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

21 de enero de 2025

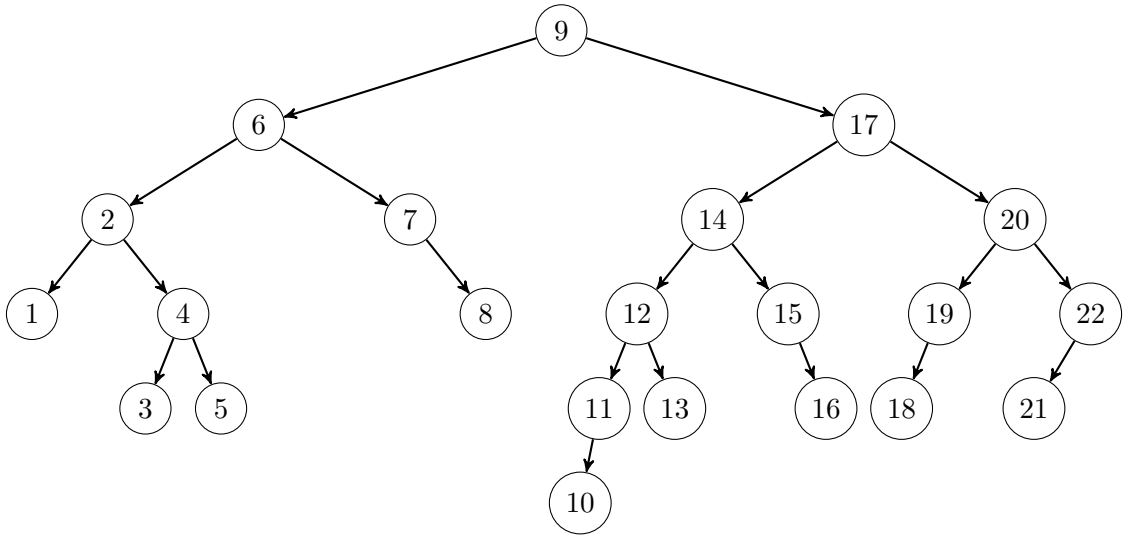
Nombre:

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 4 — 0,5 puntos

Aplica el algoritmo de eliminación de los árboles AVL para eliminar el dato 7 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 simple. Si se realiza alguna rotación doble, dibuja por separado el resultado de cada una de las dos rotaciones simples de que consta.



Ejercicio 5 — 0,5 puntos

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

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

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

Ejercicio 6 — 0,5 puntos

A continuación se presenta una implementación incorrecta del algoritmo que, dados $n \geq 2$ puntos 2D, calcula la distancia entre los dos puntos más cercanos utilizando la estrategia Divide y Vencerás. Los errores no son detalles menores, sino bucles incorrectos y/o faltantes en dos áreas.

Indica qué líneas quitarías por ser incorrectas y qué añadirías para corregir la implementación, escribiendo en C++ los cambios necesarios. Puedes referenciar los números de línea o realizar los cambios directamente en esta misma página. No modifiques lo que sea correcto, aunque haya otras formas de hacerlo.

```
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> & puntosX, const vector<Punto> & puntosY) {
11    int talla = puntosX.size();
12    if (talla == 2)
13       return distanciaAlCuadrado(puntosX[0], puntosX[1]);
14    if (talla == 3)
15       return min({distanciaAlCuadrado(puntosX[0], puntosX[1]),
16                   distanciaAlCuadrado(puntosX[0], puntosX[2]),
17                   distanciaAlCuadrado(puntosX[1], puntosX[2])});
18    int tallaIzquierda = talla / 2, tallaDerecha = talla - tallaIzquierda;
19    vector<Punto> izquierdaX(tallaIzquierda), derechaX(tallaDerecha),
20                  izquierdaY(tallaIzquierda), derechaY(tallaDerecha);
21    for (int i = 0; i < tallaIzquierda; i++) {
22       izquierdaX[i] = puntosX[i];
23       izquierdaY[i] = puntosY[i];
24    }
25    for (int i = 0; i < tallaDerecha; i++) {
26       derechaX[i] = puntosX[i + tallaIzquierda];
27       derechaY[i] = puntosY[i + tallaIzquierda];
28    }
29    float minima = min(distanciaMinima(izquierdaX, izquierdaY),
30                       distanciaMinima(derechaX, derechaY));
31    vector<Punto> centro;
32    float frontera = izquierdaX[tallaIzquierda - 1].first;
33    for (int i = 0; i < talla; i++)
34       if( abs(frontera - puntosY[i].first)  < minima )
35          centro.push_back(puntosY[i]);
36    for (int i = 0; i < centro.size() - 1; i++)
37       minima = min(minima, distanciaAlCuadrado(centro[i], centro[i + 1]));
38    return minima;
39 }
40 float distanciaMinima(const vector<Punto> & puntos) {
41    vector<Punto> puntosX(puntos), puntosY(puntos);
42    sort(puntosX.begin(), puntosX.end());
43    for (int i = 0; i < puntosX.size() - 1; i++)
44       if (puntosX[i] == puntosX[i + 1])
45          return 0;
46    sort(puntosY.begin(), puntosY.end(), compararY);
47    return sqrt(distanciaMinima(puntosX, puntosY));
48 }
```

Ejercicio 7 — 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<int> & v, int dato, int inicio, int fin) {
   if (inicio > fin)
      return false;
   if (inicio == fin)
      return v[inicio] == dato;
   int medio = (inicio + fin) / 2;
   int cuartoIzquierda = (inicio + medio) / 2;
   int cuartoDerecha = (medio + 1 + fin) / 2;
   if (buscar(v, dato, inicio, cuartoIzquierda))
      return true;
   for (int i = cuartoIzquierda + 1; i <= cuartoDerecha; i++)
      if (v[i] == dato)
         return true;
   return buscar(v, dato, cuartoDerecha + 1, fin);
}
bool buscar(const vector<int> & v, int dato) {
   return buscar(v, dato, 0, v.size() - 1);
}
```

Utilizando el Teorema Maestro, y siendo n el tamaño del vector:

a) **[0,25 puntos]** Analiza el coste temporal del algoritmo en el peor caso en funcion de n.

b) **[0,25 puntos]** Analiza el coste temporal del algoritmo en el mejor caso en función de n.

En cada apartado, 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 8 — 0,5 puntos

Tenemos un montículo binario de mínimos (*min-heap*) en un vector de tamaño n. Explica detalladamente, como si le estuvieras explicando a un programador cómo hacerlo, los pasos a seguir para transformar este vector hasta que contenga los datos ordenados de mayor a menor sin utilizar ningún otro vector ni estructura de datos auxiliar. No es necesario que lo implementes. Además, analiza el coste temporal de tu solución en el peor caso (sin incluir el coste de construir el montículo).

Ejercicio 9 — 0,5 puntos

El siguiente algoritmo recursivo averigua si un vector está ordenado de menor a mayor:

```
bool ordenado(const vector<int> & v, int inicio, int fin) {
   if (fin <= inicio)
      return true;
   int medio = (inicio + fin) / 2;
   return ordenado(v, inicio, medio)
          && ordenado(v, medio + 1, fin)
          && v[medio] <= v[medio + 1];
}
bool ordenado(const vector<int> & v) {
   return ordenado(v, 0, v.size() - 1);
}
```

Analiza los costes que se piden a continuación en función de n, siendo n el tamaño del vector. Recuerda que el operador `&&` evalúa en cortocircuito: si su operando izquierdo vale *false*, no evalúa el operando derecho.

En los dos últimos apartados, se pide determinar los costes temporal y espacial si el vector se pasase por valor en ambas funciones, cambiando `const vector<int> & v` por `vector<int> v`, sin realizar ningún otro cambio.

No es necesario que justifiques tus respuestas.

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

Ejercicio 10 — 2 puntos

Todo número entero positivo n puede expresarse como la suma de otros números enteros positivos elevados al cuadrado. La máxima cantidad de cuadrados necesaria para sumar así n es exactamente n, correspondiente a la suma de n veces 1^2. Sin embargo, necesitamos una funcion que, dado n, calcule no la máxima sino la mínima cantidad de cuadrados necesarios para sumar n.

```
int minimaCantidadCuadrados(int n)
```

Por ejemplo, para $n = 73$, el resultado de la función sería 2. Aunque hay muchas formas de expresar 73 como suma de cuadrados ($73 = 6^2+6^2+1^2$, $73 = 8^2+2^2+2^2+1^2$, $73 = 5^2+5^2+4^2+2^2+1^2+1^2+1^2$, etc.), la mínima cantidad de cuadrados necesaria es 2, correspondiente a $73 = 8^2 + 3^2$. Para $n = 12$, el resultado sería 3, correspondiente a $12 = 2^2 + 2^2 + 2^2$.

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:

	Recursiva	**No recursiva**
Coste temporal en el peor caso	$O($ $)$	$O($ $)$
Coste espacial en el peor caso	$O($ $)$	$O($ $)$