Prueba de Diseño de Algoritmos – Curso 2014-2015 – 5 de mayo de 2015

Diseño y Análisis de Algoritmos – Grado en Ingeniería Informática

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

Se compensa el examen con 4 puntos de los 10 posibles

Soluciones

Ejercicio 1 [**2 puntos**]

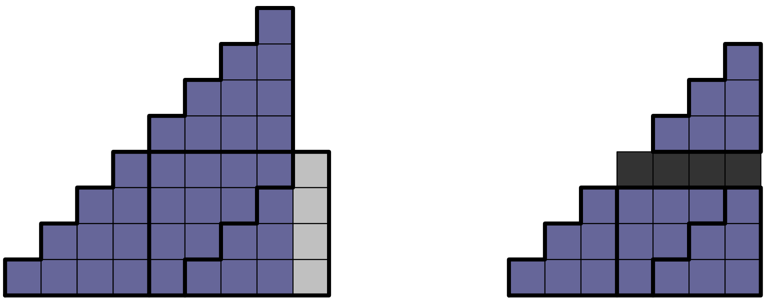
Se pide describir la suma de los primeros n números naturales mediante una función recursiva $S(n)$ la cual divida el tamaño del problema en dos.

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

Se considera que $n > 0$ pudiendo ser par o impar. Describir el/los caso(s) base y recursivo(s).

Posible solución:

Podemos pensar que el sumatorio consiste en sumar los cuadrados de una pirámide con n cuadrados en su base, $n-1$ en el segundo nivel, etc., hasta un cuadrado en el nivel superior. Dividiendo el problema por dos, podemos sumar los cuadrados de cuatro pirámides de tamaño $n/2$ (división entera) tal y como se representa en la siguiente figura cuando n es par (izquierda) e impar (derecha):



Cuando n es par sobran $n/2$ cuadrados (claros), mientras que cuando n es impar necesitamos $n/2+1$ cuadrados (oscuros) adicionales para completar la pirámide original.

De esta manera, la función recursiva sería:

$$S(n) = \begin{cases} 1 & \text{si } n = 1 \\ 4S(\frac{n}{2}) - \frac{n}{2} & \text{si } n > 1 \text{ y } n \text{ par} \\ 4S(\frac{n}{2}) + \frac{n}{2} + 1 & \text{si } n > 1 \text{ y } n \text{ impar} \end{cases}$$

Ejercicio 2 [1.5 puntos]

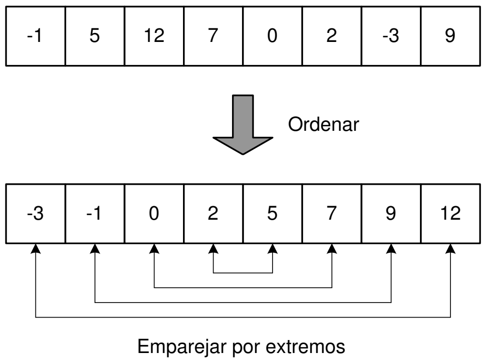
Sea un vector **v** formado por un número n par de enteros. Hay que agrupar a los elementos de **v** en $n/2$ parejas de forma que si se halla la suma de los dos miembros de cada pareja y se toma el máximo de estas sumas, el número resultante es el menor posible.

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

Se pide diseñar una estrategia voraz que obtenga una solución óptima al problema propuesto. Justifíquese informalmente la optimalidad del criterio elegido.

Solución:

La estrategia consiste en ordenar el vector **v** y emparejar a los elementos por los extremos del vector como ilustra la siguiente figura:



Por tanto, se cogerían las parejas $(\mathbf{v}[i], \mathbf{v}[n-i+1])$, para $i = 1, \ldots, n/2$.

Se demuestra que la estrategia es óptima por contradicción. Sean M el mayor elemento de **v**, y m el menor. Supongamos que en la solución óptima tenemos las parejas (M, a) y (m, b), donde a y b son otros elementos de **v**. Como $M \geq a$ y $b \geq m$, entonces $M + a = \text{máx}(M + a, m + b)$. Además $M + a \geq \text{máx}(M + m, a + b)$. Por tanto, $\text{máx}(M + a, m + b) \geq \text{máx}(M + m, a + b)$.

De esta manera, reemparejando las parejas como (M, m) y (a, b) obtenemos un valor total menor para el problema, lo cual es una contradicción con la hipótesis inicial. Finalmente, una vez seleccionada la pareja (M, m), se tendría que volver a aplicar la misma estrategia voraz al subproblema de $n - 2$ elementos, luego al de $n - 4$, y así sucesivamente.

Ejercicio 3 [2.5 puntos]

Se pide diseñar e implementar un algoritmo basado en la técnica de divide y vencerás para hallar el menor elemento de una matriz $\mathbf{A}$ de dimensiones conocidas $n \times m$. El problema original debe partirse en 4 subproblemas idénticos al original, dividiendo a la matriz en los siguientes 4 bloques:

$$\left(\begin{array}{c|c} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \hline \mathbf{A}_{21} & \mathbf{A}_{22} \end{array}\right)$$

Posible solución:

```java
public static void main(String args[]) throws Exception {
  BufferedReader cin = new BufferedReader(new InputStreamReader(System.in));
  String[] tresEnteros = cin.readLine().split(" ");
  int n = Integer.parseInt(tresEnteros[0]);
  int m = Integer.parseInt(tresEnteros[1]);
  int[][] A = new int[n][m];

  for(int i=0; i<n; i++){
    String[] n_Enteros = cin.readLine().split(" ");
    for(int j=0; j<m; j++){
      A[i][j] = Integer.parseInt(n_Enteros[j]);
    }
  }

  System.out.println(minMat(A,0,n-1,0,m-1));
}

public static int minimo(int a, int b){
  if(a<b)
    return a;
  else
    return b;
}

public static int minMat(int A[][], int ini_f, int fin_f, int ini_c, int fin_c){
  int mitad_f = (ini_f + fin_f)/2;
  int mitad_c = (ini_c + fin_c)/2;

  if ((ini_f==fin_f) && (ini_c==fin_c)) {
    return A[ini_f][ini_c];

  } else if (ini_f==fin_f) {
      return minimo(minMat(A,ini_f,fin_f,ini_c,mitad_c),
                    minMat(A,ini_f,fin_f,mitad_c+1,fin_c));

  } else if (ini_c==fin_c) {
      return minimo(minMat(A,ini_f,mitad_f,ini_c,fin_c),
                    minMat(A,mitad_f+1,fin_f,ini_c,fin_c));

  } else
      return minimo(minimo(minMat(A,ini_f,mitad_f,ini_c,mitad_c),
                           minMat(A,mitad_f+1,fin_f,ini_c,mitad_c)),
                    minimo(minMat(A,ini_f,mitad_f,mitad_c+1,fin_c),
                           minMat(A,mitad_f+1,fin_f,mitad_c+1,fin_c)));
}
```

Ejercicio 4 [4 puntos]

Nos vamos a ir de camping, y queremos llevarnos toda la comida y bebida que podamos. En casa tenemos n productos que podemos llevar, los cuales pesan p_i kilos, para $i = 1, \ldots, n$. Tenemos una mochila en la que introduciremos productos, que pueden llegar a transportar como mucho C kilos.

Se pide implementar un algoritmo basado en la técnica de *backtracking* para averiguar el peso máximo de los productos que podemos transportar en la mochila. Es decir, se trata de maximizar:

$$\sum_{i \in S} p_i$$

Con la restricción:

$$\sum_{i \in S} p_i \leq C$$

donde S representa el conjunto de índices de productos introducidos en la mochila.

Posible solución:

```
public static void mochila_peso (int[] ps, int c) {
  int[] solParcial = new int[ps.length];
  int[] solOptima = new int[ps.length];

  int pOpt = buscar_peso (ps.length, 0, 0, solParcial, solOptima, -1, ps, c);

  System.out.println(pOpt);
}

private static int buscar_peso (int n, int i, int p, int[] solParc,
                                int[] solOpt, int pOpt,
                                int[] ps, int c) {
  for (int k=0; k<=1; k++) {
    if (p+k*ps[i]<=c) {
      solParc[i] = k;
      int np = p + k*ps[i];

      if (i==n-1) {
        if (np>pOpt) {
          pOpt = np;
          for (int j=0; j<ps.length; j++)
            solOpt[j] = solParc[j];
        }
      }
      else
        pOpt = buscar_peso (n, i+1, np, solParc, solOpt, pOpt, ps, c);
    }
  }

  return pOpt;
}
```