Algoritmos y Estructuras de Datos (VJ1215) - Universitat Jaume I
Examen final - 2024/2025 - Segunda parte

5 de junio de 2025

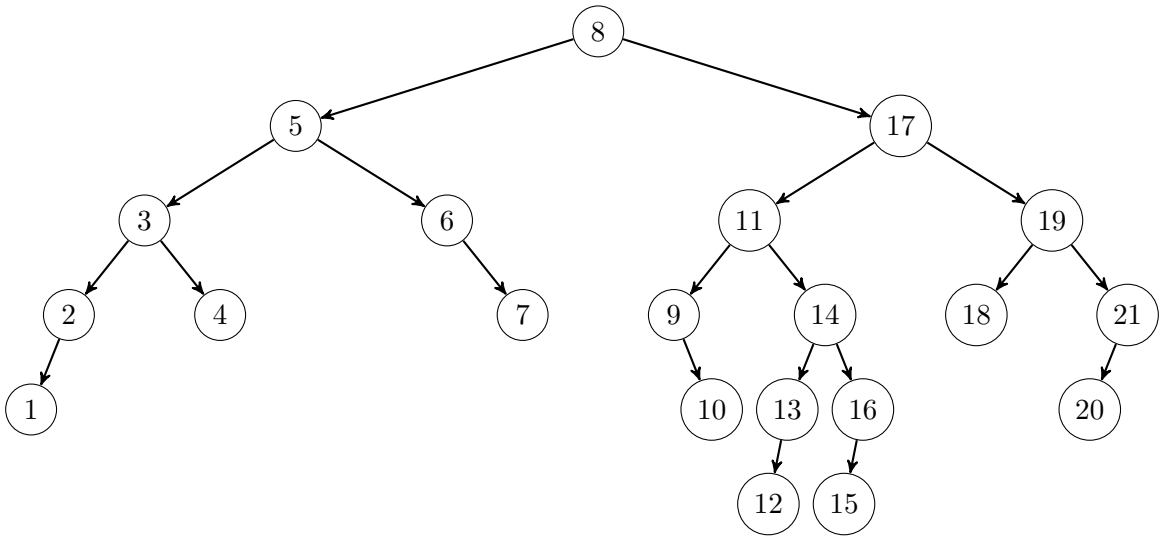
Nombre:

El examen final consta de dos partes, cada una con una duración de dos horas. La primera parte está destinada a aquellos estudiantes que han solicitado intentar mejorar la calificación obtenida en cualquiera de los exámenes parciales, renunciando a la misma. La segunda parte del examen final evalúa el resto del temario.

Este es el enunciado de la segunda parte, que suma 5 puntos. La prueba es individual. No puedes consultar libros, apuntes ni dispositivos electrónicos. Al finalizar entrega tus soluciones junto con el enunciado (no es necesario que entregues todas las hojas). Pon tu nombre en cada hoja que entregues.

Ejercicio 5 — 0,5 puntos

Aplica el algoritmo de eliminación de los árboles AVL para eliminar el dato 18 en el siguiente árbol. Indica qué rotaciones se realizan (diciendo si son a derecha o a izquierda y dónde) y dibuja cómo queda el árbol tras cada rotación. Si se realiza alguna rotación doble, dibuja por separado el resultado de cada una de las dos rotaciones simples de que consta.



Ejercicio 6 — 0,5 puntos

Un montículo binomial perezoso es una estructura de datos que permite implementar el Tipo Abstracto de Datos *Cola de Prioridad* con los siguientes costes temporales, siendo k el tamaño de la cola de prioridad:

Operación	**Coste en el peor caso**	**Coste amortizado**
insertar	$O(1)$	$O(1)$
consultarMínimo	$O(1)$	$O(1)$
eliminarMínimo	$O(k)$	$O(\log k)$
disminuirPrioridad	$O(\log k)$	$O(\log k)$

Analiza, en función de c, el coste temporal en el peor caso del algoritmo de Huffman cuando se aplica a un alfabeto de c caracteres, suponiendo que la cola de prioridad utilizada por el algoritmo se implementa con un montículo binomial perezoso.

Explica cómo obtienes el resultado, detallando cuántas veces se ejecuta y qué coste total tiene cada una de las operaciones.

Ejercicio 7 — 0,5 puntos

Se presenta a continuación una implementación del algoritmo que, dados $n \geq 2$ puntos en un espacio 2D, calcula la distancia entre los dos puntos más cercanos utilizando la estrategia Divide y Vencerás. En ella falta un aspecto importante del algoritmo, del que depende su coste temporal en el peor caso.

Indica qué líneas eliminarías, añadirías o modificarías para corregir la implementación, haciendo referencia a los números de línea. Escribe en C++ todos los cambios necesarios. No modifiques lo que ya sea correcto y eficiente, aunque pueda existir otra manera válida de implementarlo.

```
1  typedef pair<float, float> Punto;
2  float distanciaAlCuadrado(const Punto & p1, const Punto & p2) {
3     float a = p2.first  - p1.first;
4     float b = p2.second - p1.second;
5     return a * a + b * b;
6  }
7  bool compararY(const Punto & p1, const Punto & p2) {
8     return p1.second < p2.second;
9  }
10 float distanciaMinima(const vector<Punto> & puntosOrdenadosX,
11                       const vector<Punto> & puntosOrdenadosY) {
12    int talla = puntosOrdenadosX.size();
13    if (talla == 2)
14       return distanciaAlCuadrado(puntosOrdenadosX[0], puntosOrdenadosX[1]);
15    if (talla == 3)
16       return min({distanciaAlCuadrado(puntosOrdenadosX[0], puntosOrdenadosX[1]),
17                   distanciaAlCuadrado(puntosOrdenadosX[0], puntosOrdenadosX[2]),
18                   distanciaAlCuadrado(puntosOrdenadosX[1], puntosOrdenadosX[2])});
19    int tallaIzquierda = talla / 2, tallaDerecha = talla - tallaIzquierda;
20    vector<Punto> puntosOrdenadosXIzquierda(tallaIzquierda), puntosOrdenadosXDerecha(tallaDerecha),
21                  puntosOrdenadosYIzquierda(tallaIzquierda), puntosOrdenadosYDerecha(tallaDerecha);
22    for (int i = 0; i < tallaIzquierda; i++)
23       puntosOrdenadosXIzquierda[i] = puntosOrdenadosX[i];
24    for (int i = 0; i < tallaDerecha; i++)
25       puntosOrdenadosXDerecha[i] = puntosOrdenadosX[i + tallaIzquierda];
26    for (int i = 0, j = 0, k = 0; i < talla; i++)
27       if (puntosOrdenadosY[i] < puntosOrdenadosXDerecha[0])
28          puntosOrdenadosYIzquierda[j++] = puntosOrdenadosY[i];
29       else
30          puntosOrdenadosYDerecha[k++] = puntosOrdenadosY[i];
31    float minima = min(distanciaMinima(puntosOrdenadosXIzquierda, puntosOrdenadosYIzquierda),
32                       distanciaMinima(puntosOrdenadosXDerecha, puntosOrdenadosYDerecha));
33    for (int i = 0; i < puntosOrdenadosYIzquierda.size(); i++)
34       for (int j = 0; j < puntosOrdenadosYDerecha.size(); j++)
35          minima = min(minima, distanciaAlCuadrado(puntosOrdenadosYIzquierda[i],
36                                                   puntosOrdenadosYDerecha[j]));
37    return minima;
38 }
39 float distanciaMinima(const vector<Punto> & puntos) {
40    vector<Punto> puntosOrdenadosX(puntos), puntosOrdenadosY(puntos);
41    sort(puntosOrdenadosX.begin(), puntosOrdenadosX.end());
42    for (int i = 0; i < puntosOrdenadosX.size() - 1; i++)
43       if (puntosOrdenadosX[i] == puntosOrdenadosX[i + 1])
44          return 0;
45    sort(puntosOrdenadosY.begin(), puntosOrdenadosY.end(), compararY);
46    return sqrt(distanciaMinima(puntosOrdenadosX, puntosOrdenadosY));
47 }
```

Ejercicio 8 — 0,5 puntos

Teorema Maestro: La solución de la ecuación $T(N) = aT(N/b) + \theta(N^k \log^p N)$, con $a \geq 1$, $b > 1$ y $p \geq 0$, es

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{si } a > b^k \\ O(N^k \log^{p+1} N) & \text{si } a = b^k \\ O(N^k \log^p N) & \text{si } a < b^k \end{cases}$$

El siguiente algoritmo recursivo determina si un dato aparece en un vector:

```
bool buscar(const vector<float> & v, float dato) {
   if (v.size() == 0)
      return false;
   if (v.size() == 1)
      return v[0] == dato;
   int mitad = v.size() / 2;
   vector<float> izquierda(mitad);
   for (int i = 0; i < mitad; i++)
      izquierda[i] = v[i];
   if (buscar(izquierda, dato))
       return true;
   vector<float> derecha(v.size() - mitad - 1);
   for (int i = mitad + 1, j = 0; i < v.size(); i++, j++)
      derecha[j] = v[i];
   if (buscar(derecha, dato))
      return true;
   return v[mitad] == dato;
}
```

Utilizando el Teorema Maestro, analiza los costes que se piden a continuación en función de n, siendo n el tamaño del vector `v` en la primera llamada.

Coste temporal en el mejor caso	$O($ $)$
Coste temporal en el peor caso	$O($ $)$

Para cada uno de los dos costes, indica (i) qué ecuación recursiva obtienes para el coste temporal y cuáles son en ella los valores de a, b, k y p; y (ii) a qué solución llegas aplicando el teorema a partir de esa ecuación. No es necesario que des más explicaciones.

Ejercicio 9 — 0,5 puntos

En un vector, a partir de la posición 1 (inclusive), se encuentra un montículo binario de mínimos (*min-heap*).

a) Describe cómo implementarías una operación que convierta el montículo de mínimos en un montículo de máximos (*max-heap*) manteniendo los mismos datos. No es necesario proporcionar el código.

b) Ilustra el proceso de conversión aplicándolo al siguiente vector inicial. Muestra paso a paso cómo evoluciona la estructura, dibujando el árbol correspondiente que representa cada fase del proceso. Muestra también el vector definitivo después de la conversión.

0	1	2	3	4	5	6	7	8	9	10
	10	30	20	40	60	100	90	80	50	70

c) Indica cuál es el siguiente coste de tu solución en función de n, siendo n el tamaño del montículo.

Coste temporal en el peor caso	$O($ $)$

Ejercicio 10 — 0,5 puntos

El siguiente algoritmo recursivo determina si un dato aparece en un vector:

```
bool buscar(const vector<float> & v, float dato) {
   if (v.size() == 0)
      return false;
   if (v.size() == 1)
      return v[0] == dato;
   int mitad = v.size() / 2;
   vector<float> izquierda(mitad);
   for (int i = 0; i < mitad; i++)
      izquierda[i] = v[i];
   if (buscar(izquierda, dato))
       return true;
   for (int i = mitad ; i < v.size(); i++)
      if (v[i] == dato)
         return true;
   return false;
}
```

Sin utilizar el Teorema Maestro, analiza los costes que se piden a continuación en función de n, siendo n el tamaño del vector `v` en la primera llamada. En el último apartado, se pide determinar el coste temporal si el vector se pasase por valor, sustituyendo `const vector<float> & v` por `vector<float> v`, sin realizar ningún otro cambio.

Coste temporal en el mejor caso	$O($ $)$
Coste temporal en el peor caso	$O($ $)$
Coste espacial en el mejor caso	$O($ $)$
Coste espacial en el peor caso	$O($ $)$
Coste temporal en el peor caso pasando v por valor	$O($ $)$

Para cada caso, explica el cálculo realizado y, si el análisis implica algún sumatorio, describe cuál es.

Ejercicio 11 — 2 puntos

En el interior de una torre hay una escalera con n escalones, numerados desde 0 hasta $n-1$, que permite subir hasta la cima. Se nos proporciona un vector de enteros no negativos *saltoMáximo* de tamaño n. Desde el escalón i (para cada i entre 0 y $n-2$), el jugador puede saltar hacia arriba hasta cualquier escalón $j > i$, siempre que la distancia $j - i$ no supere *saltoMáximo*$[i]$. Si *saltoMáximo*$[i]$ vale 0, significa que desde ese escalón no es posible avanzar; en caso contrario, se puede saltar hasta cualquier escalón comprendido entre $i+1$ e $i +$ *saltoMáximo*$[i]$, ambos inclusive.

Necesitamos una función:

```
int minSaltos(const vector<int> & saltoMaximo)
```

que devuelva la mínima cantidad de saltos necesarios para llegar desde el escalón 0 hasta el escalón $n-1$, si esto es posible. En caso contrario, la función debe lanzar una excepción.

Por ejemplo, con el siguiente vector *saltoMáximo* de tamaño 10:

0	1	2	3	4	5	6	7	8	9
5	4	2	0	3	1	1	2	1	0

Desde el escalón 0, se puede saltar hasta el $1, 2, 3, 4$ o 5. Desde el escalón 1, se puede alcanzar el $2, 3, 4$ o 5. Desde el escalón 2, se puede saltar al 3 o 4, y desde el escalón 3, no se puede avanzar. Para llegar al escalón 9 con la mínima cantidad de saltos, una posible solución es $0 \to 4 \to 7 \to 9$, requiriendo 3 saltos en total.

Diseña empleando Programación Dinámica, e implementa en C++, un algoritmo eficiente para resolver el problema a) de forma recursiva y b) de forma no recursiva.

Indica el coste temporal y espacial de cada una de tus dos soluciones en el peor caso, en función de n.