats.npv++;
1107        stats.npr++;
1108    }
1109
1110    if (currentBest.cost > 0)
1111    {
1112        stats.npv++;
1113        stats.npr++;
1114    }
1115
1116    if (currentBest.cost > 0)
1117    {
1118        stats.npv++;
1119        stats.npr++;
1120    }
1121
1122    if (currentBest.cost > 0)
1123    {
1124        stats.npv++;
1125        stats.npr++;
1126    }
1127
1128    if (currentBest.cost > 0)
1129    {
1130        stats.npv++;
1131        stats.npr++;
1132    }
1133
1134    if (currentBest.cost > 0)
1135    {
1136        stats.npv++;
1137        stats.npr++;
1138    }
1139
1140    if (currentBest.cost > 0)
1141    {
1142        stats.npv++;
1143        stats.npr++;
1144    }
1145
1146    if (currentBest.cost > 0)
1147    {
1148        stats.npv++;
1149        stats.npr++;
1150    }
1151
1152    if (currentBest.cost > 0)
1153    {
1154        stats.npv++;
1155        stats.npr++;
1156    }
1157
1158    if (currentBest.cost > 0)
1159    {
1160        stats.npv++;
1161        stats.npr++;
1162    }
1163
1164    if (currentBest.cost > 0)
1165    {
1166        stats.npv++;
1167        stats.npr++;
1168    }
1169
1170    if (currentBest.cost > 0)
1171    {
1172        stats.npv++;
1173        stats.npr++;
1174    }
1175
1176    if (currentBest.cost > 0)
1177    {
1178        stats.npv++;
1179        stats.npr++;
1180    }
1181
1182    if (currentBest.cost > 0)
1183    {
1184        stats.npv++;
1185        stats.npr++;
1186    }
1187
1188    if (currentBest.cost > 0)
1189    {
1190        stats.npv++;
1191        stats.npr++;
1192    }
1193
1194    if (currentBest.cost > 0)
1195    {
1196        stats.npv++;
1197        stats.npr++;
1198    }
1199
1200    if (currentBest.cost > 0)
1201    {
1202        stats.npv++;
1203        stats.npr++;
1204    }
1205
1206    if (currentBest.cost > 0)
1207    {
1208        stats.npv++;
1209        stats.npr++;
1210    }
1211
1212    if (currentBest.cost > 0)
1213    {
1214        stats.npv++;
1215        stats.npr++;
1216    }
1217
1218    if (currentBest.cost > 0)
1219    {
1220        stats.npv++;
1221        stats.npr++;
1222    }
1223
1224    if (currentBest.cost > 0)
1225    {
1226        stats.npv++;
1227        stats.npr++;
1228    }
1
```

```

27         {
28             stats.ne++;
29
30             maze[child.x][child.y] = -1; // Marcamos como
31             visitado
32
33             maze_bt(maze, n, m, child, currentBest, stats,
34             debug);
35
36         }
37     }
38     else
39         stats.nnf++;
40     }
41 }
```

Las funciones personalizadas de dentro del código contienen la siguiente implementación:

```

1 int chebyshev(const Node &node, int n, int m)
2 {
3     return max(abs(node.x - (n - 1)), abs(node.y - (m - 1)));
4 }
5
6 bool isLeaf(const Node &node, int n, int m)
7 {
8     return node.x == n - 1 && node.y == m - 1;
9 }
10
11 int solution(const Node &node)
12 {
13     return node.cost; // Retorna el coste del nodo
14 }
15
16 bool isBest(int sol, int currentBest)
17 {
18     return sol < currentBest;
19 }
20
21 vector<Node> expand(const Node &node, int n, int m)
22 {
23     vector<Node> children; // Vector para almacenar los hijos
24     // del nodo actual
25
26     // Direcciones posibles
27     map<Step, tuple<int, int>> steps_inc = {
```

```

27     {N, make_tuple(-1, 0)},
28     {NE, make_tuple(-1, 1)},
29     {E, make_tuple(0, 1)},
30     {SE, make_tuple(1, 1)},
31     {S, make_tuple(1, 0)},
32     {SW, make_tuple(1, -1)},
33     {W, make_tuple(0, -1)},
34     {NW, make_tuple(-1, -1)}};

35
36 for (auto it = steps_inc.begin(); it != steps_inc.end(); ++it) // Para cada direccion...
37 {
38     // Obtenemos la nueva posicion
39     Step move = it->first;
40
41     // Obtenemos el incremento de fila y columna
42     int incx, incy;
43     tie(incx, incy) = it->second;
44
45     // Calculamos la nueva posicion
46     int newX = node.x + incx;
47     int newY = node.y + incy;
48
49     // Creamos al nuevo hijo
50     Node child;
51     child.x = newX;
52     child.y = newY;
53     child.path = node.path;
54     child.path.push_back(move); // Anadimos el movimiento al
                                   camino
55     child.cost = node.cost + 1;
56
57     // Lo anadimos al vector de hijos
58     children.push_back(child);
59 }
60
61 // Ordenamos los hijos por heuristica f(n) = cost + h
62 std::sort(children.begin(), children.end(), [n, m](const
63     Node &a, const Node &b)
64     {
65         int fA = a.cost + chebyshev(a, n, m);
66         int fB = b.cost + chebyshev(b, n, m);
67         return fA < fB; });
68
69     return children;
70 }
71 bool isFeasible(const Node &node, const vector<vector<int>> &
72     maze, int n, int m)

```

```
72  {
73      return node.x >= 0 && node.x < n &&
74          node.y >= 0 && node.y < m &&
75          maze[node.x][node.y] == 1;
76  }
77
78 bool isPromising(const Node &node, const Node &currentBest, int
79     n, int m)
80 {
81     // return node.cost < currentBest.cost; <-- Con esta
82     // heuristica obtenemos casi 150k nodos visitados en el
83     // laberinto 09.
84     return node.cost + chebyshev(node, n, m) < currentBest.cost;
85     // <-- Con esta heuristica (de Chebyshev) obtenemos
86     // apenas 220 nodos visitados en el laberinto 09.
87 }
```

8.8. Práctica Final - Camino de coste mínimo y IV

Memoria de la práctica final. Paso de volver a explicarla.

En esta memoria se pretende documentar el desarrollo de la práctica final de Análisis y Diseño de Algoritmos¹. El problema a resolver será una batería de laberintos diferentes y la metodología a aplicar será ramificación y poda.

Índice

| | |
|---|----------|
| 1. Estructuras de datos. | 2 |
| 1.1. Nodo. | 2 |
| 1.2. Lista de nodos vivos. | 2 |
| 1.3. Laberinto. | 3 |
| 1.4. Estadísticas | 3 |
| 2. Mecanismos de poda. | 3 |
| 2.1. Poda de nodos no factibles. | 3 |
| 2.2. Poda de nodos no prometedores. | 3 |
| 2.3. Esquema de implementación | 4 |
| 3. Cotas pesimistas y optimistas. | 5 |
| 3.1. Cota pesimista inicial (inicialización). | 5 |
| 3.2. Cota pesimista del resto de nodos. | 5 |
| 3.3. Cota optimista. | 6 |
| 4. Otros medios empleados para acelerar la búsqueda. | 7 |
| 4.1. Expansión de hijos | 7 |
| 5. Estudio comparativo de distintas estrategias de búsqueda. | 7 |
| 6. Tiempos de ejecución. | 8 |
| 6.1. Tiempos de BFS | 8 |

1. Estructuras de datos.

Deberemos usar una estructura organizada para poder comparar las diferentes características de los nodos y poder ordenarlos de mejor a peor en la cola de prioridad.

1.1. Nodo.

Para representar un nodo en nuestro problema crearemos la siguiente estructura:

```
1 struct Node
2 {
3     int x = 0, y = 0;
4     vector<Step> path;
5     int cost = 0;
6 };
```

El nodo consta de dos variables enteras que representan la posición del mismo, el coste del propio nodo y el camino codificado que se ha seguido para llegar hasta él. Este camino es un vector de otra struct personalizada llamada **Step**. Esta estructura representa todas las direcciones posibles y está declarada en el código como:

```
1 enum Step
2 {
3     N, NE, E, SE, S, SW, W, NW
4 };
```

1.2. Lista de nodos vivos.

La lista de nodos vivos será una *priority queue* con la siguiente estructura:

```
1 priority_queue<Node, vector<Node>, compareNodes> q;
```

Esta cola de prioridad se instancia dentro del algoritmo principal. El método **maze.bb** contiene el algoritmo.

La estructura **compareNodes** comparará dos nodos dados para poder elegir el de menor coste. Contiene la siguiente lógica:

```
1 struct compareNodes
2 {
3     bool operator()(const Node &a, const Node &b)
4     {
5         return a.cost > b.cost;
6     }
7 };
```

La función usa el operador **>** en estructuras personalizadas. Se ha sobrecargado este operador para que pueda comparar dos nodos dados. Contiene la siguiente lógica:

```
1 bool operator<(const Node &a, const Node &b)
2 {
3     return (a.cost + optimist(a)) > (b.cost + optimist(b));
4 }
```

La evaluación se realiza teniendo en cuenta el coste que tenga cada nodo añadiéndole a este la cota optimista asociada a ese mismo nodo. A mayor valor de la suma, mayor posición en la lista de prioridad tendrá el nodo y viceversa.

Las implementación de las cotas optimistas se pueden ver en la sección [3.3](#).

¹El código plasmado en la memoria no tiene por qué ser igual al que haya sido entregado. En la memoria se pretende documentar el desarrollo del código y sus respectivas mejoras a lo largo de la realización de la práctica.

1.3. Laberinto

Se ha creado una estructura personalizada para poder guardar el laberinto y que sea accesible desde cualquier contexto:

```
1 struct Maze
2 {
3     int n = 0;
4     int m = 0;
5     vector<vector<int>> maze = {};
6 }
```

Con esta estructura en el código, se ha creado una variable global que almacena el laberinto llamada `mazeStruct`. Esta variable será la que uses todas las funciones que necesiten acceder al laberinto. Se ha hecho de esta manera para evitar el paso del laberinto como de parámetro a todas las funciones y por claridad en el código.

1.4. Estadísticas

Para poder llevar un recuento de las estadísticas de manera organizada se ha creado otra `struct` personalizada llamada `Stats` que cuenta con los siguientes campos:

```
1 struct Stats
2 {
3     int nv = 0; // Número de nodos visitados
4     int ne = 0; // Número de nodos explorados
5     int nhv = 0; // Número de nodos hoja visitados
6     int nnf = 0; // Número de nodos no factibles
7     int npn = 0; // Número de nodos no prometedores
8     int ppd = 0; // Número de nodos prometedores pero
9     descartados
10    int msah = 0; // Número de veces que la mejor
11        solución se ha actualizado desde un nodo hoja
12    int msap = 0; // Número de veces que la mejor
13        solución se ha actualizado a partir de la cota pesimista de un nodo
14        sin completar
15    std::chrono::duration<double> t; // Tiempo de ejecución
16 }
```

Cuenta con las estadísticas sobre los nodos necesarias y el tiempo de ejecución.

2. Mecanismos de poda.

Es importante instanciar de manera correcta los mecanismos de poda que seguiremos para la resolución de este problema.

2.1. Poda de nodos no factibles.

Los nodos que no sean factibles serán evaluados con el siguiente método:

```
1 bool isFeasible(const Node &node)
2 {
3     return node.x >= 0 && node.x < mazeStruct.n &&
4         node.y >= 0 && node.y < mazeStruct.m &&
5         mazeStruct.maze[node.x][node.y] == 1;
6 }
```

A este método se le llama dentro del bucle de expansión del nodo actual. Se puede ver el esquema de la implementación seguida en la sección 2.3.

2.2. Poda de nodos no prometedores.

Los nodos no prometedores se evaluarán mediante la siguiente función:

3

4

3. Cotas pesimistas y optimistas.

En esta sección de la memoria veremos las tres cotas que usaremos durante el algoritmo.

3.1. Cota pesimista inicial (inicialización).

Para inicializar los nodos se usa la siguiente función:

```
1 Node initNode()
2 {
3     Node initial = Node();
4     initial.x = 0;
5     initial.y = 0;
6     initial.cost = 0;
7     initial.path = vector<Step>();
8
9     return initial;
10 }
```

Dentro de `maze_bb()` llamamos a esta función `initNode()`. Justo después de ella, llamamos a otra función llamada `pessimist()`. Este último método se encargará de establecer la cota pesimista para un nodo dado siendo, en este caso, el nodo inicial.

La función `pessimist()` es la siguiente:

```
1 Node pessimist(const Node &node, bool debug)
2 {
3     Node pesimista = node;
4     pesimista.cost = INT_MAX;
5
6     Node bfsNode = bfs();
7
8     if (bfsNode.cost >= 0)
9         pesimista = bfsNode;
10
11    return pesimista;
12 }
```

Se establece un valor `INT_MAX` por si el algoritmo pesimista no encuentra solución.

El algoritmo escogido para la poda pesimista es una búsqueda en anchura. Antes de escoger esta opción, se probó con el algoritmo `Greedy` de la práctica 7, pero obtenemos mejores resultados con la búsqueda en anchura (ver sección 6).

3.2. Cota pesimista del resto de nodos.

La cota pesimista es la que usaremos para comprobar si un nodo puede ser podado, es decir, si podemos descartar un estado de la resolución del laberinto sin explorarlo más a fondo porque se sabe que no conducirá a una solución óptima.

Para todos los nodos usamos el resultado que ofrece la búsqueda en anchura. El esquema de este algoritmo es el siguiente:

```
1 Node bfs()
2 {
3     // ...
4     vector<vector<bool>> seen(N, vector<bool>(M, false));
5     vector<vector<Step>> parentStep(N, vector<Step>(M));
6     vector<vector<pair<int, int>>> parentPos(N, vector<pair<int, int>>(M));
7     queue<tuple<int, int, int>> q;
8
9     if (mazeStruct.maze[0][0] == 0)
10        // Devolvemos INT_MAX
```

```
1     bool isPromising(const Node &node, const Node &currentBest)
2     {
3         return (node.cost + optimist(node)) < currentBest.cost;
4     }
```

La función `optimist` se verá en la sección 3.3. Esta función que comprueba si un nodo es prometedor es llamada dos veces:

- Cuando se elige el primer nodo de la cola de prioridad para comprobar si puede ser descartado al ser no prometedor (línea 14 del esquema de la sección 2.3).
- Después de comprobar que un nodo expandido es factible para comprobar si es prometedor (línea 34 del esquema de la sección 2.3).

2.3. Esquema de implementación

El esquema que se ha seguido para implementar el código de esta práctica ha sido el siguiente:

```
1 void maze_bb(Node &currentBest, Stats &stats, bool debug)
2 {
3     Node initial = initNode();
4
5     // ...
6
7     if (mazeStruct.maze[0][0] == 0)
8         // ...
9
10    while (!q.empty()) {
11        Node n = q.top();
12        q.pop();
13
14        if (!isPromising(n, currentBest))
15            // ...
16
17        if (isLeaf(n)) {
18            // ...
19
20            if (isBetter(solution(n), solution(currentBest)))
21                // ...
22            continue;
23
24        for (Node a : expand(n)) {
25            if (isFeasible(a)) {
26                // ...
27
28                if (isBetter(solution(pes(a, debug)), solution(currentBest)))
29                    // ...
30
31                if (isPromising(a, currentBest)) {
32                    q.push(a);
33                    // ...
34                }
35                else
36                    // ...
37            }
38        }
39    }
40
41 }
42 }
```

```
11     q.push({0, 0, 1});
12     seen[0][0] = true;
13
14     // ...
15
16     while (!q.empty()) {
17         auto [i, j, d] = q.front();
18         q.pop();
19
20         if (i == N - 1 && j == M - 1) { // Si final del laberinto...
21             // ...
22
23             while (!(ci == N && cj == M))
24                 // Reconstruimos el camino
25
26                 // Creamos el nodo resultado y lo devolvemos
27
28                 return result;
29
30
31             for (int k = 0; k < 8; ++k) { // Expandimos el nodo
32                 // ...
33
34                 if /* Es factible y no visitado antes */ {
35                     // Pusheamos la nueva posición
36                 }
37             }
38
39
40             // Devolvemos INT_MAX
41     }
```

Con este algoritmo obtenemos una solución rápida y factible para realizar una poda.

Es una buena cota pesimista porque su coste nunca será más de $O(n * m)$ y siempre nos garantiza una solución de manera eficiente.

3.3. Cota optimista.

La cota optimista es la mejor solución que se le puede dar al problema del laberinto.

La estrategia que se ha elegido para esta práctica se puede ver reflejada en el código en la función llamada `optimist`:

```
1 int optimist(const Node &node)
2 {
3     // return chebyshev(node);
4     // return euclidean(node);
5     return dist[node.x][node.y];
6 }
```

Como se puede ver en los comentarios de dentro de la función, se ha probado con varias metodologías:

- **Distancia de Chebyshev:** Distancia desde la celda actual al final del laberinto usando la heurística $\max(\text{abs}(\text{node.x} - (\text{mazeStruct.n} - 1)), \text{abs}(\text{node.y} - (\text{mazeStruct.m} - 1)))$.
- **Distancia de Manhattan o Euclídea:** Distancia desde la celda actual al final del laberinto usando la heurística $\sqrt{(\text{node.x} - (\text{mazeStruct.n} - 1))^2 + (\text{node.y} - (\text{mazeStruct.m} - 1))^2}$.
- **Distancia al final desde cada celda:** Distancia desde cada celda al final del laberinto. Se usa una función llamada `precomputeDistances` que guarda la distancia mínima desde cada

celda camino al final del laberinto. Estas distancias se guardan en un vector de int doble llamado `dist`.

4. Otros medios empleados para acelerar la búsqueda.

Los métodos usados para mejorar el rendimiento del algoritmo se han comentado a lo largo de la memoria hasta ahora.

Cabe destacar que el algoritmo de búsqueda en anchura obtiene el resultado y el camino correcto en tiempos muy bajos para todos los laberintos que tenemos disponibles en la batería de ficheros de pruebas (ver sección 6). Esto se debe a que la complejidad en el peor caso de esta heurística se acota por $O(n * m)$ siendo n y m el tamaño del laberinto. Usando ramificación y poda, el orden es exponencial acorde con el número de nodos explorados aunque se realicen podas en la mayoría de ellos.

4.1. Expansión de hijos

Para acelerar la búsqueda, se ordenan los hijos `node` cuando se expande el nodo padre para que el primero en ser explorado sea el que menos `node.cost + chebyshev(node)` tenga. En el código, este aspecto se ve en la función `expand`, concretamente al final y está programado de la siguiente manera:

```

1 std::sort(children.begin(), children.end(), [n, m](const Node &a, const
2   Node &b) {
3     int fA = a.cost + chebyshev(a);
4     int fB = b.cost + chebyshev(b);
5     return fA < fB; });
6
7 return children;

```

5. Estudio comparativo de distintas estrategias de búsqueda.

En esta sección de la memoria veremos los tiempos que se obtienen para los laberintos de esta práctica usando *backtracking*. No podemos usar las anteriores estrategias a *backtracking* porque sólo contaban con tres tipos de movimientos diferentes (abajo, derecha y abajo-derecha) y la mayoría de los laberintos de esta práctica no tendrían solución usando esos algoritmos.

| Nº laberinto | Backtracking (s) | Ramificación y poda (s) |
|--------------|------------------|-------------------------|
| 0 | 0.000 | 0.000 |
| 1 | 0.000 | 0.000 |
| 2 | 0.000 | 0.000 |
| 3 | 0.000 | 0.000 |
| 4 | 0.000 | 0.000 |
| 5 | 0.000 | 0.000 |
| 6 | 0.000 | 0.000 |
| 7 | 0.008 | 0.000 |
| 8 | 0.000 | 0.002 |
| 9 | 0.000 | 0.000 |
| 10 | ? | 0.028 |
| 11 | 0.002 | 0.006 |
| 12 | 0.002 | 0.000 |
| 13 | 0.668 | ? |
| ... | ? | ? |

Se puede observar que los dos algoritmos desarrollados están a la par en cuanto a tiempo, aunque ninguno sabe resolver a partir del laberinto 13.

6. Tiempos de ejecución.

En esta sección se recogen los tiempos para todos los laberintos que han sido resueltos por ramificación y poda.

| Fichero de test | Tiempo (ms) |
|-----------------|-------------|
| 00-bb.maze | 0.000 |
| 01-bb.maze | 0.011 |
| 02-bb.maze | 0.098 |
| 03-bb.maze | 0.020 |
| 04-bb.maze | 0.027 |
| 05-bb.maze | 0.139 |
| 06-bb.maze | 0.012 |
| 07-bb.maze | 0.219 |
| 08-bb.maze | 1.866 |
| 09-bb.maze | 0.491 |
| 10-bb.maze | 28.480 |
| 11-bb.maze | 7.351 |
| 12-bb.maze | 0.368 |

6.1. Tiempos de BFS

Sección extra para indicar los tiempos que obtenemos mediante BFS (*Breadth-First Search*) o búsqueda en anchura. Para esta sección se ha extraído el método `Node bfs()` en un fichero aparte, se ha compilado y se ha probado de manera independiente. Sólo recogeremos algunos de los laberintos más grandes de la fase de pruebas.

| Fichero de test | Tiempo (ms) |
|-----------------|-------------|
| k01-bb.maze | 56.687 |
| k02-bb.maze | 76.130 |
| k03-bb.maze | 519.598 |
| k05-bb.maze | 1537.237 |
| k10-bb.maze | 3078.001 |

9. Repaso

Algunas posibles preguntas de examen.

9.1. Primer parcial

1. **¿Qué tienen en común MergeSort y QuickSort?**: Que ambos usan la estrategia Divide y Vencerás. MergeSort usa espacio adicional y QuickSort no. MergeSort se ejecuta en tiempo $O(n)$ y QuickSort $\Omega(n \log n)$ y $O(n^2)$.
2. **Complejidad de juntar una lista ordenada n_o con una desordenada n_d :** $O(n_d n_o)$.
3. **Dada la siguiente relación de recurrencia:** $T(n) =$

$$\begin{cases} 1 & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

- Si $g(n) = O(n^2)$ NO se trata de inserción binaria. La inserción binaria tiene $O(n^2)$, pero NO parte los subproblemas en $\frac{n}{2}$.
 - Si $g(n) = O(n)$ se trata de QuickSelect.
 - Si $g(n) = O(1)$ se trata de búsqueda binaria.
4. En el mejor de los casos de un algoritmo recursivo, la complejidad temporal **NO**:
 - Coincide con el valor del caso base de la ecuación de recurrencia que expresa la complejidad del propio algoritmo.
 - Es la complejidad temporal de las instancias que están en el caso base.
 5. Estas dos siguientes implementaciones de la función **pow** tienen el mismo coste temporal $O(n)$.

```

1  unsigned long pot2_1(unsigned n) {
2      if (n==0)
3          return 1;
4      if (n%2==0)
5          return pot2_1(n/2) * pot2_1(n/2);
6      else
7          return 2 * pot2_1(n/2) * pot2_1(n/2);
8  }
9
10 unsigned long pot2_2(unsigned n) {
11     if (n==0)
12         return 1;
13     return 2 * pot2_2(n-1);
14 }
```

6. La complejidad temporal en el mejor de los casos NO es el tiempo que tarda el algoritmo en resolver la talla más pequeña que se le puede presentar. Es una función de la talla o tamaño del problema que tiene que estar definida para todos los valores de esta.
7. En cuanto a la complejidad espacial de los algoritmos Quicksort, Heapsort y Mergesort , sin considerar el espacio que ocupan las pilas de recursión, Mergesort tiene un coste lineal con el tamaño del vector. Quicksort y Heapsort tienen un coste constante.
8. La complejidad espacial del algoritmo de Mergesort viene dada por la siguiente relación de recurrencia: $T(n) =$

$$\begin{cases} T(n) = 1 & \text{para } n \leq 1 \\ T(n) = n + 2T\left(\frac{n}{2}\right) & \text{para } n > 1 \end{cases}$$

9. La complejidad espacial del algoritmo Quicksort, sin considerar el espacio de la pila de recursión, es $O(1)$.
10. Mergesort siempre va a ser más rápido o igual (salvo una constante) que Quicksort.

9.2. Segundo parcial

1. Si se quiere aplicar memoización para una función dada del siguiente estilo:

```

1 int f (int x, int y) {
2     if (x > y)
3         return 1;
4     return x*(y-1) + f(x, y-2);
5 }
```

Las complejidades temporales y espaciales son $O(y - x)$, ambas iguales.

2. Para una función de este otro estilo:

```

1 unsigned f( unsigned y, unsigned x) { // suponemos y >= x
2     if (x==0 || y==x) return 1;
3     return f(y-1, x-1) + f(y-1, x);
4 }
```

La mejor complejidad que podemos conseguir es $O(yx)$. Nótese la doble llamada a la función f. La cuestión es que aquí se realizan muchas llamadas repetidas, como se puede ver en la imagen [20](#).

3. El valor que se obtiene con el método **voraz** para el problema de:

- **la mochila discreta:** es una cota inferior para el valor óptimo que a veces puede ser igual a este¹⁰, pero no garantiza la solución óptima.

¹⁰si todos los objetos caben entonces es la solución óptima

- **la mochila continua:** garantiza la solución óptima. selecciona por mayor peso y fracciona el último objeto de la mejor solución.
4. El valor que se obtiene con **programación dinámica** para el problema de:
- **la mochila discreta:** garantiza la solución óptima siempre.
 - **la mochila continua:** no tiene subestructura óptima, no se puede aplicar.
5. Para acelerar el algoritmo de Prim se mantiene para cada vértice el vértice origen de la arista más corta hasta él.
6. El problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado se puede resolver siempre con una estrategia voraz.
7. La solución óptima al problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado puede construir un único árbol que va creciendo o bien construir un bosque de árboles que al final se injertan en un único árbol.
8. Un tubo de n centímetros de largo se puede cortar en segmentos de 1 centímetro, 2 centímetros, etc. Existe una lista de los precios a los que se venden los segmentos de cada longitud. Una de las maneras de cortar el tubo es la que más ingresos nos producirá. Di cuál de estas tres afirmaciones es falsa:
- (a) Es posible evitar hacer la evaluación exhaustiva de fuerza bruta guardando para cada posible longitud $j < n$ el precio más elevado posible que se puede obtener dividiendo el tubo correspondiente. **VERDADERA: Memoización**
 - (b) Hacer una evaluación exhaustiva de fuerza bruta de todas las posibles maneras de cortar el tubo consume un tiempo $\Theta(2^n)$. **VERDADERA.**
 - (c) Hacer una evaluación exhaustiva de fuerza bruta de todas las posibles maneras de cortar el tubo consume un tiempo $\Theta(n!)$. **FALSA.**
9. La mejora que en general aporta la programación dinámica frente a la solución ingenua se consigue gracias al hecho de que en la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos.
10. Un informático quiere subir una montaña y para ello decide que tras cada paso, el siguiente debe tomarlo en la dirección de máxima pendiente hacia arriba. Además, entenderá que ha alcanzado la cima cuando llegue a un punto en el que no haya ninguna dirección que sea cuesta arriba. **Está usando un algoritmo voraz.**
11. En el método voraz es habitual preparar los datos para disminuir el coste temporal de la función que determina cuál es la siguiente decisión a tomar.
12. La estrategia más eficiente para calcular el n -ésimo elemento de la serie de Fibonacci ($f(n) = f(n-1) + f(n-2)$, $f(1) = f(2) = 1$) es programación dinámica.
13. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```

1 int f( int x, int y ) {
2     if (x <= y) return 1;
3     return x + f(x-1, y);
4 }
```

La mejor complejidad espacial que se puede conseguir es $O(1)$:

```

1 int f_iterativo(int x, int y) {
2     int resultado = 1;
3     while (x > y) {
4         resultado += x;
5         x--;
6     }
7     return resultado;
8 }
```

14. El principio de optimalidad es una condición **necesaria** para que se pueda aplicar programación dinámica a un problema (y para divide y vencerás también).
15. La mochila discreta presenta una subestructura óptima.
16. La PD recursiva puede ser más eficiente en cuanto a coste temporal. La PD iterativa puede ser más eficiente en cuanto a coste espacial.
17. La diferencia entre Prim y Kruskal es que Prim es más efectivo donde hay muchas aristas y Kruskal en grafos dispersos con pocas aristas.

Resumen de complejidades de los problemas vistos en las diapositivas:

En el caso del Puzzle: b = Límite de hijos a la hora de expandir, d = Límite de profundidad a la hora de explorar.

9.3. Final

1. El subgrafo que paso a paso que va generando el algoritmo de Prim siempre contiene una única componente conexa. Kruskal no tiene por qué.
2. Mergesort tiene una complejidad espacial lineal con el tamaño del vector, la de Quicksort y Heapsort es constante.
3.
 - Si $f \in \Theta(g)$ entonces $O(f) \in O(g)$. **VERDADERO**
 - Si $f \notin \Omega(g)$ entonces $O(f) = \Omega(g)$. **FALSO**
 - Si $f \in O(g)$ entonces $g \notin O(f)$. **FALSO**
4. ¿Qué hace la siguiente función?

| Problema | Complejidad Temporal | Complejidad Espacial |
|---|---|--|
| Mochila discreta (recursiva) | $\Omega(n), O(2^n)$ | – |
| Mochila discreta (PD iterativa) | $\Theta(nW)$, reducible a $\Theta(n)^{11}$ | $\Theta(nW)$, reducible a $\Theta(W)$ |
| Corte de tubos (recursiva) | $O(2^n)$ | – |
| Corte de tubos (PD iterativa) | $O(n^2)$ | $O(n)$ |
| Corte de tubos (memoization) | $O(n^2)$ | $O(n)$ |
| Coeficiente binomial (recursiva) | $O(2^{\min(r,n-r)})$ | – |
| Coeficiente binomial (memoización) | $O(rn)$ | – |
| Mochila continua (voraz) | $O(n \log n)$ | – |
| Mochila discreta (voraz) | $O(n \log n)^{12}$ | – |
| Prim ¹³ (naive) | $O(V^3)$, reducible a $O(V^2)$ | – |
| Kruskal | $O(E \log E + V \log V)^{14}$ | – |
| Cambio de monedas (PD iterativa) | $O(nV)$ | $O(V)$ |
| Cambio de monedas (voraz) | $O(n)$ | $O(1)$ |
| Fontanero diligente | $O(n \log n)$ | $O(n)$ |
| Selección de tareas (voraz) | $O(n!)$ | $O(n)$ |
| Selección de tareas (Hungría) | $O(n^3)$ | $O(n^2)$ |
| Coloreado de grafos (backtracking) | $O(k^n)$ | $O(n + E)$ |
| Recorrido del caballo (Knight's Tour) | $O(8^{n^2})$ (o $O(8^{64})$ para 8×8) | $O(n^2)$ |
| Laberinto ($n \times m$) | $O(4^{nm})$ | $O(nm)$ |
| Asignación de tareas (perm. backtracking) | $O(n!)$ | $O(n)$ |
| Empresa naviera (mochila 0–1) | $O(2^N)$ | $O(N)$ |
| Asignación de turnos (backtracking) | $O((T + 1)^N)$ | $O(N + T)$ |
| Sudoku (9×9, backtracking) | $O(9^{81})$ | $O(81)$ |
| Puzzle deslizante (ramificación y poda) | $O(b^d)$ (exponencial en n^2) | $O(b^d)$ (exponencial en n^2) |

Table 4. Problemas con sus complejidades temporales y espaciales

```

1 void f( vector<int> &A ) {
2     priority_queue<int> pq;
3     for( int i = A.size()-1; i >= 0; i-- ) {
4         pq.push(A[i]);
5     }
6     A.clear();
7     while( !pq.empty() ) {
8         A.push_back(pq.top());
9         pq.pop();
10    }
11 }
```

Ordena el vector A.

5. • $O(2^n) \in O(n!)$
 • $O(n^n) \notin O(n!)$
 • $O(3^n) \notin O(2^n)$

6. Si $f(n) \in O(g(n))$:

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

- Es imposible que $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$.
7. Si resolvemos un problema de optimización mediante el método de la vuelta atrás, ¿se puede usar, además de una cota optimista, una cota pesimista para reducir el número de soluciones exploradas? **SÍ**
 8. Un problema tiene subestructura óptima cuando su solución se puede construir eficientemente a partir de soluciones de subproblemas suyos.
 9. El problema del alfarero (solución discreta con tiempos discretos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $m_i \in N$; El valor de cada pieza terminada, $v_i \in N$ y el tiempo necesario para su fabricación $t_i \in N, i \in [0..n - 1]$. ¿Cuántos objetos de cada clase hay que fabricar para maximizar la ganancia teniendo en cuenta que el tiempo total está limitado por $T \in N$? Si T no es muy grande con respecto a n ¿Cuál de los siguientes esquemas algorítmicos resultaría más eficiente para resolverlo? **PROGRAMACIÓN DINÁMICA**. Es el problema de la mochila con otro enfoque.
 10. En el esquema de vuelta atrás, los mecanismos de poda basados en la mejor solución hasta el momento pueden eliminar nodos que representan posibles soluciones factibles.
 11. Complejidad de la siguiente relación de recurrencia:

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

Seleccione una:

- $f(n) \in \Theta(n^3)$ **SÍ**
 - $f(n) \in \Theta(n^4)$ **NO**
 - $f(n) \in \Theta(n^2)$ **NO**
12. ¿Tiene sentido usar una función que indique cómo de prometedor es un nodo cuando resolvemos un problema que no es de optimización mediante ramificación y poda? **Sí, si se puede diseñar de manera que intente predecir si un nodo conducirá o no a la solución.**
 13. Indica cuál es la complejidad, en función de n , del fragmento siguiente:

```

1  for ( int i = 0; i < n; i++ ) {
2      A[i] = 0;
3      for ( int j = 0; j < 2*n; j++ )
4          A[i] += B[j];
5  }
```

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

14. Indica la distinta.

- $n + n \log_2 n \in \Omega(n + n \log_2 n)$ **VERDADERO**
- $\Omega(n^2) \subset \Omega(n)$ **FALSO**
- $O(n^2) \subset O(2^{\log_2 n})$ **FALSO** Esto es muy fácil pues $2^{\log_2 n} = n$ y $O(n^2) \not\subset O(n)$.

15. ¿De qué clase de complejidad es la solución de la siguiente relación de recurrencia?

$$\begin{cases} f(n) = 1 + f\left(\frac{n}{b}\right), & \text{si } n > 1 \\ f(1) = 1, & \text{con } b \in \mathbb{N}, b > 1 \end{cases}$$

Seleccione una:

- Depende del valor de b . **FALSO**
 - $f(n) \in \Theta(\log n)$ **VERDADERO**
 - $f(n) \in \Theta(n)$ **FALSO**
16. En los algoritmos de ramificación y poda, ¿el valor de una cota pesimista es menor que el valor de una cota optimista? (se entiende que ambas cotas se aplican sobre el mismo nodo). **En general sí, si se trata de un problema de maximización, aunque en ocasiones ambos valores pueden coincidir.**
17. Con respecto al tamaño del problema, ¿Cuál es el orden de complejidad temporal asintótica de la siguiente función? (asumimos que A es una matriz cuadrada)

```

1 void traspuesta( vector < vector < int >> A) {
2     for( int i = 1; i < A.size(); i++ )
3         for( int j = 0; j < i ; j++ )
4             swap( A[i][j], A[j][i] );
5 }
```

Es lineal. El tamaño del problema es A y $A = n^2$ por ser una matriz cuadrada. El código es n^2 , pero como A también lo es, se trata de una complejidad lineal.

18. Relaciones de recurrencia del algoritmo k-ésimo (Quickselect)

- **Peor caso:**

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1 \\ T(n - 1) + n, & \text{en otro caso} \end{cases}$$

- **Mejor caso:**

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + n, & \text{en otro caso} \end{cases}$$

19. En un algoritmo de ramificación y poda se puede encontrar una solución óptima sin haber alcanzado nunca un nodo hoja sólo si se hace uso de cotas pesimistas.

20. Si $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)}$ resulta ser una constante positiva no nula, ¿cuál de las siguientes expresiones NO puede darse?

- $f(n) \in \Theta(g(n))$
- $f(n) \in \Omega(g(n))$ y $g(n) \in \Omega(f(n))$
- $g(n) \notin \Theta(f(n))$ **ESTA NO SE PUEDE DAR**

21. Complejidad en función de n donde k es una constante que no depende de n :

```

1  for ( int i = k; i < n - k; i++) {
2      A[i] = 0;
3      for ( int j = i - k; j < i + k; j++ )
4          A[i] += B[j];
5  }
```

$O(n)$.

22. El valor que se obtiene con el método voraz para el problema de la mochila discreta es un valor inferior al óptimo que a veces puede ser igual a este.
23. Compejidad en función de n del fragmento siguiente (A es un `vector<int>` y `sort` es la función de ordenación de la STL):

```

1  sort(begin(A), end(A));
2  int acc = 0;
3  for(auto i : A)
4      acc += i;
```

$\Theta(n \log n)$. `sort` tiene complejidad de $O(n \log n)$ y el `for` de $O(n)$.

24. La solución de programación dinámica iterativa del problema de la mochila discreta tiene la restricción de que los pesos tienen que ser enteros positivos.
25. Sea el problema "Camino de coste mínimo" con ocho movimientos: todas las casillas colindantes. Utilizando la técnica ramificación y poda, se pretende obtener la dificultad del camino más favorable desde el origen hasta la casilla destino. A igualdad de coste temporal, ¿qué se prefiere una buena cota pesimista o una buena cota optimista? **No hay distinción, la más rentable siempre será la que mejor se aproxime a la solución del nodo.**
26. Sea el problema "Camino de coste mínimo" con tres movimientos. Este, sureste y sur. En cuanto a la complejidad espacial y temporal. ¿Cuál de los siguientes esquemas algorítmicos resulta más eficiente para resolverlo? **Programación dinámica.**
27. Mientras que el algoritmo de Prim va construyendo un único árbol de recubrimiento seleccionando vértices uno a uno, el algoritmo de Kruskal va construyendo un bosque que va uniendo añadiendo aristas, hasta obtener un único árbol de recubrimiento.

28. Sea el vector $v[8] = 8, 6, 4, 5, 4, 3, 2, 2$, se trata de un montículo máximo.

29. ¿Cuál es el coste temporal asintótico de la siguiente función?

```

1  void f(int n, int arr[]) {
2      int i = 0, j = 0;
3      for (; i < n; ++i)
4          while (j < n && arr[i] < arr[j])
5              j++;
6  }
```

Tiene complejidad de $O(n)$.

30. Marca la diferente:

- $O(4^{\log_2(n)}) \subset O(n) \subset O(2^n)$
- $O(2^{\log_2(n)}) \subset O(n^2) \subset O(n!)$ **VERDADERA** Porque $O(2^{\log_2(n)}) \subset O(n^2) \subset O(n!)$ es lo mismo que $O(n) \subset O(n^2) \subset O(n!)$
- $O(n^2) \subset O(2^{\log_2(n)}) \subset O(2^n)$

31. Sea el problema de "Camino de coste mínimo" con tres movimientos. Este, sureste y sur. Utilizando la técnica ramificación y poda se pretende obtener la dificultad del camino más favorable desde el origen hasta la casilla destino. Para tratar de acelerar la búsqueda, calculamos una cota para cada uno de los nodos visitados que consiste en resolver el mismo problema pero aplicando la técnica de programación dinámica con la condición de que el camino debe pasar por la casilla asociada al nodo que acabamos de visitar. ¿Qué podemos decir de esa cota? **Que en realidad no se trata de una cota, ni pesimista ni optimista.**

32. La estrategia más eficiente para resolver el problema del caballo del ajedrez es **vueltra atrás**.

33. Quicksort y k-ésimo tienen en común la división del problema en subproblemas.

34. Si queremos sacar todos los caminos posibles entre dos casillas cualquiera en el problema del laberinto deberemos usar **vuelta atrás**.

35. En función de n , la complejidad temporal de la siguiente función es $\Theta(n)$.

```

1  int f(int n) {
2      int k = 0;
3      for (int i = n; i > 0; i /= 3)
4          for (int j = 1; j < i; j += a)
5              k++;
6      return k;
7  }
```

36. La serie denominada *tribonacci* se define de la siguiente manera: $T(0) = T(1) = 1$, $T(2) = 2$, y $T(n) = T(n-3)+T(n-2)+T(n-1)$ para $n > 3$. **Un algoritmo de programación dinámica iterativa permite calcular el valor de $T(n)$ en tiempo $\Theta(n)$.**
37. Una de las respuestas siguientes es **falsa**. ¿Cuál es? El problema del viajante de comercio...
- se puede resolver exactamente usando un algoritmo de programación dinámica. **VERDADERO**
 - se puede resolver exactamente usando un algoritmo de búsqueda y enumeración como es el de vuelta atrás o el de ramificación y poda. **VERDADERO**
 - se puede resolver exactamente usando un algoritmo voraz derivado del de Kruskal. **FALSO**
38. La función `test()` procesa una lista de n elementos y devuelve un real. La definición de la función es recursiva. Primero descompone la lista en dos sublistas de la misma longitud usando un segmento de código que tiene una complejidad lineal con la longitud de la lista, envía cada una de dos sublistas a `test()` para que la procese, hace una serie de operaciones, con el resultado y el valor de retorno, de coste temporal constante. ¿Cuál es el coste temporal asintótico de la función `test()` en función de n ? $\Theta(n \log n)$ porque hace una llamada $O(n)$ y luego descompone en dos sublistas $O(\frac{n}{2})$.
39. ¿Cuál de las siguientes formulaciones expresa mejor la complejidad temporal, en función del parámetro n , de la siguiente función? (asumimos que n es potencia exacta de 2)
40. ¿Cuál de las siguientes formulaciones expresa mejor la complejidad temporal, en función del parámetro , de la siguiente función? (asumimos que n es potencia exacta de 2)

```

1 int f(int n){
2     int k=0;
3     for (int i = 2; i <= n; i*=2)
4         for (int j=i; j > 0; j-=2)
5             k++;
6     return k;
7 }
```

$$\sum_{p=1}^{\log n} 2^{p-1}$$

41. El problema del cambio. Se dispone de un conjunto finito de números naturales y se pretende obtener el subconjunto de menor tamaño cuyos elementos suman una cierta cantidad C. ¿Qué estrategia es la más apropiada para resolverlo? **Programación dinámica.**

42. El funcionamiento del algoritmo de ordenación Heapsort es similar al algoritmo de ordenación por selección, ya que localiza el valor más grande y lo sitúa en la posición final del vector; a continuación, localiza el siguiente valor más grande y lo sitúa en la posición anterior a la última, etc. **El algoritmo Heapsort tiene una complejidad $O(n \log n)$ en el caso peor, mejor que la complejidad $O(n^2)$ del algoritmo de selección porque Heapsort utiliza un algoritmo mucho más eficiente para localizar los valores del vector que valen más.**
43. Una es falsa y las otras dos verdaderas:
- $O(n^n) \in O(n!)$ **VERDADERO**
 - $O(3^n) \in O(2^n)$ **VERDADERO**
 - La complejidad temporal de Quicksort es $O(n^2)$ y $\Omega(n \log n)$. **FALSO** La complejidad de Quicksort es $O(n \log n)$, $\Omega(n \log n)$ y $\Theta(n^2)$.
44. En el método voraz es habitual preparar los datos para disminuir el coste temporal de la función que determina cuál es la siguiente decisión a tomar.
45. Las soluciones factibles a un problema de optimización deben cumplir dos restricciones y queremos resolver el problema mediante vuelta atrás o ramificación y poda. **La cota optimista usada para podar se puede basar en relajar cualquiera de las dos restricciones.**
46. Sea el vector $v = 1, 3, 2, 7, 4, 6, 8$ cuyos elementos están dispuestos formando un montículo de mínimos. Posteriormente añadimos en la última posición del vector un elemento nuevo con valor 5. ¿Qué operación hay que hacer para que el vector siga representando un montículo de mínimos? **Intercambiar el 7 con el 5.**
47. Indica cuál es la complejidad, en función de $n (n \geq 0)$, del fragmento siguiente:

```

1 int f( int n ) {
2     if( n == 0 )
3         return n;
4     return f(n/2) *f(n/2);
5 }
```

$$\Theta(n)$$

48. Dadas las siguientes funciones construidas mediante la técnica de memoización:

```

1 int f( vector<unsigned> &x, unsigned i ) {
2     if( x[i] != SENTINEL )
3         return x[i];
4     if( i < 5 )
5         return i;
6     return x[i] = f(x, i-1) + f(x, i-3);
7 }
```

```

1 int f( unsigned i ){
2     vector<unsigned> x(i, SENTINEL);
3     return f( x, i );
4 }
```

La declaración para SENTINEL más adecuada es `const unsigned SENTINEL = 0;`

49. ¿Cuál es la complejidad de la siguiente función? Suponed que la cola de prioridad está implementada como un heap y que $n = A.size()$. `priority_queue<int> pq(begin(A), end(A))` construye un *heap* a partir de los datos que hay en el vector A.

```

1 void f( vector<int> &A ) {
2     priority_queue<int> pq( begin(A), end(A) );
3     A.clear();
4     while( !pq.empty() ) {
5         A.push_back(pq.top());
6         pq.pop();
7     }
8 }
```

$\Theta(n \log n)$ **No se puede aplicar la técnica de memoización** porque se trata de valores reales.

50. La función que mejor expresa el número de llamadas recursivas que hace Quicksort en el mejor de los casos es $\sum_{i=0}^{\log n} 2i$. Esto se debe a que esta función es igual a $\Theta(n)$ y ese es el número de llamadas que realiza Quicksort. **Complejidad y número de llamadas es algo totalmente distinto.**
51. De los problemas siguientes, indicad cuál no se puede tratar eficientemente como los otros dos:
- **El problema de la mochila sin fraccionamiento y sin restricciones en cuanto al dominio de los pesos de los objetos y de sus valores.** No se puede aplicar programación dinámica aquí.
 - El problema del cambio, o sea, el de encontrar la manera de entregar una cantidad de dinero usando el mínimo número de monedas posibles. **PROGRAMACIÓN DINÁMICA**
 - El problema de cortar un tubo de forma que se obtenga el máximo beneficio posible.**PROGRAMACIÓN DINÁMICA**
52. Un informático quiere subir a una montaña y para ello decide que tras cada paso, el siguiente debe tomarlo en dirección de máxima pendiente hacia arriba. Además, entenderá que ha alcanzado la cima cuando llegue a un punto en el que no haya ninguna dirección que sea cuesta arriba. ¿Qué tipo de algoritmo está usando nuestro informático? **VORAZ**

53. La mejora que en general aporta la programación dinámica frente a la solución ingenua se consigue gracias al hecho de que **en la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos**.
54. El problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado **se puede resolver siempre con una estrategia voraz**.
55. ¿Cuál de estas tres estrategias voraces obtiene un mejor valor para la mochila discreta?
- **Meter primero los elementos de mayor valor específico o valor por unidad de peso.**
 - Meter primero los elementos de menor peso.
 - Meter primero los elementos de mayor valor.
56. El mecanismo que se usa para acelerar el algoritmo de Prim se basa en **mantener para cada vértice el vértice origen de la arista más corta hasta él o mantener para cada vértice su padre más cercano**.
57. ¿Cómo se vería afectada la solución voraz al problema de la asignación de tareas en el caso de que se incorporaran restricciones que contemplen que ciertas tareas no pueden ser adjudicadas a ciertos trabajadores?
- **La solución factible ya no estaría garantizada, es decir, pudiera ser que el algoritmo no llegue a solución ninguna.**
 - Habría que replantearse el criterio de selección para comenzar por aquellos trabajadores con más restricciones en cuanto a las tareas que no pueden realizar para asegurar, al menos, una solución factible.
 - Ya no se garantizaría la solución óptima pero sí una factible.
58. El valor que se obtiene con el método voraz para el problema de la mochila discreta es **una cota inferior para el valor óptimo que a veces puede ser igual a este**.
59. Cuando se calculan los coeficientes binomiales usando la recursión, ¿con qué problema se da y cómo se puede resolver? **Se repiten muchos cálculos y ello se puede evitar usando programación dinámica**.
60. Los algoritmos de programación dinámica hacen uso de **una estrategia que ahorra cálculos innecesarios / de que se puede ahorrar cálculos guardando resultados anteriores en un almacén**.
61. La eficiencia de los algoritmos voraces se basa en el hecho de que **las decisiones tomadas nunca se reconsideran**.
62. En la solución al problema de la mochila continua, ¿por qué es conveniente la ordenación previa de los objetos? **Para reducir la complejidad temporal en la toma de cada decisión: de $O(n)$ a $O(1)$ donde n es el número de objetos a considerar**.

63. Dado un problema de optimización, el método voraz **garantiza la solución óptima sólo para determinados problemas.**
64. Un tubo de un centímetro de largo se puede cortar en segmentos de 1cm, 2cm, etc. Existe una lista de los precios a los que se venden los segmentos de cada longitus. Una manera de cortar el tubo es la que más ingresos nos producirá. Di cuál de estas tres afirmaciones es falsa.
- Hacer una evaluación exhaustiva “de fuerza bruta” de todas las posibles maneras de cortar consume un tiempo $\Theta(2^n)$.
 - **Hacer una evaluación exhaustiva “de fuerza bruta” de todas las posibles maneras de cortar consume un tiempo $\Theta(n!)$.**
 - Es posible evitar hacer la evaluación exhaustiva “de fuerza bruta” guardando, para cada posible longitud $i < n$, el precio más elevado posible que se puede obtener dividiendo el tubo correspondiente.
65. Supongamos que una solución recursiva a un problema de optimización muestra estas dos características: por un lado, se basa en obtener soluciones óptimas a problemas parciales más pequeños, y por otro, estos subproblemas se resuelven más de una vez durante el proceso recursivo. Este problema es candidato a tener una solución alternativa basada en **un algoritmo de programación dinámica**.
66. ¿Cuál de estos tres problemas de optimización no tiene, o no se le conoce, una solución óptima?
- El árbol de coste mínimo de un grafo conexo. **SÍ**
 - El problema de la mochila discreta o sin fraccionamiento. **NO**
 - El problema de la mochila continua o con fraccionamiento. **SÍ**
67. La programación dinámica...
- ...en algunos casos se puede utilizar para resolver problemas de optimización con dominios continuos pero probablemente pierda su eficacia ya que puede disminuir drásticamente el número de subproblemas repetidos.
 - Las otras dos opciones son ciertas.
 - ...normalmente se usa para resolver problemas de optimización con dominios discretizables puesto que las tablas se han de indexar con este tipo de valores.
68. Si ante un problema de decisión existe un criterio de selección voraz entonces:
- la solución óptima está garantizada.
 - al menos una solución factible está garantizada
 - **Ninguna de las otras dos opciones es cierta.**
69. Cuando la descomposición recursiva de un problema da lugar a subproblemas de tamaño similar, ¿qué esquema podría ser más apropiado? **Programación dinámica**.

70. ¿Cuál de los siguientes pares de problemas son equivalentes en cuanto al tipo de solución (óptima, factible, etc.) aportada por el método voraz?

- El fontanero diligente y el problema del cambio.
- La mochila continua y la asignación de tareas.
- **La mochila discreta y la asignación de tareas.**

71. **Programación dinámica** es la estrategia más eficiente para calcular el n -ésimo elemento de la serie de Fibonacci.

72. El TAD *Union-find* se usa en Kruskal para **comprobar si un arco forma ciclos**.

73. Se pretende aplicar la técnica memoización a la siguiente función recursiva:

```

1 int f( int x, int y ) {
2     if( x > y ) return 1;
3     return x*(y-1) + f(x,y-2);
4 }
```

En el caso más desfavorable ¿qué complejidades temporal y espacial cabe esperar de la función resultante?

- Ninguna de las otras dos opciones es correcta.
- Temporal $O(xy)$ y espacial $O(x)$.
- $O(y - x)$ tanto temporal como espacial. Por la condición `if (x>y)`, sólo se hace la recursión cuando x es menor o igual a y .

74. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```

1 unsigned f( unsigned y, unsigned x) { // suponemos y >= x
2     if (x==0 || y==x) return 1;
3     return f(y-1, x-1) + f(y-1, x);
4 }
```

¿Cuál es la mejor complejidad temporal y espacial que se puede conseguir? $O(xy)$ y $O(y)$.

75. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```

1 int f( int x, int y ) {
2     if( x <= y ) return 1;
3     return x + f(x-1,y);
4 }
```

¿Cuál es la mejor complejidad espacial que se puede conseguir? $O(1)$

76. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```

1 unsigned f( unsigned x, unsigned v[ ] ) {
2     if (x==0)
3         return 0;
4     unsigned m = 0;
5     for(unsigned y = 0; y<x; y++)
6         m=max(m, v[y] + f(x-y, v));
7     return m;
8 }
```

¿Cuál es la mejor complejidad espacial que se puede conseguir? $O(x)$ La mejor estructura para el almacén es `int A[]`.

77. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```

1 float f(unsigned x, int y) {
2     if(y<0) return 0;
3     float a = 0.0;
4     if(v1[y] <= x)
5         a = v2[y] + f(x-v1[y], y-1);
6     float b = f(x, y-1);
7     return min(a, 2+b);
8 }
```

La mejor estructura para el almacén es `unsigned A[][]`.

78. La complejidad temporal en el mejor de los casos es una función de la talla que tiene que estar definida para todos los posibles valores de esta.
79. Con respecto al esquema Divide y Vencerás, ¿es cierta la siguiente afirmación? Si la talla se reparte equitativamente entre los subproblemas, entonces la complejidad temporal resultante es una función logarítmica. **No tiene porqué, la complejidad temporal no depende únicamente del tamaño resultante de los subproblemas.**
80. La versión de *Quicksort* que utiliza como pivote la mediana del vector **no presenta caso mejor y peor distintos para instancias del mismo tamaño**. Esto se debe a que forzar la mediana "fuerza" también al mejor caso.
81. Dada la siguiente relación de recurrencia, ¿qué cota es verdadera?

$$f(n) = \begin{cases} 1, & n = 1 \\ n + 2f(n - 1) & n > 1 \end{cases}$$

La cota que se cumple es $f(n) \in \Theta(n2^n)$ porque, si desarrollamos:

$$\begin{aligned}
 f(n) &= n + 2f(n-1) \\
 &= n + 2(n-1 + 2f(n-2)) \\
 &= n + 2(n-1) + 2^2f(n-2) \\
 &= n + 2(n-1) + 2^2(n-2) + 2^3f(n-3) \\
 &\dots \\
 &= \sum_{k=0}^{n-1} 2^k(n-k)
 \end{aligned}$$

82. ¿Cuál es la solución a al siguiente relación de recurrencia?

$$f(n) = \begin{cases} \Theta(1) & n = 0 \\ \Theta(1) + f(\frac{n}{3}) & n > 0 \end{cases}$$

Sacamos la relación de recurrencia:

$$\begin{aligned}
 f(n) &= \Theta(1) + f\left(\frac{n}{3}\right) \\
 &= \Theta(1) + \Theta(1) + f\left(\frac{n}{9}\right) \\
 &= \Theta(1) + \Theta(1) + \Theta(1) + f\left(\frac{n}{27}\right) \\
 &\dots \\
 &= \sum_{i=0}^{\log_3 n} \Theta(1) = \Theta(\log_3 n) = \Theta(\log n)
 \end{aligned}$$

Entonces, la complejidad es $\Theta(\log n)$.

83. Indica cuál es la complejidad, en función de n , del fragmento siguiente:

```

1 | for ( int i = n; i > 0; i /= 2)
2 |   for ( int j = n; j > 0; j /= 2)
3 |     a += A[i][j];
  
```

$$O(\log^2(n))$$

84. Los algoritmos de ordenación *Quicksort* y *Mergesort* tienen en común **que aplican la estrategia de divide y vencerás**.

85. Sea la siguiente relación de recurrencia:

$$T(n) = \begin{cases} 1 & \sin \leq 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

Si $T(n) \in O(n)$, ¿en cuál de estos tres casos nos podemos encontrar? Seleccione una:

- $g(n) = 1$ **VERDADERO**
- $g(n) = n$ **FALSO**
- $g(n) = \log n$ **VERDADERO TAMBIÉN**

86. Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$ entonces:

- **Las otras dos opciones son ambas ciertas.**
- $f \in \Theta(\max(g_1, g_2))$
- $f^2 \in \Theta(g_1 g_2)$

87. ¿Cuál de los siguientes algoritmos de ordenación necesita un espacio de almacenamiento adicional al vector que se ordena con complejidad $O(n)$?

- **Mergesort**
- Bubblesort
- Quicksort

88. Se queiren ordenar d números distintos comprendidos entre 1 y n . Para ello se usa un array de n booleanos que se inicializan primero a false. A continuación se recorren los d números cambiando los valores del elemento del vector de booleanos correspondiente a su número a true. Por último, se recorre el vector de booleanos escribiendo los índices de los elementos del vector de booleanos que son true. ¿Es este algoritmo más rápido que el Mergesort?

- **Sólo si** $d \log d > kn$ **donde k es una constante que depende de la implementación.**
- No, ya que este algoritmo ha de recorrer varias veces el vector de booleanos.
- Sí, ya que Mergesort es $O(n \log n)$ y este es $O(n)$.

89. Marca la diferente:

- $\Theta(\log_2(n)) = \Theta(\log_3(n))$ **VERDADERO** Cambiar la base sólo introduce una constante multiplicativa que NO afecta a la notación.
- $\Theta(\log(n^2)) = \Theta(\log(n^3))$ **VERDADERO** Es lo mismo que $\Theta(2 \log(n)) = \Theta(3 \log(n))$ y, por tanto, $\Theta(\log(n)) = \Theta(\log(n))$
- $\Theta(\log^2(n)) = \Theta(\log^3(n))$ **FALSO**

90. La siguiente relación de recurrencia expresa la complejidad de un algoritmo recursivo, donde $g(n)$ es una función polinómica:

$$T(n) = \begin{cases} 1 & \sin \leq 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

Di cuál de las siguientes afirmaciones es cierta:

- Si $g(n) \in \Theta(n^2)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación mediante inserción binaria.

- Si $g(n) \in \Theta(1)$ la relación de recurrencia representa la complejidad temporal del algoritmo de búsqueda dicotómica.
- Si $g(n) \in \Theta(n)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación Mergesort.

91. Marca la diferente:

- $O(n^2) \subset O(2^{\log_2 n}) \subset O(2^n)$ **FALSO**
- $O(4^{\log_2 n}) \subseteq O(n^2) \subset O(2^n)$ **VERDADERO**
- $O(2^{\log_2 n}) \subseteq O(n^2) \subset O(n!)$ **VERDADERO**

92. La complejidad temporal en el mejor de los casos de un algoritmo recursivo:

- coincide con el valor del caso base de la ecuación de recurrencia que expresa la complejidad temporal del algoritmo.
- es la complejidad temporal de las instancias que están en el caso base.
- **ninguna de las dos opciones es verdadera.**

93. Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces:

- $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$
- $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$
- **Las otras dos opciones son ambas ciertas.**

94. La relación de recurrencia

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

- **Expresa el número de llamadas recursivas que hace el algoritmo Quicksort en el peor de los casos.**
- Ninguna de las otras dos opciones es cierta.
- Expresa la complejidad temporal asintótica en el peor de los casos del algoritmo de búsqueda binaria.

95. Tenemos un vector ordenado de tamaño n_0 y un vector desordenado de tamaño n_d y queremos obtener un vector ordenado con todos los elementos. ¿Qué será más rápido?

- Insertar los elementos del vector desordenado (uno a uno) en el vector ordenado.
- Ordenar el desordenado y luego mezclar las listas.
- **Depende de si $n_0 > n_d$ o no.** Porque:
 - Si n_d es más pequeño, la inserción individual podría ser más eficiente.
 - Si n_d es más grande, ordenar y fusionar podría ser mejor.

96. La relación de recurrencia que representa la complejidad en el peor caso del algoritmo de búsqueda del k-ésimo elemento más pequeño de un vector es

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ T(n-1) + n & \text{en otro caso} \end{cases}$$

97. Dada la siguiente relación de recurrencia, ¿qué cota es verdadera?

$$f(n) = \begin{cases} 1 & n = 1 \\ n^2 + 2f\left(\frac{n}{4}\right) & n > 1 \end{cases}$$

- $f(n) \in \Theta(n^2)$ **VERDADERO**
- $f(n) \in \Theta(n^2 \log n)$ **FALSO**
- $f(n) \in \Theta(n^3)$ **FALSO**

Siguiendo el teorema maestro (sección 3.2):

$$f(n) = \begin{cases} 1 & n = 1 \\ c + af\left(\frac{n}{b}\right) & n > 1 \end{cases}$$

Cuando:

- $a < b : f(n) \in \Theta(1)$
- $a = b : f(n) \in \Theta(\log n)$
- $a > b : f(n) \in \Theta(n^{\log_b a})$

Ojo, porque esto es sólo para la $f(n)$ de dentro de los casos, hay que sumarle la complejidad de c y ya sacar la complejidad real a partir de $c + f(n)$.

98. De las siguientes igualdades, o bien dos son ciertas y una es falsa, o bien una es cierta y dos son falsas. Marca la que es diferente a las otras dos.

- $\Theta(n) = \mathcal{O}(n) \cup \Omega(n)$ **FALSO**
- $\Theta(n) = \mathcal{O}(n) \cap \Omega(n)$ **VERDADERO** Por definición.
- $\mathcal{O}(n) = \mathcal{O}(n) \cap \Theta(n)$ **FALSO**

99. ¿Cuál es la condición base para terminar la recursión en la implementación por divide y vencerás del problema restringido del laberinto?

- Llegar al inicio o al final del laberinto, según cómo se establezca la recursión.
- Llegar a una celda con valor 0 en el laberinto.
- **Las otras dos opciones son ambas ciertas.** La recursión termina cuando se llega a una pared o al final o al inicio del laberinto. En estos dos casos no se profundiza más.

100. En la implementación naive de la versión restringida del problema del laberinto, en el caso de que las tres direcciones sean posibles, ¿qué hay que hacer con las longitudes de los caminos obtenidos con las llamadas recursivas?
- Elegir la más grande y sumarle uno.
 - **Elegir la más pequeña y sumarle uno.**
 - Elegir la más cercana y sumarle uno.
101. ¿Qué estructuras de datos utilizaste en la implementación naive de la versión restringida del problema del laberinto? (excluyendo el laberinto)
- Una cola de prioridad. **RAMIFICACIÓN Y PODA**
 - Una matriz bidimensional. **PROGRAMACIÓN DINÁMICA**
 - **Ninguna de las otras dos opciones es cierta.**
102. ¿En qué parte de los algoritmos que hacen uso de la memoización se realiza la verificación de si un subproblema ha sido ya resuelto?
- **Al inicio del algoritmo.**
 - Antes de realizar la llamada recursiva.
 - Despues de realizar la llamada recursiva.
103. Queremos conocer el camino de mínima longitud en el problema restringido del laberinto usando un algoritmo de programación dinámica con la técnica de complejidad espacial mejorada (con dos vectores). ¿Cómo se puede obtener?
- Recorriendo el primer vector, empezando por la casilla de salida y buscando los resultados de los subproblemas accesibles a partir de ella.
 - Recorriendo el primer vector, empezando por la casilla de entrada y buscando los resultados de los subproblemas accesibles a partir de ella.
 - **Ninguna de las dos opciones es correcta.**
104. ¿Qué estrategia es mejor en los algoritmos voraces para resolver el problema restringido del laberinto?
- **Elegir siempre la dirección hacia la salida más cercana que no esté bloqueada.**
 - Elegir la dirección con menos obstáculos en cada paso.
 - Elegir aleatoriamente una dirección en cada paso.
105. ¿Qué estrategia se sigue para realizar la exploración del laberinto con la técnica de vuelta atrás?
- **Realizar una búsqueda en profundidad siguiendo un orden predefinido de movimientos.**
 - Utilizar una búsqueda en amplitud para explorar todas las posibles soluciones en paralelo. **RAMIFICACIÓN Y PODA**

- Realizar una búsqueda heurística utilizando una función de evaluación para guiar la exploración hacia la solución óptima. **RAMIFICACIÓN Y PODA**
106. En la aplicación al problema general del laberinto del algoritmo de ramificación y poda utilizando una cola de prioridad, ¿qué función cumple la cola de prioridad en el proceso de exploración del laberinto?
- **Ordenar los caminos explorados en función de un criterio de evaluación para seleccionar el siguiente camino a explorar.**
 - Almacenar todos los caminos explorados para su posterior análisis y comparación.
 - Realizar una búsqueda en amplitud de todos los caminos posibles antes de tomar una decisión.
107. Si estamos procesando la casilla (i, j) y la casilla de salida del laberinto es la (m, n) , ¿cuál de las siguientes opciones será una buena cota pesimista al camino que queda por recorrer?
- $\max(|m - i|, |n - j|)$
 - $|m - i| + |m - j|$
 - **Ninguna de las otras dos opciones es cierta.**
108. Di cuál de estas tres afirmaciones sobre el problema de la mochila sin división de los objetos y con pesos discretos es cierta.
- El algoritmo de programación dinámica iterativa que lo resuelve calcula el mismo número de subproblemas que un algoritmo de programación dinámica recursiva (del tipo "divide y vencerás" que gestiona un almacén de memorización). **FALSO** Las versiones recursivas sólo resuelven "lo que necesitan", las iterativas lo resuelven todo.
 - El algoritmo de programación dinámica iterativa que lo resuelve precisa como mucho un espacio del orden de $\Theta(n)$ donde n es el número de objetos. **FALSO** Representa las filas de la matriz.
 - El algoritmo de programación dinámica iterativa que lo resuelve precisa como mucho un espacio del orden de $\Theta(P)$ donde P es el peso máximo de la mochila. **VERDADERO** Representa las columnas de la matriz. La versión optimizada que cumple con $O(P)$.
109. ¿Qué estructura de datos es la más eficiente en la programación dinámica iterativa para almacenar los subproblema resueltos del problema de la mochila sin división de los objetos y con pesos discretos?
- Una cola de prioridad.
 - **Una matriz unidimensional.** Un vector. El de la pregunta de antes de $O(P)$.
 - Un árbol binario.

110. Di cuál de estas tres afirmaciones sobre las soluciones basadas en algoritmos de búsqueda y enumeración para el problema de la mochila sin división de los objetos es cierta.

- **No requieren que los pesos sean discretos o discretizables.**
- Requieren que los pesos sean discretos o discretizables para definir los niveles del árbol de búsqueda.
- Deben visitar obligatoriamente todas las soluciones factibles para determinar cuál es la solución óptima.

111. ¿Cuál de estos tres algoritmos de ordenación de vectores tiene una complejidad temporal en el caso peor que es cuadrática con la longitud del vector?

- **Quicksort**
- Mergesort
- Heapsort

112. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```

1 unsigned f( unsigned y, unsigned x) { // suponemos y >= x
2     if (x==0 || y==x)
3         return 1;
4     return f(y-1, x-1) + f(y-1, x);
5 }
```

Mejor complejidad temporal: $O(xy)$. Mejor complejidad espacial: $O(y)$.

113. Se pretende aplicar la técnica memoización a la siguiente función recursiva:

```

1 int f(int x, int y) {
2     if ( x > y )
3         return 1;
4     return x*(y-1) + f(x, y-2);
5 }
```

En el caso más desfavorable, ¿qué complejidades temporal y espacial cabe esperar de la función resultante?

- $O(y - x)$, tanto temporal como espacial.
- Temporal $O(xy)$ y espacial $O(x)$.
- Ninguna de las otras dos opciones es correcta.

114. Se pretende aplicar la técnica memoización a la siguiente función recursiva:

```

1 int f(int x, int y) {
2     if( x<= y)
3         return 1;
4     return x*f(x-1,y)+y;
5 }
```

En el caso más desfavorable, ¿qué complejidades temporal y espacial cabe esperar de la función resultante?

- Temporal $O(x - y)$ y espacial $O(1)$.
- $O(x - y)$, tanto temporal como espacial.
- Ninguna de las otras dos opciones es correcta.

115. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```

1 int f(int x, int y) {
2     if( x <= y)
3         return 1;
4     return x + f(x-1,y);
5 }
```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

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

¿Y temporal?

- $O(y)$
- $O(xy)$
- $O(x)$ VERDADERO

Si aplicásemos memoización, en el caso más desfavorable, ¿qué complejidades temporal y espacial cabe esperar de la función resultante?

- $O(x - y)$, tanto temporal como espacial.
- Temporal $O(x - y)$ y espacial $O(1)$.
- Ninguna de las otras dos opciones es correcta.

116. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```

1 float f(unsigned x, int y) {
2     if (y<0)
3         return 0;
4     float A = 0.0;
5     if (v1[y] <= x)
6         A = v2[y] + f(x-v1[y], y-1);
7     float B = f(x, y-1);
8     return min(A, 2+B);
9 }
```

¿Cuál es la mejor complejidad temporal que se puede conseguir?

- $O(y)$
- $O(x)$
- $O(xy)$ **VERDADERO**

117. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```

1 unsigned f(unsigned x, unsigned v[])
2     if (x==0)
3         return 0;
4     unsigned m = 0;
5
6     for (unsigend k = 0; k < x; k++)
7         m = max( m, v[k] + f(x-k, v) );
8     return m;
9 }
```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

- $O(1)$
- $O(x)$ **VERDADERO**
- $O(x^2)$

118. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```

1 unsigned f(unsigned x, unsigned v[])
2     if (x==0)
3         return 0;
4     unsigned m = 0;
5
6     for (unsigend y = 0; y < x; y++)
7         m = max( m, v[k] + f(x-y, v) );
8     return m;
9 }
```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

- $O(y)$
- $O(x)$ **VERDADERO**
- $O(xy)$

119. Sobre la complejidad temporal de la siguiente función:

```

1  unsigned desperdicio (unsigned n) {
2      if (n<=1)
3          return 0;
4      unsigned sum = desperdicio(n/2) + desperdicio(n/2) +
5                  desperdicio(n/2);
6
7      for (unsigned i=1; i<n-1; i++)
8          for (unsigned j=1; j<=i; j++)
9              for (unsigned k=1; k<=j; k++)
10                 sum+=i*j*k;
11
12 }
```

- Ninguna de las otras dos alternativas es cierta.
- Las complejidades en los casos mejor y peor son distintas.
- El mejor de los caso se da cuando $n \leq 1$ y en tal caso la complejidad es constante.

120. Con respecto al esquema Divide y vencerás, ¿es cierta la siguiente afirmación? Si la talla se reparte equitativamente entre los subproblemas, entonces la complejidad temporal resultante es una función logarítmica.

- No, nunca, puesto que también hay que añadir el coste de la división en subproblemas y la posterior combinación.
- **No tiene porqué, la complejidad temporal no depende únicamente del tamaño resultante de los subproblemas.**
- Sí, siempre, en Divide y vencerás la complejidad temporal depende únicamente del tamaño de los subproblemas.

121. ¿Qué cota se deduce de la siguiente relación de recurrencia?

$$T(n) = \begin{cases} 1 & n = 1 \\ n + 4f\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

- $f(n) \in \Theta(n^2)$ **VERDADERO** 3.2
- $f(n) \in \Theta(n)$
- $f(n) \in \Theta(n \log n)$

Recordamos:

$$f(n) = \begin{cases} 1 & n = 1 \\ c + af\left(\frac{n}{b}\right) & n > 1 \end{cases}$$

Cuando:

- $a < b : f(n) \in \Theta(1)$
- $a = b : f(n) \in \Theta(\log n)$
- $a > b : f(n) \in \Theta(n^{\log_b a})$

Entonces, tenemos que $a = 4$ y $b = 2$. Aplicamos $f(n) \in \Theta(n^{\log_b a})$ y tenemos que $f(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$

122. Marca la diferente:

- $2n^3 - 10n^2 + 1 \in O(n^3)$ **VERDADERO**
- $n + n\sqrt{n} \in \Omega(n)$ **VERDADERO**
- $n + n\sqrt{n} \in \Theta(n)$ **FALSO**

Ponemos que $f(n) = n$ y que $g(n) = n\sqrt{n} = n^{1.5}$. Entonces, $g(n) > f(n)$ siempre.

123. Sea $f(n) = n \log n + n$:

- $f(n) \in \Omega(n \log n)$
- $f(n) \in O(n \log n)$
- **Las otras dos opciones son ciertas.**

124. Dada la siguiente relación de recurrencia, ¿qué cota es verdadera?

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

- $f(n) \in \Omega(2^n)$ **FALSA**
- $f(n) \in \Theta(n^2)$ **FALSA**
- $f(n) \in \Theta(n2^n)$ **VERDADERA**

125. Indica cuál es la complejidad, en función de n , del fragmento siguiente:

```

1 | for (int i=n; i>0; i/=2)
2 |   for (int j=n; j>0; j/=2)
3 |     a+=A[i][j];

```

- $O(\log^2(n))$ **VERDADERO**
- $O(n \log(n))$ **FALSO**
- $O(n^2)$ **FALSO**

126. ¿Cuál de las formulaciones expresa mejor el coste temporal de la siguiente función?

```

1 int f(int n) {
2     int count = 0;
3     for (int i = 2; i < n; i += 2) // A
4         for (int j = 1; j < i; j*=2) // B
5             count += 1;
6     return count;
7 }
```

- $C_e(n) = \sum_{i=1}^{n/2} (1 + \log_2 i)$ **VERDADERO** El bucle A ejecuta $\frac{n}{2}$ iteraciones. El bucle B recorre $\log_2 i + 1$ iteraciones, es decir, $1 + 2 + 4 + 8\dots$
- $C_e(n) = \sum_{i=1}^{n/2} (\log_2(n/2))$ **FALSO**
- $C_e(n) = \frac{\sum_{i=1}^{n-1} (\log_2 2i)}{2}$ **FALSO**

127. Di cuál de estos resultados de coste temporal asintótico es falsa:

- **La ordenación de un vector usando el algoritmo Mergesort requiere en el peor caso un tiempo en $O(n^2)$.**
- La búsqueda binaria en un vector ordenado requiere en el peor caso un tiempo en $O(\log n)$
- La ordenación de un vector usando el algoritmo Quicksort requiere en el peor caso un tiempo $O(n^2)$.

128. ¿Qué tienen en común el algoritmo que obtiene el k-ésimo elemento más pequeño de un vector (estudiado en clase) y el algoritmo de ordenación Quicksort?

- **La división del problema en subproblemas.**
- La combinación de las soluciones a los subproblemas.
- El número de llamadas recursivas que se hacen.

129. Sea la siguiente relación de recurrencia:

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

Si $T(n) \in O(n)$, ¿en cuál de estos tres casos nos podemos encontrar?

- $g(n) = 1$ **VERDADERO** porque $g(n) = O(n)$, y por el Teorema Maestro se tiene $T(n) = \Theta(n)$
- $g(n) = n$ **FALSO** porque entonces $T(n) = \Theta(n \log n) \notin O(n)$
- $g(n) = \log n$ **VERDADERO** porque $g(n) = O(n)$, y por el Teorema Maestro se tiene $T(n) = \Theta(n)$

130. Tenemos una lista ordenada de tamaño n_o y una lista desordenada de tamaño n_d , queremos obtener una lista ordenada con todos los elementos. ¿Cuál sería la complejidad de insertar, uno a uno, todos los elementos de la lista desordenada en la ordenada?

- $O(n_d n_o)$ **FALSO** Sería verdadero si se usase búsqueda lineal en lugar de binaria.
- $O(n_d \log n_o)$ **VERDADERO** Usando búsqueda binaria (como hace std::vector en C++).
- $O(n_d(n_o + n_d))$ **FALSO**

131. ¿Qué se entiende por tamaño de problema?

- El valor máximo que puede tomar una instancia cualquiera de ese problema. **FALSO**
- La cantidad de espacio en memoria que se necesita para codificar una instancia de ese problema. **FALSO**
- El número de parámetros que componen el problema. **VERDADERO**

No son correctas todas al 100%, así que esta pregunta es a la que mejor suene y a lo que el profe le salga de los cojones.

132. Sea la siguiente relación de recurrencia:

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

Si $T(n) \in O(n^2)$, ¿en cuál de estos tres casos nos podemos encontrar?

- $g(n) = n^2$ **VERDADERO**
- $g(n) = n \log n$ **FALSO**
- $g(n) = n$ **FALSO**

133. La eficiencia de los algoritmos voraces se basa en el hecho de que **las decisiones tomadas nunca se reconsideran**.

134. En la solución al problema de la mochila continua, ¿por qué es conveniente la ordenación previa de los objetos?

- Para reducir la complejidad temporal en la toma de decisión de $O(n)$ a $O(1)$, donde n es el número de objetos a considerar.
- Para reducir la complejidad temporal en la toma de decisión de $O(n^2)$ a $O(n \log n)$, donde n es el número de objetos a considerar.
- Porque si no se hace no es posible garantizar que la toma de decisiones siga un criterio voraz.

135. ¿Cómo se vería afectada la solución voraz al problema de la asignación de tareas en el caso de que se incorporaran restricciones que contemplen que ciertas tareas no pueden ser adjudicadas a ciertos trabajadores?

- La solución factible ya no estaría garantizada, es decir, pudiera ser que el algoritmo no llegue a solución alguna.
- Ya no se garantizaría la solución óptima pero si una factible.
- Habría que replantearse el criterio de selección para comenzar por aquellos trabajadores con más restricciones en cuanto a las tareas que no pueden realizar para asegurar, al menos, una solución factible.

136. Se pretende aplicar la técnica memoización a la siguiente función recursiva:

```

1 int f(int x, int y) {
2     if(x>y)
3         return 1;
4     return x+f(x,y-2);
5 }
```

En el caso más desfavorable, ¿qué complejidades temporal y espacial cabe esperar de la función resultante?

- Temporal $O(x * y)$ y espacial $O(x)$
- $O(y - x)$, tanto temporal como espacial.
- Ninguna de las otras dos opciones es correcta.

137. La mejora que en general aporta la programación dinámica frente a la solución ingenua se consigue gracias al hecho de que **en la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos**.

138. La solución óptima al problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado...

- **puede construir un único árbol que va creciendo o bien construir un bosque de árboles que al final se injertan en un único árbol.**
- se construye haciendo crecer un único árbol.
- se construye haciendo crecer varios árboles que al final acaban injertados en un único árbol.

139. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```

1 unsigned f(unsigned x, unsigend v[] ) {
2     if(x==0)
3         return 0;
4     unsigned m = 0;
5     for (unsigned k = 0; k < x; k++)
```

```

6         m = max(m, v[k] + f(x-k, v));
7         return m;
8     }

```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

- $O(x^2)$ FALSO
- $O(1)$ FALSO
- $O(x)$ VERDADERO

140. ¿Cuál de estos tres problemas de optimización no tiene, o no se le conoce, una solución voraz óptima?

- El árbol de cobertura de coste mínimo de un grafo conexo.
- **El problema de la mochila discreta o sin fraccionamiento.**
- El problema de la mochila continua o con fraccionamiento.

141. De los problemas siguientes , indicad cuál no se puede tratar eficientemente como los otros dos:

- El problema de cortar un tubo de forma que se obtenga el máximo beneficio posible. **PROGRAMACIÓN DINÁMICA**
- **El problema de la mochila sin fraccionamiento y sin restricciones en cuanto al dominio de los pesos de los objetos y de sus valores.**
- El problema del cambio, o sea, el de encontrar la manera de entregar una cantidad de dinero usando el mínimo de monedas posibles. **PROGRAMACIÓN DINÁMICA**

142. Cuando se calculan los coeficientes binomiales usando la recurrencia $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$, con $\binom{n}{0} = \binom{n}{n} = 1$, ¿qué problema se da y cómo se puede resolver?

- La recursión puede ser infinita y por tanto es necesario organizarla según el esquema iterativo de programación dinámica.
- Se repiten muchos cálculos y ello se puede evitar haciendo uso de una estrategia voraz.
- **Se repiten muchos cálculos y ello se puede evitar usando programación dinámica.**

143. ¿Qué mecanismo se usa para acelerar el algoritmo de Prim?

- Mantener una lista de los arcos ordenados según su peso.
- El TAD Union-find.
- Mantener para cada vértice el vértice origen de la arista más corta hasta él.

144. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
1 int f(int x, int y) {
2     if (x<=y)
3         return 1;
4     return x+f(x-1,y);
5 }
```

¿Cuál es la mejor complejidad temporal que se puede conseguir?

- $O(y)$ **FALSO**
- $O(xy)$ **FALSO**
- $O(x)$ **VERDADERO**

9.4. Exámenes (míos)

9.4.1. Primer parcial (2024/25)

The screenshot shows the UACloud virtual classroom interface for the first partial exam. The top navigation bar includes 'Página Principal', 'Área personal', and 'Mis cursos'. The main content area displays the exam details and 12 questions.

Exam Details:

- Estado: Finalizado
- Comenzado: lunes, 26 de marzo de 2023, 13:07
- Completo
- Duración: 9 minutos 23 segundos
- Puntos: 6.50/12.00
- Calificación: 5.42 de 10.00 (54.17%)

Questions:

- Pregunta 1** (Correcta)

Se puntuó 1.00 sobre 1.00

Se marcó pregunta

¿De cuál de estos resultados de corte temporal asintótico es falso?

Selección una:

 - La ordenación de un vector usando el algoritmo MergeSort requiere en el peor caso un tiempo en $O(n^2)$. ✓
 - La búsqueda binaria en un vector ordenado requiere en el peor caso un tiempo en $O(\log n)$
 - No contexto (equivalente a no marcar nada).
 - La ordenación de un vector usando el algoritmo Quicksort requiere en el peor caso un tiempo en $O(n^2)$
- Pregunta 2** (Correcta)

Se puntuó 1.00 sobre 1.00

Se marcó pregunta

¿Qué tienen en común el algoritmo que obtiene el λ -ésimo elemento más pequeño de un vector (estudiado en clase) y el algoritmo de ordenación Quicksort?

Selección una:

 - La división del problema en subproblemas. ✓
 - No contexto (equivalente a no marcar nada).
 - La combinación de las soluciones a los subproblemas.
 - El número de llamadas recursivas que se hacen.
- Pregunta 3** (Incorrecta)

Se puntuó 0.00 sobre 1.00

Se marcó pregunta

¿Cuál de las formulaciones expresa mejor el coste temporal de la siguiente función?

```
int f(int n) {
    int count = 0;
    for (int i = 2; i < n; i++) {
        for (int j = i; j < i + 2) {
            count++;
        }
    }
    return count;
}
```

Selección una:

 - $C_1(n) = \sum_{i=2}^{n-1} (1 + \log_2 i)$
 - No contexto (equivalente a no marcar nada).
 - $C_2(n) = \frac{n^2 \log_2 n}{2}$ ✗
 - $C_3(n) = \sum_{i=2}^{n-1} (\log_2 (n/2))$
- Pregunta 4** (Correcta)

Se puntuó 1.00 sobre 1.00

Se marcó pregunta

Si $f \in \Omega(g_1)$ y $f \in \Omega(g_2)$ entonces

Selección una:

 - $f \in \Omega(g_1 \cdot g_2)$
 - $f \in \Omega(g_1 + g_2)$ ✓
 - $f \notin \Omega(g_1 \cdot g_2)$
 - No contexto (equivalente a no marcar nada).
- Pregunta 5** (Correcta)

Se puntuó 1.00 sobre 1.00

Se marcó pregunta

Con respecto al parámetro n , ¿cuál es la complejidad temporal de la siguiente función?

```
void f1( unsigned n ) {
    if ( n < 1 ) return;
    for ( int i = 0; i < n; i++ )
        for ( int j = 0; j < n; j++ )
            for ( int k = 0; k < n; k++ )
                cout << * * *;
    for ( int l = 0; l < n; l++ )
        cout << n / 2;
}
```

Selección una:

 - $\Theta(n^3)$
 - No contexto (equivalente a no marcar nada).
 - $\Theta(n^2 \log n)$
 - $\Theta(n^3 \log n)$ ✓
- Pregunta 6** (Incorrecta)

Se puntuó 0.00 sobre 1.00

Se marcó pregunta

¿Cuál de las siguientes formulaciones expresa mejor el coste temporal asintótico de la siguiente función?

```
int f(int n) {
    int count = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < 2 * i + 2; j++)
            count++;
    return count;
}
```

Selección una:

 - $f(n) = \sum_{i=0}^{\lfloor n/2 \rfloor} 4n \left(\frac{1}{2}\right)^i$ ✓
 - $f(n) = \sum_{i=0}^{\lfloor n/2 \rfloor} 2n$.
 - No contexto (equivalente a no marcar nada).
 - Ninguna de las otras dos opciones es correcta.
- Pregunta 7** (Incorrecta)

Se puntuó 0.00 sobre 1.00

Se marcó pregunta

De las siguientes expresiones, o bien dos son verdaderas y una es falsa, o bien dos son falsas y una es verdadera. Marca la que (en este sentido) es distinta a las otras dos.

Selección una:

 - $\Theta(\log(n^2)) = \Theta(\log(n^2))$
 - $\Theta(\log(n)) = \Theta(\log(n))$
 - $\Theta(\log(n)) = \Theta(\log^2(n))$
 - No contexto (equivalente a no marcar nada). ✗
- Pregunta 8** (Correcta)

Se puntuó 1.00 sobre 1.00

Se marcó pregunta

La siguiente relación de recurrencia expresa la complejidad de un algoritmo recursivo, donde $g(n)$ es una función polinómica:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

¿Cuál de las siguientes afirmaciones es cierta?

Selección una:

 - $S_1(g(n)) \in O(1)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación mediante inserción binaria.
 - $S_2(g(n)) \in O(1)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación MergeSort. ✓
 - No contexto (equivalente a no marcar nada).
 - $S_3(g(n)) \in O(1)$ la relación de recurrencia representa la complejidad temporal del algoritmo de búsqueda dicotómica.
- Pregunta 9** (Incorrecta)

Se puntuó -0.50 sobre 1.00

Se marcó pregunta

Tenemos una lista ordenada de tamaño n_L y una lista desordenada de tamaño n_U , queremos obtener una lista ordenada con todos los elementos. ¿Cuál será la complejidad de insertar, uno a uno, todos los elementos de la lista desordenada en la ordenada.

Selección una:

 - $O(n_L \cdot n_U)$ ✗
 - No contexto (equivalente a no marcar nada).
 - $O(n_U \log n_L)$
 - $O(n_L \log n_U)$
- Pregunta 10** (Incorrecta)

Se puntuó -0.50 sobre 1.00

Se marcó pregunta

JQué se entiende por tamaño del problema?

Selección una:

 - No contexto (equivalente a no marcar nada).
 - El valor máximo que puede tener una instancia cualquiera de ese problema.
 - La cantidad de espacio en memoria que se necesita para codificar una instancia de ese problema.
 - El número de parámetros que componen el problema.
- Pregunta 11** (Correcta)

Se puntuó 1.00 sobre 1.00

Se marcó pregunta

Sea la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n < 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

Si $T(n) = O(n^2)$, en cuál de estos tres casos nos podemos encontrar?

Selección una:

 - $g(n) = n^2$ ✓
 - No contexto (equivalente a no marcar nada).
 - $g(n) = -n \log n$
 - $g(n) = n$
- Pregunta 12** (Incorrecta)

Se puntuó 1.00 sobre 1.00

Se marcó pregunta

Con respecto a la complejidad temporal de la siguiente función, ¿cuál de las siguientes afirmaciones es cierta?

```
int post2_1( unsigned n ) {
    if (n==0)
        return 1;
    else
        return post2_1( n/2 ) * post2_1( n/2 );
    else
        return 2 * post2_1( n/2 ) + post2_1( n/2 );
}
```

Selección una:

 - El costo temporal exacto de la función es $O(n)$.
 - La complejidad temporal en el mejor de los casos es constante.
 - No contexto (equivalente a no marcar nada).
 - Las otras dos afirmaciones son ambas falsas.

9.4.2. Segundo parcial (2024/25)

Página Principal Área personal Mis cursos

Práctica final Entregado de la práctica ...

Ficheros de test

Entrega de prácticas Pract-1 Pract-2 Pract-3 Pract-4 Pract-5 Pract-6 Pract-7 Pract-8 Práctica final

Simulacro del primer examen Entrenamiento, primer_parcial... Primer examen parcial... Segundo examen parcial... Segundo parcial

Página 1 Correcta Se puntuó 1,00 sobre 1,00 Marcar pregunta

Pregunta 1 Estado Finalizado Comenzado lunes, 5 de mayo de 2025, 13:07 Completado lunes, 5 de mayo de 2025, 13:17 Duración 9 minutos 41 segundos Puntos 9,00 de 12,00 Clasificación 7,50 de 10,00 (75%)

La eficiencia de los algoritmos voraces se basa en el hecho de que ...

Selección una:

- a. ... con antelación, las posibles decisiones se ordenan de mejor a peor.
- b. No contexto (equivalente a no marcar nada).
- c. ... antes de tomar una decisión se comprueba si satisface las restricciones del problema.
- d. ... las decisiones tomadas nunca se reconsideran. ✓

Pregunta 2 Correcta Se puntuó 1,00 sobre 1,00 Marcar pregunta

En la solución al problema de la mochila continua ¿por qué es conveniente la ordenación previa de los objetos?

Selección una:

- a. Para reducir la complejidad temporal en la toma de cada decisión: $O(n) \times O(1)$, donde n es el número de objetos a considerar. ✓
- b. Para reducir la complejidad temporal en la toma de cada decisión: $O(n^2) \times O(n \log n)$, donde n es el número de objetos a considerar.
- c. Póngase si no se hace es posible garantizar que la toma de decisiones siga un criterio voraz.
- d. No contexto (equivalente a no marcar nada).

Pregunta 3 Incorrecta Se puntuó 0,00 sobre 1,00 Marcar pregunta

¿Cómo se veía afectada la solución voraz al problema de la asignación de tareas en el caso de que se incorporaran restricciones que contemplen que ciertas tareas no pueden ser adjudicadas a ciertos trabajadores?

Selección una:

- a. No contexto (equivalente a no marcar nada). ✗
- b. La solución factible ya no estaría garantizada, es decir, podría ser que el algoritmo no llegue a una solución alguna.
- c. Ya no se garantizaría la solución óptima pero sí una factible.
- d. Habría que replantearse el criterio de selección para comenzar por aquellos trabajadores con más restricciones en cuanto a las tareas que no pueden realizar para asegurar, al menos, una solución factible.

Pregunta 4 Correcta Se puntuó 1,00 sobre 1,00 Marcar pregunta

Se pretende aplicar la técnica memoización a la siguiente función recursiva:

```
int f1(int x, int y) {
    if (x > y) return 1;
    return x + f1(x,y-2);
}
```

En el caso más desfavorable, ¿qué complejidad temporal y espacial cabe esperar de la función resultante?

Selección una:

- a. Temporal $O(x - y)$ y espacial $O(x)$
- b. $O(y - x)$, tanto temporal como espacial. ✓
- c. Ninguna de las otras dos opciones es correcta.
- d. No contexto (equivalente a no marcar nada).

Pregunta 5 Correcta Se puntuó 1,00 sobre 1,00 Marcar pregunta

La mejora que en general aporta la programación dinámica frente a la solución ingenua se consigue gracias al hecho de que ...

Selección una:

- a. No contexto (equivalente a no marcar nada).
- b. El número de veces que se resuelven los subproblemas no tiene nada que ver con la eficiencia de los problemas resueltos mediante programación dinámica.
- c. ... en la solución ingenua se resuelve pocas veces un número relativamente grande de subproblemas distintos.
- d. ... en la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos.

Pregunta 6 Correcta Se puntuó 1,00 sobre 1,00 Marcar pregunta

La solución óptima al problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado ...

Selección una:

- a. ... puede construir un único árbol que va creciendo o bien construir un bosque de árboles que al final se injetan en un único árbol ✓
- b. ... se construye haciendo crecer un único árbol.
- c. No contexto (equivalente a no marcar nada).
- d. ... se construye haciendo crecer varios árboles que al final acaban injetados en un único árbol.

Pregunta 7 Incorrecta Se puntuó 0,00 sobre 1,00 Marcar pregunta

Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
unsigned f1(unsigned x, unsigned v1) {
    if (v1 == 0)
        unsigned n = 0;
    else
        for ( unsigned k = 0; k < x; k++)
            n = max( n, v1) + f1( x-k, v1-1 );
    return n;
}
```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

Selección una:

- a. $O(x^2)$
- b. $O(1)$
- c. $O(x)$
- d. No contexto (equivalente a no marcar nada). ✗

Pregunta 8 Correcta Se puntuó 1,00 sobre 1,00 Marcar pregunta

¿Cuál de estos tres problemas de optimización no tiene, o no se lo conoce, una solución voraz óptima?

Selección una:

- a. El problema de cortar un tubo de forma que se obtenga el máximo beneficio posible.
- b. No contexto (equivalente a no marcar nada).
- c. El problema de la mochila discreta o sin fraccionamiento. ✓
- d. El problema de la mochila continua o con fraccionamiento.

Pregunta 9 Correcta Se puntuó 1,00 sobre 1,00 Marcar pregunta

De los problemas siguientes, indicad cuál no se puede tratar eficientemente como los otros dos:

Selección una:

- a. El problema de cortar un tubo de forma que se obtenga el máximo beneficio posible.
- b. El problema de la mochila sin fraccionamiento y sin restricciones en cuanto al dominio de los pesos de los objetos y de sus valores. ✓
- c. No contexto (equivalente a no marcar nada).
- d. El problema del cambio, o sea, el de encontrar la manera de entregar una cantidad de dinero usando el mínimo de monedas posibles.

Pregunta 10 Correcta Se puntuó 1,00 sobre 1,00 Marcar pregunta

Cuando se calculan los coeficientes binomiales usando la recursión $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$ con $\binom{n}{0} = \binom{n}{n} = 1$, qué problema se da y cómo se puede resolver?

Selección una:

- a. La recursión puede ser infinita y por tanto es necesario organizarla según el esquema iterativo de programación dinámica.
- b. Se repiten muchos cálculos y ello se puede evitar haciendo uso de una estrategia voraz.
- c. No contexto (equivalente a no marcar nada).
- d. Se repiten muchos cálculos y ello se puede evitar usando programación dinámica. ✓

Pregunta 11 Correcta Se puntuó 1,00 sobre 1,00 Marcar pregunta

¿Qué mecanismo se usa para acelerar el algoritmo de Prim?

Selección una:

- a. Mantener una lista de los arcos ordenados según su peso.
- b. El TAO "Union-find"
- c. No contexto (equivalente a no marcar nada).
- d. Mantener para cada vértice el vértice origen de la arista más corta hasta él. ✓

Pregunta 12 Incorrecta Se puntuó 0,00 sobre 1,00 Marcar pregunta

Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
int f1(int x, int y) {
    if (x < y) return 1;
    return x + f1(x-1,y);
}
```

¿Cuál es la mejor complejidad temporal que se puede conseguir?

Selección una:

- a. No contexto (equivalente a no marcar nada). ✗
- b. $O(1)$
- c. $O(x - y)$
- d. $O(x)$

9.5. Preguntas de otros años

26. Para resolver la versión general del problema del laberinto mediante vuelta atrás, utilizamos el estimador que aparece en el listado. ¿Sería como cota optimista para cualquier nodo del árbol de búsqueda?

```
using Maze = vector<vector<bool>>;
using Point = tuple<size_t, size_t>;
size_t estimator(const Maze &maze, const Point &p) {
    return max(maze.size()) - get<0>(p), maze[0].size() - get<1>(p)) - 1;
}
```

- (a) No, puesto que no sería una cota optimista válida para todos los nodos.
 (b) Sí, el estimador por si solo ya representaría una buena cota optimista.
 ✓(c) Sí, pero para mejorar la poda habría que sumarle la longitud del camino que representa cada nodo.

27. Las siguientes afirmaciones se refieren a un nodo cualquiera del árbol de búsqueda de ramificación y poda en un problema de **minimización**. De ellas, o bien dos son ciertas y una es falsa, o bien una es cierta y dos son falsas. Marca la que es diferente a las otras dos.

- (a) El nodo no es prometedor si su cota pesimista es menor que su cota optimista. F
 (b) El nodo no es prometedor si su cota pesimista es mayor que su cota optimista. C
 ✓(c) El nodo no es prometedor si su cota optimista es mayor que la mejor cota pesimista encontrada hasta el momento.

28. Se pretende resolver la versión general del problema del laberinto. ¿Resultaría útil disponer de un almacén de resultados parciales obtenido mediante la técnica de programación dinámica iterativa aplicándola a la versión restringida del mismo problema?

- (a) Las otras dos opciones son ambas ciertas.
 (b) Probablemente sí, llenando un almacén hacia delante desde la casilla de entrada al laberinto.
 (c) Probablemente sí, llenando un almacén hacia atrás comenzando en la casilla de salida del laberinto.

29. ¿Cuál de las siguientes formulaciones expresa mejor la complejidad temporal, en función del parámetro n , de la siguiente función? (asumimos que n es potencia exacta de 2)

```
int f(int n) {
    int k=0;
    for (int i = 2; i <= n; i*=2)
        for (int j=n; j >0; j-=2)
            k++;
    return k;
}
```

- (a) $\sum_{p=2}^{n/2} \frac{p-1}{2}$
 (b) $\sum_{p=1}^{\log n} 2^{p-1}$
 ✓(c) $\sum_{p=1}^{\log n} \frac{n}{2}$

ryp sería peor pq tiene q mantener la cola de prioridad
 Como hay q pasar por todos los combis se puebla aplicar pero no es lo optimo

35. ¿Cuál de las siguientes afirmaciones es cierta sobre la complejidad temporal, en el caso peor, de los algoritmos de vuelta atrás y ramificación y poda aplicados a la versión general del problema de la mochila?

- ✓(a) Ambos algoritmos tienen la misma complejidad temporal.
 (b) El algoritmo de vuelta atrás tiene una complejidad temporal peor que el de ramificación y poda.
 ✓(c) El algoritmo de ramificación y poda tiene una complejidad temporal peor que el de vuelta atrás.

36. Una empresa de transportes dispone de M vehículos para repartir N paquetes, todos al mismo destino. Cada paquete i tiene un peso P_i y se tiene que entregar antes de que transcurra un tiempo T_P . Por otro lado, cada vehículo j puede transportar una carga máxima C_j , tarda un tiempo T_V para llegar al destino y consume una cantidad L_j de litros de combustible, independientemente de la carga que transporta. Imaginad un algoritmo de vuelta atrás que obtenga la manera en que se tienen que transportar los objetos (en qué vehículo j tiene que ir cada objeto i) para que el consumo sea el mínimo. ¿Cuál sería una buena cota pesimista?

- (a) La solución voraz del problema de cargar cada paquete en el camión de menor consumo donde cada paquete llega a tiempo, sin tener en cuenta si el camión se sobrecarga o no.
 ✓(b) La solución voraz del problema de cargar cada paquete en el camión de menor consumo donde cada paquete llega a tiempo, sin sobrecargar ningún camión.
 (c) Ninguna de las otras dos opciones es correcta.

37. Las siguientes afirmaciones se refieren a un nodo cualquiera del árbol de búsqueda de ramificación y poda en un problema de **maximización**. De ellas, o bien dos son ciertas y una es falsa, o bien una es cierta y dos son falsas. Marca la que es diferente a las otras dos.

- ✓(a) Si su cota pesimista es menor que el mejor valor encontrado hasta el momento, entonces el nodo no podrá mejorar dicho valor. F
 (b) Si su cota optimista es menor que el mejor valor encontrado hasta el momento, entonces el nodo no podrá mejorar dicho valor. V
 (c) Su cota optimista es un valor que podría llegar a alcanzarse. V

38. Debemos calcular A^n , donde A una matriz de $m \times m$ elementos. ¿Existe una solución que evite realizar $n-1$ multiplicaciones de coste m^3 ?

- (a) No, la única manera de calcular A^n es iterativamente, $A^n = AA^{n-1}$ y se deben realizar $n-1$ multiplicaciones. tempo
 ✓(b) Sí, existe una solución más eficiente del estilo "divide y vencerás", que usa un espacio $\Theta(m^3 \log n)$.
 (c) Sí, existe una solución más eficiente del estilo "divide y vencerás", que usa un espacio $\Theta(m^2 n)$.

39. Sean dos funciones de coste $f : \mathbb{N} \rightarrow \mathbb{R}^+$ y $g : \mathbb{N} \rightarrow \mathbb{R}^+$ tales que $f \in O(g)$. De las siguientes situaciones, o bien dos pueden suceder y la otra no, o bien dos no pueden suceder y la otra sí. Indica la que es diferente de las otras dos.

- (a) $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ V
 (b) $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k$, $k \in \mathbb{R}^+$, $k \neq 0$ V
 ✓(c) $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ P → esto solo ocurre si f > g

40. Una de las siguientes afirmaciones es cierta. Di cuál.

- ✓(a) $\Theta(2^n) \neq \Theta(3^n)$
 (b) $O(2^n) \subset O(n^2)$ P las exponentiales para ser equivalentes
 ✓(c) $\Omega(2^n) \subset \Omega(n!)$ P se encuentran la misma base

30. Se pretende resolver la versión general del problema del laberinto mediante un algoritmo de búsqueda y enumeración. Para reducir los nodos visitados, utilizamos una matriz de enteros donde se almacena la longitud del mejor camino encontrado hasta el momento que llega a cada casilla del laberinto. ¿de qué poda se trata?

- (a) De una poda basada en la cota optimista.
 (b) De una poda basada en la cota pesimista.
 ✓(c) Ninguna de las otras dos opciones es cierta.

31. Se pretende resolver la versión general del problema del laberinto mediante un algoritmo de búsqueda y enumeración. Para agilizar la búsqueda, hemos pensado utilizar una matriz en la que cada casilla (i, j) se inicializa con la solución de la versión restringida del problema asumiendo que el origen del laberinto es (i, j) . ¿Qué utilidad puede tener esta matriz?

- (a) Ninguna, en todo caso habrá que inicializar cada casilla (i, j) con la solución de la versión restringida del problema asumiendo que el origen del laberinto es $(0, 0)$ y el destino (i, j) .
 (b) A priori poca, ya que cada vez que un nodo visita una casilla (i, j) habrá que actualizar la matriz, lo que perjudicaría la eficiencia del algoritmo resultante.
 ✓(c) Puede resultar eficaz para ajustar la mejor cota pesimista encontrada hasta el momento y además, es imprescindible actualizar la matriz durante todo el proceso de búsqueda.

32. Con la técnica de programación dinámica aplicada a la versión restringida del problema del laberinto...

- (a) ...no solo puede obtenerse el camino de salida más corto, también puede obtenerse, sin incrementar la complejidad temporal, el camino más corto desde el origen hasta cualquier casilla accesible del laberinto.
 (b) ...puede obtenerse el camino más corto desde cualquier casilla accesible hasta la salida si es que existe.
 ✓(c) Las otras dos opciones son ambas ciertas.

33. Dada la versión restringida del problema del laberinto, se puede modificar la solución de programación dinámica con complejidad espacial reducida para obtener también un camino de salida de longitud mínima?

- (a) No.
 ✓(b) Sí, pero a costa de aumentar la complejidad espacial del almacén.
 (c) Sí, añadiendo un vector de movimientos.

34. El problema de la moneda consiste en formar una suma M con el número mínimo de monedas tomadas (con repetición) de un conjunto donde hay una cantidad suficientemente grande de monedas de cada posible valor facial. Asumiremos que los contendores de monedas están previamente ordenados por valor facial. ¿Cuál de estas afirmaciones sobre el algoritmo voraz obvio es falsa?

- (a) Dependiendo de cuáles sean los valores faciales y la suma, puede ser que el algoritmo voraz no encuentre ninguna solución. V
 (b) Dependiendo de cuáles sean los valores faciales y la suma, puede ser que el algoritmo voraz no encuentre la solución que menos monedas usa. V
 ✓(c) Tiene una complejidad temporal $\Theta(M)$. F

Nombre:
 DNI / NIE:

Análisis y Diseño de Algoritmos
 Examen final C3 - 16 de junio de 2023

Instrucciones:

- Antes de comenzar el examen poned vuestros datos en el cuadernillo de preguntas y también en la hoja de respuestas.
- No olvidéis poner la modalidad en la hoja de respuestas.
- Dejad encima de la mesa vuestro DNI o vuestra TIU.
- No podéis consultar ningún material ni hablar con nadie.
- Cada pregunta solo tiene una opción correcta, marcadla como se indica en la hoja de respuestas.
- Cada respuesta incorrecta resta la mitad de una correcta. Las preguntas sin contestar ni suman ni restan puntos.
- Tenéis 90 min. para hacer el examen.

Normativa: *(Reglamento para la evaluación de los aprendizajes, 27-11-2015)*

- Está prohibido acceder al aula del examen con cualquier tipo de dispositivo electrónico.
- Además de dos bolígrafos o lápices, del documento de identificación personal y del material suministrado por el profesorado, **no** se permite tener ningún objeto o documento, ni en la mesa ni en sus inmediaciones.
- Si tienes dudas acerca de un objeto o dispositivo concreto pregunta al profesor antes de que comience la prueba.
- El incumplimiento de esta normativa puede conllevar, entre otras, la expulsión del aula del examen sin posibilidad de realizar la prueba.

Algunas preguntas hacen referencia al problema con el que hemos trabajado en las sesiones de prácticas. Su enunciado común es el siguiente:

Problema del laberinto: Se dispone de una cuadrícula $n \times m$ de valores $\{0, 1\}$ que representa un laberinto. Un valor 0 indica que la posición es inaccesible; por el contrario, con el valor 1 se simbolizan las casillas accesibles. Se pretende conocer la longitud del camino de salida más corto o un camino de salida con esa longitud (o ambas cosas).

Se define *longitud del camino* como el número de casillas que lo componen. Se define *camino de salida* como un camino que partiendo del origen del laberinto $(0, 0)$, conduce a la salida ($n-1, m-1$). De este problema hemos distinguido dos versiones según los movimientos permitidos:

- **versión restringida:** Asumimos solo tres movimientos posibles: derecha; diagonal y abajo, siempre que la casilla de destino sea una posición válida.
- **versión general:** Desde una celda se puede acceder a cualquier de sus ocho colindantes siempre que la casilla destino que sea una posición válida.

Preguntas:

1. En un algoritmo de búsqueda y enumeración, ¿qué podemos decir acerca de la heurística que se utiliza para determinar si un nodo debe expandirse o no?

- (a) Que puede equivocarse, por eso se le llama heurística.
- (b) Que también puede usarse como estrategia de búsqueda.
- (c) Las otras dos opciones son ambas ciertas.

2. ¿Cuál es el coste de monticularizar (*heapify*) un vector de tamaño N ?

- (a) $O(N \log N)$ y $\Omega(N)$
- (b) $\Theta(N)$
- (c) $O(N)$ y $\Omega(1)$

3. Con los valores numéricos almacenados en un fichero, queremos construir un *heap* (montículo). ¿cuál es la forma más eficiente de proceder?

- (a) Almacenar esos valores en un vector y después, reorganizar sus elementos para que estén dispuestos en forma de *heap*.
- (b) Almacenar esos valores directamente en un *heap* que inicialmente está vacío y va creciendo desde cada uno de los valores insertados.
- (c) Ambas formas de proceder son equivalentes en cuanto a eficiencia.

4. En cuanto a la posibilidad de aplicar la técnica de programación dinámica iterativa para resolver un problema:

- (a) Se debe conocer de antemano todos los posibles subproblemas y además, se debe disponer de una ordenación entre todos ellos según tamaño.
- (b) No necesariamente ha de conocerse de antemano todos los posibles subproblemas pero sí debe saberse datos de los cualesquiero, cuál es más pequeño.
- (c) Se debe conocer de antemano todos los posibles subproblemas pero no necesariamente se debe disponer de una ordenación entre todos ellos según tamaño.

5. ¿Cuál es la complejidad temporal en función de n , del siguiente fragmento:

```
for( int i = 0; i < n; i++ ) {
    A[i] = 0;
    for( int j = 0; j < 20; j++ )
        A[i] += B[j];
}
```

- (a) $\Theta(n \log n)$
- (b) $\Theta(n^2)$
- (c) $\Theta(n)$

6. Sea el vector $v = \{1, 3, 2, 7, 4, 6, 8\}$ cuyos elementos están dispuestos formando un montículo de mínimos. Posteriormente añadimos en la última posición del vector un elemento nuevo con valor 5. ¿Qué operación hay que hacer para que el vector siga representando un montículo de mínimos?

- (a) Intercambiar el 7 con el 5.
- (b) Intercambiar el 8 con el 5.
- (c) No hay que hacer nada pues el vector $v = \{1, 3, 2, 7, 4, 6, 8, 5\}$ también es un montículo de mínimos.

12. Queremos aplicar la técnica de memoización a la siguiente función recursiva:

```
double f( double x ) {
    if( x <= 2 )
        return x;
    return f(sqrt(x-1)) + f(sqrt(x-2));
}
```

¿Cuál sería un buen candidato para el almacén?

(La función `sqrt()` obtiene la raíz cuadrada; x_{Max} es el valor de x en la primera llamada.)

- (a) `vector<vector<double>> M(xMax+1, vector<double>(xMax+1))`
- (b) `vector<double> M(xMax+1)`
- (c) Ninguna de las otras dos opciones es válida.

13. Estamos resolviendo la versión restringida del problema del laberinto. Ya hemos obtenido el almacén de resultados parciales. ¿Con qué complejidad temporal podemos obtener un camino de salida de longitud mínima si es que existe?

- (a) $\Theta(m + n)$
- (b) $\Omega(\min(m, n))$ y $O(mn)$
- (c) $\Theta(mn)$

14. ¿Cuál es la característica principal de un algoritmo voraz aplicado al problema del laberinto?

- (a) Explora exhaustivamente todas las posibles soluciones del laberinto.
- (b) Toma decisiones locales óptimas en cada paso sin considerar el panorama general.
- (c) Utiliza una estrategia basada en estimaciones heurísticas para encontrar la solución óptima.

15. Se pretende resolver el problema del viajante de comercio (*travelling salesman problem*) mediante el esquema de vuelta atrás. ¿Cuál de los siguientes valores se espera que se comporte mejor como cota optimista para un nodo?

- (a) La suma de los pesos de las k aristas restantes más cortas, donde k es el número de ciudades que quedan por visitar.
- (b) El valor que se obtiene de multiplicar k por el peso de la arista más corta de entre las restantes, donde k es el número de ciudades que quedan por visitar.
- (c) La suma de los pesos de las aristas que completan la solución paso a paso visitando el vértice más cercano al último visitado.

16. Se pretende resolver la versión general del problema del laberinto mediante ramificación y poda. Para obtener la cota optimista de un nodo cualquiera procedemos así: calculamos el número de casillas que quedan, por la diagonal derecha-abajo, hasta llegar a una pared del laberinto. A ese valor le sumamos el número de casillas que podrían quedar desde esa pared hasta la salida utilizando únicamente movimientos del tipo abajo o derecha, según corresponda. ¿Qué podemos decir acerca de la cota optimista que se obtendría?

- (a) Que no cumple la condición para ser una cota optimista correcta.
- (b) Que no sería eficiente con respecto a otras cotas optimistas, ya que su cálculo requiere una complejidad temporal que no es constante.
- (c) Que es una cota optimista correcta y además puede obtenerse con complejidad temporal constante.

7. ¿Cuál es la complejidad, en función de n , del siguiente fragmento:

(suponed que \tilde{A} está definido como `vector<int> A(n)` y `sort()` es la función de ordenación de la librería estándar de C++, que tiene la mejor complejidad, temporal y espacial, posible para un algoritmo de ordenación de propósito general.)

```
std::sort(begin(A), end(A));
int acc = 0;
for( auto i : A )
    acc += i;
```

- (a) $\Theta(n \log n)$
- (b) $\Theta(n^2)$
- (c) $\Theta(n)$

8. Dada la versión restringida del problema del laberinto, ¿cuál de las estrategias siguientes proveeira una cota optimista para ramificación y poda?

- (a) Suponer que en adelante todas las casillas del laberinto son accesibles.
- (b) Las otras dos estrategias son ambas válidas.
- (c) Suponer que ya no se van a realizar más movimientos.

9. Para resolver la versión general del problema de la mochila con n objetos y carga máxima W , hemos escrito un algoritmo de divide y vencerás que, sucesivamente, divide el problema en dos subproblemas; cada uno de ellos toma la mitad de los objetos y la mitad de la carga máxima de la mochila. El caso base ocurre cuando solo hay un objeto que se añade a la solución si cabe en la fracción de carga máxima que corresponde a ese subproblema, y si no cabe se descarta. Asumiendo que n y W son potencias exactas de 2, ¿qué podemos decir de esta solución?

- (a) Que con los resultados de los subproblemas no siempre se puede componer la solución del problema original.
- (b) Que no cumple el teorema de reducción.
- (c) Que, aunque con los resultados de los subproblemas se puede componer la solución del problema original, esta formulación no mejora la solución estudiada en clase.

10. Dada la versión general del problema del laberinto, tratamos de completar un nodo cualquiera del árbol de búsqueda con el camino que, en caso de existir, obtendríamos asumiendo solo los tres movimientos de la versión restringida. ¿Qué obtendríamos en el caso de completar el nodo?

- (a) Una cota optimista para ese nodo.
- (b) Un nodo hoja del subárbol generado por aquel nodo.
- (c) Nada de interés, puesto que el camino resultante no contemplaría todos los movimientos permitidos.

11. Con respecto a la técnica *poda con memoria*, solo una de las siguientes afirmaciones es cierta. ¿Cuál es?

- (a) No resulta eficaz para resolver la versión general del problema del laberinto mediante ramificación y poda.
- (b) No se puede aplicar para resolver la versión restringida del problema del laberinto.
- (c) En una solución de vuelta atrás para la versión general del problema del laberinto, sirve también para descartar ciclos.

17. ¿Cómo se utiliza la cola de prioridad en el algoritmo de ramificación y poda para resolver el problema del laberinto?

- (a) Se agregan los nodos a la cola de prioridad en orden aleatorio para explorar todas las posibilidades.
- (b) Los nodos se agregan a la cola de prioridad según una estimación heurística para priorizar los caminos más prometedores.
- (c) Los nodos se extraen de la cola de prioridad según se vayan obteniendo.

18. Un fontanero tiene una jornada de Q cuartos de hora (es así como se organiza la agenda) y tiene C clientes. El trabajo del cliente i tarda q_i cuartos de hora y el fontanero le cobra un precio p_i . Es posible que no pueda atender todos los clientes en la jornada, que nunca pue de alargar. Este problema tiene una solución bien conocida que permite elegir qué clientes visitar para que la suma cobrada al final de la jornada sea la máxima. ¿Qué podemos decir de esta solución?

- (a) Que se ha de implementar forzosamente con un algoritmo de búsqueda y enumeración como el de vuelta atrás.
- (b) Que la organización de la agenda en cuartos de hora permite obtener una solución de complejidad temporal $\Theta(QC)$ y complejidad espacial $\Theta(Q)$.
- (c) Que no se puede implementar con una solución de "divide y vencerás" con memoización.

19. Dada la solución *naive* de la versión restringida del problema del laberinto, en general, ¿cuántas veces se llega al caso base de la recursión?

- (a) Solo una.
- (b) Un valor que puede crecer exponencialmente con el tamaño del laberinto.
- (c) Un valor que a lo sumo es $n \cdot m$.

20. Con respecto a los algoritmos estudiados durante el curso que encuentran el árbol de recubrimiento de mínimo coste, de las afirmaciones siguientes, o bien dos son verdaderas y una es falsa, o bien dos son falsas y una es verdadera. Marca la que (en este sentido) es diferente de las otras dos.

- (a) El algoritmo de Kruskal va construyendo un bosque de áboles que va uniendo hasta que acaba con un árbol de recubrimiento de coste mínimo.
- (b) El algoritmo de Prim se puede acelerar notablemente si los vértices se organizan en una estructura *union-find*.
- (c) La complejidad temporal del algoritmo de Prim es cúbica con respecto al número de vértices del grafo.

21. Se dispone de un conjunto de n valores numéricos dispuestos en forma de montículo y se desea obtener el valor de la suma de todos los que al menos tienen un hijo (es decir, no son nodos hoja). ¿Cuál es la complejidad temporal del mejor algoritmo que se puede escribir?

- (a) $O(n)$
- (b) $O(\log n)$
- (c) $O(n \log n)$

22. Dada la versión restringida del problema del laberinto, ¿se podría modificar la solución *naive* de divide y vencerás para que obtenga el número total de caminos diferentes, de cualquier longitud, que conducen a la salida desde el origen?

- (a) No se puede, ya que no se trataría de un problema de optimización.
- (b) No se puede, puesto que trata de un nuevo problema que no cumple las condiciones de aplicación de divide y vencerás.
- (c) Sí, pero puesto que se trata de un nuevo problema, hay que cambiar la recurrencia matemática inicial.

23. ¿Qué inconveniente presenta la siguiente función?

```
void maze(const Maze &maze, vector<vector<bool>> &visited, Point &currentPoint,
         size_t currentLength, size_t &currentBestLength) {
    if (currentPoint == maze.exit()) {
        currentBestLength = min(currentBestLength, currentLength);
        return;
    }

    for (auto step : allSteps) {
        Point next = currentPoint.next(step);
        if (maze.is_valid(next) && !visited[next.x()][next.y()] &&
            currentLength < currentBestLength) {
            visited[next.x()][next.y()] = true;
            maze(maze, visited, next, currentLength + 1, bestLength);
            visited[next.x()][next.y()] = false;
        }
    }
    return;
}
```

- (a) Que no detecta todos los subproblemas repetidos que pueden aparecer.
- (b) Que la poda basada en la mejor solución hasta el momento se puede mejorar fácilmente.
- (c) Las otras dos opciones son ambas ciertas.

24. Una empresa de transportes dispone de M vehículos para repartir N paquetes, todos al mismo destino. Cada paquete i tiene un peso P_i y se tiene que entregar antes de que transcurra un tiempo T_{ki} . Por otro lado, cada vehículo j puede transportar una carga máxima C_j , tarda un tiempo T_{kj} para llegar al destino y consume una cantidad L_j de litros de combustible, independientemente de la carga que transporta. Imaginad un algoritmo de vuelta atrás que obtenga la manera en que se tienen que transportar los objetos (en qué vehículo j tiene que ir cada objeto i) para que el consumo sea el mínimo. ¿Cuál sería una buena cota optimista?

- (a) La solución voraz del problema de cargar cada paquete en el camión de menor consumo donde cada paquete llega a tiempo, sin tener en cuenta si el camión se sobrecarga o no.
- (b) La solución voraz del problema de cargar cada paquete en el camión de menor consumo, sin sobrecargarlo, sin tener en cuenta si el paquete llega a tiempo o no.
- (c) Ambas son cotas optimistas válidas.

25. De las siguientes expresiones, o bien dos son ciertas y una es falsa, o bien al contrario, una es cierta y dos son falsas. Marca la que en este sentido es diferente a las otras dos.

- (a) $\sum_{i=1}^{\frac{n}{2}} \sum_{j=1}^i 2^j \in O(n \log n)$
- (b) $\sum_{i=1}^n \sum_{j=1}^{\log i} 2^j \in O(n^2)$
- (c) $\sum_{i=1}^{\log n} \sum_{j=1}^n 2^j \in O(n \log n)$

26. ¿Cuál de los siguientes enfoques es más eficiente para resolver la versión restringida del problema del laberinto?

- (a) Enfoque voraz.
- (b) Enfoque de programación dinámica.
- (c) Enfoque de vuelta atrás.

27. Indica cuál es la complejidad temporal en función de n , donde A es un vector de enteros y k es una constante que no depende de n , del fragmento siguiente:

```
for( int i = k; i < n - k; i++ ){
    A[i] = 0;
    for( int j = i - k; j < i + k; j++ )
        A[i] += B[j];
}
(a) Θ(k)
(b) Θ(n2)
(c) Θ(n)
```

28. Dada la versión restringida del problema del laberinto, estamos interesados en obtener, para cada casilla accesible del laberinto, el camino más corto entre el origen y esa casilla. ¿Qué esquema es el más apropiado en este caso?

- (a) La versión iterativa de programación dinámica.
- (b) Memoización.
- (c) Ambas técnicas resultan equivalentes.

29. Se pretende resolver la versión general del problema del laberinto mediante ramificación y poda y para ello se usa una estrategia que consiste en priorizar las expansiones de los nodos que contienen un camino explorado más corto. ¿Qué podemos decir del algoritmo resultante?

- (a) Que la primera hoja a la que se llegue es la solución del problema y por lo tanto, ya no será necesario explorar más nodos de la lista de nodos vivos, aunque no esté vacía.
- (b) Que el recorrido en el árbol de búsqueda será equivalente a un recorrido por niveles, por lo que no es necesario utilizar una cola de prioridad.
- (c) Las otras dos opciones son ambas ciertas.

30. ¿Qué obtenemos con la siguiente declaración de C++: priority_queue<nodo> pq; ?

- (a) Un heap o montículo de máximos.
- (b) Un heap o montículo de mínimos.
- (c) Un heap o montículo sin orden establecido ya que no se ha definido la función de comparación.

31. Se dispone de un conjunto de n valores numéricos dispuestos en un vector sin orden preestablecido. Se desea escribir una función que reciba ese vector y un valor k ($n/2 \leq k \leq n$) y que devuelva los k valores más pequeños dispuestos en otro vector de manera ordenada. ¿Cuál es la complejidad temporal del mejor algoritmo que se puede escribir?

- (a) $O(kn)$
- (b) Ninguna de las otras dos opciones es cierta.
- (c) $O(k \log n)$

32. Dada la versión general del problema del laberinto ¿Qué ocurre si la cota optimista de un nodo resulta ser el valor que se obtiene de una solución factible pero que no es la mejor en el subárbol generado por ese nodo?

- (a) Nada especial, las cotas optimistas se corresponden con soluciones factibles que no tienen por qué ser las mejores.
- (b) Que el algoritmo sería incorrecto pues podría descartarse el nodo que conduce a la solución óptima.
- (c) Que el algoritmo sería más lento pues se explorarían más nodos de los necesarios.

33. ¿Qué hace la siguiente función?

```
void f( vector<int>&A ) {
    priority_queue<int> pq;
    for( auto i : A )
        pq.push(A[i]);
    A.clear();
    while( !pq.empty() ) {
        A.push_back(pq.top());
        pq.pop();
    }
}
```

- (a) Invierte el vector A (el último elemento quedará el primero).
- (b) Nada, deja el vector A como estaba.
- (c) Ordena el vector A.

34. Dada la versión restringida del problema del laberinto, si solo se desea conocer la longitud del camino de salida más corto, ¿cuál es la mejor complejidad temporal y espacial que se puede conseguir si se aplica programación dinámica?

- (a) Temporal $\Theta(nm)$ y espacial $\Theta(\min\{n,m\})$
- (b) Ninguna de las otras dos opciones es cierta.
- (c) Temporal $\Theta(nm)$ y espacial $\Theta(nm)$

35. Dados dos nodos cualesquiera del árbol de búsqueda de ramificación y poda, en general, ¿se puede saber con certeza cuál está más cerca de la solución óptima del problema a resolver?

- (a) Sí, el que tiene mejor cota optimista.
- (b) Sí, el que tiene mejor cota pesimista.
- (c) Sí, pero solo si ambos nodos son hoja.

36. Se pretende resolver la versión general del problema del laberinto mediante un algoritmo de búsqueda y enumeración. Para reducir el número de nodos explorados, ¿qué mecanismo de los relacionados garantiza encontrar la solución y resulta a priori más eficaz?

- (a) Usar una matriz de booleanos para descartar caminos que llegan a una casilla ya visitada. Cada casilla (i, j) de la matriz se inicializa con el valor *false*.
- (b) Usar una matriz de enteros donde se almacena la longitud del mejor camino encontrado hasta el momento que llega a cada casilla del laberinto. Cada casilla (i, j) de la matriz se inicializa con el valor infinito.
- (c) Usar una matriz de enteros donde se almacena la longitud del mejor camino encontrado hasta el momento que llega a cada casilla del laberinto. Cada casilla (i, j) de la matriz se inicializa con la solución de programación dinámica para la versión restringida del problema asumiendo que el destino es (i, j) .

37. Dada la versión general del problema del laberinto, se pretende resolver mediante vuelta atrás haciendo uso de un mecanismo de poda basado en la mejor solución encontrada hasta el momento. ¿Cuándo se pondría un nodo?

- (a) Cuando el valor de su cota optimista supere al valor de la mejor cota pesimista encontrada hasta el momento.
- (b) Cuando el valor de su cota pesimista supere al valor de su cota optimista.
- (c) Cuando el valor de su cota optimista supere al valor de su cota pesimista.

38. La distancia de Manhattan (d) entre dos puntos (x_1, y_1) y (x_2, y_2) viene dada por la expresión $d = |x_1 - x_2| + |y_1 - y_2|$. ¿Podría utilizarse como cota optimista para resolver la versión general de problema del laberinto?

- (a) No, puesto que incumple la condición para ser una cota optimista correcta.
- (b) Sí, pero hay otros estimadores que se suelen comportar mejor ante este problema.
- (c) Sí, y además tiene la ventaja de que se puede calcular con complejidad temporal constante.

39. ¿De qué clase de complejidad es la solución de la siguiente relación de recurrencia? $f(n) = n(n-1) + f(n-1) \quad \text{si } n > 0$
 $f(0) = 1 \quad \text{si } n = 0$

- (a) $f(n) \in \Theta(n^2)$
- (b) $f(n) \in \Theta(n^3)$
- (c) Ninguna de las otras dos opciones es cierta.

40. El problema de la moneda consiste a formar una suma M con el número mínimo de monedas tomadas (con repetición) de un conjunto C donde hay una cantidad suficientemente grande de monedas con cada posible valor facial $C = \{c_1, c_2, \dots, c_{|C|}\}$, con $c_1 = 1$. ¿Cuál de estas afirmaciones sobre un algoritmo recursivo de la forma

$$n_{\text{opt}}(M) = 1 + \min_{1 \leq i \leq |C|} n_{\text{opt}}(M - c_i); \quad n_{\text{opt}}(0) = 0; \quad n_{\text{opt}}(x) = \infty \text{ para } x < 0$$

es falsa?

- (a) Dependiendo de cuáles sean los valores faciales y la suma, puede ser que el algoritmo recursivo no encuentre solución.
- (b) Tiene un coste temporal prohibitivo, ya que puede calcular $n_{\text{opt}}(x)$ para el mismo valor de x más de una vez.
- (c) Encuentra siempre la solución óptima.

Nombre:

DNI / NIE:

Análisis y Diseño de Algoritmos
Examen final C3 - 15 de junio de 2022

Instrucciones:

- ANTES DE COMENZAR EL EXAMEN poned vuestros datos en el cuadernillo de preguntas y también en la hoja de respuestas.
- NO OLVIDES poner la modalidad en la hoja de respuestas.
- DEJAD encima de la mesa vuestro DNI o vuestra TIU.
- No podéis consultar ningún material ni hablar con nadie.
- Cada pregunta solo tiene una opción correcta, marcadla como se indica en la hoja de respuestas.
- Cada respuesta incorrecta resta la mitad de una correcta. Las preguntas sin contestar ni suman ni restan puntos.
- Tenéis 90 min. para hacer el examen.

Normativa: (Reglamento para la evaluación de los aprendizajes, 27-11-2015)

- Está prohibido acceder al aula del examen con cualquier tipo de dispositivo electrónico.
- Además de dos bolígrafos o lápices, del documento de identificación personal y del material suministrado por el profesorado, NO se permite tener ningún objeto o documento, ni en la mesa ni en sus inmediaciones.
- Si tienes dudas acerca de un objeto o dispositivo concreto pregunta al profesor antes de que comience la prueba.
- El incumplimiento de esta normativa puede conllevar, entre otras, la expulsión del aula del examen sin posibilidad de realizar la prueba.

Algunas preguntas hacen referencia al problema con el que hemos trabajado en las sesiones de prácticas. Su enunciado común es el siguiente:

Encaminamiento óptimo: Se deben situar m puertas de acceso en una red de n servidores. Cada servidor i tiene una capacidad de almacenamiento $c_i \geq 0$ y d_{i,p_j} es la distancia de i a la puerta de enlace más cercana p_j . El tráfico total de la red es $\sum_{i=1}^n c_i d_{i,p_j}$, y debe minimizarse disponiendo de manera óptima las m puertas.

De este problema hemos distinguido dos versiones según cómo están interconectados los nodos:

- **versión general:** Los nodos y sus interconexiones constituyen un grafo.
- **versión simplificada:** Los nodos se conectan entre sí mediante un único bus bidireccional.

Modalidad 1

8. Indica cuál de los siguientes conjuntos es el conjunto $O(f)$:

- $\{g : \mathbb{N} \rightarrow \mathbb{R}^+ | \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c f(n)\}$
- $\{g : \mathbb{N} \rightarrow \mathbb{R}^+ | \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0, g(n) \geq c f(n)\}$
- $\{g : \mathbb{N} \rightarrow \mathbb{R}^+ | \exists c, d > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0, c f(n) \leq g(n) \leq d f(n)\}$

9. Se pretende resolver el problema del encaminamiento óptimo (versión general) mediante vuelta atrás. La solución se expresa mediante un vector x de n valores booleanos en el cual x_i indica si se coloca una puerta en el i -ésimo servidor. Dado un nodo del árbol de búsqueda que se ha completado hasta la posición k del vector, ¿cuál de los siguientes fragmentos de código obtiene una cota optimista correcta y, además, resulta ser la más rápida, en su cálculo, de entre las posibles alternativas válidas? (Fíjate únicamente en el recorrido de los bucles, el resto es idéntico en las tres opciones. Descarta posibles opciones que no calculan una cota optimista correcta)

(a)

```
double optimistic_bound=0;
for (int i=0; i<k; i++){
    double min_d=0;
    if (!x[i]){
        min_d= numeric_limits<double>::max();
        for (int j=0; j<n; j++)
            if (x[j] && d[i][j] < min_d)
                min_d= d[i][j];
    }
    optimistic_bound+= c[i] * min_d;
}
return optimistic_bound;
```

(b)

```
double optimistic_bound=0;
for (int i=0; i<k; i++){
    double min_d=0;
    if (!x[i]){
        min_d= numeric_limits<double>::max();
        for (int j=0; j<n; j++)
            if (x[j] && d[i][j] < min_d)
                min_d= d[i][j];
    }
    optimistic_bound+= c[i] * min_d;
}
return optimistic_bound;
```

(c)

```
double optimistic_bound=0;
for (int i=0; i<n; i++){
    double min_d=0;
    if (!x[i]){
        min_d= numeric_limits<double>::max();
        for (int j=0; j<n; j++)
            if (x[j] && d[i][j] < min_d)
                min_d= d[i][j];
    }
    optimistic_bound+= c[i] * min_d;
}
return optimistic_bound;
```

10. El elemento n -ésimo de la serie tribonacci, $T(n)$, se define como sigue: $T(n) = T(n-3) + T(n-2) + T(n-1)$ para $n \geq 3$; $T(0) = 0$; $T(1) = 1$ y $T(2) = 1$. ¿Cuál de estas afirmaciones es falsa?

- Una implementación ingenua de la función $T(n)$, la cual llamaría a $T(n-1)$, $T(n-2)$ y $T(n-3)$ tendría una complejidad prohibitiva por la repetición de cálculos que se produciría.
- Es posible una implementación de programación dinámica iterativa con complejidad $\Theta(n)$.
- El problema no tiene una solución de programación dinámica iterativa pero se puede de resolver añadiendo memoización al cálculo recursivo ingenuo en el que el cálculo de $T(n)$ comporta realizar las llamadas a $T(n-1)$, $T(n-2)$ y $T(n-3)$.

11. A partir del problema del encaminamiento óptimo (versión simplificada), modificamos la expresión que obtiene el tráfico estimado de la red; ahora es $\sum_{i=0}^n c_i (d_i - d_{p_i})^2$, d_i es la distancia de i al primer servidor del bus; p_i es la puerta de acceso más cercana a i). Disponemos de la nueva función $ogw(k, n)$ que obtiene el mínimo tráfico de la subred colocando solo una puerta de enlace en el lugar apropiado entre el servidor k y el $n-1$ (ambos incluidos). Estamos interesados en una función $met(m, n)$ que calcule el mínimo tráfico de la red colocando, en las mejores ubicaciones, m puertas de enlace. Sabiendo que $met(1, n) = ogw(0, n)$, ¿cómo se podría calcular ahora, de forma recursiva, $met(m, n)$, con $1 \leq m \leq n^2$?
- De la misma manera que en el problema original que se ha trabajado en las sesiones de prácticas.
 - No sucede pues, con la modificación planteada, no se cumple la propiedad de subestructura óptima.
 - Como la distancia está elevada al cuadrado, esta función no se puede calcular recursivamente.

12. Sobre el teorema de reducción:

- Que se cumpla es condición necesaria para poder aplicar divide y vencerás.
- Las otras dos opciones son ambas falsas.
- Que se cumpla es condición necesaria para poder aplicar programación dinámica.

13. De las siguientes situaciones, o bien dos son posibles y una no lo es, o bien al contrario, solo una es posible y las otras dos no lo son. Marca la que, en este sentido, es diferente a las demás.

- $f(n) \in O(n)$ y $f(n) \in \Omega(1)$.
- $f(n) \in O(n)$ y $f(n) \in \Omega(n^2)$.
- $f(n) \in O(n)$ y $f(n) \in O(n^2)$.

14. ¿Puede ocurrir que la solución recursiva de estilo "divide y vencerás" pero con memoización de un problema resuelva menos subproblemas que la mejor solución iterativa posible de programación dinámica?

- No, nunca.
- Sí, porque la mejor solución iterativa posible de programación dinámica puede resolver subproblemas que no sean necesarios al resolver subproblemas posteriores.
- Sí, porque no existe garantía de que la mejor solución iterativa posible no resuelva problemas repetidos, mientras que la técnica de memoización lo garantiza directamente mediante el uso de un almacén.

15. En la estrategia de ramificación y poda se usa una cola de prioridad para decidir en qué orden se expanden los nodos. Imaginemos un problema de optimización. ¿Puede ser que el valor por el cual se ordenan los nodos sea una cota pesimista del nodo?
- Sí.
 - No, porque para podar necesitamos una cota optimista.
 - No, porque una cota pesimista es típicamente el valor que se encuentra en una de las hojas que cuelga del nodo.

16. Sobre la propiedad de *subestructura óptima* de un problema de optimización (por selección discreta):
- Es condición necesaria para poder aplicar divide y vencerás.
 - Es condición necesaria para poder aplicar programación dinámica.
 - Las otras dos opciones son ambas ciertas.

17. En cuanto a la complejidad temporal de la siguiente función, ¿cuál de las siguientes formulaciones expresa mejor su complejidad en el peor de los casos?

```
int f(vector<int>& v) {
    int n=v.size(), i=2, k=0;
    while (i<n) {
        int j=i;
        while (v[j] != v[1]) {
            k++;
            j=j/2;
        }
        i=i+2;
    }
    return k;
}
```

- $c_s(n) = \sum_{k=1}^{\lfloor \frac{n-1}{2} \rfloor} \log_2 2k \in O(n \log n)$
- $c_s(n) = \sum_{k=1}^{\lfloor \frac{n-1}{2} \rfloor} \log_2 2k \in O(\log^2 n)$
- Las otras dos opciones son ambas falsas.

18. La costa de un país tiene n núcleos de población, todos ellos unidos por una línea costera de tren. La industria de cada núcleo de población j produce T_j toneladas de productos para la exportación y se encuentra en el kilómetro k_j de la línea de tren. Las exportaciones son básicas para su economía y debe realizarlas por mar ya que ha roto relaciones con los países con los que linda por tierra. El gobierno ha decidido promover el transporte marítimo y ha presupuestado la cantidad necesaria para construir p puertos de manera que se minimice el tráfico por la línea de tren. El tráfico es $\sum_{j=1}^n T_j |k_j - k_{s(j)}|$ donde $s(j)$ es el puerto de salida más cercano al núcleo j . ¿Es posible resolver el problema mediante una técnica de programación dinámica?

- No. Debe resolverse usando una técnica de búsqueda y enumeración (vuelta atrás, ramificación y poda) ya que el problema no tiene subestructura óptima.
- Sí. El problema goza de subestructura óptima: podemos resolver el problema asumiendo que conocemos la solución para las m primeras ciudades y para $p-1$ puertos, determinar la posición óptima para que el puerto p sirva a las $n-m$ ciudades restantes, y buscar el valor óptimo de m .
- No, pero el problema tiene una solución voraz exacta que consiste en empezar por asignar puerto a todos los núcleos de población e ir quitando uno a uno los puertos de manera que el tráfico que resulte de quitarlos aumente lo mínimo posible.

19. ¿Cuál de las siguientes formulaciones expresa mejor la complejidad temporal, en función del parámetro n , de la siguiente función? (asumimos que n es potencia exacta de 2)

```
int f(int n) {
    int k=0;
    for (int i = 2; i <= n; i*=2)
        for (int j=i; j>0; j-=2)
            k++;
    return k;
}
```

- $\sum_{p=2}^{n/2} \frac{p-1}{2}$
- $\sum_{p=1}^{\log n} 2^{p-1}$
- $\sum_{p=1}^{\log n} 2 \cdot (p-1)$

20. ¿Cuál es el coste temporal de crear un montículo a partir de un vector no ordenado?

- $\Theta(n)$
- $\Theta(n \log n)$
- $\Omega(n \log n)$ y $O(n^2)$.

21. Existen dos algoritmos que para ordenar un vector de n elementos, buscan el máximo de esos n elementos, lo intercambian con el n -ésimo elemento para ponerlo al final, y luego ordenan, usando el mismo algoritmo, el vector de las primeras $n-1$ componentes. ¿Cuál de las afirmaciones siguientes es cierta?

- Uno de los algoritmos es *heapsort* y el otro es una de las posibles maneras de realizar la ordenación por selección; el primero tiene un coste temporal $O(n \log n)$ y el segundo, $O(n^2)$.
- Uno de los algoritmos es *heapsort* y el otro es una de las posibles maneras de realizar la ordenación por burbuja o *bubblesort*; el primero tiene un coste temporal $O(n \log n)$ y el segundo, $O(n^2)$.
- Uno de los algoritmos es *heapsort* y el otro es una de las posibles maneras de realizar la ordenación por selección; el primero tiene un coste temporal $O(n)$ y el segundo, $O(n^2)$.

22. Se pretende resolver la versión general del problema del encaminamiento óptimo mediante la técnica "divide y vencerás". ¿Se podría obtener la mejor disposición de las pueras?

- No, ya que no se cumple la propiedad "subestructura óptima".
- No, ya que no cumple el teorema de reducción.
- Sí, pero a costa de una complejidad temporal prohibitiva.

23. ¿Cuál es la complejidad temporal en función del tamaño del problema (n) de multiplicar dos matrices cuadradas?

- $O(n^2)$
- $O(n^{3/2})$
- $O(n^3)$

24. Se pretende resolver el problema del encaminamiento óptimo mediante ramificación y poda (versión general) y para ello se implementa la lista de nodos vivos con una pila. ¿Qué podemos decir del algoritmo resultante frente a una solución de vuelta atrás?

- Que para mejorar la solución de vuelta atrás será necesario hacer uso de las cotas pesimistas de los nodos intermedios.
- Que será más eficiente que la solución de vuelta atrás, tanto en tiempo como en espacio requerido.
- Que ambos, el de vuelta atrás y el de ramificación y poda, estarán utilizando la misma estrategia de búsqueda.

25. Se quiere resolver el problema del encaminamiento óptimo (versión simplificada) mediante un algoritmo voraz que sigue la siguiente estrategia:

- suponer que no hay pueras de acceso.
 - buscar cual es el servidor (que aún no es puerta de acceso) que al hacerlo puerta de acceso produce tráfico menor.
 - marcar el servidor encontrado como puerta de acceso.
 - repetir los dos últimos pasos hasta que se tenga m pueras de acceso.
- El algoritmo no devuelve en general la solución óptima del problema.
 - El algoritmo no es voraz.
 - El algoritmo siempre devuelve la solución óptima del problema.

26. Para un problema concreto, ¿puede ser que la técnica de ramificación y poda pade más que la de vuelta atrás pero al final sea más lenta?

- Sí, puede pasar incluso con implementaciones muy eficientes.
- No, la que poda más es siempre más rápida.
- No, a menos que la implementación sea muy ineficiente.

27. Se quiere resolver el problema del encaminamiento óptimo (versión simplificada) mediante un algoritmo voraz que sigue la siguiente estrategia:

- marcar todos los servidores como pueras de acceso (por tanto, el tráfico será cero).
 - buscar de qué nodo tenemos que quitar la puerta de acceso para que el aumento de tráfico sea mínimo.
 - quitar la puerta de acceso encontrada.
 - repetir los dos últimos pasos hasta que se tenga m pueras de acceso.
- El algoritmo no es voraz.
 - El algoritmo siempre devuelve la solución óptima del problema.
 - El algoritmo no devuelve en general la solución óptima del problema.

28. Una de las tres afirmaciones siguientes sobre los algoritmos que obtienen el árbol de recubrimiento mínimo de un grafo ponderado no dirige es cierta. ¿Cuál es?

- El algoritmo de Kruskal va ampliando un único árbol de recubrimiento mínimo.
- El algoritmo de Prim va ampliando un único árbol de recubrimiento mínimo.
- El algoritmo de Prim se puede acelerar usando una estructura de datos de conjuntos disjuntos con las operaciones *union* y *find*.

29. Se pretende resolver la versión general del problema del encaminamiento óptimo mediante la técnica de programación dinámica. ¿Con qué inconveniente nos encontraríamos?

- Las otras dos opciones son ambas verdaderas.
- Que el programa resultante solo sería capaz de resolver los casos sencillos.
- No se cumple la propiedad de subestructura óptima.

30. Dado el problema del encaminamiento óptimo (versión simplificada), la siguiente función pretende encontrar la mejor ubicación de las pueras de enlace haciendo uso del almacén M . ¿Es correcta?

Nota: la función `pair<int, double>ogw(...)` devuelve, respectivamente, la mejor ubicación de una puerta de enlace entre dos nodos; y el tráfico asociado a esa ubicación de la puerta de enlace)

```
double met_naive(vector<int>& M, const vector<double>& d,
                  const vector<double>& c, int n, int m)
{
    double best_traffic;
    int best_gw;

    if( m == 1 ) {
        tie(best_gw, best_traffic) = ogw( d, c, 0, n );
    }
    else {
        best_traffic = numeric_limits<double>::max();
        for( int node = m ; node < n ; node++ ) {
            auto [candidate_gw, traffic] = ogw( d, c, node, n );
            traffic += met_naive( M, d, c, node, m-1 );
            if( traffic < best_traffic ) {
                best_traffic = traffic;
                best_gw = candidate_gw;
            }
        }
    }

    M[n] = best_gw;
    return best_traffic;
}
```

- Sí, aunque solo resolverá ejemplos sencillos.
- No, ya que no resuelve todos los subproblemas necesarios.
- No, puesto que no será posible extraer la mejor ubicación de las pueras a partir del almacén M .

31. Un ladrón entra por la noche en la quesería más prestigiosa del mercado central con una larga mochila cilíndrica que tiene exactamente el diámetro de los quesos (todos los quesos son cilindros del mismo diámetro y altura, siguiendo un nuevo estándar de la UE) en la que puede cargar exactamente un metro de quesos y con una sofisticada sierra radial quesera (con baterías) que le permite cortar un queso horizontalmente en lugar de hacer cuñas, de manera que se lleve un cilindro. Cada queso tiene un precio único, que no se repite en la tienda. Si quiere llevar queso por el máximo importe posible. Indica cuál de las siguientes afirmaciones sobre la carga óptima es falsa.

- Lleva la mochila llena hasta arriba y como mucho ha usado la sierra radial para cortar un queso.
- Los primeros quesos que ha cargado son, enteros, los quesos más caros de toda la quesería.
- Lleva la mochila llena hasta arriba y ha usado la sierra radial más de una vez para llevarse porciones bien calculadas de los quesos más caros.

32. Dado el problema del encaminamiento óptimo (versión general), el valor que se obtiene con el método voraz es ...
- ... una cota superior para el valor óptimo, pero que nunca coincide con este.
 - ... una cota inferior para el valor óptimo.
 - ... una cota superior para el valor óptimo que a veces coincide con este.
33. El problema del cambio es el de formar una suma M con el número mínimo de monedas tomadas (con repetición) de un conjunto C en el que el valor facial de la moneda de tipo i es c_i . ¿Cuál de las siguientes afirmaciones es falsa?
- Una versión (memoizada) de la siguiente recursión da la solución óptima en caso de que exista: $n(M) = 1 + \min_{1 \leq i \leq |C|} n(M - c_i)$; $n(0) = 0$; $n(x) = \infty$ para $x < 0$
 - La solución voraz consistente en coger siempre la moneda de valor facial más grande de cuyo valor es menor que la cantidad M así: $n(M) = 1 + n(M - c^*)$ donde $c^* = \max\{c \in C | c \leq M\}$ puede no encontrar solución para cualquier M y C .
 - La solución voraz consistente en coger siempre la moneda de valor facial más grande de cuyo valor es menor que la cantidad M así: $n(M) = 1 + n(M - c^*)$ donde $c^* = \max\{c \in C | c \leq M\}$ encuentra siempre la solución óptima para cualquier M y C si existe dicha solución.
34. Hemos resuelto el problema del encaminamiento óptimo (versión simplificada) mediante un algoritmo de programación dinámica cuya complejidad espacial ha resultado ser $O(n)$. ¿Qué podemos asegurar de esta solución?
- Que no es correcta si lo que se pretende es encontrar el valor del tráfico estimado para la mejor disposición de las puertas de enlace.
 - Que no podemos determinar dónde colocar las puertas de enlace.
 - Que no es la más eficiente si lo que se pretende es encontrar, utilizando las técnicas estudiadas durante el curso, el valor del tráfico estimado para la mejor disposición de las puertas de enlace.
35. Sea un problema de optimización por selección discreta, con restricciones, en el que se deben tomar n decisiones booleanas para optimizar un indicador, y se abordará mediante un método de búsqueda y enumeración (vuelta atrás, ramificación y poda). ¿Cuál de las siguientes afirmaciones es correcta?
- La complejidad temporal será como mucho $O(n \log n)$ porque en general basta con ordenar adecuadamente las decisiones para convertir cualquier problema de este tipo en un problema de complejidad temporal lineal.
 - Puede haber problemas para los que la complejidad será exponencial o peor; ninguna estrategia de poda puede garantizar que esto no va a ocurrir.
 - La complejidad temporal en el peor caso será $O(n^2)$ ya que se toman n decisiones binarias.

36. Se pretende resolver el problema del encaminamiento óptimo (versión general) mediante ramificación y poda. Si un nodo interno del árbol de búsqueda se completa colocando las restantes puertas de enlace en los servidores con más carga de entre los que quedan por analizar. ¿Qué obtenemos?
- las otras dos opciones son ambas falsas.
 - Una cota pesimista, es decir, una hoja del árbol de búsqueda situada más abajo que ese nodo interno, en la misma rama.
 - Una cota optimista, es decir, una hoja del árbol de búsqueda situada más abajo que ese nodo interno, en la misma rama.

37. Tenemos un "superprocesador" que tiene una instrucción que permite la ordenación de 100 elementos en un tiempo constante. Para este superprocesador, adaptamos el algoritmo Mergesort de forma que cada vez que queremos ordenar menos de 100 elementos, en lugar de hacer las llamadas recursivas, llama a esta instrucción. ¿Cuál sería la complejidad de este algoritmo?

- $O(n \log n)$
- $O(n)$
- $O(1)$

38. En cuanto a la complejidad temporal de la siguiente función, ¿qué podemos decir acerca del mejor de los casos?

```
int f(vector<int> &v) {
    int n=v.size(), i=2, k=0;
    while (i<n) {
        int j=i;
        while (v[j] != v[1]) {
            k++;
            j=j/2;
        }
        i=i+2;
    }
    return k;
}
```

- Que el mejor de los casos ocurre cuando el vector tiene 2 elementos o menos y la complejidad es $\Omega(1)$.
- Que uno de los mejores casos ocurre cuando $v[j] = v[1] \forall j \in \mathbb{N}$ y la complejidad es $\Omega(n)$.
- Las otras dos opciones son ambas falsas.

39. Se pretende calcular el valor 2^n , $n \in \mathbb{N}$, haciendo una transcripción literal de la expresión $2^n = 1 + \sum_{i=0}^{n-1} \prod_{j=1}^i 2$. ¿Cuál sería la complejidad temporal asintótica, en función de n , del algoritmo resultante?

- $O(2^n)$
- $O(n^2)$
- $O(n)$

40. Teniendo en cuenta que $ogw(k, n)$ obtiene el tráfico estimado si se coloca una sola puerta de enlace entre los nodos k y $n-1$, ¿cuál de las siguientes recurrencias es la más apropiada para resolver la versión simplificada del problema del encaminamiento óptimo mediante divide y vencerás?

- $\text{met}(m, n) = \begin{cases} \text{ogw}(0, n) & \text{si } m = 1 \\ \min_{k \in [m-1, n-1]} (\text{met}(m-1, k) + \text{ogw}(k, n)) & \text{si } m > 1 \end{cases}$
- $\text{met}(m, n) = \begin{cases} \text{ogw}(0, n) & \text{si } m = 1 \\ \min_{k \in [0, n-1]} (\text{met}(m-1, k) + \text{ogw}(k, n)) & \text{si } m > 1 \end{cases}$
- $\text{met}(m, n) = \begin{cases} \text{ogw}(0, n) & \text{si } m = 1 \\ \min_{k \in [0, m-1]} (\text{met}(m-1, k) + \text{ogw}(k, n)) & \text{si } m > 1 \end{cases}$

Área personal / Mis cursos / 24018_2020-21 / Examen_final_C3 / [castellano] Examen final C3 – Grupos Valenciano/J2ADE-ARA

| | |
|---------------|---------------------------------------|
| Comenzado el | miércoles, 16 de junio de 2021, 08:30 |
| Estado | Finalizado |
| Finalizado en | miércoles, 16 de junio de 2021, 09:26 |
| Tiempo | 55 minutos 32 segundos |
| empleado | |
| Puntos | 9,50/30,00 |
| Calificación | 3,17 de 10,00 (32%) |

Pregunta 1
Correcta
Puntuía 1,00 sobre 1,00

Qué diferencia (entre otras) hay entre el algoritmo de Prim y el de Kruskal?

- Seleccione una:
- a. Aún siendo el grafo de partida totalmente conexo, el algoritmo de Kruskal garantiza la solución óptima mientras que el de Prim sólo garantiza un subóptimo.
 - b. El subgrafo que pasa a paso va generando el algoritmo de Prim siempre contiene una única componente conexa mientras que el de Kruskal no tiene por qué.
 - c. No contesto (equivalente a no marcar nada).
 - d. El algoritmo de Prim es voraz y el de Kruskal no.

Pregunta 2
Correcta
Puntuía 1,00 sobre 1,00

Con respecto a la complejidad espacial de los algoritmos de ordenación Quicksort, Heapsort y Mergesort:

- Seleccione una:
- a. Las complejidad espacial de todos ellos es lineal con el tamaño del vector a ordenar.
 - b. Mergesort tiene complejidad espacial lineal con el tamaño del vector a ordenar, la de los otros dos es constante.
 - c. No contesto (equivalente a no marcar nada).
 - d. Mergesort y Heapsort tienen complejidad espacial lineal con el tamaño del vector a ordenar, la de Quicksort es constante.

Pregunta 3
Correcta
Puntuía 1,00 sobre 1,00

De las expresiones siguientes, o bien dos son verdaderas y una es falsa, o bien dos son falsas y una es verdadera. Marca la que (en este sentido) es diferente de las otras dos.

- Seleccione una:
- a. Si $f \in \Theta(g)$ entonces $O(f) = O(g)$.
 - b. No contesto (equivalente a no marcar nada).
 - c. Si $f \notin \Omega(g)$ entonces $O(f) = \Omega(g)$.
 - d. Si $f \in O(g)$ entonces $g \notin O(f)$.

Pregunta 4
Incorrecta
Puntuía -0,50 sobre 1,00

¿Qué hace la siguiente función?

```
void f(vector<int> &A) {
    priority_queue<int> pq;
    for (int i = A.size()-1; i >= 0; i--) {
        pq.push(A[i]);
    }
    A.clear();
    while (!pq.empty()) {
        A.push_back(pq.top());
        pq.pop();
    }
}
```

- a. No contesto (equivalente a no marcar nada).
- b. Invierte el vector A (el último elemento quedará el primero).
- c. Nada, deja el vector como estaba.
- d. Ordena el vector A.

Pregunta 5
Incorrecta
Puntuía -0,50 sobre 1,00

Una de las afirmaciones siguientes es cierta y las otras dos falsas. Indicad cuál es la cierta.

- Seleccione una:
- a. No contesto (equivalente a no marcar nada).
 - b. $O(2^n) \in O(n!)$
 - c. $O(n^n) \in O(n!)$
 - d. $O(3^n) \in O(2^n)$

Pregunta 6
Correcta
Puntuía 1,00 sobre 1,00

Si $f(n) \in O(g(n))$ ¿cuál de estas situaciones no es posible?

- Seleccione una:
- a. $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$
 - b. $f(n) \in \Omega(g(n))$
 - c. No contesto (equivalente a no marcar nada).
 - d. $g(n) \in O(f(n))$

Pregunta 7
Incorrecta
Puntuía -0,50 sobre 1,00

Si resolvemos un problema de optimización mediante el método de la vuelta atrás, ¿se puede usar, además de una cota optimista, una cota pesimista para reducir el número de soluciones exploradas?

- Seleccione una:
- a. Si. Aunque las cotas pesimistas no se hayan explicado en clase hasta llegar al tema de ramificación y poda, no hay ninguna razón que impida su uso en el método de la vuelta de atrás.
 - b. No, porque las podas las determina la cota optimista, y lo hace independientemente de cuál sea el valor de la mejor solución en curso.
 - c. No. En el método de la vuelta atrás siempre es necesario visitar las hojas para actualizar la mejor solución en curso.
 - d. No contesto (equivalente a no marcar nada).

Pregunta 8
Correcta
Puntuía 1,00 sobre 1,00

Un problema tiene subestructura óptima cuando....

- Seleccione una:
- a. ... es posible escribir una solución voraz para el problema.
 - b. ... se trata de un problema con complejidad inherentemente prohibitiva.
 - c. No contesto (equivalente a no marcar nada).
 - d. ... su solución se puede construir eficientemente a partir de soluciones de subproblemas suyos.

Pregunta 12
Incorrecta
Puntuía -0,50 sobre 1,00

¿De qué clase de complejidad es la solución de la siguiente relación de recurrencia?
$$\begin{array}{l} f(0) = 1 \\ f(n) = n(n-1) + f(n-1) \end{array}$$

- Seleccione una:
- a. $\Theta(n^3)$
 - b. $\Theta(n^4)$
 - c. $\Theta(n^2)$
 - d. No contesto (equivalente a no marcar nada).

Pregunta 13
Incorrecta
Puntuía -0,50 sobre 1,00

¿Tiene sentido usar una función que indique cómo de prometedor es un nodo cuando resolvemos un problema que no es de optimización mediante ramificación y poda?

- Seleccione una:
- a. Sí. Solo tiene sentido usar una función de promesa en problemas de optimización, y esta ha de ser necesariamente una cota optimista.
 - b. Los problemas que no son de optimización no se pueden resolver mediante ramificación y poda.
 - c. No contesto (equivalente a no marcar nada).
 - d. Sí, si se puede diseñar de manera que intente predecir si un nodo conducirá o no a la solución.

Pregunta 14
Correcta
Puntuía 1,00 sobre 1,00

Indica cuál es la complejidad, en función de n , del fragmento siguiente:

```
for( int i = 0; i < n; i++ ) {  
    A[i] = 0;  
    for( int j = 0; j < 2*n; j++ )  
        A[i] += B[j];  
}
```

- a. No contesto (equivalente a no marcar nada)
- b. $\Theta(n \log n)$
- c. $\Theta(n)$
- d. $\Theta(n^2)$

Pregunta 9
Incorrecta
Puntuía -0,50 sobre 1,00

El problema del alfarero (solución discreta con tiempos discretos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $m_i \in \mathbb{N}$. El valor de cada pieza terminada, $v_i \in \mathbb{N}$ y el tiempo necesario para su fabricación $t_i \in \mathbb{N}$, $i \in \{0, \dots, n-1\}$. ¿Cuántos objetos de cada clase hay que fabricar para maximizar la ganancia teniendo en cuenta que el tiempo total está limitado por $T \in \mathbb{N}$?

Si T no es muy grande con respecto a n ¿Cuál de los siguientes esquemas algorítmicos resultaría más eficiente para resolverlo?

- Seleccione una:
- a. Un algoritmo voraz.
 - b. Vuelta atrás.
 - c. Programación dinámica.
 - d. No contesto (equivalente a no marcar nada).

Pregunta 10
Correcta
Puntuía 1,00 sobre 1,00

En el esquema de vuelta atrás, los mecanismos de poda basados en la mejor solución hasta el momento...

- Seleccione una:
- a. ... garantizan que no se va a explorar todo el espacio de soluciones posibles.
 - b. Las otras dos opciones son ambas verdaderas.
 - c. ... pueden eliminar nodos que representan posibles soluciones factibles.
 - d. No contesto (equivalente a no marcar nada).

Pregunta 11
Correcta
Puntuía 1,00 sobre 1,00

El problema del alfarero (solución discreta con tiempos continuos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $m_i \in \mathbb{N}$. El valor de cada pieza terminada, $v_i \in \mathbb{N}$ y el tiempo necesario para su fabricación $t_i \in \mathbb{R}$, $i \in \{0, \dots, n-1\}$. ¿Cuántos objetos de cada clase hay que fabricar para maximizar la ganancia teniendo en cuenta que el tiempo total está limitado por $T \in \mathbb{R}$?

Se pretende resolver mediante un algoritmo de ramificación y poda. Para determinar si un nodo es prometedor se estima su ganancia máxima haciendo uso de la solución voraz discreta (sin fraccionamientos) de la parte aún sin completar. ¿Qué podemos decir del algoritmo resultante?

- Seleccione una:
- a. Que presumiblemente explorará menos nodos de los necesarios.
 - b. No contesto (equivalente a no marcar nada).
 - c. Que presumiblemente explorará más nodos de los necesarios.
 - d. Que si comienza con una solución subóptima encontrará antes la óptima.

Pregunta 15
Correcta
Puntuía 1,00 sobre 1,00

¿Cuál sería la complejidad temporal de la siguiente función tras aplicar programación dinámica?

```
double f(int n, int m) {  
    if (n <= 1) return 1;  
    return m * f(n-1, m);  
}
```

- Seleccione una:
- a. $\Theta(n \cdot \log m)$
 - b. No contesto (equivalente a no marcar nada).
 - c. $\Theta(n^2)$
 - d. $\Theta(n)$

Pregunta 16
Incorrecta
Puntuía -0,50 sobre 1,00

De las siguientes expresiones, o bien dos son verdaderas y una es falsa, o bien dos son falsas y una es verdadera. Marca la que (en este sentido) es distinta a las otras dos.

- Seleccione una:
- a. $\Theta(n + n \log_2 n)$
 - b. $\Theta(n^2)$
 - c. No contesto (equivalente a no marcar nada).
 - d. $\Theta(2^n)$

Pregunta 17
Correcta
Puntuía 1,00 sobre 1,00

¿De qué clase de complejidad es la solución de la siguiente relación de recurrencia?

```
\begin{array}{l} f(0) = 1 \\ f(n) = 1 + f(n/b) \end{array}
```

- Seleccione una:
- a. Depende del valor de b .
 - b. No contesto (equivalente a no marcar nada).
 - c. $\Theta(\log n)$
 - d. $\Theta(n)$

Pregunta 18

Correcta

Puntuía 1,00 sobre 1,00

En los algoritmos de ramificación y poda, ¿el valor de una cota pesimista es menor que el valor de una cota optimista? (se entiende que ambas cotas se aplican sobre el mismo nodo)

Seleccione una:

- a. Sí, siempre es así.
 - b. No contesto (equivalente a no marcar nada).
 - c. En general sí, si se trata de un problema de minimización, aunque en ocasiones ambos valores pueden coincidir.
 - d. En general sí, si se trata de un problema de maximización, aunque en ocasiones ambos valores pueden coincidir.
- ✓

Pregunta 19

Incorrecta

Puntuía -0,50 sobre 1,00

Con respecto al tamaño del problema, ¿Cuál es el orden de complejidad temporal asintótica de la siguiente función? (asumimos que $\|A\|$ es una matriz cuadrada)

```
void traspueta( vector < vector < int >> A) {
    for (int i = 1; i < A.size(); i++)
        for (int j = 0; j < i; j++)
            swap( A[i][j], A[j][i] );
}
```

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
 - b. cuadrático
 - c. lineal
 - d. constante
- ✗

Pregunta 20

Incorrecta

Puntuía -0,50 sobre 1,00

¿Cuál es la relación de recurrencia que representa la complejidad en el peor caso del algoritmo de búsqueda del k-ésimo elemento más pequeño de un vector (estudiado en clase).

Seleccione una:

- a. $T(n) = \left\lfloor \frac{n}{k} \right\rfloor + 1$
 - b. $T(n) = \left\lfloor \frac{n}{k} \right\rfloor + n$
 - c. No contesto (equivalente a no marcar nada).
 - d. $T(n) = \left\lfloor \frac{n}{k} \right\rfloor + 1$
- ✗

Pregunta 24

Correcta

Puntuía 1,00 sobre 1,00

Indica cuál es la complejidad en función de n , donde $|k|$ es una constante (no depende de n). del fragmento siguiente :

```
for ( int i = k; i < n - k; i++ ) {
    A[i] = 0;
    for( int j = i - k; j < i + k; j++ )
        A[i] += B[j];
}
```

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
 - b. $O(n \log n)$
 - c. $O(n)$
 - d. $O(n^2)$
- ✓

Pregunta 25

Incorrecta

Puntuía -0,50 sobre 1,00

El problema del alfarero (solución discreta con tiempos continuos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $\|m_i\|$. El valor de cada pieza terminada, $\|v_i\|$ y el tiempo necesario para su fabricación $\|t_i\|$. El tiempo disponible para la fabricación de objetos está limitado por $\|T\|$.

Se pretende resolver mediante ramificación y poda y para ello se hace uso de una cota que consiste en asumir que de las restantes clases de objetos aún no tratadas se va a fabricar exactamente una pieza. ¿Qué podemos decir de esta cota?

Seleccione una:

- a. Que es una cota optimista.
 - b. Que es una cota pesimista.
 - c. No contesto (equivalente a no marcar nada).
 - d. Que no es cota, ni optimista ni pesimista
- ✗

Pregunta 26

Incorrecta

Puntuía -0,50 sobre 1,00

¿A qué clase de complejidad pertenece la función $t(n)$ definida por la siguiente relación de recurrencia?

$$t(n) = \left\lfloor \frac{n}{2} \right\rfloor + 1 + \sum_{j=1}^{n-1} t(j)^3$$

Seleccione una:

- a. $\Theta(n^2)$
 - b. No contesto (equivalente a no marcar nada).
 - c. $\Theta(\frac{n^2 \log n}{\log 3})$
 - d. $\Theta(n)$
- ✗

Pregunta 21

Incorrecta

Puntuía 0,00 sobre 1,00

El problema del alfarero (solución discreta con tiempos discretos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $\|m_i\|$. El valor de cada pieza terminada, $\|v_i\|$ y el tiempo necesario para su fabricación $\|t_i\|$. El tiempo disponible para la fabricación de objetos está limitado por $\|T\|$.

Utilizando la técnica programación dinámica iterativa se pretende conocer únicamente la ganancia máxima que podría obtener el alfarero. ¿Cuál es la mejor complejidad espacial y temporal que se puede conseguir?

Seleccione una:

- a. Espacial $\Theta(n)$ y temporal $\Theta(n^2)$
- b. No contesto (equivalente a no marcar nada).
- c. Espacial $\Theta(n)$ y temporal $\Theta(n^2)$
- d. $\Theta(n^2)$, tanto espacial como temporal.

Pregunta 22

Correcta

Puntuía 1,00 sobre 1,00

En un algoritmo de optimización resuelto mediante ramificación y poda ¿Podría encontrarse la solución óptima sin haber alcanzado nunca un nodo hoja?

Seleccione una:

- a. No, los nodos hojas son los nodos completados y por lo tanto hay que visitar al menos uno de ellos para almacenarlo como la mejor solución hasta el momento.
 - b. Si, esto puede ocurrir incluso si no se hace uso de cotas pesimistas.
 - c. No contesto (equivalente a no marcar nada).
 - d. Si, pero esto solo podría ocurrir si se hace uso de cotas pesimistas.
- ✓

Pregunta 23

Correcta

Puntuía 1,00 sobre 1,00

Si $\lim_{n \rightarrow \infty} (g(n)/f(n))$ resulta ser una constante positiva no nula, cuál de las siguientes expresiones NO puede darse?

Seleccione una:

- a. $f(n) \in \Theta(g(n))$
- b. No contesto (equivalente a no marcar nada).
- c. $f(n) \in \Omega(g(n))$ y $g(n) \in \Omega(f(n))$
- d. $g(n) \in o(f(n))$

Pregunta 24

Correcta

Puntuía 1,00 sobre 1,00

Pregunta 27

Correcta

Puntuía 1,00 sobre 1,00

El problema del alfarero (solución discreta con tiempos continuos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $\|m_i\|$. El valor de cada pieza terminada, $\|v_i\|$ y el tiempo necesario para su fabricación $\|t_i\|$. El tiempo disponible para la fabricación de objetos está limitado por $\|T\|$.

Utilizando una técnica de divide y vencerás, ¿se podría saber cuál es la ganancia máxima que podría alcanzar el alfarero?

Seleccione una:

- a. Sí, pero a costa de una complejidad temporal prohibitiva.
 - b. No, ya que no se cumple la propiedad "subestructura óptima".
 - c. No contesto (equivalente a no marcar nada).
 - d. No, ya que no es posible descomponer el problema en subproblemas.
- ✓

Pregunta 28

Correcta

Puntuía 1,00 sobre 1,00

El valor que se obtiene con el método voraz para el problema de la mochila discreta es ...

Seleccione una:

- a. ... un valor inferior al valor óptimo que a veces puede ser igual a este.
- b. No contesto (equivalente a no marcar nada).
- c. ... un valor inferior al valor óptimo, pero que nunca coincide con este.
- d. ... un valor superior al valor óptimo.

Pregunta 29

Incorrecta

Puntuía -0,50 sobre 1,00

El problema del alfarero (solución discreta con tiempos continuos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $\|m_i\|$. El valor de cada pieza terminada, $\|v_i\|$ y el tiempo necesario para su fabricación $\|t_i\|$. El tiempo disponible para la fabricación de objetos está limitado por $\|T\|$.

Se pretende resolver mediante un algoritmo de ramificación y poda. ¿Qué ocurre si el subóptimo de partida coincide con la cota optimista del nodo inicial?

Seleccione una:

- a. Que el algoritmo sería incorrecto ya que el subóptimo de partida es en realidad una cota pesimista para el nodo inicial.
 - b. No contesto (equivalente a no marcar nada).
 - c. Que el algoritmo no debería explorar ningún nodo.
 - d. Que la cota optimista está mal estimada.
- ✗

Pregunta 30

Incorrecta

Puntuación -0,50 sobre 1,00

Indica cuál es la complejidad, en función de n , del fragmento siguiente:
(suponed que A está definido como vector $A(n)$ y sort es la función de ordenación de la STL)

```
sort(begin(A), end(A));  
int acc = 0;  
for( auto i : A )  
    acc += i;
```

- a. $\Theta(n)$
- b. No contesto (equivalente a no marcar nada)
- c. $\Theta(n^2)$
- d. $\Theta(n \log n)$



13/6/24, 18:50

Examen 2020 Corregido

Práctica final Julio

Ir a...

Examen 1
Invierno
Puntuación: 0,00
sobre 1,00
F' Marzo
entrega

Examen 2
Invierno
Puntuación: 0,00
sobre 1,00
F' Marzo
entrega

Examen 3
Invierno
Puntuación: 0,00
sobre 1,00
F' Marzo
entrega

La solución de programación dinámica heredada del problema de la mochila discreta.

Sección correcta:

- a. Se da la restricción de que los pesos tienen que ser enteros positivos.
- b. Calcula menor peso de valor de la mochila que la correspondiente solución de programación dinámica recursiva.
- c. Tiene menor complejidad de tiempo.
- d. Tiene un costo temporal más grande.

La respuesta correcta es: ... tiene la restricción de que los pesos tienen que ser enteros positivos.

Sección correcta:

- a. La respuesta correcta es: ... tiene la restricción de que los pesos tienen que ser enteros positivos.
- b. La optimiza. Una buena otra optimización sería más rápida que una buena otra posiblemente.
- c. No contesto (equivalente a no marcar nada).
- d. No hay distinción, se me remitió porque era lo que mejor se adaptaba a la solución del mío.

La respuesta correcta es: No hay distinción, la más remarcable siempre será la que mejor se adapte a la solución del mío.

about:blank

1/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

Examen 1
Invierno
Puntuación: 0,00
sobre 1,00
F' Marzo
entrega

Examen 2
Invierno
Puntuación: 0,00
sobre 1,00
F' Marzo
entrega

Examen 3
Invierno
Puntuación: 0,00
sobre 1,00
F' Marzo
entrega

Sé el problema "Camino de coste mínimo" con tres movimientos: Este, Sureste y Sur. Se pretende encontrar el camino más favorable con la menor complejidad temporal posible. ¿Cuál sería la más ajustada para realizar todo el proceso?

Selección una:

- a. $O(n \cdot m \cdot i \cdot n)$
- b. $O(n \cdot m \cdot i \cdot n \cdot m)$
- c. $O(n \cdot m \cdot i \cdot m)$
- d. No contesto (equivalente a no marcar nada).

La respuesta correcta es: $O(n \cdot m \cdot i \cdot n \cdot m)$

Sé el problema "Camino de coste mínimo" con tres movimientos: Este, Sureste y Sur. En cuanto a la complejidad espacial y temporal: ¿Cuál de los siguientes esquemas algorítmicos resulta más eficiente para resolverlo?

Selección una:

- a. Programación dinámica
- b. No contesto (equivalente a no marcar nada).
- c. Divide y vencida
- d. Ramificación y poda

La respuesta correcta es: Programación dinámica

about:blank

2/45

about:blank

3/45

Preguntas 5
Resuelto 0/0
Puntuación 0,00
Última modificación 13/6/24
0º Marcar pregunta

Sólo una de las tres afirmaciones siguientes es cierta. Cuál?

Seleccione una:

- a. Igualmente que el algoritmo de Prim va construyendo un único árbol de recubrimiento seleccionando vértices uno a uno, el algoritmo de Kruskal va construyendo un bosque que va uniendo añadiendo aristas, hasta obtener un único árbol de recubrimiento.
- b. Los algoritmos de Prim y de Kruskal mantienen en todo momento un único árbol de recubrimiento, que crece añadiendo aristas o vértices.
- c. Mientras que el algoritmo de Prim va construyendo un único árbol de recubrimiento seleccionando aristas una a una, el algoritmo de Kruskal va construyendo un bosque que va uniendo añadiendo vértices hasta obtener un único árbol de recubrimiento.
- d. Los dos son iguales.

La respuesta correcta es: Igualmente que el algoritmo de Prim va construyendo un único árbol de recubrimiento seleccionando vértices uno a uno, el algoritmo de Kruskal va construyendo un bosque que va uniendo añadiendo aristas, hasta obtener un único árbol de recubrimiento.

Preguntas 6
Resuelto 0/0
Puntuación 0,00
Última modificación 13/6/24
0º Marcar pregunta

Sa el vector $\mathbf{v}[0] = [6, 8, 6, 4, 3, 2, 2]$. Indica cuál de las siguientes opciones es cierta. (Se asume la notación del lenguaje C/C++ en la que el primer elemento del vector está en la posición 0, es decir, en $\mathbf{v}[0]$).

Seleccione una:

- a. El vector tiene un módulo máximo porque el elemento $v[1]=6$ debe ser "máximo" (desplazado hacia la izquierda).
- b. El vector tiene un módulo máximo porque el elemento $v[2]=4$ debe ser "máximo" (desplazado hacia la derecha).
- c. No contiene equivalencia a no marcar nada.
- d. El vector tiene un módulo mínimo.

La respuesta correcta es: El vector \mathbf{v} es un módulo máximo.

Preguntas 7
Resuelto 0/0
Puntuación 0,00
Última modificación 13/6/24
0º Marcar pregunta

¿Cuál es el costo temporal asintótico de la siguiente función?

```
Void f(int n, int arr[])
{
    int i;
    For( i = 1; i < n; i++)
        While( i < n && arr[i] < arr[i])
            i++;
}
```

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $O(n \log n)$
- c. $O(n)$
- d. $O(n^3)$

La respuesta correcta es: $O(n)$.

Preguntas 8
Resuelto 0/0
Puntuación 0,00
Última modificación 13/6/24
0º Marcar pregunta

De las siguientes expresiones, o bien dos son verdaderas y una es falsa, o bien dos son falsas y una es verdadera. Marca la que (en este sentido) es distinta a las otras dos.

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $O(4^{n/2}) < O(n) < O(2^n)$
- c. $O(4^{n/2}) < O(n) < O(4^n)$
- d. $O(n^2) < O(2^{n/2}) < O(2^n)$

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:5

Examen 2020 Corregido

La respuesta correcta es: $O(2^{n^2}) \subset O(n^2) \subset O(n!)$

Se el problema "Caminó de noche inferior" con tres movimientos: Baja, Subida y Suj. Utilizando la técnica ramificada y para ello pretende obtener la cifración del código más favorable desde la casilla destino. Para tratar de acelerar la búsqueda calculamos una cota para cada uno de los nodos visitados que coincide en recibir el mismo problema para aplicarle la técnica programación dinámica con la condición de que el camino debe parar por la casilla destinada al medio que acabamos de visitar. ¿Qué podemos decir de esa cota?

Selección una:

- a. Que es una cosa pesimista pero no optimista.
- b. Que es una cosa pesimista y optimista al mismo tiempo.
- c. Me comiendo (equivalente a no morder nada).
- d. Que en realidad no se trata de una cosa, ni pesimista ni optimista.

La respuesta correcta es: Que en realidad no se trata de una nota, ni pasimata ni agitadora.

0 Se quiere desarrollar un programa que comprueba si es posible que un caballo de ajedrez, mediante una secuencia de los movimientos permitidos, recorra todas las casillas de un tablero $N \times N$ a partir de una determinada casilla dada como entrada y sin repetir ninguna casilla. De entre las estrategias que se crean, ¿cuál será la eficiente para resolver el problema?

Selecione una

- a. Agobiado todo.
- b. Programador ordinaria.
- c. Te comiendo (explicacion a no mas de media).
- d. Vuelta aslo.

La respuesta correcta es: Vuelta atrás.

about:blank

8/45

about:blank

9/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:5

Examen 2020 Corregido

Ejercicio 11
Dado el siguiente código, analiza y responde las siguientes preguntas:

```
unsigned long pat2(); //signed int  
{  
    if (n>0)  
        return 1;  
    else  
        return 2 * pat2(n+1);  
}
```

```

unsigned long pot2_2(unsigned n)
{
    if (n <= 0)
        return 1;
    if ((n <= 2))
        return pot2_2(n)*pot2_2(n);
    else
        return 2*pot2_2(n-2)*pot2_2(n-2);
}

```

Selecione una:

- a. La primera es más eficiente que la segunda
- b. La segunda es más eficiente que la primera
- c. No contesto (equivalente a no marcar nada)

- Si Ambas son equivalentes en cuanto a eficiencia

La respuesta correcta es: Ambas son equivalentes en cuanto a eficiencia

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

Selección una:
 a. La división del problema en subproblemas.
 b. La combinación de las soluciones a los subproblemas.
 c. El número de fórmulas recursivas que se hacen.
 d. No contiene igualdades a ni menor ni más.

La respuesta correcta es: La división del problema en subproblemas

↓ Document continues below

Discover more from:

Análisis Y Diseño De

Algoritmos

34018



Universidad de Alicante

43 documents

Go to course



13/45

about:blank

12/45

about:blank

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

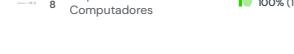
Examen 2020 Corregido

Preguntas 13
Puntaje: 0/10
Tiempo: 1000 ms
Último intento: 1/10

Sea el problema "Camino de coste mínimo" con tres movimientos: Este, Sureste y Sur. Utilizando la técnica programación dinámica

Preguntas 14
Puntaje: 0/10
Tiempo: 1000 ms
Último intento: 1/10

Selección una:
 a. Especifica $\Theta(n^2)$ / temporal $\Theta(n \cdot m)$.
 b. $\Theta(n \cdot m)$ tanto espacio como temporal.
 c. No contiene igualdades a ni menor ni más.
 d. Especifica $\Theta(\min(n, m))$ / temporal $\Theta(n \cdot m)$.

La respuesta correcta es: Especifica $\Theta(\min(n, m))$ / temporal $\Theta(n \cdot m)$ 

Selección una:
 a. Si $g(n) \in \Theta(1)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación MergeSort.
 b. No contiene igualdades a ni menor ni más.
 c. Si $g(n) \in \Theta(n^2)$ la relación de recurrencia representa la complejidad temporal del algoritmo de búsqueda por inserción.
 d. Si $g(n) \in \Theta(n)$ la relación de recurrencia representa el número de fórmulas recursivas que hace el algoritmo de ordenación MergeSort.

La respuesta correcta es: Si $g(n) \in \Theta(n^2)$ la relación de recurrencia representa la complejidad temporal del algoritmo de búsqueda por inserción.

about:blank

14/45

about:blank

15/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

Con respecto a la complejidad espacial de los algoritmos de ordenación Quicksort, Heapsort y Mergesort:

- Seleccione una:
- a. La complejidad espacial de todos ellos es lineal con el tamaño del vector a ordenar.
 - b. No contesto (equivalente a no marcar nada).
 - c. Mergesort tiene complejidad espacial lineal con el tamaño del vector a ordenar, la de los otros dos es constante.
 - d. Mergesort y Heapsort tienen complejidad espacial lineal con el tamaño del vector a ordenar; la de Quicksort es constante.

La respuesta correcta es: Mergesort tiene complejidad espacial lineal con el tamaño del vector a ordenar; la de los otros dos es constante.

- Seleccione una:
- a. La complejidad temporal de todos ellos es lineal con el tamaño del vector a ordenar.
 - b. No contesto (equivalente a no marcar nada).
 - c. Mergesort tiene complejidad temporal lineal con el tamaño del vector a ordenar, la de los otros dos es constante.
 - d. Mergesort y Heapsort tienen complejidad temporal lineal con el tamaño del vector a ordenar; la de Quicksort es constante.

La respuesta correcta es: Mergesort tiene complejidad temporal lineal con el tamaño del vector a ordenar; la de los otros dos es constante.

¿Cuál es la complejidad temporal de la siguiente función recursiva?

```
unsigned despedida (unsigned n) {
    if (n == 0)
        return 0;
    unsigned num = despedida (n/2) + despedida (n/2);
    for (unsigned i=1; i<n-1; i++)
        for (unsigned j=i; j<n; j++)
            num+=i+j;
    return num;
}
```

16/45

about:blank

17/45

about:blank

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

¿Cuál es el dominio de $f(x) \cdot g(x)$ si se considera $\lim_{x \rightarrow \infty} [f(x)/g(x)] = k > 0$?

- Seleccione una:
- a. $f(x) = O(g(x))$ y $g(x) = O(f(x))$
 - b. $O(g(x)) = O(f(x))$
 - c. No contesto (equivalente a no marcar nada).
 - d. $g(x) = O(f(x))$ pero $f(x) \neq O(g(x))$

La respuesta correcta es: $f(x) = O(g(x))$ y $g(x) = O(f(x))$ Se presentó el problema: "Cálculo de costa italiana con los instrumentos finos. Supongamos que el punto de partida en una costa italiana (L_0) del mar con número M ($1 \leq i \leq N \leq j \leq M$) se divide en la costa (S_{ij}). Utilizando la técnica directa y sencilla se pretende conocer la dificultad del camino más favorable".

¿Cuál es la respuesta correcta?

- Seleccione una:
- a. No contesto (equivalente a no marcar nada).
 - b. $m(p_i, m_j) = \begin{cases} M_{ij}[m] & \text{si } i < j \text{ y } m_i < j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} & \text{si } i < j \text{ y } m_i > j \\ m & \text{si } i = j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} + M_{ij}[m] & \text{si } i < j \text{ y } m_i = j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} - M_{ij}[m] & \text{si } i < j \text{ y } m_i < j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} & \text{si } i < j \text{ y } m_i > j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} + M_{ij}[m] & \text{si } i < j \text{ y } m_i = j \end{cases}$
 - c. $m(p_i, m_j) = \begin{cases} M_{ij}[m] & \text{si } i < j \text{ y } m_i < j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} & \text{si } i < j \text{ y } m_i > j \\ m & \text{si } i = j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} + M_{ij}[m] & \text{si } i < j \text{ y } m_i = j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} - M_{ij}[m] & \text{si } i < j \text{ y } m_i < j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} & \text{si } i < j \text{ y } m_i > j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} + M_{ij}[m] & \text{si } i < j \text{ y } m_i = j \end{cases}$

La respuesta correcta es: $m(p_i, m_j) = \begin{cases} M_{ij}[m] & \text{si } i < j \text{ y } m_i < j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} & \text{si } i < j \text{ y } m_i > j \\ m & \text{si } i = j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} + M_{ij}[m] & \text{si } i < j \text{ y } m_i = j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} - M_{ij}[m] & \text{si } i < j \text{ y } m_i < j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} & \text{si } i < j \text{ y } m_i > j \\ \min\{m_p, (i-1), m_{p+1}, m-1, m_{p+2}, (m-1), m_{p+3}, m-2, m_{p+4}\} + M_{ij}[m] & \text{si } i < j \text{ y } m_i = j \end{cases}$ Activar Windo
Ir a Configuració

about:blank

18/45

about:blank

19/45

13/6/24, 18:50

Examen 2020 Corregido

Se el problema "Camino de coste mínimo" con tres movimientos: Este, Sureste y Sur. Se pretende limitar todos los caminos posibles entre dos casillas cualesquiera. ¿Qué técnica algorítmica debemos utilizar?

- Selecciona una:
- a. Programación dinámica.
 - b. Backtracking.
 - c. Ramificación y poda.
 - d. No correcto (equivalente a no marcar nada).

La respuesta correcta es: backtracking.

Es función del parámetro n ¿cuál es la complejidad temporal de la siguiente función?

```
int f(int n) {
    int k;
    for (int i = 0; i < n; i >> 2) {
        for (int j = 0; j < i; j += 2)
            k++;
    }
    return k;
}
```

- Selecciona una:
- a. $\Theta(n \log(n))$
 - b. No correcto (equivalente a no marcar nada).
 - c. $\Theta(n^2)$
 - d. $\Theta(n)$

about:blank

13/6/24, 18:50

Examen 2020 Corregido

La respuesta correcta es: $\Theta(n)$

20/45

about:blank

21/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

about:blank

22/45

about:blank

23/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

about:blank

24/45

about:blank

25/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

about:blank

26/45

about:blank

27/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

about:blank

28/45

about:blank

29/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

about:blank

30/45

about:blank

31/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

about:blank

32/45

about:blank

33/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

about:blank

34/45

about:blank

35/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

about:blank

36/45

about:blank

37/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

about:blank

38/45

about:blank

39/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

about:blank

40/45

about:blank

41/45

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

More from:

[Análisis Y Diseño De Algoritmos](#)

34018

Universidad de Alicante

43 documents



[Go to course](#)

Examen 2020 Corregido

Ada Valenciano Parte 4

Analisis Y Diseño De Algoritmos

64

100% (1)

Ada Valenciano Parte 3

Analisis Y Diseño De Algoritmos

27

100% (1)

Ejercicio 2 práctica 4

Analisis Y Diseño De Algoritmos

1

67% (3)

ADA pr-7-es

Analisis Y Diseño De Algoritmos

4

None

about:blank

42/45

about:blank

43/45

More from:

ALBERCA 16 ❤ 999+

ibc

Universidad de Alicante

13/6/24, 18:50

Examen 2020 Corregido

13/6/24, 18:50

Examen 2020 Corregido

Discover more

- SOL Práctica 3 FC**
Fundamentos De Los Computadores 100% (1)
- TODAS las prácticas de EC en un pdf**
Estructura De Los Computadores 100% (1)
- Práctica 4 STI (memoria)**
Sistemas Y Tecnologías De Información 100% (1)
- SOLUCIÓN Sesión 2**
Fundamentos Fisicos De La Informatica 100% (1)

Recommended for you

- Parcial Fase III**
Arquitectura De Los Computadores 100% (2)
- Formulaario**
Arquitectura De Los Computadores 100% (1)
- Resumen Libro Investigación Comercial**
Investigación Comercial 100% (13)
- Apuntes, temas 1-10**
Investigación Comercial 100% (4)

about:blank

44/45

about:blank

45/45

[Área personal](#) / Mis cursos / [24018_2020-21](#) / [Examen_final_C4](#) / [castellano] Examen final C4 - Grupo "Tardes"

| | |
|---------------|--------------------------------------|
| Comenzado el | miércoles, 7 de julio de 2021, 08:44 |
| Estado | Finalizado |
| Finalizado en | miércoles, 7 de julio de 2021, 09:31 |
| Tiempo | 57 minutos 39 segundos |
| empleado | |
| Puntos | 11,50/30,00 |
| Calificación | 3,83 de 10,00 (38%) |

Pregunta 1
Incorrecta Puntuación 0,00 sobre 1,00

La serie denominada tribonacci se define de la siguiente manera: $T(0) = T(1) = 1$, $T(2) = 2$, i . $T(n) = T(n - 3) + T(n - 2) + T(n - 1)$ para $n > 3$. Sólo una de las afirmaciones siguientes es cierta. ¿Cuál es?

Seleccione una:

- a. Un algoritmo de programación dinámica iterativa para calcular $T(n)$ tendría un coste espacial $\Theta(n)$ y este coste no se podría reducir a $\Theta(1)$.
- b. No contesto (equivalente a no marcar nada).
- c. Un algoritmo de programación dinámica iterativa permite calcular el valor de $T(n)$ en tiempo $\Theta(n)$.
- d. Un algoritmo recursivo con memoización para calcular $T(n)$ para a un n grande tendría una complejidad prohibitiva.

Pregunta 2
Incorrecta Puntuación -0,50 sobre 1,00

Una de las respuestas siguientes es falsa. ¿Cuál es? El problema del viajante de comercio ...

Seleccione una:

- a. ... se puede resolver exactamente usando un algoritmo de programación dinámica.
- b. No contesto (equivalente a no marcar nada).
- c. ... se puede resolver exactamente usando un algoritmo voraz derivado del Kruskal.
- d. ... se puede resolver exactamente usando un algoritmo de búsqueda y enumeración como es el de vuelta atrás o el de ramificación y poda.

Pregunta 3
Correcta Puntuación 1,00 sobre 1,00

El problema del alfarero (solución continua con tiempos continuos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $m_i \in N$. El valor de cada pieza terminada, $v_i \in N$ y el tiempo necesario para su fabricación $t_i \in R$, $i \in [0..n - 1]$. ¿Cuántos objetos de cada clase hay que fabricar para maximizar la ganancia teniendo en cuenta que el tiempo total está limitado por $T \in R$?

Si el alfarero pudiera vender objetos sin terminar a un precio proporcional al estado de terminación. ¿Cuál de las siguientes estrategias sería más apropiada para resolverla?

Seleccione una:

- a. Un algoritmo voraz.
- b. Programación dinámica.
- c. Vuelta atrás.
- d. No contesto (equivalente a no marcar nada).

Pregunta 4
Correcta Puntuación 1,00 sobre 1,00

El problema del alfarero (solución discreta con tiempos discretos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $m_i \in N$. El valor de cada pieza terminada, $v_i \in N$ y el tiempo necesario para su fabricación $t_i \in N$, $i \in [0..n - 1]$. El tiempo total disponible viene dado por $T \in N$

Se pretende listar todas las posibilidades de fabricación de objetos. ¿Qué estrategia es la más adecuada?

Seleccione una:

- a. Un algoritmo voraz.
- b. No contesto (equivalente a no marcar nada).
- c. Ramificación y poda.
- d. Vuelta atrás.

Pregunta 5
Incorrecta Puntuación 0,00 sobre 1,00

Si $\lim_{n \rightarrow \infty} (g(n)/f(n)) = 0$, ¿Cuál de las siguientes expresiones NO puede darse?

Seleccione una:

- a. $g(n) \in \Omega(f(n))$
- b. No contesto (equivalente a no marcar nada).
- c. $g(n) \notin \Theta(f(n))$
- d. $f(n) \notin \Theta(g(n))$

Pregunta 6
Incorrecta
Puntuá -0,50 sobre 1,00

La función `test()` procesa una lista de n elementos y devuelve un real. La definición de la función es recursiva. Primero descompone la lista en dos sublistas de la misma longitud usando un segmento de código que tiene una complejidad lineal con la longitud de la lista, envía cada una de las dos sublistas a `test()` para que la procese, hace una serie de operaciones, con el resultado y el valor de retorno, de coste temporal constante. ¿Cuál es el coste temporal asintótico de la función `test()` en función de n ?

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $\Theta(n \log n)$
- c. $\Theta(\log n)$
- d. $\Theta(n)$



Pregunta 7
Incorrecta
Puntuá -0,50 sobre 1,00

¿Cuál de las siguientes formulaciones expresa mejor la complejidad temporal, en función del parámetro n , de la siguiente función? (asumimos que n es potencia exacta de 2)

```
int f(int n) {
    int k=0;
    for (int i = 2; i <= n; i*=2)
        for (int j=i; j > 0; j-=2)
            k++;
    return k;
}
```

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $\sum_{p=2}^{n/2} \frac{p-1}{2}$
- c. $\sum_{p=1}^{\log n} 2 \cdot (p-1)$
- d. $\sum_{p=1}^{\log n} 2^{p-1}$



Pregunta 8
Correcta
Puntuá 1,00 sobre 1,00

Una de estas afirmaciones es falsa. ¿Cuál es?

Seleccione una:

- a. El algoritmo de Kruskal se puede acelerar notablemente si los vértices se organizan en una estructura union-find.
- b. El algoritmo de Prim va construyendo un bosque de árboles que va uniendo hasta que acaba con un árbol de recubrimiento de coste mínimo.
- c. No contesto (equivalente a no marcar nada).
- d. El algoritmo de Prim se puede acelerar notablemente si se guarda, para cada vértice no visitado, los datos de la arista de mínimo peso que lo une a un vértice visitado.



Pregunta 9
Incorrecta
Puntuá -0,50 sobre 1,00

La función `test()` procesa una lista de n elementos y devuelve un real. La definición de la función es recursiva. Primero descompone la lista en dos sublistas de la misma longitud usando un segmento de código que tiene una complejidad lineal con la longitud de la lista, y después envía una de las dos sublistas a `test()` para que la procese, hace una serie de operaciones, con el resultado y el retorno, de coste temporal constante. ¿Cuál es el coste temporal asintótico de la función `test()` en función de n ?

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $\Theta(n)$
- c. $\Theta(\log n)$
- d. $\Theta(n \log n)$



Pregunta 10
Incorrecta
Puntuá -0,50 sobre 1,00

¿Cuál de las siguientes es la complejidad temporal más ajustada para un algoritmo que calcula la potencia n -ésima de una matriz cuadrada, expresada en función de n ?

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $O(n)$
- c. $O(n \log n)$
- d. $O(\log n)$



Pregunta 11
Correcta
Puntuá 1,00 sobre 1,00

El problema del cambio: Se dispone de un conjunto finito de números naturales y se pretende obtener el subconjunto de menor tamaño cuyos elementos suman una cierta cantidad C . ¿Qué estrategia es la más apropiada para resolverlo?

Seleccione una:

- a. Un algoritmo voraz.
- b. No contesto (equivalente a no marcar nada).
- c. Programación dinámica.
- d. Ramificación y poda.



Pregunta 12
Correcta
Puntuá 1,00 sobre 1,00

El funcionamiento del algoritmo de ordenación Heapsort es similar al algoritmo de ordenación por selección, ya que localiza el valor más grande y lo sitúa en la posición final del vector; a continuación, localiza el siguiente valor más grande y lo sitúa en la posición anterior a la última, etc.

¿Cuál de las afirmaciones siguientes es cierta?

Seleccione una:

- a. El algoritmo Heapsort tiene una complejidad $O(n)$ en el caso peor, mejor que la complejidad $O(n^2)$ del algoritmo de selección, porque Heapsort utiliza un algoritmo mucho más eficiente para localizar los valores del vector que valen más.
- b. No contesto (equivalente a no marcar nada).
- c. El algoritmo Heapsort tiene una complejidad $O(n \log n)$ en el caso peor, mejor que la complejidad $O(n^2)$ del algoritmo de selección, porque Heapsort utiliza un algoritmo mucho más eficiente para localizar los valores del vector que valen más.
- d. Por ello, los dos algoritmos tienen la misma complejidad en el caso peor, $O(n^2)$, aunque la complejidad en el caso mejor de Heapsort es $O(n \log n)$.



Pregunta 14
Incorrecta
Puntuá -0,50 sobre 1,00

El problema del alfarero (solución discreta con tiempos continuos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $m_i \in \mathbb{N}$. El valor de cada pieza terminada, $v_i \in \mathbb{N}$ y el tiempo necesario para su fabricación $t_i \in [0..n-1]$. El tiempo disponible para la fabricación de objetos está limitado por $T \in \mathbb{R}$

Se pretende resolver mediante ramificación y poda y para ello se hace uso de una cota que consiste en coger, de entre las claves aún no consideradas, un número al azar de objetos a fabricar siempre que se cumpla las restricciones del problema. ¿Qué podemos decir de esta cota?

Seleccione una:

- a. Que es una cota optimista.
- b. Que no es cota, ni optimista ni pesimista
- c. No contesto (equivalente a no marcar nada).
- d. Que es una cota pesimista.



Pregunta 15
Correcta
Puntuá 1,00 sobre 1,00

En el método voraz ...

Seleccione una:

- a. ... para garantizar la solución óptima, las decisiones solo pueden pertenecer a dominios discretos o discretizables.
- b. ... es habitual preparar los datos para disminuir el coste temporal de la función que determina cuál es la siguiente decisión a tomar.
- c. No contesto (equivalente a no marcar nada).
- d. ... para garantizar la solución óptima, las decisiones solo pueden pertenecer a dominios continuos.



Pregunta 13
Correcta
Puntuá 1,00 sobre 1,00

Una de las afirmaciones siguientes es cierta y las otras dos falsas. Indicad cuál es la falsa.

Seleccione una:

- a. $O(n^n) \in O(n!)$
- b. $O(3^n) \in O(2^n)$
- c. La complejidad temporal de Quicksort es $O(n^2)$ y $\Omega(n \log n)$
- d. No contesto (equivalente a no marcar nada).



Pregunta 16

Correcta

Puntuá 1,00 sobre 1,00

El problema del alfarero (solución discreta con valores y tiempos continuos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $m_i \in N$. El valor de cada pieza terminada, $v_i \in R$ y el tiempo necesario para su fabricación, $t_i \in R$, $i \in [0, n - 1]$. ¿Cuántos objetos de cada clase hay que fabricar para maximizar la ganancia teniendo en cuenta que el tiempo total está limitado por $T \in R$?

Se pretende resolverlo mediante ramificación y poda. Las siguientes funciones tratan de estimar una ganancia aproximada para la parte del nodo aún sin completar. ¿Cuál es la mejor para usarla como parte de la cota optimista?

- a. No contesto (equivalente a no marcar nada).

b.

```
double optimistic( const vector<int> &m, const vector<double> &v, const vector<double> &t, double T, size_t from) {
    double gain = 0.0;
    for( size_t i = from; i < m.size() && T > 0; i++ ) {
        double num_objs = min( T/t[i], double(m[i]));
        gain += num_objs * v[i];
        T -= num_objs * t[i];
    }
    return gain;
}
```

c.

```
double optimistic( const vector<int> &m, const vector<double> &v, const vector<double> &t, double T, size_t from) {
    double gain = 0.0;
    for( size_t i = from; i < m.size() && T > 0; i++ ){
        for( int j = 1; j <= m[i]; j++ ) {
            gain += v[j];
            T -= t[j];
        }
    }
    return gain;
}
```

d.

```
double optimistic( const vector<int> &m, const vector<double> &v, const vector<double> &t, double T, size_t from) {
    double gain = 0.0;
    for( size_t i = from; i < m.size() && T > 0; i++ ){
        for( int j = 1; j <= m[i]; j++ ) {
            if( t[i] < T ) {
                gain += v[j];
                T -= t[j];
            }
        }
    }
    return gain;
}
```

Pregunta 17

Correcta

Puntuá 1,00 sobre 1,00

Las soluciones factibles a un problema de optimización deben cumplir dos restricciones y queremos resolver el problema mediante vuelta atrás o ramificación y poda. ¿Cuál de las siguientes afirmaciones es **cierta**?

Seleccione una:

- a. La cota optimista usada para podar nunca se puede basar en la relajación de ninguna de las restricciones que deben cumplir las soluciones factibles.
- b. La cota optimista usada para podar se debe basar en relajar ambas restricciones simultáneamente.
- c. No contesto (equivalente a no marcar nada).
- d. La cota optimista usada para podar se puede basar en relajar una cualquiera de las dos restricciones.

Pregunta 18

Correcta

Puntuá 1,00 sobre 1,00

Indica cuál es la complejidad, en función de n , del fragmento siguiente:

```
for( int i = 0; i < n; i++ ) {
    A[i] = 0;
    for( int j = 0; j < 20; j++ )
        A[i] += B[j];
}
```

- a. $\Theta(n \log n)$
- b. $\Theta(n^2)$
- c. $\Theta(n)$
- d. No contesto (equivalente a no marcar nada)

Pregunta 19

Correcta

Puntuá 1,00 sobre 1,00

¿Qué hace la siguiente función?

```
void f( vector<int> &A ) {
    priority_queue<int> pq;
    for( auto a: A )
        pq.push(a);
    A.clear();
    while( !pq.empty() ) {
        A.push_back(pq.top());
        pq.pop();
    }
}
```

- a. No contesto (equivalente a no marcar nada)
- b. Invierte el vector A (el último elemento quedará el primero)
- c. Nada, deja el vector como estaba
- d. Ordena el vector A

Pregunta 20

Incorrecta

Puntuá 0,00 sobre 1,00

Una empresa tiene M referencias en su stock. Cada referencia $j \in [1, M]$ tiene un peso p_j y un valor v_j y dispone de n_j unidades en su stock. Dispone de un solo camión en el que puede cargar como máximo un peso P . Indicad cuál de las tres funciones siguientes representa una posible solución voraz aproximada al problema de cargar el camión de manera que se transporte un valor máximo.

Seleccione una:

- a. int f{
 const vector<int> &p,
 const vector<int> &v,
 const vector<int> &n,
 int P,
 int k
}
if(k == 0 || P == 0)
 return 0;
int gain = 0
for(int num_objs = 0; num_objs <= n[k-1]; num_objs++)
 gain = max(gain, f(p, v, n, P - num_objs * p[k-1], k-1));
return gain;
}

 b. int f{
 const vector<int> &p,
 const vector<int> &v,
 const vector<int> &n,
 int P,
 int k
}
if(k == 0 || P == 0)
 return 0;
int gain = 0
for(int num_objs = 0; num_objs <= 1; num_objs++)
 gain = max(gain, f(p, v, n, P - num_objs * p[k-1], k-1));
return gain;
}

 c. int f{
 const vector<int> &p,
 const vector<int> &v,
 const vector<int> &n,
 int P,
 int k
}
if(k == 0 || P == 0)
 return 0;
int num_objs = min(P/p[k-1], n[k-1]);
return num_objs * v[k-1] + f(p, v, n, P - num_objs * p[k-1], k-1);
}

 d. No contesto (equivalente a no marcar nada).

Pregunta **21**
Incorrecta
Puntúa -0,50 sobre 1,00

Pregunta **23**
Incorrecta
Puntuó 0,00 sobre 1,00

El problema del alfarero (solución discreta con tiempos continuos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $m_i \in \mathbb{N}$. El valor de cada pieza terminada, $v_i \in \mathbb{N}$ y el tiempo necesario para su fabricación $t_i \in R : t_i \in [0..n - 1]$. ¿Cuántos objetos de cada clase hay que fabricar para maximizar la ganancia teniendo en cuenta que el tiempo total está limitado por $T \in R$?

¿Cuál de los siguientes esquemas algorítmicos resultaría más eficiente para resolverlo?

Seleccione una:

- a. Programación dinámica
- b. Un algoritmo voraz
- c. No contesto (equivalente)

Pregunta **22**
Correcta
Puntúa 1,00 sobre

El problema del alfarero (solución discreta con valores y tiempos discretos): Se dispone de n clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, $m_i \in N$; El valor de cada pieza terminada, $v_i \in N$ y el tiempo necesario para su fabricación $t_i \in N$, $i \in [0..n-1]$. ¿Cuál es el valor máximo de los objetos que puede fabricar teniendo en cuenta que el tiempo total está limitado por $T \in N$?

Para ello se escribe la siguiente función siguiendo la técnica de divide y vencerá

```

int potter( const vector<int> &v, const vector<int> &m, const vector<int> &t, int T, int k )
{
    if( k == 0 )
        return 0;
    int max_earnings = -1;
    for( int c = 0; c <= m[k-1]; c++ ) {
        int earnings = 0;
        if( T >= c * t[k-1] )
            earnings += v[c];
        max_earnings = max( max_earnings, earnings );
    }
    return max_earnings;
}

```

¿Cuál es la línea que falta?

- a. No contesto (equivalente a no marcar nada)
 - b. `earnings = c * v[k-1] + potter(v, m, t, k-1, T - c * t[k-1]);`
 - c. `earnings = potter(v, m, t, k-1, T - c * t[k-1]);`
 - d. `earnings = potter(v, m, t, k-1, T);`

Pregunta **25**
Correcta
Puntúa 1,00 sobre 1,00

Dada la siguiente función recursiva

```
unsigned f( unsigned a, unsigned b )
{
    if( a < 3 )
        return a + 2*b;
    return f(a-1, (7*b)%10);
}
```

donde suponemos que siempre se va a invocar la función con $b < 1$.

Queremos acelerarla aplicando la técnica de programación dinámica iterativa. ¿Cómo quedaría?

- a.

```
unsigned f( unsigned a, unsigned b ) {
    vector<vector<unsigned>> M(a+1, vector<unsigned>(10));
    for( unsigned i = 0; i < a; i++ )
        for( unsigned j = 0; j < 10; j++ )
            if( i < 3 )
                M[i][j] = i + 2*j;
            else
                M[i][j] = M[i-1][(7*j)%10];
    return M[a][b];
}
```

b. No contesto (equivalente a marcar nada)

c.

```
unsigned f( unsigned a, unsigned b ) {
    vector<vector<unsigned>> M(a+1, vector<unsigned>(10));
    for( unsigned i = 0; i <= a; i++ )
        for( unsigned j = 0; j < 10; j++ )
            if( i < 3 )
                M[i][j] = i + 2*j;
            else
                M[i][j] = M[i-1][(7*j)%10];
    return M[a][b];
}
```

d.

```
unsigned f( unsigned a, unsigned b ) {
    vector<vector<unsigned>> M(a+1, vector<unsigned>(10));
    for( unsigned i = 0; i < a; i++ )
        for( unsigned j = 0; j < 10; j++ )
            if( i < 3 )
                M[i][j] = i + 2*j;
            else
                M[i][j] = M[i-1][(7*j)%10];
    return M[a][b];
}
```

Sea el vector $v = \{1, 3, 2, 7, 4, 6, 8\}$ cuyos elementos están dispuestos formando un montículo de mínimos. Posteriormente añadimos en la última posición del vector un elemento nuevo con valor 5. ¿Qué operación hay que hacer para que el vector siga representando un montículo de mínimos?

Seleccione una:

- a. No contesto (equivalente a)
- b. No hay que hacer nada pu
- c. Intercambiar el 7 con el 5.
- d. Intercambiar el 8 con el 5.

Pregunta 2.
Correcta
Puntúa 1,00

Indica cuál es la complejidad, en función de n ($n \geq 0$), del fragmento siguiente.

```
int f( int n ) {  
    if( n == 0)  
        return n;  
  
    return f(n/2)*f(n/  
}
```

- a. No contesto (equivalente a no marcar na questão)
 - b. $\Theta(n)$
 - c. $\Theta(n \log n)$
 - d. $\Theta(\log \log n)$

Pregunta 20

Dada la siguiente función construida mediante la técnica memoización

```

int f( vector<unsigned> &x, unsigned i )
{
    if( x[i] != SENTINEL )
        return x[i];
    if( i < 5 )
        return i;
    return x[i] = f(x, i-1) + f(x, i-2);
}

```

¿Cuál es la declaración para SENTINEL más adecuada?

- a. `const unsigned SENTINEL = -1;`
 - b. `const unsigned SENTINEL = numeric_limits<unsigned>::max();`
 - c. `const unsigned SENTINEL = 0;`
 - d. No contesto (equivalente a no marcar nada)

Pregunta 21
Correcta
Puntúa 1,00

¿Qué complejidad tiene la siguiente función?

```
A.clear();
while( !pq.empty() ) {
    A.push_back(pq.top());
    pq.pop();
}
```

Suponed que la cola de prioridad está implementada como un heap y que $n = A.size()$. priority_queue<int> pq(begin(A), end(A)) construye un heap a partir de los datos que hay en el vector A.

- a. $\Theta(n^2)$
 - b. $\backslash(\backslash\Theta(n))$
 - c. $\backslash(\backslash\Theta(n \log n))$
 - d. No contesto (equivalente a no marcar na questão)

Pregunta 28

Incorrecta

Puntuía 0,00 sobre 1,00

Queremos aplicar la técnica de memoización a la siguiente función recursiva:

```
double f( double x ) {
    if( x <= 2 )
        return x;
    return f(sqrt(x-1)) + f(sqrt(x-2));
}
```

¿Cuál sería un buen candidato para el almacén? [la función sqrt() calcula la raíz cuadrada]

- a. vector<double> M(xMax+1); (donde xMax es el valor de x en la primera llamada)
- b. No se puede aplicar la técnica de memoización
- c. vector<double, double> M(xMax+1,xMax+1); (donde xMax es el valor de x en la primera llamada)
- d. No contesto (equivalente a no marcar nada) ✗

Pregunta 29

Incorrecta

Puntuía -0,50 sobre 1,00

¿Cuál de las siguientes formulaciones expresa mejor el número de llamadas recursivas que hace Quicksort en el mejor de los casos?

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $\sum_{i=0}^n \log(n)$
- c. $\sum_{i=0}^n (\log(n))^2$
- d. $\sum_{i=0}^n (\log(n))$ ✗

Pregunta 30

Incorrecta

Puntuía -0,50 sobre 1,00

Se dispone de $\binom{n}{k}$ clases de objetos. De cada una de ellas se conoce el número máximo de piezas que se puede fabricar, m_i y el tiempo necesario para su fabricación t_i . Queremos listar todas las posibilidades de fabricación de objetos teniendo en cuenta que el tiempo total está limitado por T .

Para ello hemos hecho el siguiente programa donde faltan unas líneas:

```
void combinations( const vector<int> &m, const vector<double> &t, double T, size_t k, vector<int> &x ) {
    if( k == m.size() ) {
        print_comb(x);
        return;
    }
    // ==> Aquí falta código <==

    void combinations( const vector<int> &m, const vector<double> &t, double T ) {
        vector<int> x(m.size());
        combinations(m, t, T, 0, x);
    }
}
```

¿Cuáles son las líneas que faltan? [suponed que print_comb() imprime correctamente la combinación que hay codificada en x]

- a.

```
for( int j = 0; j < m[k]; j++ ) {
    x[j] = k;
    if( T >= j * t[k] )
        combinations( m, t, T - j * t[k], k+1, x );
}
```
 - b. No contesto (equivalente a no marcar nada)
 - c.

```
for( int j = 0; j < m[k]; j++ ) {
    x[k] = j;
    if( T >= j * t[k] )
        combinations( m, t, T - j * t[k], k+1, x );
}
```
 - d.

```
for( int j = 0; j <= m[k]; j++ ) {
    x[k] = j;
    if( T >= j * t[k] )
        combinations( m, t, T - j * t[k], k+1, x );
}
```
- ✗

► [valencià] Examen final C4 - Grup "Vesprades"

Ir a...

FRAGMENTOS DE CÓDIGO:

Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
unsigned f( unsigned y, unsigned x){ // suponemos y >= x
    if( x==0 || y==x ) return 1;
    return f(y-1, x-1) + f(y-1, x);
}
```

¿Cuál es la mejor complejidad temporal que se puede conseguir?

Seleccione una:

- a. $O(x)$
- b. $O(x \cdot y)$ ✓
- c. No contesto (equivalente a no marcar nada).
- d. $O(y)$

Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
unsigned f( unsigned y, unsigned x){ // suponemos y >= x
    if( x==0 || y==x ) return 1;
    return f(y-1, x-1) + f(y-1, x);
}
```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

Seleccione una:

- a. No contesto (equivalente a no marcar nada). ✗
- b. $O(1)$
- c. $O(y)$
- d. $O(y^2)$

Se pretende aplicar la técnica memoización a la siguiente función recursiva:

```
int f( int x, int y ) {
    if( x <= y ) return 1;
    return x * f(x-1,y);
}
```

En el caso más desfavorable, ¿qué complejidades temporal y espacial cabe esperar de la función resultante?

Seleccione una:

- a. Ninguna de las otras dos opciones es correcta.
- b. $O(y - x)$, tanto temporal como espacial. ✓
- c. No contesto (equivalente a no marcar nada).
- d. Temporal $O(x \cdot y)$ y espacial $O(x)$

Se pretende aplicar la técnica memoización a la siguiente función recursiva:

```
int f( int x, int y ) {
    if( x <= y ) return 1;
    return x * f(x-1,y) + y;
}
```

En el caso más desfavorable, ¿qué complejidades temporal y espacial cabe esperar de la función resultante?

Seleccione una:

- a. Ninguna de las otras dos opciones es correcta.
- b. No contesto (equivalente a no marcar nada).
- c. $O(x - y)$, tanto temporal como espacial. ✓
- d. Temporal $O(x - y)$ y espacial $O(1)$

Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
int f( int x, int y ) {
    if( x <= y ) return 1;
    return x + f(x-1,y);
}
```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

Seleccione una:

- a. $O(1)$ ✓
- b. $O(x^2)$
- c. $O(x)$
- d. No contesto (equivalente a no marcar nada).

Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
int f( int x, int y ) {
    if( x <= y ) return 1;
    return x + f(x-1,y);
}
```

¿Cuál es la mejor complejidad temporal que se puede conseguir?

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $O(y)$
- c. $O(x \cdot y)$
- d. $O(x)$

Se pretende aplicar la técnica memoización a la siguiente función recursiva:

```
int f( int x, int y ) {
    if( x <= y ) return 1;
    return x + f(x-1,y);
}
```

En el caso más desfavorable, ¿qué complejidades temporal y espacial cabe esperar de la función resultante?

Seleccione una:

- a. $O(x - y)$, tanto temporal como espacial. ✓
- b. No contesto (equivalente a no marcar nada).
- c. Ninguna de las otras dos opciones es correcta.
- d. Temporal $O(x - y)$ y espacial $O(1)$

Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
float f(unsigned x, int y) {
    if( y < 0 ) return 0;
    float A = 0.0;
    if( v1[y] <= x )
        A = v2[y] + f( x-v1[y], y-1 );
    float B = f( x, y-1 );
    return min(A,2+B);
}
```

¿Cuál es la mejor complejidad temporal que se puede conseguir?

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $O(x \cdot y)$ ✓
- c. $O(x)$
- d. $O(y)$

Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
float f(unsigned x, int y) {
    if( y < 0 ) return 0;
    float A = 0.0;
    if( v1[y] <= x )
        A = v2[y] + f( x-v1[y], y-1 );
    float B = f( x, y-1 );
    return min(A,2+B);
}
```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

Seleccione una:

- a. $O(y)$ ✓
- b. $O(1)$
- c. No contesto (equivalente a no marcar nada).
- d. $O(y^2)$

Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
float f(unsigned x, int y) {
    if( y < 0 ) return 0;
    float A = 0.0;
    if( v1[y] <= x )
        A = v2[y] + f( x-v1[y], y-1 );
    float B = f( x, y-1 );
    return min(A,2+B);
}
```

¿Cuál es la mejor complejidad temporal que se puede conseguir?

Seleccione una:

- a. $O(y)$
- b. $O(x)$
- c. No contesto (equivalente a no marcar nada).
- d. $O(x \cdot y)$ ✓

Se pretende aplicar la técnica memoización a la siguiente función recursiva:

```
int f( int x, int y ) {
    if( x > y ) return 1;
    return x*(y-1) + f(x,y-2);
}
```

En el caso más desfavorable, ¿qué complejidades temporal y espacial cabe esperar de la función resultante?

Seleccione una:

- a. Ninguna de las otras dos opciones es correcta.
- b. Temporal $O(x \cdot y)$ y espacial $O(x)$
- c. No contesto (equivalente a no marcar nada).
- d. $O(y - x)$, tanto temporal como espacial. ✓

Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
unsigned f( unsigned x, unsigned v[] ) {
    if( x==0 )
        return 0;
    unsigned m = 0;
    for ( unsigned k = 0; k < x; k++ )
        m = max( m, v[k] + f( x-k, v ) );
    return m;
}
```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

Seleccione una:

- a. $O(1)$
- b. $O(x)$ ✓
- c. $O(x^2)$
- d. No contesto (equivalente a no marcar nada).

Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
unsigned f( unsigned x, unsigned v[] ) {
    if( x==0 )
        return 0;
    unsigned m = 0;
    for ( unsigned y = 0; y < x; y++ )
        m = max( m, v[y] + f( x-y, v ) );
    return m;
}
```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $O(y)$
- c. $O(x \cdot y)$
- d. $O(x)$ ✓

TEORÍA:

La eficiencia de los algoritmos voraces se basa en el hecho de que ...

Seleccione una:

- a. ... las decisiones tomadas nunca se reconsideran. ✓
- b. No contesto (equivalente a no marcar nada).
- c. ... antes de tomar una decisión se comprueba si satisface las restricciones del problema.
- d. ... con antelación, las posibles decisiones se ordenan de mejor a peor.

En la solución al problema de la mochila continua ¿por qué es conveniente la ordenación previa de los objetos?

Seleccione una:

- a. Para reducir la complejidad temporal en la toma de cada decisión: de $O(n^2)$ a $O(n \log n)$, donde n es el número de objetos a considerar.
- b. Porque si no se hace no es posible garantizar que la toma de decisiones siga un criterio voraz.
- c. No contesto (equivalente a no marcar nada).
- d. Para reducir la complejidad temporal en la toma de cada decisión: de $O(n)$ a $O(1)$, donde n es el número de objetos a considerar.

El problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado ...

Seleccione una:

- a. ... se puede resolver siempre con una estrategia voraz. ✓
- b. ... sólo se puede resolver con una estrategia voraz si existe una arista para cualquier par de vértices del grafo.
- c. No contesto (equivalente a no marcar nada).
- d. ... no se puede resolver en general con una estrategia voraz.

¿Cuál de estas estrategias para calcular el n -ésimo elemento de la serie de Fibonacci ($f(n) = f(n-1) + f(n-2)$, $f(1) = f(2) = 1$) es más eficiente?

Seleccione una:

- a. La estrategia voraz.
- b. No contesto (equivalente a no marcar nada).
- c. Las dos estrategias citadas serían similares en cuanto a eficiencia.
- d. Programación dinámica. ✓

Supongamos que una solución recursiva a un problema de optimización muestra estas dos características: por un lado, se basa en obtener soluciones óptimas a problemas parciales más pequeños, y por otro, estos subproblemas se resuelven más de una vez durante el proceso recursivo. Este problema es candidato a tener una solución alternativa basada en ...

Seleccione una:

- a. ... un algoritmo de programación dinámica.
- b. ... un algoritmo voraz.
- c. ... un algoritmo del estilo de divide y vencerás.
- d. No contesto (equivalente a no marcar nada).

Un tubo de n centímetros de largo se puede cortar en segmentos de 1 centímetro, 2 centímetros, etc. Existe una lista de los precios a los que se venden los segmentos de cada longitud. Una de las maneras de cortar el tubo es la que más ingresos nos producirá. Di cuál de estas tres afirmaciones es falsa.

Seleccione una:

- a. Hacer una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar el tubo consume un tiempo $\Theta(2^n)$.
- b. Es posible evitar hacer la evaluación exhaustiva "de fuerza bruta" guardando, para cada posible longitud $j < n$ el precio más elevado posible que se puede obtener dividiendo el tubo correspondiente.
- c. Hacer una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar el tubo consume un tiempo $\Theta(n!)$. ✓
- d. No contesto (equivalente a no marcar nada).

¿Para qué se utiliza el TAD "Union-find" en el algoritmo de Kruskal?

Seleccione una:

- a. Para comprobar si un arco forma ciclos. ✓
- b. No contesto (equivalente a no marcar nada).
- c. Para comprobar si un vértice ya ha sido visitado.
- d. Para comprobar si dos vértices son equivalentes.

La solución de programación dinámica iterativa del problema de la mochila discreta ...

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. ... tiene la restricción de que los valores de los objetos tienen que ser números discretos o discretizables.
- c. ... tiene la restricción de que los pesos de los objetos tienen que ser números discretos o discretizables. ✓
- d. ... calcula menos veces el valor de la mochila que la correspondiente solución de programación dinámica recursiva.

La solución óptima al problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado ...

Seleccione una:

- a. ... se construye haciendo crecer un único árbol.
- b. No contesto (equivalente a no marcar nada).
- c. ... se construye haciendo crecer varios árboles que al final acaban injertados en un único árbol.
- d. ... puede construir un único árbol que va creciendo o bien construir un bosque de árboles que al final se injertan en un único árbol. ✓

Un tubo de n centímetros de largo se puede cortar en segmentos de 1 centímetro, 2 centímetros, etc. Existe una lista de los precios a los que se venden los segmentos de cada longitud. Una de las maneras de cortar el tubo es la que más ingresos nos producirá. Di cuál de estas tres afirmaciones es falsa.

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. Es posible evitar hacer la evaluación exhaustiva "de fuerza bruta" guardando, para cada posible longitud $j < n$ el precio más elevado posible que se puede obtener dividiendo el tubo correspondiente.
- c. Hacer una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar el tubo consume un tiempo $\Theta(n!)$. ✓
- d. Hacer una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar el tubo consume un tiempo $\Theta(2^n)$.

¿Cuál de estos tres problemas de optimización no tiene, o no se le conoce, una solución voraz óptima?

Seleccione una:

- a. El árbol de cobertura de coste mínimo de un grafo conexo.
- b. No contesto (equivalente a no marcar nada).
- c. El problema de la mochila discreta o sin fraccionamiento. ✓
- d. El problema de la mochila continua o con fraccionamiento.

El valor que se obtiene con el método voraz para el problema de la mochila discreta es ...

Seleccione una:

- a. ... una cota inferior para el valor óptimo, pero que nunca coincide con este.
- b. ... una cota superior para el valor óptimo.
- c. No contesto (equivalente a no marcar nada).
- d. ... una cota inferior para el valor óptimo que a veces puede ser igual a este. ✓

En la solución al problema de la mochila continua ¿por qué es conveniente la ordenación previa de los objetos?

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. Para reducir la complejidad temporal en la toma de cada decisión: de $O(n)$ a $O(1)$, donde n es el número de objetos a considerar.
- c. Para reducir la complejidad temporal en la toma de cada decisión: de $O(n^2)$ a $O(n \log n)$, donde n es el número de objetos a considerar.
- d. Porque si no se hace no es posible garantizar que la toma de decisiones siga un criterio voraz.

Supongamos que una solución recursiva a un problema de optimización muestra estas dos características: por un lado, se basa en obtener soluciones óptimas a problemas parciales más pequeños, y por otro, estos subproblemas se resuelven más de una vez durante el proceso recursivo. Este problema es candidato a tener una solución alternativa basada en ...

Seleccione una:

- a. ... un algoritmo del estilo de divide y vencerás.
- b. No contesto (equivalente a no marcar nada).
- c. ... un algoritmo de programación dinámica. ✓
- d. ... un algoritmo voraz.

En el método voraz ...

Seleccione una:

- a. ... es habitual preparar los datos para disminuir el coste temporal de la función que determina cuál es la siguiente decisión a tomar.
- b. ... siempre se encuentra solución pero puede que no sea la óptima.
- c. No contesto (equivalente a no marcar nada).
- d. ... el dominio de las decisiones sólo pueden ser conjuntos discretos o discretizables. ✓

Supongamos que una solución recursiva a un problema de optimización muestra estas dos características: por un lado, se basa en obtener soluciones óptimas a problemas parciales más pequeños, y por otro, estos subproblemas se resuelven más de una vez durante el proceso recursivo. Este problema es candidato a tener una solución alternativa basada en ...

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. ... un algoritmo de programación dinámica. ✓
- c. ... un algoritmo voraz.
- d. ... un algoritmo del estilo de divide y vencerás.

La solución de programación dinámica iterativa del problema de la mochila discreta ...

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. ... tiene la restricción de que los pesos de los objetos tienen que ser números discretos o discretizables. ✓
- c. ... calcula menos veces el valor de la mochila que la correspondiente solución de programación dinámica recursiva.
- d. ... tiene la restricción de que los valores de los objetos tienen que ser números discretos o discretizables.

¿Cuál de estos tres problemas de optimización no tiene, o no se le conoce, una solución voraz óptima?

Seleccione una:

- a. El problema de la mochila continua o con fraccionamiento.
- b. El árbol de cobertura de coste mínimo de un grafo conexo.
- c. No contesto (equivalente a no marcar nada).
- d. El problema de la mochila discreta o sin fraccionamiento. ✓

¿Qué mecanismo se usa para acelerar el algoritmo de Prim?

Seleccione una:

- a. El TAD "Union-find"
- b. Mantener una lista de los arcos ordenados según su peso.
- c. No contesto (equivalente a no marcar nada).
- d. Mantener para cada vértice el vértice origen de la arista más corta hasta él. ✓

¿Cuál de los siguientes pares de problemas son equivalentes en cuanto al tipo de solución (óptima, factible, etc.) aportada por el método voraz?

Seleccione una:

- a. El fontanero diligente y la asignación de tareas.
- b. El fontanero diligente y el problema del cambio.
- c. No contesto (equivalente a no marcar nada).
- d. El fontanero diligente y la mochila continua. ✓

La solución óptima al problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado ...

Seleccione una:

- a. ... se construye haciendo crecer varios árboles que al final acaban injertados en un único árbol.
- b. No contesto (equivalente a no marcar nada).
- c. ... se construye haciendo crecer un único árbol.
- d. ... puede construir un único árbol que va creciendo o bien construir un bosque de árboles que al final se injertan en un único árbol. ✓

De los problemas siguientes, indicad cuál no se puede tratar eficientemente como los otros dos:

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. El problema del cambio, o sea, el de encontrar la manera de entregar una cantidad de dinero usando el mínimo de monedas posibles.
- c. El problema de la mochila sin fraccionamiento y sin restricciones en cuanto al dominio de los pesos de los objetos y de sus valores.
- d. El problema de cortar un tubo de forma que se obtenga el máximo beneficio posible. ✓

¿Cuál de los siguientes pares de problemas son equivalentes en cuanto al tipo de solución (óptima, factible, etc.) aportada por el método voraz?

Seleccione una:

- a. El fontanero diligente y la asignación de tareas.
- b. El fontanero diligente y el problema del cambio.
- c. No contesto (equivalente a no marcar nada).
- d. El fontanero diligente y la mochila continua. ✓

¿Cómo se vería afectada la solución voraz al problema de la asignación de tareas en el caso de que se incorporaran restricciones que contemplen que ciertas tareas no pueden ser adjudicadas a ciertos trabajadores ?

Seleccione una:

- a. Ya no se garantizaría la solución óptima pero si una factible.
- b. La solución factible ya no estaría garantizada, es decir, pudiera ser que el algoritmo no llegue a una solución alguna. ✓
- c. Habría que replantearse el criterio de selección para comenzar por aquellos trabajadores con más restricciones en cuanto a las tareas que no pueden realizar para asegurar, al menos, una solución factible.
- d. No contesto (equivalente a no marcar nada).

Un tubo de n centímetros de largo se puede cortar en segmentos de 1 centímetro, 2 centímetros, etc. Existe una lista de los precios a los que se venden los segmentos de cada longitud. Una de las maneras de cortar el tubo es la que más ingresos nos producirá. Di cuál de estas tres afirmaciones es falsa.

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. Hacer una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar el tubo consume un tiempo $\Theta(n!)$. ✓
- c. Es posible evitar hacer la evaluación exhaustiva "de fuerza bruta" guardando, para cada posible longitud $j < n$, el precio más elevado posible que se puede obtener dividiendo el tubo correspondiente.
- d. Hacer una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar el tubo consume un tiempo $\Theta(2^n)$.

¿Para qué se utiliza el TAD "Union-find" en el algoritmo de Kruskal?

Seleccione una:

- a. Para comprobar si un vértice ya ha sido visitado.
- b. No contesto (equivalente a no marcar nada).
- c. Para comprobar si un arco forma ciclos. ✓
- d. Para comprobar si dos vértices son equivalentes.

Cuando se calculan los coeficientes binomiales usando la recursión $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$, con $\binom{n}{0} = \binom{n}{n} = 1$, qué problema se da y cómo se puede resolver?

Seleccione una:

- a. La recursión puede ser infinita y por tanto es necesario organizarla según el esquema iterativo de programación dinámica.
- b. Se repiten muchos cálculos y ello se puede evitar haciendo uso de una estrategia voraz.
- c. No contesto (equivalente a no marcar nada).
- d. Se repiten muchos cálculos y ello se puede evitar usando programación dinámica. ✓

La programación dinámica...

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. Las otras dos opciones son ciertas. ✓
- c. ... en algunos casos se puede utilizar para resolver problemas de optimización con dominios continuos pero probablemente pierda su eficacia ya que puede disminuir drásticamente el número de subproblemas repetidos.
- d. ... normalmente se usa para resolver problemas de optimización con dominios discretizables puesto que las tablas se han de indexar con este tipo de valores.

Supongamos que una solución recursiva a un problema de optimización muestra estas dos características: por un lado, se basa en obtener soluciones óptimas a problemas parciales más pequeños, y por otro, estos subproblemas se resuelven más de una vez durante el proceso recursivo. Este problema es candidato a tener una solución alternativa basada en ...

Seleccione una:

- a. ... un algoritmo del estilo de divide y vencerás. ✓
- b. ... un algoritmo de programación dinámica.
- c. ... un algoritmo voraz.
- d. No contesto (equivalente a no marcar nada).

¿Qué mecanismo se usa para acelerar el algoritmo de Prim?

Seleccione una:

- a. El TAD "Union-find"
- b. Mantener para cada vértice el vértice origen de la arista más corta hasta él. ✓
- c. No contesto (equivalente a no marcar nada).
- d. Mantener una lista de los arcos ordenados según su peso.

La solución óptima al problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado ...

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. ... debe construirlo arista a vértice: vértice a vértice no puede ser.
- c. ... debe construirlo vértice a vértice: arista a arista no puede ser.
- d. ... puede construirlo tanto vértice a vértice como arista a arista. ✓

La solución de programación dinámica iterativa del problema de la mochila discreta ...

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. ... calcula menos veces el valor de la mochila que la correspondiente solución de programación dinámica recursiva.
- c. ... tiene la restricción de que los pesos de los objetos tienen que ser números discretos o discretizables.
- d. ... tiene la restricción de que los valores de los objetos tienen que ser números discretos o discretizables.

La solución óptima al problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado ...

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. ... debe construirlo arista a vértice: vértice a vértice no puede ser.
- c. ... debe construirlo vértice a vértice: arista a arista no puede ser.
- d. ... puede construirlo tanto vértice a vértice como arista a arista. ✓

De los problemas siguientes, indicad cuál no se puede tratar eficientemente como los otros dos:

- El problema de la mochila sin fraccionamiento y sin restricciones en cuanto al dominio de los pesos de los objetos y de sus valores.
- El problema del cambio, o sea, el de encontrar la manera de entregar una cantidad de dinero usando el mínimo de monedas posibles.
- El problema de cortar un tubo de forma que se obtenga el máximo beneficio posible.

Un informático quiere subir a una montaña y para ello decide que tras cada paso, el siguiente debe tomarlo en la dirección de máxima pendiente hacia arriba. Además, entenderá que ha alcanzado la cima cuando llegue a un punto en el que no haya ninguna dirección que sea cuesta arriba. ¿qué tipo de algoritmo está usando nuestro informático?

- un algoritmo divide y vencerás.
- un algoritmo de programación dinámica.
- un algoritmo voraz.

La mejora que en general aporta la programación dinámica frente a la solución ingenua se consigue gracias al hecho de que ...

- ... en la solución ingenua se resuelve pocas veces un número relativamente grande de subproblemas distintos.
- ... en la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos.
- El número de veces que se resuelven los subproblemas no tiene nada que ver con la eficiencia de los problemas resueltos mediante programación dinámica.

El problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado ...

- solo se puede resolver con una estrategia voraz si existe una arista para cualquier par de vértices del grafo.
- ... no se puede resolver en general con una estrategia voraz.
- ... se puede resolver siempre con una estrategia voraz.

¿Cuál de estas tres estrategias voraces obtiene un mejor valor para la mochila discreta?

- Meter primero los elementos de mayor valor específico o valor por unidad de peso.
- Meter primero los elementos de menor peso.
- Meter primero los elementos de mayor valor.

1. ¿Qué mecanismo se usa para acelerar el algoritmo de Prim?

- Mantener para cada vértice el vértice origen de la arista más corta hasta él / Mantener para cada vértice su "padre" más cercano
- El TAD "Union-find"
- Mantener una lista de los arcos ordenados según su peso.

2. ¿Cómo se vería afectada la solución voraz al problema de la asignación de tareas en el caso de que se incorporaran restricciones que contemplen que ciertas tareas no pueden ser adjudicadas a ciertos trabajadores ?

- La solución factible ya no estaría garantizada, es decir, pudiera ser que el algoritmo no llega a solución alguna.
- Habrá que replantearse el criterio de selección para comenzar por aquellos trabajadores con más restricciones en cuanto a las tareas que no pueden realizar para asegurar, al menos, una solución factible.
- Ya no se garantizaría la solución óptima pero sí una factible.

3. El valor que se obtiene con el método voraz para el problema de la mochila discreta es ...

- ... una cota inferior para el valor óptimo que a veces puede ser igual a este.
- ... una cota superior para el valor óptimo.
- ... una cota inferior para el valor óptimo, pero que nunca coincide con este.

4. Cuando se calculan los coeficientes binomiales usando la recursión , con , qué problema se da y de cómo se puede resolver?

- La recursión puede ser infinita y por tanto es necesario organizarla según el esquema iterativo de programación dinámica.
- Se repiten muchos cálculos y ello se puede evitar usando programación dinámica.
- Se repiten muchos cálculos y ello se puede evitar haciendo uso de una estrategia voraz.

5. Los algoritmos de programación dinámica hacen uso ...

- ... de que la solución óptima se puede construir añadiendo a la solución el elemento óptimo de los elementos restantes, uno a uno.
- ... de una estrategia que ahorra cálculos innecesarios / ... de que se puede ahorrar cálculos guardando resultados anteriores en un almacén.
- ... de una estrategia trivial consistente en examinar todas las soluciones posibles.

6. La eficiencia de los algoritmos voraces se basa en el hecho de que ...

- ... las decisiones tomadas nunca se reconsideran.
- ... con antelación, las posibles decisiones se ordenan de mejor a peor.
- ... antes de tomar una decisión se comprueba si satisface las restricciones del problema.

7. Un informático quiere subir a una montaña y para ello decide que tras cada paso, el siguiente debe tomarlo en la dirección de máxima pendiente hacia arriba. Además, entenderá que ha alcanzado la cima cuando llegue a un punto en el que no haya ninguna dirección que sea cuesta arriba. ¿qué tipo de algoritmo está usando nuestro informático?

- un algoritmo divide y vencerás.
- un algoritmo de programación dinámica.
- un algoritmo voraz.

8. De los problemas siguientes, indicad cuál no se puede tratar eficientemente como los otros dos:

- El problema de la mochila sin fraccionamiento y sin restricciones en cuanto al dominio de los pesos de los objetos y de sus valores.
- El problema del cambio, o sea, el de encontrar la manera de entregar una cantidad de dinero usando el mínimo de monedas posibles.
- El problema de cortar un tubo de forma que se obtenga el máximo beneficio posible.

9. La solución óptima al problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado

- ... puede construir un único árbol que va creciendo o bien construir un bosque de árboles que al final se insertan en un único árbol.
- ... se construye haciendo crecer varios árboles que al final acaban injertados en un único árbol.
- ... se construye haciendo crecer un único árbol.

10. El problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado ...

- solo se puede resolver con una estrategia voraz si existe una arista para cualquier par de vértices del grafo.
- ... no se puede resolver en general con una estrategia voraz.
- ... se puede resolver siempre con una estrategia voraz.

11. La mejora que en general aporta la programación dinámica frente a la solución ingenua se consigue gracias al hecho de que ...

- ... en la solución ingenua se resuelve pocas veces un número relativamente grande de subproblemas distintos.
- ... en la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos.
- El número de veces que se resuelven los subproblemas no tiene nada que ver con la eficiencia de los problemas resueltos mediante programación dinámica.

12. En la solución al problema de la mochila continua ¿por qué es conveniente la ordenación previa de los objetos?

- Para reducir la complejidad temporal en la toma de cada decisión: de $O(n)$ a $O(1)$, donde n es el número de objetos a considerar.
- Porque si no se hace no es posible garantizar que la toma de decisiones siga un criterio voraz.
- Para reducir la complejidad temporal en la toma de cada decisión: de $O(n^2)$ a $O(n\log n)$, donde n es el número de objetos a considerar.

13. En el método voraz ...

- ... siempre se encuentra una solución pero puede que no sea la óptima.
- ... es habitual preparar los datos para disminuir el coste temporal de la función que determina cuál es la siguiente decisión a tomar.
- ... el dominio de las decisiones solo pueden ser conjuntos discretos o discretizables.

14. ¿Cuál de estas tres estrategias voraces obtiene un mejor valor para la mochila discreta?

- Meter primero los elementos de mayor valor específico o valor por unidad de peso.
- Meter primero los elementos de menor peso.
- Meter primero los elementos de mayor valor.

15. Dado un problema de optimización, el método voraz ...

- Siempre obtiene la solución óptima.
- Garantiza la solución óptima sólo para determinados problemas.
- Siempre obtiene la solución factible.

16. Un tubo de un centímetro de largo se puede cortar en segmentos de 1 centímetro, 2 centímetros, etc. Existe una lista de los precios a los que se venden los segmentos de cada longitud. Una manera de cortar el tubo es la que más ingresos nos producirá. Di cuál de estas tres afirmaciones es FALSA

- Hacer una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar consume un tiempo $\Theta(2^n)$.
- Hacer una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar consume un tiempo $\Theta(n!)$.
- Es posible evitar hacer la evaluación exhaustiva "de fuerza bruta" guardando, para cada posible longitud i-n, el precio más elevado posible que se puede obtener dividiendo el tubo correspondiente.

17. Supongamos que una solución recursiva a un problema de optimización muestra estas dos características: por un lado, se basa en obtener soluciones óptimas a problemas parciales más pequeños, y por otro , estos subproblemas se resuelven más de una vez durante el proceso recursivo. Este problema es candidato a tener una solución alternativa basada en ...

- ... un algoritmo voraz.
- ... un algoritmo de programación dinámica.
- ... un algoritmo del estilo divide y vencerás.

18. ¿Cuál de estos tres problemas de optimización no tiene, o no se le conoce, una solución voraz óptima?
- El árbol de coste mínimo de un grafo conexo.
 - El problema de la mochila discreta o sin fraccionamiento.**
 - El problema de la mochila continua o con fraccionamiento.

19. La programación dinámica ...
- ... en algunos casos se puede utilizar para resolver problemas de optimización con dominios continuos pero probablemente pierda su eficacia ya que puede disminuir drásticamente el número de subproblemas repetidos.
 - Las otras dos opciones son ciertas.**
 - ... normalmente se usa para resolver problemas de optimización con dominios discretizables puesto que las tablas se han de indexar con este tipo de valores.

20. Si ante un problema de decisión existe un criterio de selección voraz entonces...
- la solución óptima está garantizada.
 - ... al menos una solución factible está garantizada.
 - Ninguna de las otras dos opciones es cierta.**

21. Cuando la descomposición recursiva de un problema da lugar a subproblemas de tamaño similar, ¿qué esquema podría ser más apropiado?
- Programación dinámica.**
 - Divide y vencerás, siempre que se garantice que los subproblemas no son del mismo tamaño.
 - El método voraz.

22. ¿Cuál de los siguientes pares de problemas son equivalentes en cuanto al tipo de solución (óptima, factible, etc.) aportada por el método voraz?
- El fontanero diligente y el problema del cambio.
 - La mochila continua y la asignación de tareas.
 - La mochila discreta y la asignación de tareas.**

23. ¿Cuál de estas estrategias para calcular el n -ésimo elemento de la serie de Fibonacci es más eficiente?
- La estrategia voraz.
 - Las dos estrategias citadas serían similares en cuanto a eficiencia.
 - Programación dinámica.**

24. ¿Por qué se utiliza el TAD "Union-find" en el algoritmo de Kruskal?
- Para comprobar si dos vértices son equivalentes
 - Para comprobar si un arco forma ciclos**
 - Para comprobar si un vértice ya ha sido visitado

29. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
unsigned f( unsigned x, unsigned v[ ]){
    if (x==0)
        return 0;
    unsigned m = 0;
    for(unsigned y = 0; y<x; y++)
        m = max(m, v[y] + f(x-y, v));
    return m;
}
```

- ¿Cuál es la mejor complejidad espacial que se puede conseguir?
- $\mathcal{O}(x * y)$
 - $\mathcal{O}(x)$**
 - $\mathcal{O}(y)$

30. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
unsigned f( unsigned x, unsigned v[ ]){
    if (x==0)
        return 0;
    unsigned m = 0;
    for(unsigned y = 0; y<x; y++)
        m = max(m, v[y] + f(x-y, v));
    return m;
}
```

- ¿Cuál es la mejor estructura para el almacén?
- int A[]**
 - int A
 - int A[]

31. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
float f(unsigned x, int y){
    if(y<0) return 0;
    float a = 0.0;
    if(v1[y] <= x)
        a = v2[y] + f(x-v1[y], y-1);
    float b = f(x, y-1);
    return min(a, 2+b);
}
```

- ¿Cuál es la mejor estructura para el almacén?
- unsigned A[][]**
 - unsigned A
 - unsigned A[]

25. Se pretende aplicar la técnica memoización a la siguiente función recursiva:

```
int f( int x, int y ) {
    if( x > y ) return 1;
    return x*(y-1) + f(x-1, y-2);
}
```

En el caso más desfavorable, ¿qué complejidades temporal y espacial cabe esperar de la función resultante?

- Ninguna de las otras dos opciones es correcta.
- Temporal $\mathcal{O}(x \cdot y)$ y espacial $\mathcal{O}(x)$
- $\mathcal{O}((O(x-y))$, tanto temporal como espacial.**

26. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
unsigned f( unsigned y, unsigned x){ // suponemos y >= x
    if (x==0 || y==x) return 1;
    return f(y-1, x-1) + f(y-1, x);
}
```

¿Cuál es la mejor complejidad temporal que se puede conseguir?

- $\mathcal{O}(y \cdot x)$**
- $\mathcal{O}(y)$
- $\mathcal{O}(x)$

27. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
unsigned f( unsigned y, unsigned x){ // suponemos y >= x
    if (x==0 || y==x) return 1;
    return f(y-1, x-1) + f(y-1, x);
}
```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

- $\mathcal{O}(1)$
- $\mathcal{O}(y)$**
- $\mathcal{O}(y^2)$

28. Se pretende implementar mediante programación dinámica iterativa la función recursiva:

```
int f( int x, int y ) {
    if( x <= y ) return 1;
    return x + f(x-1,y);
}
```

¿Cuál es la mejor complejidad espacial que se puede conseguir?

- $\mathcal{O}(x)$
- $\mathcal{O}(1)$**
- $\mathcal{O}(x^2)$

Se pretende obtener la complejidad temporal en el caso más desfavorable de la siguiente función:

```
int max( vector < int > > v ) {
    if (v.size() == 1)
        return v[0];
    int n = v.size();
    int j = n/2;
    while (j < n-1 && v[j] < v[j+1])
        j++;
    if (j < n-2)
        return max( v[j+1], v[j+2] );
    else
        return v[j];
}
```

Imagen de stocktrek/2011-01-14-10-26-27.jpg

¿Cuál de las siguientes formulaciones expresa dicho conteo?

Selecione una:

- $c_1(n) = \sum_{j=1}^{n-1} \left(\frac{n}{2}\right)^j \in \mathcal{O}(n \log n)$ ✓
- $c_2(n) = \sum_{j=1}^{n-1} (n - n/2)^j \in \mathcal{O}(n \log n)$ ✓
- Las otras dos opciones son también correctas.
- No contiene (equivalente a no marcar nada).

¿Cuál es la solución a la siguiente relación de recurrencia?

$$f(0) = 0$$

$$f(1) = 1$$

$$f(2) = f(1) + f(0) = 1$$

$$f(3) = f(2) + f(1) = 2$$

$$\vdots$$

$$f(n) = f(n-1) + f(n-2)$$

Selecione una:

- $f(n) \in \Theta(n \log n)$ ✓
- $f(n) \in \Theta(n^2)$
- Ninguna de las otras dos es correcta.

Quíz mi elección:

De los siguientes expresiones, ¿también dos son verdaderas otras dos:

Selecione una:

- $O(n^2) = O(2^{2n})$ ✓
- $O(n^2) = O(2^n)$ ✓
- $O(n^2) = O(2^{2n}) \leq O(2^n) \leq O(n^2)$
- $O(2^{2n}) \leq O(n^2) \leq O(2^n)$

Selecione una:

- No contiene (equivalente a no marcar nada).
- Es igual a la constante de crecimiento.
- Es menor que la constante de crecimiento.
- Es mayor que la constante de crecimiento.

Selecione una:

- $O(n^2) = O(2^{2n})$ ✓
- $O(n^2) = O(2^n)$
- $O(n^2) = O(2^{2n}) \leq O(2^n) \leq O(n^2)$
- $O(2^{2n}) \leq O(n^2) \leq O(2^n)$

Selecione una:

- Es igual a la constante de crecimiento.
- Es menor que la constante de crecimiento.
- Es mayor que la constante de crecimiento.

Selecione una:

- $O(n^2) = O(2^{2n})$ ✓
- $O(n^2) = O(2^n)$
- $O(n^2) = O(2^{2n}) \leq O(2^n) \leq O(n^2)$
- $O(2^{2n}) \leq O(n^2) \leq O(2^n)$

Selecione una:

- Es igual a la constante de crecimiento.
- Es menor que la constante de crecimiento.
- Es mayor que la constante de crecimiento.

Indica cuál es la complejidad en el peor caso de la función replace:

```
unsigned bound( const vector<int> & v ) {
    for( unsigned i = 0; i < v.size(); i++ )
        if( v[i] == 0 ) v[i] = 1;
    return v.size();
}

void replace( vector<int> & v, int n ) {
    for( unsigned i = 0; i < v.size(); i++ )
        v[i] = n;
}
```

Selecione una:

- a $O(n \log n)$
- b $O(n^2)$
- c $O(n)$

¿Cuál es la complejidad temporal de la siguiente función recursiva?

```
unsigned recursive( unsigned n ) {
    if( n == 1 ) return 1;
    unsigned sum = 2 * recursive( n / 2 ) + n * recursive( n / 2 );
    for( unsigned i = 1; i < n; i++ )
        sum += i;
    return sum;
}

T(n) = 2T(n/2) + \Theta(n^2)
```

Selecione una:

- a $\Theta(n^2)$
- b $\Theta(n^3)$
- c $\Theta(2^n)$

$$h^{\Theta(n^2)} = n$$

... donde $h = 2$ y $n = n$

$$\Theta(n) = O(n) + \Theta(n) \text{ OK}$$

$$\Theta(n) = O(n) + \Omega(n) \text{ Toda } \Theta(n) \text{ NO OK}$$

33. ¿Cuál es la definición correcta de $O(f)$?

- a $O(f) = \{g : N \rightarrow \mathbb{R}^+ \mid \forall c \in \mathbb{R}, \forall n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \leq cg(n)\}$
- b $O(f) = \{g : N \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq cf(n)\}$
- c $O(f) = \{g : N \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \leq cg(n)\}$

18. Sean dos funciones de coste $f : N \rightarrow \mathbb{R}^+$ y $g : N \rightarrow \mathbb{R}^+$ tales que $f \in O(g)$. De las siguientes situaciones, o bien dos pueden suceder y la otra no, o bien los no pueden suceder y la otra sí. Indica la que es diferente de las otras dos.

- a $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k, k \in \mathbb{R}^+, k \neq 0 \text{ OK (Si son del mismo orden)}$
- b $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \text{ NO OK}$
- c $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \text{ OK}$

39. Dada la relación de recursividad:

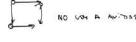
$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ pT\left(\frac{n}{k}\right) + g(n) & \text{en otro caso} \end{cases}$$

(donde p y a son enteros mayores que 1 y $g(n) = n^k$), ¿qué tiene que ocurrir para que se cumpla $T(n) \in \Theta(n^k \log_a n)$?

- a $p > a^k \quad h = p \quad p = a^k \quad MTO$
- b $p = a^k \quad b = a \quad p = a^k$
- c $p < a^k \quad k = k$

27. En una cuadrícula se quiere dibujar el contorno de un cuadrado de n casillas de lado. ¿Cuál será la complejidad temporal del mejor algoritmo que pueda existir?

- a $O(n)$
- b $O(n^2)$
- c $O(\sqrt{n})$



40. Si $T(n) \in O(n^3)$, ¿qué puede pasar que $f(n) \in O(n^2)$?

- a No, porque $n^3 \notin O(n^2)$
- b Solo para valores bajos de n
- c Es perfectamente posible, ya que $O(n^2) \subset O(n^3)$

15. El algoritmo de ordenación Quicksort divide el problema en dos subproblemas. ¿Cuál es la complejidad temporal asintótica de realizar esa división?

- a $O(n \log n)$
- b $O(n)$ y $O(n^2)$
- c $O(n)$

20. ¿Qué tienen en común el algoritmo que obtiene el k -ésimo elemento más pequeño de un vector (estudiado en clase) y el algoritmo de ordenación Quicksort?

- a El número de llamadas recursivas que se hacen.
- b La combinación de las soluciones a los subproblemas.
- c La división del problema en subproblemas.

21. ¿Cuál es el coste temporal asintótico de la siguiente función?

```
void f( int n, int arr[] ) {
    int i = 0, j = 0;
    for( ; i < n; ++i )
        while( j < n && arr[i] < arr[j] ) j++;
    i++;
}

(a)  $O(n^2)$ 
(b)  $O(n)$ 
(c)  $O(n \log n)$ 
```

18. Si $\lim_{n \rightarrow \infty} (f(n)/n^2) = k$, y $k \neq 0$, ¿cuál de estas tres afirmaciones es falsa?

- a $f(n) \in O(n^3) \subset \{f(n) \in \Theta(n^2)\}$
- b $f(n) \in \Theta(n^2)$
- c $f(n) \in \Theta(n^2) \subset \{f(n) \in \Theta(n^2)\}$

39. Dada la relación de recursividad:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ pT\left(\frac{n}{k}\right) + g(n) & \text{en otro caso} \end{cases}$$

(donde p y a son enteros mayores que 1 y $g(n) = n^k$), ¿qué tiene que ocurrir para que se cumpla $T(n) \in \Theta(n^k \log_a n)$?

- a $p > a^k$
- b $p = a^k$
- c $p < a^k$

25.

De las siguientes expresiones, o bien dos son ciertas y una es falsa, o bien al contrario, una es cierta y las dos son falsas. Marca la que en este sentido es diferente a las otras dos.

- a $\sum_{i=1}^n 2^i \in O(n \log n) \quad \text{y} \quad \sum_{i=1}^n 2^i \in \Theta(n^2) \quad \text{y} \quad \sum_{i=1}^n 2^i \in \Theta(n^3)$
- b $\sum_{i=1}^n \sum_{j=1}^{2^i} 2^j \in O(n^2) \quad \text{y} \quad \sum_{i=1}^n \sum_{j=1}^{2^i} 2^j \in \Theta(n^3)$
- c $\sum_{i=1}^{\log n} \sum_{j=1}^{2^i} 2^j \in O(n \log n) \quad \text{y} \quad \sum_{i=1}^{\log n} \sum_{j=1}^{2^i} 2^j \in \Theta(n^2)$

22. Una de estas tres situaciones no es posible:

- a $f(n) \in O(n)$ y $f(n) \in \Omega(1)$
- b $f(n) \in \Omega(n^2)$ y $f(n) \in O(n)$
- c $f(n) \in O(n)$ y $f(n) \in O(n^2)$

30. Marca el uso de una estrategia "divide y vencerás" la existencia de una solución de complejidad temporal polinómica a cualquier problema:

- a Si, en cualquier caso.
- b No
- c Si, pero siempre que la complejidad temporal conjunta de las operaciones de descomposición del problema y la combinación de las soluciones sea polinómica.

12. Sea $g(n) = \sum_{i=0}^k a_i n^i$. Di cuál de las siguientes afirmaciones es falsa

- a Las otras dos afirmaciones son ambas falsas.
- b $g(n) \in \Theta(n^K)$
- c $g(n) \in \Omega(n^K)$

13. Si $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ entonces ...

- a ... $f(n) \in O(g(n))$
- b ... $f(n) \in O(g(n)) \text{ OK}$
- c ... $f(n) \in O(f(g(n)))$

18. Sean dos funciones de coste $f : N \rightarrow \mathbb{R}^+$ y $g : N \rightarrow \mathbb{R}^+$ tales que $f \in O(g)$. De las siguientes situaciones, o bien dos pueden suceder y la otra no, o bien los no pueden suceder y la otra sí. Indica la que es diferente de las otras dos.

12. Sea $g(n) = \sum_{i=0}^k a_i n^i$. Di cuál de las siguientes afirmaciones es falsa

- a Las otras dos afirmaciones son ambas falsas.
- b $g(n) \in \Theta(n^K)$
- c $g(n) \in \Omega(n^K)$

13. Si $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ entonces ...

- a ... $f(n) \in O(g(n))$
- b ... $f(n) \in O(g(n)) \text{ OK}$
- c ... $f(n) \in O(f(g(n)))$

18. Sean dos funciones de coste $f : N \rightarrow \mathbb{R}^+$ y $g : N \rightarrow \mathbb{R}^+$ tales que $f \in O(g)$. De las siguientes situaciones, o bien dos pueden suceder y la otra no, o bien los no pueden suceder y la otra sí. Indica la que es diferente de las otras dos.

22. El algoritmo de ordenación MergeSort divide el problema en dos subproblemas. ¿Cuál es la complejidad temporal asintótica de realizar esa división?

- a $O(n)$
- b $O(n \log n)$
- c $O(1)$

33. Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$ entonces

- a $f^2 \in \Theta(g_1 g_2)$
- b Las otras dos opciones son ambas ciertas.
- c $f \in \Theta(\max(g_1, g_2))$

21. ¿Cuál es el coste temporal asintótico de la siguiente función?

```
void f( int n, int arr[] ) {
    int i = 0, j = 0;
    for( ; i < n; ++i )
        while( j < n && arr[i] < arr[j] ) j++;
    i++;
}

(a)  $O(n^2)$ 
(b)  $O(n)$ 
(c)  $O(n \log n)$ 
```

18. Si $\lim_{n \rightarrow \infty} (f(n)/n^2) = k$, y $k \neq 0$, ¿cuál de estas tres afirmaciones es falsa?

- a $f(n) \in O(n^3) \subset \{f(n) \in \Theta(n^2)\}$
- b $f(n) \in \Theta(n^2)$
- c $f(n) \in \Theta(n^2) \subset \{f(n) \in \Theta(n^2)\}$

39. Dada la relación de recursividad:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ pT\left(\frac{n}{k}\right) + g(n) & \text{en otro caso} \end{cases}$$

(donde p y a son enteros mayores que 1 y $g(n) = n^k$), ¿qué tiene que ocurrir para que se cumpla $T(n) \in \Theta(n^k \log_a n)$?

- a $p > a^k$
- b $p = a^k$
- c $p < a^k$

18. Sean dos funciones de coste $f : N \rightarrow \mathbb{R}^+$ y $g : N \rightarrow \mathbb{R}^+$ tales que $f \in O(g)$. De las siguientes situaciones, o bien dos pueden suceder y la otra no, o bien dos no pueden suceder y la otra sí. Indica la que es diferente a las otras dos.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k, \quad k \in \mathbb{R}^+, \quad k \neq 0$$

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

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

22. El algoritmo de ordenación Mergesort divide el problema en dos subproblemas. ¿Cuál es la complejidad temporal asintótica de realizar esa división?

- (a) $O(n)$
 (b) $O(n \log n)$
 (c) $O(1)$

33. Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$ entonces

(a) $f^2 \in \Theta(g_1 \cdot g_2)$

(b) Las otras dos opciones son ambas ciertas.

(c) $f \in \Theta(\max(g_1, g_2))$

22. Una de estas tres situaciones no es posible:

(a) $f(n) \in O(n)$ y $f(n) \in \Omega(1)$

(b) $f(n) \in \Omega(n^2)$ y $f(n) \in O(n)$

(c) $f(n) \in O(n)$ y $f(n) \in O(n^2)$

30. Garantiza el uso de una estrategia "divide y vencerás" la existencia de una solución de complejidad temporal polinómica a cualquier problema?

- (a) Sí, en cualquier caso.

(b) No.

(c) Poco siempre que la complejidad temporal conjunta de las operaciones de descomposición del problema y la combinación de las soluciones sea polinómica.

12. Sea $g(n) = \sum_{i=0}^n a_i n^i$. Di cuál de las siguientes afirmaciones es falsa:

- (a) Las otras dos afirmaciones son ambas falsas.

(b) $g(n) \in \Theta(n^k)$

(c) $g(n) \in \Omega(n^k)$

13. Si $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ entonces ...

(a) ... $f(n) \in \Theta(g(n))$

(b) ... $f(n) \in O(g(n))$

(c) ... $g(n) \in O(f(n))$

27. En una cuadrícula se quiere dibujar el contorno de un cuadrado de n casillas de lado. ¿cuál será la complejidad temporal del mejor algoritmo que pueda sacar?

- (a) $O(n^2)$
 (b) $O(n)$
 (c) $O(\sqrt{n})$

40. Si $f(n) \in O(n^3)$, ¿puede pasar que $f(n) \in O(n^2)$?

- (a) No, porque $n^3 \notin O(n^2)$
 (b) Sólo para valores bajos de n
 (c) Es perfectamente posible, ya que $O(n^2) \subset O(n^3)$

15. El algoritmo de ordenación Quicksort divide el problema en dos subproblemas. ¿Cuál es la complejidad temporal asintótica de realizar esa división?

- (a) $O(n \log n)$
 (b) $\Omega(n) \vee O(n^2)$
 (c) $O(n)$

20. ¿Qué tienen en común el algoritmo que obtiene el k -ésimo elemento más pequeño de un vector (estudiado en clase) y el algoritmo de ordenación Quicksort?

- (a) El número de llamadas recursivas que se hacen.
 (b) La combinación de las soluciones a los subproblemas.
 (c) La división del problema en subproblemas.

39. Dada la relación de recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ pT\left(\frac{n}{a}\right) + g(n) & \text{en otro caso} \end{cases}$$

(donde p y a son enteros mayores que 1 y $g(n) = n^k$). ¿Qué tiene que ocurrir para que se cumpla $T(n) \in \Theta(n^k \log_a n)$?

- (a) $p > a^k$
 (b) $p = a^k$
 (c) $p < a^k$

30. Si el coste temporal de un algoritmo es $T(n)$, ¿cuál de las siguientes situaciones es imposible?

- (a) $T(n) \in O(n) \wedge T(n) \in \Omega(n^2)$
 (b) $T(n) \in \Omega(n) \vee T(n) \in \Theta(n^2)$
 (c) $T(n) \in \Theta(n) \wedge T(n) \in \Omega(n^2)$

31. El coste temporal asintótico de invertir un elemento en un vector ordenado de forma que continúa ordenado.

- (a) ... $\Theta(n)$ a desplazamientos
 (b) ... $\Theta(\log n)$
 (c) ... $\Theta(n^2)$

32. ¿Qué nos proporciona la media entre el coste temporal asintótico (o complejidad temporal) en el peor caso y el coste temporal asintótico en el mejor caso?

- (a) El coste temporal promedio.
 (b) El coste temporal asintótico en el caso medio.

(c) Nada de interés.

33. El coste temporal asintótico del programa

$\theta(n)=0; \text{for}(i=0; i<n; i++) \text{ for}(j=i; j<n; j++) a+=i+j;$

som...

- (a) ... el del primero, menor que el del segundo.
 (b) ... el del segundo, menor que el del primero.

(c) Igual.

- (a) $T(n) \in \Omega(n^2) \wedge T(n) \in O(n^2)$
 (b) Siendo por encima de n^2 no está por debajo de n^2

- Un algoritmo recursivo basado en el esquema divide y vencerás

- Selección una:
 (a) es más eficiente que la más equitativa de la división en subproblemas

- (b) Las demás opciones son verdaderas.

- (c) nunca tendrá una complejidad exponencial → da porciones del coste de comp

Indica cuál es la complejidad de la función siguiente:

Un problema de tiempo $T(n)$ que divide el vector en tres partes y calcula la suma de cada parte.

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

Un problema de tiempo $T(n)$ que calcula la suma de los elementos de un vector de longitud n .

7. El algoritmo de ordenación *Mergesort* divide el problema en dos subproblemas. ¿Cuál es la complejidad temporal asintótica de realizar esa división?

- a) $O(1)$
- b) $O(n)$
- c) $O(n \log n)$

8. ¿Cuál es la definición correcta de $\Omega(f)$?

- a) $\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ | \exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} \text{ } \forall n \geq n_0, g(n) \geq c f(n)\}$
- b) $\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ | \exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \geq c g(n)\}$
- c) $\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ | \forall c \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \geq c g(n)\}$

34. La siguiente relación de recurrencia expresa la complejidad de un algoritmo recursivo, donde $g(n)$ es una función polinómica:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

Di cuál de las siguientes afirmaciones es cierta.

- a) Si $g(n) \in \Theta(n)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación *mergesort*.
- b) Si $g(n) \in \Theta(1)$ la relación de recurrencia representa la complejidad temporal del algoritmo de búsqueda dicotómica.
- c) Si $g(n) \in \Theta(n)$ la relación de recurrencia representa la complejidad temporal en el caso mejor del algoritmo de ordenación *quicksort*.



La siguiente relación de recurrencia expresa la complejidad de un algoritmo recursivo, donde $g(n)$ es una función polinómica:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

Di cuál de las siguientes afirmaciones es cierta:

- Selección una:
- a. Si $g(n) \in O(n^2)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación mediante inserción.
 - b. Si $g(n) \in O(1)$ la relación de recurrencia representa la complejidad temporal del algoritmo de búsqueda dicotómica.
 - c. Si $g(n) \in O(n)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación *Mergesort*.
 - d. No contesto (equivale a no marcar nada).



Pregunta 1

La complejidad temporal en el mejor de los casos...

Selección una:

- a. ... es una función de la talla que tiene que estar definida para todos los posibles valores de ésta.
- b. ... es el tiempo que tarda el algoritmo en resolver la talla más pequeña que se le puede presentar.
- c. Las demás opciones son verdaderas.

Pregunta 2

Sobre la complejidad temporal de la siguiente función:

```
unsigned desperdicio (unsigned n)
{
    if (n<=1)
        return 0;
    unsigned sum = desperdicio (n/2) + desperdicio (n/2) + desperdicio (n/2);
    for (unsigned i=1; i<n-1; i++)
        for (unsigned j=i; j<i; j++)
            for (unsigned k=1; k<j; k++)
                sum+=i*j*k;
    return sum;
}
```

Selección una:

Ninguna de las otras dos alternativas es cierta.

Las complejidades en los casos mejor y peor son distintas.

El mejor de los casos se da cuando $n \leq 1$ y en tal caso la complejidad es constante.

Pregunta 3

Con respecto al esquema *Divide y vencerás*, ¿es cierta la siguiente afirmación?

Si la talla se reparte equitativamente entre los subproblemas, entonces la complejidad temporal resultante es una función logarítmica.

Selección una:

No, nunca, puesto que también hay que añadir el coste de la división en subproblemas y la posterior combinación.

No tiene porque, la complejidad temporal no depende únicamente del tamaño resultante de los subproblemas.

Si, siempre, en *Divide y Vencerás* la complejidad temporal depende únicamente del tamaño de los subproblemas.

Pregunta 4

¿Qué cota se deduce de la siguiente relación de recurrencia?

$$f(n) = \begin{cases} 1 & n = 1 \\ n + 4f(n/2) & n > 1 \end{cases}$$

Selección una:

- a) $f(n) \in \Theta(n^2)$
- b) $f(n) \in \Theta(n)$
- c) $f(n) \in \Theta(n \log n)$

Pregunta 8

La versión de *Quicksort* que utiliza como pivote la mediana del vector...

Selección una:

- a. ... no presenta caso mejor y peor distintos para instancias del mismo tamaño.
- b. ... es más eficiente si el vector ya está ordenado.
- c. ... es la versión con mejor complejidad en el mejor de los casos.

Pregunta 9

El siguiente fragmento del algoritmo de ordenación *Quicksort* reorganiza los elementos del vector para obtener una subsecuencia de elementos menores que el pivote y otra de mayores. Su complejidad temporal, con respecto al tamaño del vector v , que está delimitado por los valores p , i y p_f , es...

```
x = v[p];
i = p+1;
j = p_f;
do {
    while( i<p_f && v[i] < x ) i++;
    while( v[j] > x ) j--;
    if( i <= j ) {
        swap( v[i], v[j] );
        i++;
        j--;
    }
} while( i < j );
swap( v[p], v[j] );
```

Nota: La función *swap* tiene una complejidad temporal constante.

Selección una:

- a. ... lineal en cualquier caso.
- b. ... cuadrática en el peor de los casos.
- c. ... lineal en el caso peor y constante en el caso mejor.

Pregunta 6

Sea $f(n) = n \log n + n$.

Selección una:

- a. ... $f(n) \in \Omega(n \log n)$
- b. ... $f(n) \in O(n \log n)$
- c. Las otras dos opciones son ciertas

Pregunta 7

Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces...

Selección una:

- a. Las otras dos alternativas son ciertas.
- b. $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$
- c. $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$

Pregunta 10

Dada la siguiente relación de recurrencia, ¿Qué cota es verdadera?

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

Selección una:

- a. $f(n) \in \Omega(2^n)$
- b. $f(n) \in \Theta(n^2)$
- c. $f(n) \in \Theta(n^{2^n})$

Pregunta 11

¿Cuál es la solución a la siguiente relación de recurrencia?

$$f(n) = \begin{cases} \Theta(1) & n = 0 \\ \Theta(1) + f(n/3) & n > 0 \end{cases}$$

pregunta

Seleccione una:

- a. $f(n) \in \Theta(\log n)$.
- b. $f(n) \in \Theta(n/3)$.
- c. Ninguna de las otras dos es cierta.

Pregunta 2

Incorrecta

¿Cuál de las formulaciones expresa mejor el coste temporal de la siguiente función?

```
int f(int n){
    int count = 0;
    for (int i = 2; i < n; i += 2)
        for (int j = 1; j < i; j+=2)
            count += 1;
    return count;
}
```

Seleccione una:

- a. $C_u(n) = \sum_{i=2}^{n/2} (1 + \log_2 i)$
- b. $C_v(n) = \sum_{i=1}^{n/2} (\log_2(n/2))$
- c. No contesto (equivalente a no marcar nada).
- d. $C_e(n) = \frac{\sum_{i=2}^{n-1} \log_2 2i}{2}$

Pregunta 12

Indica cuál es la complejidad, en función de n , del fragmento siguiente:

```
for( int i = n; i > 0; i /= 2 )
    for( int j = n; j > 0; j /= 2 )
        a += A[i][j];
```

pregunta

Seleccione una:

- a. $O(\log^2(n))$
- b. $O(n \log(n))$
- c. $O(n^2)$

Pregunta 3

Los algoritmos de ordenación Quicksort y Mergesort tienen en común ...

Seleccione una:

- a. ... que se ejecutan en tiempo $O(n)$.
- b. ... que aplican la estrategia de divide y vencerás.
- c. No contesto (equivalente a no marcar nada).
- d. ... que ordenan el vector sin usar espacio adicional.

Pregunta 1

Con respecto a la complejidad temporal de la siguiente función, ¿cuál de las siguientes afirmaciones es cierta?

```
unsigned long pot2_1(unsigned n){
    if (n==0)
        return 1;
    if (n%2==0)
        return pot2_1( n/2 ) * pot2_1( n/2 );
    else
        return 2 * pot2_1( n/2 ) * pot2_1( n/2 );
}
```

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. Las otras dos afirmaciones son ambas falsas.
- c. El coste temporal exacto de la función es $O(n)$.
- d. La complejidad temporal en el mejor de los casos es constante.

Pregunta 4

Sea la siguiente relación de recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

Si $T(n) \in O(n)$, ¿en cuál de estos tres casos nos podemos encontrar?

Seleccione una:

- a. $g(n) = 1$
- b. No contesto (equivalente a no marcar nada).
- c. $g(n) = n$
- d. $g(n) = \log n$

Pregunta 5

Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$ entonces

Seleccione una:

- a. Las otras dos opciones son ambas ciertas.
- b. No contesto (equivalente a no marcar nada).
- c. $f \in \Theta(\max(g_1, g_2))$
- d. $f^2 \in \Theta(g_1 \cdot g_2)$

Pregunta 8

¿Cuál de los siguientes algoritmos de ordenación necesita un espacio de almacenamiento adicional al vector que se ordena con complejidad $O(n)$?

Seleccione una:

- a. Mergesort.
- b. Bubblesort.
- c. No contesto (equivalente a no marcar nada).
- d. Quicksort.

Pregunta 6

La complejidad temporal (o coste temporal asintótico) en el mejor de los casos...

Seleccione una:

- a. ... es el tiempo que tarda el algoritmo en resolver la talla más pequeña que se le puede presentar.
- b. ... es una función de la talla, o tamaño del problema, que tiene que estar definida para todos los posibles valores de ésta.
- c. Las otras dos opciones son ambas verdaderas.
- d. No contesto (equivalente a no marcar nada).

Pregunta 9

Se quieren ordenar d números distintos comprendidos entre 1 y n . Para ello se usa un array de n booleanos que se inicializan primero a false. A continuación se recorren los d números cambiando los valores del elemento del vector de booleanos correspondiente a su número a true. Por último se recorre el vector de booleanos escribiendo los índices de los elementos del vector de booleanos que son true.

¿Es este algoritmo más rápido (asintóticamente) que el Mergesort?

Seleccione una:

- a. No contesto (equivalente a no marcar nada)
- b. Sólo si $d \cdot \log d > k \cdot n$ (donde k es una constante que depende de la implementación)
- c. No, ya que este algoritmo ha de recorrer varias veces el vector de booleanos.
- d. Si, ya que el Mergesort es $O(n \log n)$ y este es $O(n)$

Pregunta 7

Con respecto al parámetro n , ¿Cuál es la complejidad temporal de la siguiente función?

```
void f( unsigned n ){
    if( n < 1 ) return;
    for( int i = 0; i < n; i++ )
        for( int j = 0; j < n; j++ )
            for( int k = 0; k < n; k++ )
                cout << "*";
    for( int i = 0; i < 8; i++ )
        f( n / 2 );
}
```

Seleccione una:

- a. $\Theta(n^3)$
- b. No contesto (equivalente a no marcar nada).
- c. $\Theta(n^2 \log n)$
- d. $\Theta(n^3 \log n)$

Pregunta 10

pregunta

[] siguientes expresiones, o bien dos son verdaderas y una es falsa, o bien dos son falsas y una es verdadera. Marca la que (en este sentido) es distinta a las otras dos.

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $\Theta(\log_2(n)) = \Theta(\log_3(n))$
- c. $\Theta(\log(n^2)) = \Theta(\log(n^3))$
- d. $\Theta(\log^2(n)) = \Theta(\log^3(n))$

Pregunta 11

La siguiente relación de recurrencia expresa la complejidad de un algoritmo recursivo, donde $g(n)$ es una función polinómica:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

Di cuál de las siguientes afirmaciones es cierta:

Seleccione una:

- a. Si $g(n) \in \Theta(n^2)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación mediante inserción binaria.
- b. Si $g(n) \in \Theta(1)$ la relación de recurrencia representa la complejidad temporal del algoritmo de búsqueda dicotómica.
- c. Si $g(n) \in \Theta(n)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación Mergesort.
- d. No contesto (equivalente a no marcar nada).

Comenzado el martes, 21 de marzo de 2023, 15:19

Estado Finalizado

Finalizado el martes, 21 de marzo de 2023, 15:37

Tiempo empleado 18 minutos 19 segundos

Puntos 2,50/12,00

Calificación 2,08 de 10,00 (20,83%)

Pregunta 1

Incorrecta

Se puntuó -0,50 sobre 1,00

De las siguientes expresiones, o bien dos son verdaderas y una es falsa, o bien dos son falsas y una es verdadera. Marca la que (en este sentido) es distinta a las otras dos.

Seleccione una:

- a. $O(n^2) \subset O(2^{\log_2(n)}) \subset O(2^n)$
- b. No contesto (equivalente a no marcar nada).
- c. $O(4^{\log_2(n)}) \subseteq O(n^2) \subset O(2^n)$
- d. $O(2^{\log_2(n)}) \subseteq O(n^2) \subset O(n!)$ ✗

Pregunta 2

Sin contestar

Puntuó como 1,00

¿Cuál de las formulaciones expresa mejor el coste temporal de la siguiente función?

```
int f(int n){  
    int count = 0;  
    for (int i = n; i >= 2; i -= 2)  
        for (int j = 1; j < i; j*=2)  
            count += 1;  
    return count;  
}
```

Seleccione una:

- a. $\sum_{i=1}^{\log n/2} \sum_{j=1}^{2^i} \log_2 j$
- b. $\sum_{i=1}^{\log n/2} \log_2 2^i$
- c. No contesto (equivalente a no marcar nada).
- d. $\sum_{i=1}^{\log n/2} \log_2 n/2$

Pregunta 12

Se pretende obtener la complejidad temporal en el caso más desfavorable de la siguiente función.

```
int exa (vector < int > & v){  
    int i, sum = 0, n = v.size();  
    if (n > 0){  
        int j = n;  
        while (sum < 100 and j != 0 ){  
            j = j / 2;  
            sum = 0;  
            for (i = j; i < n; i++)  
                sum += v[i];  
        }  
        return j;  
    }  
    else return -1;  
}
```

¿Cuál de las siguientes formulaciones expresa dicho coste?

Seleccione una:

- a. Las otras dos opciones son ambas ciertas.
- b. $c_s(n) = n \sum_{j=1}^{\log n} \sum_{i=1}^{2^j} \left(\frac{1}{2}\right)^i \in O(n \log n)$
- c. $c_s(n) = \sum_{k=1}^{\log n+1} (n - n/2^k) \in O(n \log n)$
- d. No contesto (equivalente a no marcar nada).

Pregunta 3

Sin contestar

Puntuó como 1,00

La complejidad temporal en el mejor de los casos de un algoritmo recursivo...

Seleccione una:

- a. ... coincide con el valor del caso base de la ecuación de recurrencia que expresa la complejidad temporal del algoritmo.
- b. ninguna de las otras dos opciones es verdadera.
- c. ... es la complejidad temporal de las instancias que están en el caso base.
- d. No contesto (equivalente a no marcar nada).

Pregunta 4

Correcta

Se puntuó 1,00 sobre 1,00

Los algoritmos de ordenación Quicksort y Mergesort tienen en común ...

Seleccione una:

- a. ... que ordenan el vector sin usar espacio adicional.
- b. No contesto (equivalente a no marcar nada).
- c. ... que aplican la estrategia de divide y vencerás. ✓
- d. ... que se ejecutan en tiempo $O(n)$.

Pregunta 5

Correcta

Se puntuó 1,00 sobre 1,00

Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces...

Seleccione una:

- a. $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$
- b. $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$
- c. Las otras dos opciones son ambas ciertas. ✓
- d. No contesto (equivalente a no marcar nada).

Pregunta 6

Incorrecta

Se puntuó -0,50 sobre 1,00

La relación de recurrencia $T(n) = 1 + T(n - 1)$ si $n > 1$; $T(1) = 1$...

Seleccione una:

- a. Expresa el número de llamadas recursivas que hace el algoritmo Quicksort en el peor de los casos.
- b. Ninguna de las otras dos opciones es cierta. ✗
- c. No contesto (equivalente a no marcar nada).
- d. Expresa la complejidad temporal asintótica en el peor de los casos del algoritmo de búsqueda binaria.

Pregunta 7

Correcta

Se puntuó 1,00 sobre 1,00

Se pretende obtener la complejidad temporal en el caso más desfavorable de la siguiente función.

```
int exa (vector < int > & v){  
    int i, sum = 0, n = v.size();  
    if (n > 0){  
        int j = n;  
        while (sum < 100 and j != 0 ){  
            j = j / 2;  
            sum = 0;  
            for (i = j; i < n; i++)  
                sum += v[i];  
        }  
        return j;  
    }  
    else return -1;  
}
```

¿Cuál de las siguientes formulaciones expresa dicho coste?

- a. $c_s(n) = n \sum_{j=1}^{\log n} \sum_{i=1}^{2^j} \left(\frac{1}{2}\right)^i \in O(n \log n)$
- b. Las otras dos opciones son ambas ciertas. ✓
- c. $c_s(n) = \sum_{k=1}^{\log n+1} (n - n/2^k) \in O(n \log n)$
- d. No contesto (equivalente a no marcar nada).

Pregunta 8

Incorrecta

Se puntuó -0,50 sobre 1,00

Se quieren ordenar d números distintos comprendidos entre 1 y n . Para ello se usa un array de n booleanos que se inicializan primero a false. A continuación se recorren los d números cambiando los valores del elemento del vector de booleanos correspondiente a su número a true. Por último se recorre el vector de booleanos escribiendo los índices de los elementos del vector de booleanos que son true.

¿Es este algoritmo más rápido (asintóticamente) que el Mergesort?

Seleccione una:

- a. No, ya que este algoritmo ha de recorrer varias veces el vector de booleanos. ✗
- b. Sí, ya que el Mergesort es $\mathcal{O}(n \log n)$ y este es $\mathcal{O}(n)$.
- c. No contesto (equivalente a no marcar nada).
- d. Sólo si $d \ll \log n$ (donde \ll es una constante que depende de la implementación)

Pregunta 9

Incorrecta

Se puntuó -0,50 sobre 1,00

Con respecto al parámetro $\backslash(n\backslash)$, ¿Cuál es la complejidad temporal de la siguiente función?

```
void f( unsigned n ){
    if ( n < 1 ) return;
    for( int i = 0; i < n; i++ )
        for( int j = 0; j < n; j++ )
            for( int k = 0; k < n; k++ )
                cout << " ";
    for( int i = 0; i < 8; i++ )
        f( n / 2 );
}
```

Seleccione una:

- a. $\backslash\backslash\Theta(n^2 \log n)\backslash\backslash$
- b. No contesto (equivalente a no marcar nada).
- c. $\backslash\backslash\Theta(n^3 \log n)\backslash\backslash$
- d. $\backslash\backslash\Theta(n^3)\backslash\backslash$ x

Pregunta 10

Correcta

Se puntuó 1,00 sobre 1,00

Sea la siguiente relación de recurrencia:

$$\backslash(T(n) = \left\{ \begin{array}{ll} 1 & \text{si } n = 1 \\ 2T(\frac{n}{2}) + g(n) & \text{en otro caso} \end{array} \right. \}$$

Si $\backslash(T(n) \in O(n^2))$, ¿en cuál de estos tres casos nos podemos encontrar?

Seleccione una:

- a. $\backslash(g(n)=n^2)\backslash\checkmark$
- b. $\backslash(g(n)=\log n)\backslash$
- c. No contesto (equivalente a no marcar nada).
- d. $\backslash(g(n)=n)\backslash$

Pregunta 11

Correcta

Se puntuó 1,00 sobre 1,00

La siguiente relación de recurrencia expresa la complejidad de un algoritmo recursivo, donde $\backslash(g(n)\backslash)$ es una función polinómica:

$$\backslash(T(n) = \left\{ \begin{array}{ll} 1 & \text{si } n = 1 \\ 2T(\frac{n}{2}) + g(n) & \text{en otro caso} \end{array} \right. \}$$

Di cuál de las siguientes afirmaciones es falsa:

Seleccione una:

- a. Si $\backslash(g(n)\in \backslash\Theta(1)\backslash)$ la relación de recurrencia representa la complejidad temporal del algoritmo de búsqueda dicotómica. ✓
- b. Si $\backslash(g(n)\in \backslash\Theta(n)\backslash)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación Mergesort.
- c. Si $\backslash(g(n)\in \backslash\Theta(n)\backslash)$ la relación de recurrencia representa la complejidad temporal en el caso mejor del algoritmo de ordenación Quicksort.
- d. No contesto (equivalente a no marcar nada).

Pregunta 3

Correcta

Se puntuó 1,00 sobre 1,00

¿Qué tienen en común el algoritmo que obtiene el k -ésimo elemento más pequeño de un vector (estudiado en clase) y el algoritmo de ordenación Quicksort?

Seleccione una:

- a. La combinación de las soluciones a los subproblemas.
- b. La división del problema en subproblemas. ✓
- c. El número de llamadas recursivas que se hacen.
- d. No contesto (equivalente a no marcar nada).

Pregunta 4

Incorrecta

Se puntuó -0,50 sobre 1,00

De las siguientes expresiones, o bien dos son verdaderas y una es falsa, o bien dos son falsas y una es verdadera. Marca la que (en este sentido) es distinta a las otras dos.

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $O(4^{\log_2(n)}) \subset O(n) \subset O(2^n)$ x
- c. $O(2^{\log_2(n)}) \subset O(n^2) \subset O(n!)$
- d. $O(n^2) \subset O(2^{\log_2(n)}) \subset O(2^n)$ x

Pregunta 5

Incorrecta

Se puntuó 0,00 sobre 1,00

La siguiente relación de recurrencia expresa la complejidad de un algoritmo recursivo, donde $g(n)$ es una función polinómica:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

Di cuál de las siguientes afirmaciones es falsa:

Seleccione una:

- a. Si $g(n) \in O(n)$ la relación de recurrencia representa la complejidad temporal en el mejor caso del algoritmo de búsqueda del k -ésimo elemento más pequeño de un vector (estudiado en clase). x
- b. No contesto (equivalente a no marcar nada).
- c. Si $g(n) \in O(1)$ la relación de recurrencia representa la complejidad temporal del algoritmo de búsqueda dicotómica.
- d. Si $g(n) \in O(n^2)$ la relación de recurrencia representa la complejidad temporal del algoritmo de ordenación por inserción binaria.

Pregunta 2

Incorrecta

Se puntuó -0,50 sobre 1,00

Tenemos un vector ordenado de tamaño n_o y un vector desordenado de tamaño n_d y queremos obtener un vector ordenado con todos los elementos. ¿Qué será más rápido?

Seleccione una:

- a. Insertar los elementos del vector desordenado (uno a uno) en el vector ordenado. x
- b. Ordenar el desordenado y luego mezclar las listas.
- c. No contesto (equivalente a no marcar nada).
- d. Depende de si $n_o > n_d$ o no.

Pregunta 12

Incorrecta

Se puntuó -0,50 sobre 1,00

Las siguientes funciones calculan el valor de la potencia n -ésima de dos. ¿Cuál es más eficiente en cuanto a coste temporal?

```
unsigned long pot2_1(unsigned n){
    if (n==0) return 1;
    if (n%2==0) return pot2_1(n/2) * pot2_1(n/2);
    else return 2 * pot2_1(n/2) * pot2_1(n/2);
}

unsigned long pot2_2(unsigned n){
    if (n==0) return 1;
    return 2 * pot2_2(n-1);
}
```

Seleccione una:

- a. La segunda, pot2_2(n), es más eficiente que la otra.
- b. Las dos funciones son equivalentes en cuanto a coste temporal. ✓
- c. No contesto (equivalente a no marcar nada).
- d. La primera, pot2_1(n), es más eficiente que la otra. x

[← Entrenamiento_primer_parcial](#)

Ir a...

Pregunta 6

Incorrecta

Se puntuá -0,50 sobre 1,00

Sea la siguiente relación de recurrencia $T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$. Si $T(n) \in O(n)$, ¿en cuál de estos tres casos nos podemos encontrar?

- Seleccione una:
- a. $g(n) = n^2$
 - b. $g(n) = 1$
 - c. $g(n) = n$ ✘
 - d. No contesto (equivalente a no marcar nada).

Pregunta 7

Incorrecta

Se puntuá -0,50 sobre 1,00

¿Cuál es la relación de recurrencia que representa la complejidad en el peor caso del algoritmo de búsqueda del k -ésimo elemento más pequeño de un vector?

Seleccione una:

- a. $T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + 1 & \text{en otro caso} \end{cases}$
- b. $T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + n & \text{en otro caso} \end{cases}$
- c. $T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ T(n-1) + n & \text{en otro caso} \end{cases}$
- d. No contesto (equivalente a no marcar nada).

Pregunta 8

Incorrecta

Se puntuá -0,50 sobre 1,00

Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$ entonces

- Seleccione una:
- a. $f \in \Theta(g_1 + g_2)$
 - b. No contesto (equivalente a no marcar nada).
 - c. $f \in \Theta(g_1 \cdot g_2)$
 - d. $f \notin \Theta(\max(g_1, g_2))$ ✘

Pregunta 9

Correcta

Se puntuá 1,00 sobre 1,00

Las siguientes funciones calculan el valor de la potencia n -ésima de dos. ¿Cuál es más eficiente en cuanto a coste temporal?

```
unsigned long pot2_1(unsigned n){
    if (n==0) return 1;
    if (n%2==0) return pot2_1(n/2) * pot2_1(n/2);
    else return 2 * pot2_1(n/2) * pot2_1(n/2);
}

unsigned long pot2_2(unsigned n){
    if (n==0) return 1;
    return 2 * pot2_2(n-1);
}
```

Seleccione una:

- a. La primera, pot2_1(n), es más eficiente que la otra.
- b. La segunda, pot2_2(n), es más eficiente que la otra.
- c. Las dos funciones son equivalentes en cuanto a coste temporal. ✓
- d. No contesto (equivalente a no marcar nada).

Pregunta 10

Incorrecta

Se puntuá -0,50 sobre 1,00

Con respecto al parámetro n , ¿Cuál es la complejidad temporal de la siguiente función?

```
void f( unsigned n ){
    if( n < 2 ) return;
    for( int i = 0; i < pow(n,2); i++ )
        cout << "*";
    f( n - 2 );
}
```

Seleccione una:

- a. $\Theta(n^2)$
- b. No contesto (equivalente a no marcar nada).
- c. $\Theta(n^3)$
- d. $\Theta(n^2 \log n)$ ✘

Pregunta 11

Correcta

Se puntuá 1,00 sobre 1,00

La complejidad temporal (o coste temporal asintótico) en el mejor de los casos...

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. ... es el tiempo que tarda el algoritmo en resolver la talla más pequeña que se le puede presentar.
- c. ... es una función de la talla, o tamaño del problema, que tiene que estar definida para todos los posibles valores de ésta. ✓
- d. Las otras dos opciones son ambas verdaderas.

Pregunta 12

Correcta

Se puntuá 1,00 sobre 1,00

¿Cuál de las formulaciones expresa mejor el coste temporal de la siguiente función?

```
int f(int n) {
    int count = 0;
    for (int i = 2; i < n; i += 2)
        for (int j = 1; j < i; j*=2)
            count += 1;
    return count;
}
```

Seleccione una:

- a. No contesto (equivalente a no marcar nada).
- b. $C_e(n) = \frac{\sum_{i=2}^{n-1} \log_2 2i}{2}$
- c. $C_e(n) = \sum_{i=2}^{n/2} (\log_2(n/2))$
- d. $C_e(n) = \sum_{i=1}^{n/2} (1 + \log_2 i)$ ✓

→ [Entrenamiento_primer_parcial](#)

Ir a...

Segundo parcial ▶