drprobsalgoritmia.pdf

Anónimo

Algoritmia y Estructuras de Datos Avanzadas

2º Grado en Ingeniería Informática

Escuela Politécnica Superior
Universidad Autónoma de Madrid

Algoritmos y Estructuras de Datos Avanzadas

Problemas

Segundo Curso, Grado en Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

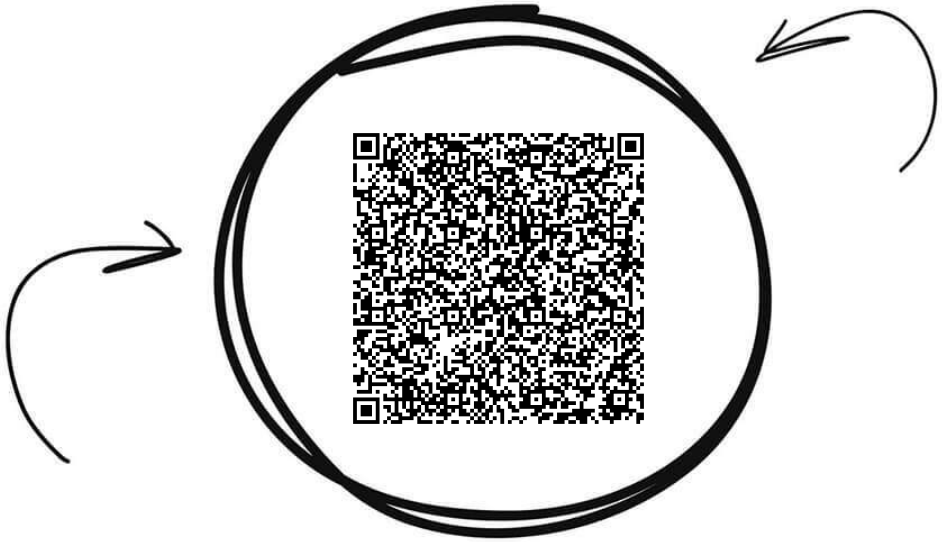
Contents

5€ DE BIENVENIDA

Con esta promo, te llevas **5€** por
tu cara bonita al subir **3 apuntes**
a Wuolah Wuolitah

WUOLAH

Algoritmia y Estructuras de...



Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

1 Problemas básicos

1. (**P; Rec**) Escribir un algoritmo para encontrar el segundo mayor elemento de una lista con n elementos. ¿Cuántas comparaciones haría como mucho?

2. (**P; Rec**) Escribir un algoritmo que nos devuelva los elementos mayor y menor de una lista con n elementos. ¿Cuántas comparaciones haría como mucho?

3. (**P; Rec**) Queremos evaluar en un valor `x` un polinomio de grado `n` con coeficientes dados por una tabla `c[0], c[1], ..., c[n]` . Dar un algoritmo iterativo que vaya evaluando y acumulando sucesivamente los términos del polinomio. ¿Cuántas multiplicaciones hará?

4. (**P; Rec**) El siguiente algoritmo, conocido como Regla de Horner, proporciona la evaluación de un polinomio de grado `n` y coeficientes `c[0], c[1], ..., c[n]` en un punto `X`:

   ```
   def Horner(n, c, x):
       p = c[n]
       para i de n-1 a 0:
           p = p * x + c[i]
       devolver p
   ```

   Comprobar que el algoritmo es correcto y estimar su rendimiento en número de multiplicaciones y compararlo con el del algoritmo "simple" de un problema anterior.

5. (**E; Python.** Muchas funciones matemáticas tienen definiciones **recursivas**. Una de estas funciones es la **factorial** $f(N) = N!$ que podemos definir como $f(1) = 1$, $f(N) = N \times f(N-1)$ si $N > 1$. Escribir una función Python recursiva `factorial_rec(n)` para calcularlo.

6. (**P; Python.**) Aunque elegantes, las funciones recursivas pueden ser bastante caras desde el punto de vista computacional y las alternativas no recursivas son muy deseables (de hecho, las CPU de los ordenadores no pueden entender las instrucciones recursivas y los lenguajes ensambladores no las tienen). Escribir una versión iterativa de `factorial(n)` que no utilice memoria adicional.

7. (**P.**) **Números de Fibonacci y biología cunicular**. Supongamos que los conejitos se hacen adultos en un mes y que una pareja de conejos adultos, sea cual sea su sexo, producen una pareja de conejitos al mes. Si dos conejitos náufragos llegan a una isla desierta, ¿cuántas parejas de conejos habrá en la isla tras N meses? (por ejemplo, tras dos meses habría dos parejas, una de conejitos y otra de conejos adultos).

8. (**P; Python.**) Los números **Fibonacci** F_n se definen recursivamente como $F_0 = F_1 = 1$ y $F_n = F_{n-1} + F_{n-2}$ para $n \geq 2$. Escribir una función Python recursiva `fibonacci_rec(n)` para calcularlas.

9. (**P; Python.**) Escribir una versión iterativa de `fibonacci(n)` que en cada paso guarde los números de Fibonacci recién calculados para que no se necesiten llamadas recursivas.

   Sugerencia: utilizar primero un dict para esto; luego, tratar de eliminarlo.

10. (**E; Python.**) Modificar nuestro código Hanoi para escribir una función `hanoi_count(n_discos, a=1, b=2, c=3)` que devuelva el número de movimientos de disco que ejecutaría el algoritmo.

    A continuación ejecutarlo para diferentes valores de `n` e intentar estimar una función $f(n)$ que dé dicho número de movimientos.

11. (**P; Rec.**) Para quedarnos tranquilos acerca de cuándo acabará el mundo, supongamos que los monjes de Hanoi mueven un disco por segundo sin parar para comer, dormir o lo que sea. ¿Cuantos segundos necesitarán para mover los 64 discos? Calcular primero una aproximación a dicho número mediante potencias de 10 aplicando la famosa ecuación informática $2^{10} = 10^3$ y a continuación escribir dicho número en letras.

12. (**E; Rec.**) **¿Y si han empezado ya y son muy rápidos?** Suponed que los monjes han empezado sus movimientos a las 0:00 del 1 de enero de 2000. ¿A qué velocidad deben mover los discos para acabar su trabajo antes de las 0:00 del 1 de enero de 2100?

13. (**P; Rec.**) El presidente de Ruritania tiene un problema de filtraciones a la prensa: uno de sus 100 empleados no para de filtrar noticias. Para descubrirlo, su CIO le propone dar diferentes presuntas noticias a los miembros del personal y comprobar si la información acaba en las noticias hasta que lo descubran (saben que el filtrador filtrará cualquier historia que le den y que los no filtradores nunca lo harán). Como inventarse noticias es difícil, quieren minimizar el número de noticias que necesiten. Diseñar una estrategia que, según lo anterior, utilice un número mínimo de noticias. ¿Cuál será dicho número?

2 Tipos abstractos de datos

1. (**E; Rec**) Al igual que hemos definido min heaps, se podrían definir max heaps como heaps donde cada nodo es mayor que sus hijos. Reescribir las funciones Python `heapify(h: List, i: int)` y `insert(h: List, k: int)` para que funcionen en max heaps.

2. (**E; Rec**) Insertar los elementos de la lista `[7, 1, 6, 2, 5, 3, 4]` tanto en un min heap como en un max heap indicando en cada caso el estado del correspondiente heap según se van insertando los elementos.

3. (**E; Rec**) Nos han dado los elementos de la lista del problema anterior como un array, que queremos convertir en un min heap **sin usar memoria auxiliar**. Para ello vamos a aplicar min heapify de izquierda a derecha y de abajo arriba sobre los nodos internos de la representación en un heap del array (esto es, aquellos nodos que no son hojas). Indicar en suficiente detalle la evolución del heap según va avanzando el algoritmo.

4. (**E; Rec**) Suponiendo que vamos a usar tablas como estructuras de datos para min y max heaps, indicar el estado de las correspondientes tablas tras las inserciones de la lista `[7, 1, 6, 2, 5, 3, 4]` del problema anterior.

5. (**E; Rec**) Tenemos una lista sobre la que queremos crear un min heap "in place", esto es, sin usar memoria auxiliar. Una forma de hacerlo es localizar el primer elemento de la lista que no sería una hoja en la representación de la lista como un heap, y aplicar `heapify` a ese nodo y, sucesivamente, a todos sus anteriores hasta llegar a la raíz del heap (esto es, al primer nodo de la lista). Comprobar que esta idea funciona aplicándola a las listas `[7, 6, 5, 4, 3, 2, 1]` y `[1, 7, 2, 6, 3, 5, 4]`.

6. (**P; Rec**) Si tenemos un heap con `n` elementos e índices entre `0` y `n-1`, ¿cuál es el índice del primer nodo que no es hoja? ¿Y cuál es el índice de la primera hoja?

7. (**E; Rec**) Vamos a insertar los elementos de la lista `[7, 1, 6, 2, 5, 3, 4]` en una cola de prioridad sobre un min heap. Dar la evolución del estado de dicha cola según se van insertando los elementos de la lista.

8. (**E; Rec**) Tras construir la cola de prioridad anterior, vamos a extraer de la misma tres elementos y reinsertarlos a continuación en la misma cola de prioridad. Indicar el estado de la cola tras cada extracción y reinserción.

9. (**P; Rec**) El problema de selección consiste en encontrar el elemento de una tabla T posiblemente desordenada que ocuparía ls posición k en una ordenación de T. Un algoritmo para ello podría consistir en ir leyendo los sucesivos elementos de T manteniendo en cada momento en un max heap H los menores k elementos leídos hasta ese momento. Argumentar que de hacerlo así, el elemento buscado es el mayor elemento del heap H obtenido en el último paso.

10. (**P; Rec**) Suponiendo disponibles las primitivas `insert, remove` para max heaps, escribir el pseudocódigo de un tal algoritmo y estimar su coste en función de k y del tamaño n de la tabla.

11. (**E; Rec**) Dar la evolución del algoritmo del problema anterior si se aplicara para seleccionar el tercer elemento en la tabla `[7, 6, 5, 4, 3, 2, 1]`, indicando para cada inserción el estado del max heap.

12. (**P+**) Una posible EdD para el TAD CD puede obtenerse almacenando los elementos de los subconjuntos en listas enlazadas. Para una mayor eficacia de la unión, dichas listas se acceden desde una tabla cuyos índices se identifican con los elementos del conjunto universal base U. En una entrada de dicha tabla de índice i se almacena

    - un puntero al primer elemento de la lista donde se almacena el elemento i (dicho primer elemento es el representante del conjunto almacenado en la lista);
    - un puntero al último elemento de la última lista de la que i fue representante;
    - el tamaño de la última lista de la que i fue representante.

    Dar el pseudocódigo de las primitivas del TAD CD que utilicen la EdD recién mencionada.

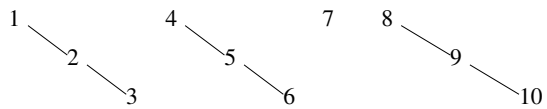
13. (**P; Rec**) ¿Cuál sería el coste de `find` y `union` en la implementación anterior?

14. (**P+**) Si sobre la EdD anterior se efectúan siempre uniones por tamaños (esto es, la lista más corta se añade a la más larga), demostrar que el coste conjunto de todas las uniones posibles es $\mathrm{O}(NlogN)$ con $N = |\mathcal{S}|$).

    Sugerencia: observar que el coste lo determina el número de veces que se cambia el puntero de un nodo e intentar estimar ese número de cambios para un nodo concreto considerando el tamaño de los conjuntos del nodo antes y después de un cambio de punteros.

15. (**E; Rec**) El bosque inferior representa el estado en un cierto momento de un conjunto disjunto. Suponiendo que los rangos de los árboles coinciden con sus alturas, indicar la evolución de dicho bosque tras efectuar las siguientes operaciones utilizando unión por rangos y compresión de caminos:

    1. `union(find(2), find(6))`
    2. `union(find(3), find(7))`
    3. `union(find(6), find(10))`

    Representar el bosque inicial en una tabla y rehacer dicha evolución sobre la misma.

16. (**P; Rec**) Justificar el que al aplicar unión por alturas la profundidad de un árbol de conjunto S cualquiera es $\lg |S|$.

    Sugerencia: teoría

17. (**P+; Rec**) Justificar el que al aplicar unión por rangos la profundidad del árbol asociado a un conjunto S cualquiera es $\leq \lg |S|$.

    Sugerencia: usar que altura $\leq$ rango y argumentar que $r(T) \leq 1 + \lg |T|$ razonando como en el caso de la unión por alturas

18. (**P; Rec**) Una alternativa a la unión por rangos es la unión por tamaños: en cada unión, el árbol con menos nodos se añade al otro (en caso de empate se une el segundo al primero). Sobre el esquema en tabla `S[]` para representar los distintos conjuntos como árboles, ¿cómo se podría incorporar la información relativa al tamaño de los árboles?

19. (**P+**) Comprobar que empleando unión por tamaños sobre un conjunto de N elementos, también se cumple que para M finds y K uniones, se tiene $n(M, N) = \mathrm{O}(K + M \log N)$.

    Sugerencia: razonar como en el caso de la unión por alturas para demostrar primero que aquí también $p(T) \leq \lg |T|$) y completar con esto el resto del argumento.

20. (**Python; Rec**) Dar una versión Python recursiva de la función `find` con compresión de caminos. Eliminar a continuación dicha recursión y simplificar lo más posible el código resultante.

21. (**P+**) Sabemos que el número máximo de `unions` que se pueden hacer en un CD con n elementos hasta llegar a un único conjunto es $n - 1$, pero ¿cuál podría ser el número mínimo?

22. (**E; Rec**) Calcular $\lg^*$ para los siguientes valores: $1024, 2048, 10^3, 10^6$. Hacerlo sin calcular explícitamente los logaritmos involucrados salvo en el caso que se apliquen a potencias de 2.

23. (**P; Rec**) La EPS quiere mandar dos estudiantes a una competición de programación, uno de los cuáles tiene que ser experto en algoritmos y el otro en EdDs. Sin embargo, en vez de tener una lista de estudiantes expertos en algoritmos y otra de expertos en EdDs, lo que tiene es una lista de parejas `(a, b)` donde los dos estudiantes o bien son expertos en algoritmos o bien lo son en EdDs; ningún estudiante puede estar a la vez en las dos listas. ¿Cómo podría la EPS derivar una lista de estudiantes expertos en algoritmos y otra de expertos en EdDs?

24. (**E; Rec**) Aplicar el método de resolver el problema anterior a la lista

    `[(0, 1), (2, 0), (4, 6), (4, 7), (7, 5), (0, 3), (4, 6), (3, 1), (4, 5), (0, 2)]`.

25. (**P; Rec**) Estamos preparando una boda y queremos sentar juntos por un lado a la familia del novio, por otro a la de la novia y por otro a los amigos pero, de nuevo, no tenemos listas de cada categoría sino una lista de parejas `(a, b)` donde los dos miembros o son familia del novio, o de la novia, o a amigos de la pareja. Como la lista no abarca todas las posibles parejas de cada categoría, suponemos que esas relaciones son transitivas. ¿Cómo podríamos obtener el número de invitados familia del novio, familia de la novia o amigos?

26. (**E; Rec**) Aplicar el método de resolver el problema anterior a las listas `[(0, 12), (10, 0), (7, 12), (1, 9), (3, 8), (3, 9), (11, 6), (2, 5), (4, 5), (0, 2)]`.

27. (**E; Rec**) Dibuja tres grafos con 7 vértices y 3, 5, y 7 componentes conexas respectivamente, represéntalos mediante listas de ramas y aplica a continuación el algoritmo basado en CDs para encontrar dichas componentes.

28. (**E; Rec**) Usar CDs para encontrar las CC de los grafos asociados a las listas

    `[(0, 2), (1, 3), (2, 3), (2, 4), (3, 4), (5, 6), (5, 7), (6, 8)]`,

    `[(0, 1), (2, 0), (4, 6), (4, 7), (7, 5), (0, 3), (4, 6), (3, 1), (4, 5), (0, 2)]`.

29. (**P; Rec**) ¿Puedes ver alguna relación entre los problemas anteriores que justifique los métodos usados para su solución?

3 Algoritmos codiciosos

1. (**E; Rec**) Dar la solución al problema de la mochila fraccionaria para las sucesiones de pesos `[1, 2, 3, 4, 5]`, valores `[5, 4, 3, 2, 1]` y cota `11`.

2. (**P; Rec**) ¿Cuál podría ser la sucesión supercreciente más pequeña?
   Sugerencia: si $a_n > \sum_1^{n-1} a_i$, *entonces* $a_n \geq 1 + \sum_1^{n-1} a_i$.

3. (**E+; Rec**) Encontrar un código prefijo óptimo para un archivo con los caracteres `a, b, c, d, e, f, g, h`, todos con una misma frecuencia 10 (en caso de coincidencia de frecuencias, unir los dos primeros elementos por orden alfabético).

4. (**P; Rec**) En general, dado un archivo con 2^N elementos equiprobables, dar razonadamente la estructura del correspondiente código prefijo.

   Si codificamos mediante Huffman un tal archivo, ¿cuántos bits tendrá dicha codificación?

5. (**E; Rec**) Construir el código Huffman de compresión máxima para un fichero con los siguientes caracteres y frecuencias:

   ```
   caracter:     a   b   c   d   e   f
   frecuencia:  13   8   5   3   2   1
   ```

   indicando razonadamente en cada paso los distintos árboles de metacaracteres. ¿Cuántos bits tendrá dicha codificación?

6. (**P; Rec**) En el problema anterior las frecuencias de los caracteres se corresponden con los primeros números de Fibonacci. En general, si en un fichero con N caracteres la frecuencia del carácter i–ésimo es F_i, ¿cuál sería su codificación Huffman?

7. (**P+; Rec**) Si codificamos mediante Huffman un archivo como el anterior, ¿cuántos bits tendrá dicha codificación?

8. (**E; Rec**) Un fichero consta de dos símbolos, uno con probabilidad 0.9999 y otro con probabilidad 0.0001. ¿Cuáles son las longitudes de los códigos Huffman? ¿Cuáles son las cotas para las longitudes de los códigos Shannon? ¿Cuál es la entropía de la distribución de símbolos? Comentar los resultados bajo un punto de vista de compresión.

9. (**E; Rec**) Encontrar el código Huffman para una distribución de frecuencias (1/3, 1/5, 1/5, 2/15, 2/15). Argumentar que dicho código también es óptimo para la distribución (1/5, 1/5, 1/5, 1/5, 1/5).

10. (**E; Rec**) ¿Cuáles de los siguientes códigos NO puede ser de Huffman: $\{0, 10, 11\}, \{00, 01, 10, 110\}, \{01, 10\}$?

11. (**P; Rec**) El código $\{0, 01\}$, ¿es prefijo?, ¿es descodificable?

12. (**E**) Encontrar los códigos de Shannon para las distribuciones de frecuencia (1/3, 1/5, 1/5, 2/15, 2/15) y (1/5, 1/5, 1/5, 1/5, 1/5).

13. (**E**) Además de variantes por reflexiones, un archivo puede tener dos códigos Huffman con longitudes distintas para los caracteres. Ilustrar esta situación para un archivo de 4 símbolos y probabilidades respectivas (1/3, 1/3, 1/4 y 1/12) deshaciendo posibles empates de manera distinta en distintas codificaciones. Calcular las longitudes teóricas óptimas y observar que para algún código, su longitud óptima Huffman puede ser mayor que su longitud teórica.

14. (**P; Rec**) Argumentar cómo se podría establecer una codificación a la Huffman sobre un alfabeto ternario (esto es, de 3 símbolos). Aplicar dicha codificación a la distribución (1/21, 2/21, 3/21, 4/21, 5/21, 6/21).

15. (**P; Rec**) Dar una versión de la desigualdad de Kraft para códigos ternarios y, en general, para códigos sobre un alfabeto de M símbolos.

16. (**P+**) ¿Para qué árboles binarios es estricta la desigualdad de Kraft?

17. **P.** ¿Da nuestro algoritmo codicioso para el problema del viajante un circuito óptimo para el grafo `["madrid", "barcelona", "sevilla", "valencia"]` de las transparencias? Encuentra ejemplos de entradas para las que se devuelve un circuito no óptimo.

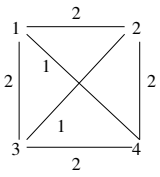
18. **E; Python.** Añadir `["vigo", "bilbao", "malaga"]` a nuestra lista de ciudades junto con sus respectivas distancias (se pueden buscar por internet) y aplicar el algoritmo codicioso para obtener un circuito. ¿Es óptimo?

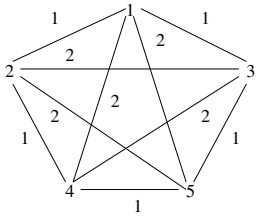
19. (**P+; Rec**) Se dice que un procesador funciona en modo no preemptivo si una vez que un trabajo toma el control del procesador, no lo cede hasta que termina. Dar un algoritmo que recibiendo N trabajos $T_1, T_2, \ldots, T_N$ de tiempos de ejecución t_i, $i = 1 \ldots N$, proporcione un orden de ejecución que minimize el tiempo medio de espera hasta la finalización. (Observar que de entrar en el orden inicial, el primer trabajo tarda T_1, el segundo $T_1 + T_2$ y así sucesivamente.)

    Sugerencia: diseñar un algoritmo codicioso razonable y demostrar a continuación que proporciona una solución correcta.

20. (**E; Rec**) Argumentar que los grafos inferiores son euclídeos y dar para ellos una solución aproximada del problema del viajante indicando los pasos intermedios necesarios y efectuándolos también por inspección.

    Dar también cotas inferiores del coste de una solución óptima del TSP.



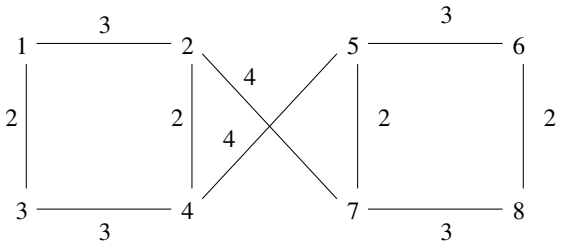


21. (**E; Rec**) Suponiendo que el grafo inferior es completo y que el coste de las ramas no dibujadas es 4,

    (a) Argumentar que dicho grafo es euclídeo.

    (b) Dar una solución aproximada del problema del viajante indicando los pasos intermedios necesarios y efectuándolos también por inspección.

    (c) Dar una cota inferior del coste de una solución óptima del TSP.



4 Algoritmos recursivos

1. (**P; Rec**) ¿Cómo se podrían multiplicar dos números complejos realizándose solamente tres multiplicaciones de números reales?

2. (**P; Rec**) Se ha demostrado que se pueden multiplicar dos matrices de tamaño 68×68 usando 132,464 productos, dos matrices 70×70 realizando 143,640 productos y 72×72 mediante 155,424 productos. ¿Cuál de estos métodos produciría un rendimiento asintótico mejor? ¿Cómo se comparan con el método de Strassen?

3. (**P; Rec**) El profesor Tragacanto nos informa que ha descubierto una manera de multiplicar matrices 3×3 mediante un cierto número K de productos y que su algoritmo mejora el de Strassen. ¿Cuál sería el valor máximo de un tal K?

4. (**P; Rec**) ¿Cuántas multiplicaciones realiza el algoritmo estándar para multiplicar matrices 4×4? ¿Cuántas realiza Strassen?

5. (**P; Rec**) La compañía DeepMind acaba de encontrar una forma de multipicar mtrices 4×4 mediante 47 multiplicaciones. ¿Cuántas multiplicaciones hará su algoritmo sobre matrices $N \times N$?

6. (**P++**) Demostrar que cualquier algoritmo de determinación del máximo de una tabla de N elementos mediante cdcs ha de efectuar como mínimo $N - 1$ cdcs. Concluir que el algoritmo habitual es óptimo.

7. (**P; Rec**) ¿Cuántas cdcs son necesarias para calcular la mediana de 5 elementos? *Hint:* `MergeSort` *ordena una tabla en 8 cdcs como máximo, pero se puede mejorar.*

8. (**E; Rec**) Dar la evolución para $K = 11$ de los algoritmos `QuickSelect` y `QuickSelect5` sobre la tabla `15 3 7 2 12 9 1 6 14 11 4 8 13 5 10`. Usar una rutina `partir` aproximada que toma como pivote el primer elemnto de una tabla, lo sitúa en su posición y delante y detrás del pivote sitúa los elementos menores y mayores, respectivamente, **en el mismo orden que en la tabla original**.

9. (**E; Rec**) Indicar la evolución del algoritmo `QSelect` para la determinación del 9 elemento de la tabla `[15 3 7 2 12 9 1 6 14]` indicando los índices y posición a buscar en las sucesivas llamadas a `QSelect`, así como los resultados de las llamadas a las rutinas `Pivote5` y `Partir`.

10. (**P++**) ¿Cuál sería la eficacia en el caso peor de QuickSort si la selección del elemento medio se hiciera como en QuickSelect5?

11. **(P; Rec)** Supongamos que una tabla ordenada de forma ascendente y sin elementos repetidos se rota sobre un pivote que desconocemos de antemano. Dar un algoritmo para encontrar el elemento mínimo de una tal tabla en tiempo $O(\log N)$.

12. **(P; Rec) Subiendo escaleras.** ¿Cuantas formas distintas hay de subir una escalera de n peldaños si los subimos bien de uno en uno o bien de dos en dos?

13. **(P+)** Los habitantes de Escalera son famosos por su capacidad de subir de golpe varios peldaños de una escalera. Además les gusta especializarse en subir un número determinado de peldaños, como por ejemplo `[1, 3, 5]`.

    Nos surge la curiosidad de saber de cuántas maneras distintas podría subir una escalera de 10 peldaños un escalerano especializado en subir `[1, 3, 5]` peldaños. Nótese que el orden de los peldaños es importante.

    O más generalmente, si un escalerano se especializa en subir `l = [p_1, p_2, ..., p_K]` peldaños, dar una función que devuelva de cuántas maneras diferentes podrá subir una escalera de `N` peldaños.

14. **(P+) La galería de tiro.** La federación de tiro con pistola de Ruritania nos ha encargado el diseño de una galería de tiro donde la distribución de los tiradores se ha de hacer de la siguiente manera:

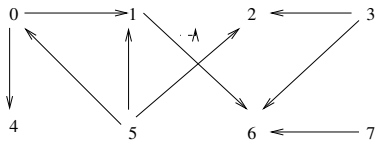
- El primer tirador que llegue ha de situarse en uno de los dos extremos izquierdo o derecho de la misma.
- Los demás han de situarse de manera que maximicen el número de puestos de tiro vacíos a su izquierda o a su derecha.
- No se admiten más tiradores si no es posible tener al menos un puesto vacío a la izquierda o a la derecha de los ya admitidos.

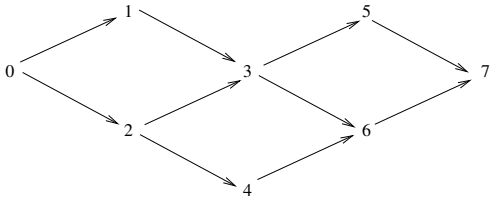
Por ejemplo, si hay 6 puestos, los dos primeros tiradores se situarán en los extremos mientra que un tercero podrá ponerse en el tercer o cuarto puesto, tras lo que no será posible acomodar más tiradores. Pero si hay 8 puestos, se pueden acomodar 4 tiradores, dos en los extremos y,por ejemplo, uno en el cuarto puesto y otro en el sexto, mientras que para nueve puestos se pueden acomodar hasta 5 tiradores. La federación no está segura de su presupuesto por lo que nos pide para un cierto número N de espacios cuál va a ser el número máximo de tiradores a acomodar.

Dar un algoritmo que calcule dicho número.

5 Grafos

1. (**E**) Para los siguientes grafos, dar su representación mediante una matriz de adyacencia y una lista de adyacencia.





2. (**E**) Junto a la lista de adyacencia de un grafo $G = (V, R)$ se puede definir su lista de incidencia, en la que `L[i]` apunta a los vértices v_j tales que $(v_j, v_i) \in R$. Escribir las listas de incidencia de los grafos del problema anterior. Definir de manera análoga la matriz de incidencia y calcularla en los mismos ejemplos.

3. (**E; Rec**) Escribir una función Python que reciba la lista de adyacencia de un grafo y devuelva su lista de incidencia, usando para ello una estructura de datos adecuada.

4. (**E; Rec**) Escribir una función Python que reciba la matriz de adyacencia de un grafo y devuelva su matriz de incidencia.

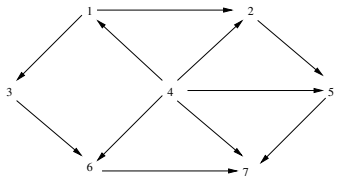
5. (**E; Rec**) La *incidencia* (o *grado*) de un vértice se define como el número de ramas que llegan a él. Escribir elpsuedocódigo de una función que calcule la tabla `inc[]` de incidencias en los vértices de un grafo a partir de la representación de un grafo en una lista de adyacencia.

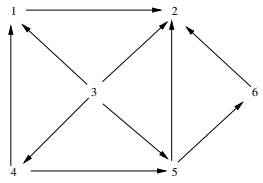
6. (**P; Rec**) Demostrar que en un grafo no dirigido $G = (V, R)$ se cumple que $\sum_{u \in V} inc(u) = 2|R|$. Deducir que un grafo no dirigido tiene un número par de vértices de incidencia impar.

7. (**P; Rec**) Demostrar que la cantidad de personas que un día dado da la mano un número impar de veces es un número par.

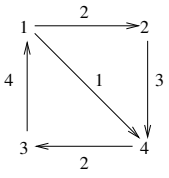
8. (**P+; Rec**) El complemento de un grafo $G = (V, R)$ es un grafo $G' = (V, R')$ en el que $(u, v) \in R'$ sii $(u, v) \notin R$. Dar una función Python que reciba la lista de adyacencia de un grafo y devuelva la de su complemento.

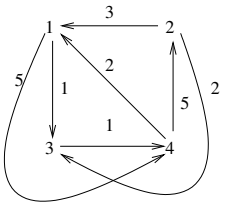
9. (**P; Rec**) Dar una función Python que reciba la matriz de adyacencia de un grafo y devuelva la de su complemento.
10. (**P**) ¿Cómo habría que modificar el algoritmo de búsqueda en profundidad para que proporcionara las ramas ascendentes y descendentes?
11. (**P**) ¿Cómo habría que modificar el algoritmo de búsqueda en profundidad para que proporcionara las ramas de cruce?
12. (**E; Rec**) Aplicar BP en los grafos inferiores para clasificar sus ramas.





13. (**P; Rec**) Argumentar que si en un grafo dirigido G hay una rama ascendente, entonces hay un ciclo en G. Y al revés, si no hay ninguna rama ascendente en G, entonces G es acíclico.
14. (**P; Rec**) Derivar un algoritmo de decisión de si un grafo es o no aciclico. ¿Cuál es su coste?
15. (**E; Rec**) Aplicar BP en los grafos inferiores para detectar si los mismos son o no acíclicos. (No considerar los costes de ramas indicados.)





16. (**E; Rec**) Aplicar el algoritmo de obtención de componentes conexas mediante búsqueda en profundidad para encontrar tales componentes en los grafos obtenidos transformando en grafos no dirigidos los del problema 12.
17. (**E; Rec**) Aplicar el algoritmo de obtención de componentes conexas mediante búsqueda en profundidad a los grafos inferiores. Verificar tambien que agrupando los índices de acuerdo a esas componentes, las matrices de adyacencia tienen una estructura de bloques diagonales.

18. (**E; Rec**) Aplicar el algoritmo de Tarjan para encontrar las componentes fuertemente conexas en los grafos dirigidos del problema 12.

19. (**E; Rec**) Una **ordenación topológica** de un grafo dirigido acíclico (GDA; DAG en inglés) $G = (V, R)$ es un orden total $\leq$ de V donde si $(u, v) \in R$, entonces $u \leq v$. Dar por inspección una ordenación topológica en los grafos del problema 12.

20. (**E; Rec**) Comprobar que si en los grafos anteriores se aplica BP computando tiempos de finalización e insertando los vértices en el inicio de una LE según termina su proceso, dicha LE da una ordenación topológica del grafo.

21. (**E; Rec**) En los grafos anteriores escribir sus vértices en una lista en el orden dado por OT y dibujar a continuación sobre la misma las ramas del grafo. Comprobar que todas ellas apuntan de izquierda a derecha.

22. (**P+; Rec**) Comprobar que una OT puede también obtenerse situando los nodos en una lista en orden inverso al de los tiempos de de finalización.

23. (**P+; Rec**) Dar el psc de un algoritmo OT basado en lo anterior y argumentar que proporciona una OT sobre cualquier GDA. ¿Cuál es su coste?

24. (**E; Rec**) El profesor Tragacanto se hace un lío cada vez que se viste. Para intentar evitarlo, ha construido un grafo dirigido $G = (V, R)$ de vértices $V = \{\ zapatos,\ calcetines,\ calzoncillos,$
$pantalones,\ cinturon,\ camiseta,\ camisa,\ corbata,\ chaqueta,\ abrigo,\ sombrero\}$ y donde $(u, v) \in R$ si la prenda u debe ponerse antes que v. Por ejemplo $(calcetin,\ zapato) \in R$. Indicar qué necesita hacer el profesor para poder vestirse sin rectificar ningún paso.

25. (**P+; Rec**) Si enumeramos los vértices de un GDA según su ordenación topológica, ¿qué estructura tiene su matriz de adyacencia? ¿Cuáles son sus autovalores? Argumentar que si G es un GDA con N nodos, su matriz de adyacencia tiene N autovalores 0.

26. (**P+; Rec**) Demostrar que en un grafo dirigido acíclico siempre hay un vértice de incidencia 0 (esto es, un vértice al que no llegan ramas) y otro de excedencia 0 (esto es, un vértice del que no salen ramas).
Pista: considerar un camino $u_1, u_2, \ldots, u_k$ de un número máximo de ramas y argumentar que $inc(u_1) = 0$ y $adj(u_k) = 0$.

6 Programación dinámica

1. (**P**) Dar una función Python `greedy_change(c, l_coins)` que implemente la solución codiciosa al problema del cambio devolviendo un dict con las monedas de `l_coins` necesarias para dar el cambio óptimo.

2. (**P; Rec**) Dar una función Python `pd_change(c, l_coins)` que implemente la solución mediante programación dinámica del problema del cambio y devuelva el número mínimo de monedas necesario para dar cambio de c.

3. (**P+; Rec**) En base al problema anterior, dar una función Python `pd_optimal_change(c, l_coins)` que devuelva un dict con las monedas de `l_coins` necesarias para dar el cambio óptimo utilizando para ello una matriz auxliar adecuada como se menciona en el problema 7.

4. (**E; Rec**) Aplicar los algoritmos anteriores para dar cambio óptimo de 7 maravedís usando monedas de 1, 3, 4, y 5 maravedís.

5. (**E; Rec.**) En algunos países del euro, como Finlandia, las monedas de 1 o 2 céntimos ya no están en circulación y todos los importes se redondean a la cantidad de 0 o 5 céntimos más cercana. Escribir una función Python `rounded_change(c, l_values)` que dé correctamente el cambio en estos países.

6. (**P**) Dar el pseudocódigo de un algoritmo PD para determinar el beneficio óptimo en el problema 0–1 de la mochila, y estimar su eficacia.

   Implementar dicho pseudocódigo en una función Python.

7. (**P; Rec**) Modificar el algoritmo anterior para que determine la composición de una carga óptima.

8. (**E; Rec**) Dar la evolución del algoritmo PD para el problema 0–1 de la mochila con cota 12 sobre los siguientes elementos

   ```
   item:    1   2   3
   weight:  3   7   8
   value:   5  10  11
   ```

9. (**E**) Dar la evolución del algoritmo PD para el problema 0–1 de la mochila con cota 17 sobre los siguientes elementos

   ```
   elemento:   1   2   3   4   5
   peso:       3   4   7   8   9
   valor:      4   5  10  11  13
   ```

   Sugerencia: Python?

10. (**P; Rec**) Modificar la solución PD para el problema 0–1 de la mochila de tal manera que de cada elemento pueda haber un número cualquiera de ejemplares.

11. (**E; Rec**) ¿Cuál sería la evolución de este algoritmo sobre el problema 8 si suponemos que cada elemento puede aparecer cualquier número de veces?

12. (**P+; Rec**) El Problema de la Suma (una variante del de la Mochila) consiste en, dada una serie finita $\mathcal{S} = s_1, \ldots, s_N$, y un valor entero V, determinar si algún subconjunto $s_{i_1}, \ldots, s_{i_K}$, $K \leq N$ de $\mathcal{S}$ verifica que $s_{i_1} + \ldots + s_{i_K} = V$.

    Un posible algoritmo de programación dinámica para dicho problema pasa por definir una matrix $d(i, v), 1 \leq i \leq N, 0 \leq v \leq V$, con valores 1 si v puede descomponerse como una suma de los elementos $s_1, \ldots, s_i$ y 0 en caso contrario.

    Comprobar que para $2 \leq i \leq n$, $0 \leq v \leq V$, se cumple

    $$d(i, v) = \max\{d(i-1, v), d(i-1, v-v_i)\}.$$

13. (**P; Rec**) A partir del problema 12, dar el pseudocódigo de un algoritmo PD para resolver el problema de la Suma a partir de una inicialización adecuada de la matriz $d(i, v)$.

14. (**E; Rec**) Resolver mediante programación dinámica el problema de la suma para la lista $1, 3, 5, 7$ y el valor 11.

15. (**P+**) Recoger el algoritmo anterior en una rutina Python `subsetSum(nums, S)` que reciba una lista `nums` de N números y un valor `S` a descomponer, y devuelva una matriz $N \times S$ con valores $d_{i,s}$ 0 o 1 según s pueda descomponerse o no como suma de los i primeros números.

16. (**P+**) Sobre la base de los problemas anteriores dar un algoritmo que indique qué elementos de la lista participan en la descomposición de un número S si tal número puede descomponerse como suma de números en dicha lista.

    Implementar dicho algoritmo en una rutina Python.

17. (**P+; Rec**) Queremos decidir si podemos repartir los elementos de un cierto conjunto de enteros positivos de tal manera que las sumas de cada uno de los dos subconjuntos coincidan. ¿Cómo lo podríamos abordar?

18. (**P**) Implementar en Python el algoritmo PD para encontrar la distancia de edición entre dos cadenas.

19. (**E**) a. Encontrar razonadamente la distancia mínima entre las cadenas `noria` y `radio`.

20. (**E; Rec**) Encontrar razonadamente la distancia mínima entre las cadenas `forraje` y `zarzajo`.

21. (**P; Rec**) Una variante del problema de distancia entre cadenas es el de encontrar la subcadena común más larga entre las cadenas S y T. Denotando por l_{ij} la longitud de la subcadena común más larga entre $S_i = [s_0, \ldots, s_i]$ y $T_j = [t_0, \ldots, t_j]$, encontrar una fórmula PD para l_{ij}.

22. (**E; Rec**) Encontrar la longitud de la subcadena común más larga entre `forraje` y `zarzajo`.

23. (**P+; Rec**) Un defecto del algoritmo PD para encontrar la subcadena común más larga es que no indica cuál podría ser la misma. ¿Cómo se podría dar un algoritmo para ello? Sugerencia: combinar en un algoritmo recursivo la tabla l_{ij} de longitudes y el conocimiento de los caracteres de S y T. Aplicar el algoritmo resultante para las cadenas `forraje` y `zarzajo`.

24. **P.** Recordemos que las ecuaciones PD para el problema de la distancia de edición son

$$\begin{aligned} d_{i,j} &= d_{i-1,j-1} \text{ si } s_i = t_j; \\ &= 1 + \min\{d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j}\} \text{ si } s_i \neq t_j \end{aligned}$$

    mientras que los del problema de la subcadena común más larga son

$$\begin{aligned} \ell_{i,j} &= 1 + \ell_{i-1,j-1} \text{ si } s_i = t_j; \\ &= \max\{\ell_{i-1,j-1}, \ell_{i,j-1}, \ell_{i-1,j}\} \text{ si } s_i \neq t_j; \end{aligned}$$

    donde no hay ningún término $\ell_{i-1,j-1}$. Argumentar por qué esto es así. ¿Podemos prescindir del término $1 + d_{i-1,j-1}$ para las distancias de edición?

25. (**P; Rec**) Dar un algoritmo para encontrar la subcadena palindrómica más larga (no necesariamente consecutiva) de una cadena S dada.

26. (**P**) Escribir una función Python que implemente el algoritmo dado en clase de obtención del número mínimo de operaciones en multiplicación de matrices.

27. (**P+**) Modificar esta función para que también proporcione la ordenación óptima.

28. (**E; Rec**) Dar la evolución del algoritmo anterior y la ordenación óptima cuando se emplea para multiplicar 4 matrices de tamaños respectivos 10×15, 15×20, 20×25 y 25×30.

29. (**P**) Estudiar razonadamente los requerimientos de memoria de los distintos algoritmos de programación dinámica de los problemas anteriores sobre cadenas.

30. (**P+; Rec**) Los métodos recursivos pueden dar lugar en ocasiones a algoritmos totalmente horribles bajo el punto de vista de la eficacia computacional. Un ejemplo de este tipo de situaciones sería un algoritmo recursivo para el cálculo de los números de Fibonacci. Escribir un tal algoritmo y estimar su rendimiento.

31. (**P; Rec**) El pésimo rendimiento de algoritmos como el anterior suele deberse a que el cálculo de ciertos valores se repite innecesariamente. Una forma de evitar este hecho es usar una tabla *caching* `c[]` para los valores intermedios, donde `c[i] = 1` si un cierto término T_i ya ha sido calculado, o bien vale `0` si no lo ha sido. Modificar el algoritmo recursivo anterior para el cálculo de los números de Fibonacci usando un tal tabla global. ¿Cuál sería la eficacia del algoritmo resultante?

32. (**P; Rec**) ¿Cómo se podrían aplicar estas ideas para dar versiones recursivas eficaces a los algoritmos anteriores de programación dinámica?

33. (**P++**) **Cruzando el río.** Para cruzar un río debemos ir de piedra en piedra situadas de manera consecutiva. El problema es que algunas son poco firmes y no podemos usarlas para avanzar pero hay otras más firmes e incluso elásticas que nos impulsan para avanzar sobre una, dos o más piedras de las siguientes. Eso sí, el impulso no se puede modular por lo que si pisamos por ejemplo en una piedra que da tres avances, los tenemos que dar enteros, sin poder quedarmos en dos o uno.

    Para nuestra conveniencia, las piedras tienen números que indican esa situación. Por ejemplo, la lista `[2, 0, 1, 0]` indica que de la primera piedra podemos ir a la tercera y de la tercera a la cuarta. Pero no podemos movernos de la segunda o de la cuarta, ni tampoco de la primera a la segunda. A su vez, en la lista `[1, 1, 0, 1]` podemos ir de la primera piedra a la segunda y de esta a la tercera, pero de la tercera no podríamos ir a la cuarta.

    Nos preguntamos si dada una tal lista podemos alcanzar la última posición a partir de la primera. Por ejemplo, ello sería posible con la primera lista pero no con la segunda.

    Escribir una función `cruzable(l_saltos)` que devuelva 1 o 0 en función de que la lista permita o no cruzar el río.

7 Algoritmos paralelos

1. (**E; Rec**) La tabla `p = [0, 0, 1, 3, 3, 4, 5, 6]` representa un conjunto disjunto en un momento dado. Dar la evolución sobre la misma del algoritmo `par_sum` indicando el estado de las tablas que dicho algoritmo ha de gestionar tras cada una de sus iteraciones.

2. (**P; Rec**) Dar el pseudocódigo de un algoritmo paralelo `par_media(t, n)` que calcule en paralelo la media de una tabla `t` de `n` elementos. Evaluar $T_1(n)$ y $T_\infty(n)$ para dicho algoritmo.

3. (**P; Rec**) Queremos usar la función anterior en un algoritmo paralelo para calcular la desviación típica de una tabla `t` de `n` elementos. Dar el pseudocódigo de una tal función y evaluar para ella $T_1(n)$ y $T_\infty(n)$.

4. (**P; Rec**) Dar los pseudocódigos de funciones paralelas para encontrar bien el máximo o bien el mínimo de una tabla `t` de `n` elementos, estimado los correspondientes valores de $T_1(n)$ y $T_\infty(n)$.

5. (**P; Rec**) Dar el pseudocódigo de una función paralela para calcular el operador `OR` de una tabla booleana `t` de `n` elementos, estimado los correspondientes valores de $T_1(n)$ y $T_\infty(n)$.

6. (**P; Rec**) Dar el pseudocódigo de una función `par_dot_prod(x, y, n)` que calcule el producto escalar de los vectores de las tablas `x, y` de `n` elementos, y estimar los correspondientes valores de $T_1(n)$ y $T_\infty(n)$.

7. (**P+; Rec**) Tenemos la representación binaria de un entero `x` mediante una tabla `b` de `n` elementos. Dar un algoritmo paralelo para calcular `x` y estimar sus correspondientes $T_1(n)$ y $T_\infty(n)$.

8. (**E; Rec**) Dar la evolución del algoritmo Merge Sort paralelo sobre la tabla `7, 6, 5, 4, 3, 2, , 0`.

9. (**P; Rec**) **Plus Scans.** Dada una tabla `t` con `n` elementos, queremos construir un algoritmo que devuelva la tabla `s`, donde `s[i]` contiene la suma de los `i-1` primeros elementos de `t`; esto es `s[0]=0, s[1]=t[0], ...`. Comprobar mediante ejemplos de tablas de tamaño 2^k que el siguiente algoritmo calcula efectivamente la tabla `s`:

```
def par_scan(t):
    if len(t) == 1:
        return [0]
    else:
```

```
tt = [ t[2*i] + t[2*i+1] for i in range(len(t) // 2) ]
ss = par_scan(tt)
#s = len(t) * [0]
s = vector(n)
for i in range(len(t)):
    if i % 2 == 0:
        s[i] = ss[i // 2]
    else:
        s[i] = ss[(i-1) // 2] + t[i-1]
return s
```

10. (**P+; Rec**) **Parallel Plus Scans.** Paralelizar el algoritmo anterior y estimar $T_1(n)$ y $T_\infty(n)$ para el mismo.

11. (**P; Rec**) **List Ranking.** Tenemos una representación simplificada de una lista enlazada dada por una tabla `n` donde `n[i]` contiene el índice del elemento siguiente a `i` ; el último elemento se detecta por la condición `n[i] = i` . Por ejemplo, la tabla

    ```
    [1, 2, 3, 4, 5, 5]
    ```

    representa una tal lista con índices entre `0` y `5` donde el elemento de índice `i` apunta al `i+1` y `5` es el último.

    Para una tal lista el problema del ranking consiste en calcular una nueva tabla `r` donde `r[i]` contiene el número de elementos entre el nodo `i` y el último nodo de la tabla, incluyendo a este pero no al nodo `i` . Comprobar mediante ejemplos de tablas adecuadas que el siguiente algoritmo calcula efectivamente la tabla `r` :

    ```
    def par_ranking(p):
        r = len(p) * [0]
        #r = vector(len(p))
        for i in range(len(p)):
            if p[i] == i:
                r[i] = 0
            else:
                r[i] = 1
        last_iter = int(np.ceil( np.log2(len(p))))
        for _ in range(last_iter):
            r = [r[i] + r[p[i]] for i in range(len(p))]
            p = [p[p[i]] for i in range(len(p))]
        return r
    ```

12. (**P+; Rec**) **Parallel List Ranking.** Paralelizar el algoritmo anterior y estimar $T_1(n)$ y $T_\infty(n)$ para el mismo.

13. (**P; Rec**) Si `p` representa ahora una lista enlazada estándar, dar un algoritmo alternativo de coste lineal para el problema del ranking de listas que añada a cada nodo de la lista su ranking.