UNIVERSIDAD DE GRANADA

Algorítmica

2º Grado en Ingeniería Informática

Guión de prácticas

Algoritmos Divide y Vencerás

DECSAI

Departamento de Ciencias de la

Algoritmos Divide y Vencerás

1. Objetivo

El objetivo de la práctica consiste en que el alumno sea capaz de analizar un problema y resolverlo mediante la técnica Divide y Vencerás. Se expondrá un problema que será resuelto en clase por el profesor. Posteriormente, se expone un conjunto de problemas que deberán ser resueltos por el estudiante.

2. Diseño de algoritmos Divide y Vencerás

2.1. Descripción del problema de ejemplo

Se dispone de un array de números enteros que se desea ordenar. Se dispone de un algoritmo básico "Burbuja" que resuelve el problema (ver Figura 1). El ejercicio trata de resolver el problema de ordenación mediante la técnica Divide y Vencerás.

```
76 void OrdenaBurbuja(int *v, int n) {
77
78     int i, j, aux;
79     bool haycambios= true;
80
81     i= 0;
82     while (haycambios) {
83
84      haycambios=false; // Suponemos vector ya ordenado
85      for (j= n-1; j>i; j--) { // Recorremos vector de final a i
86
87          if (v[j-1]>v[j]) { // Dos elementos consecutivos mal ordenados
88           aux= v[j]; // Los intercambiamos
89           v[j]= v[j-1];
90           v[j-1]= aux;
91           haycambios= true; // Al intercambiar, hay cambio
92          }
93      }
94      i++;
95     }
96 }
```

Figura 1: Algoritmo de ordenación por burbuja

2.2. Solución: Diseño de componentes

Para poder aplicar la técnica Divide y Vencerás se deben cumplir los siguientes requisitos:

- El problema inicial debe poder ser divisible en subproblemas que tengan aproximadamente el

mismo tamaño, independientes entre sí, que sean de la misma naturaleza del problema inicial y que se puedan resolver por separado.

- Las soluciones de los subproblemas debe poder combinarse entre sí para poder dar lugar a la solución del problema inicial.
- Debe existir un caso base en el que el problema sea ya indivisible o, en su defecto, un algoritmo básico que pueda resolverlo.

Las tres circunstancias se dan en el problema de ordenación a tratar. El diseño propuesto sería el siguiente:

- **División del problema en subproblemas**. Un array v[1..n] de n componentes a ordenar inicial se puede dividir en K=2 subproblemas, v1[1...floor(n/2)] y v2[floor(n/2)+1..n], correspondientes a la mitad izquierda y mitad derecha del array, respectivamente (NOTA: floor(x) denota la parte entera de x). Ambos subproblemas son de la misma naturaleza del original (ordenar v1/v2), tienen aproximadamente el mismo tamaño (n1=floor(n/2), n2=n-floor(n/2)), son independientes entre sí (ordenar v1 no depende de ordenar v2 y viceveresa) y, por tanto, se pueden resolver por separado.
- **Existencia de caso base**. En este problema el caso base se da cuando n<=1; es decir, cuando el vector sólo tiene una única componente, que es a su vez solución al problema dado que un array de 1 componente estaría ya ordenado, o el vector tiee 0 componentes y no existe. Adicionalmente, también disponemos de un método básico que resuelve el problema (algoritmo Burbuja).
- **Combinación de soluciones**. Sean v1, v2 los dos subproblemas generados desde el problema inicial v, con tamaños n1, n2 respectivamente. Sea s1 la solución de ordenar v1, y s2 la solución de ordenar v2. La solución S al problema original de ordenar v se puede obtener como S=combina(s1, s2), donde el método "combina" trata de ir copiando cada componente menor de v1/v2 iterativamente en la solución (ver apuntes de teoría).

2.3. Solución: Diseño del algoritmo

El método de combinación, con las descripciones dadas en el apartado anterior, sería el siguiente:

```
Método S=combina(s1[1..n1], s2[1..n2])
    S= array vacío de tamaño n1+n2

    j= 1, i=1, k=1
    Mientras i<n1 y j<n2, hacer:
        Si s1[i] <= s2[j], hacer:
            S[k]= s1[i]
            i= i+1
        En otro caso:
            S[k]= s2[j]
            j= j+1
        k= k+1
    Fin-Mientras

    Mientras i<n1, hacer: // Rellenar S con el resto de s1, si s2 se ha introducido en S ya
        S[k]= s1[i]
        i= i+1
```

```
            k= k+1
    Fin-Mientras

    Mientras j<n2, hacer: // Rellenar S con el resto de s2, si s1 se ha introducido en S ya
            S[k]= s2[j]
            j= j+1
            k= k+1
    Fin-Mientras
    Devolver S
```

Con este diseño de componentes, podemos pasar a adaptar la plantilla DyV para resolver el problema de ordenación de la siguiente forma:

```
Algoritmo S=DyV(v[1..n]: Array de n componentes)
    Si n<=1, hacer: # Caso Base de la recurrencia
            S=copia de v
            Devolver S
    En otro caso: # Caso General de la recurrencia
            Dividir v en dos arrays v1=v[1..floor(n/2)] , v2= v[floor(n/2)+1..n]
            s1= DyV(v1) # Resolver subproblema 1
            s2= DyV(v2) # Resolver subproblema 2
            S= combina(s1, s2)
            Devolver S
```

2.4. Solución: Implementación

La implementación de este algoritmo en C++ requiere de una adaptación para establecer componentes iniciales y finales de los subvectores v1, v2, y el uso de un array auxiliar para realizar la combinación de soluciones para que la implementación sea más eficiente. La Figura 2 muestra la implementación de la función "*combina*" diseñada, y la Figura 3 la implementación del algoritmo "*DyV*" diseñado.

```
 8  void fusionaMS(double *v, int posIni, int centro, int posFin, double *vaux) {
 9
10      int i= posIni;
11      int j= centro;
12      int k= 0;
13
14      while (i<centro && j<=posFin) {
15          if (v[i]<=v[j]) {
16              vaux[k]= v[i];
17              i++;
18          } else {
19              vaux[k]= v[j];
20              j++;
21          }
22          k++;
23      }
24
25      while (i<centro) {
26          vaux[k]= v[i];
27          i++, k++;
28      }
29      while (j<=posFin) {
30          vaux[k]= v[j];
31          j++, k++;
32      }
33
34      memcpy(v+posIni, vaux, k*sizeof(double));
35  }
```

Figura 2: Función de fusión del algoritmo MergeSort

```
38  void MergeSort(double *v, int posIni, int posFin, double *vaux) {
39
40      if (posIni>=posFin) return;
41
42      int centro= (posIni+posFin)/2;
43
44      MergeSort(v, posIni, centro, vaux);
45      MergeSort(v, centro+1, posFin, vaux);
46      fusionaMS(v, posIni, centro+1, posFin, vaux);
47  }
```

Figura 3: Función principal del algoritmo MergeSort

Suponiendo un array de entrada original "v" de "n" componentes, y la existencia de otro array auxiliar "*vaux*", la llamada para resolver el problema con el código propuesto sería la siguiente:

MergeSort(v, 0, n-1, vaux).

La implementación dada devuelve la solución dentro del mismo array "v" original de entrada, modificándolo. Requiere tener como entrada otro array auxiliar *vaux*, del mismo tamaño que el array *v*.

2.5. Estudio de eficiencia de la solución

Para conocer si la solución Divide y Vencerás propuesta es más eficiente que el algoritmo básico inicial, calcularemos la eficiencia de ambos métodos. En particular, la eficiencia de Burbuja es $O(n^2)$. Para el método DyV, al ser recursivo, tenemos la siguiente ecuación recurrente en el caso general:

$$T(n)= 2T(n/2)+n$$

Como la ecuación no tiene las variables de la forma requerida por la ecuación característica, tenemos que hacer un cambio de variable. En este caso, $n=2^m$, quedando:

$$T(2^m)= 2T(2^{m-1}) + 2^m$$

Llevamos las "T's" a la izquierda, y obtenemos que es una ecuación lineal no homogénea:

$$T(2^m) - 2T(2^{m-1}) = 2^m$$

Resolvemos primero la "parte homogénea", dando como resultado el polinomio de la parte homogénea $pH(x)= (x-2)$. A continuación, resolvemos la parte "no homogénea", quedando $2^m = b_1^n\ q_1(n)$, que se resuelve con la asignación $b_1= 2$ y $q_1(n)= 1$ con grado $d_1=0$. Esto nos permite elaborar la forma del polinomio característico de la ecuación como $p(x)= (x-2)^2$. Tenemos una raiz $r=1$, cuyo valor es $R_1=2$ con multiplicidad $M_1= 2$, Resolviendo con la técnica de la ecuación característica y deshaciendo el cambio de variable, tenemos que el tiempo de ejecución del algoritmo es $T(n)= c_{10}n + c_{11}\ n*\log_2(n)$ y que, por tanto, su orden en el peor de los casos es $O(n*\log_2(n))$.

Dado que el orden de ejecución del método básico es $O(n^2)$ y el del método DyV es $O(n*\log_2(n))$, podemos afirmar que, asintóticamente, el algoritmo DyV realiza menos operaciones cuando el tamaño del caso tiende a infinito y que, por tanto tiene una mejor eficiencia.

3. Ejercicios propuestos

Para cada uno de los ejercicios siguientes, se debe:

- Proponer e implementar un algoritmo básico (o por fuerza bruta) que resuelva el problema.
- Diseñar e implementar un algoritmo Divide y Vencerás que resuelva el problema.
- Analizar la eficiencia de ambos algoritmos, indicando si se producen mejoras en la eficiencia y/o en el tiempo de ejecución para tamaños "grandes" del tamaño del caso.
- Proponer varios ejemplos de uso de ambos algoritmos, mostrando que tanto el método básico como el DyV resuelven el problema.

3.1. Ejercicio 1

La unidad aritmético-lógica (ALU) de un microcontrolador de consumo ultrabajo no dispone de la operación de multiplicación. Sin embargo, necesitamos multiplicar un número natural *i* por otro número natural *j*, que notamos como i*j. Sabemos, por la definición matemática de la operación de multiplicación en los números naturales, que:

$$i * j = \begin{cases} i & j = 1 \\ i * (j-1) + i & j > 1 \end{cases} \forall i, j \in N$$

Asumiendo como método básico la implementación directa de la fórmula anterior: ¿Es posible crear un algoritmo Divide y Vencerás más eficiente que la implementación de este algoritmo básico? El algoritmo que se diseñe no deberá incluir la operación de multiplicación, dado que la ALU no puede ejecutarla al carecer de comando de multiplicación.

3.2. Ejercicio 2

Un número natural *n* se dice que es cuadrado perfecto si se corresponde con el cuadrado de otro número natural. Por ejemplo, 4 es perfecto dado que $4=2^2$. También son cuadrados perfectos $25=5^{2,}$ o $100=10^2$.

¿Qué método básico/por fuerza bruta podemos diseñar para calcular si un número es cuadrado perfecto o no? ¿Es posible diseñar un algoritmo Divide y Vencerás para igualar (o mejorar) la eficiencia de este método básico?

Nota: No se puede usar la función "sqrt()".

3.3. Ejercicio 3

Dado un número natural *n*, deseamos saber si existe otro número natural *y* de modo que *n=y*(y+1)*(y+2)*. Por ejemplo, para *n*=60, existe y=3 de modo que *y*(y+1)*(y+2)=3*4*5=60*.

¿Qué método básico/por fuerza bruta podemos diseñar para calcular, dado un valor de *n* de entrada, si existe el número *y* que hace cumplir la igualdad y cuál es su valor? ¿Es posible diseñar un algoritmo Divide y Vencerás para igualar (o mejorar) la eficiencia de este método básico?

4. Evaluación de la práctica

Se deben resolver todos los ejercicios propuestos en el apartado 3 de este guión. Cada ejercicio se valorará sobre 10 de la siguiente forma:

1. **(1 punto)** Diseño del algoritmo básico.
2. **(3 puntos)** Análisis y diseño de componentes del algoritmo Divide y Vencerás, o justificación de que el problema no puede resolverse por la técnica.
3. **(3 puntos)** Diseño del algoritmo Divide y Vencerás y de la función de combinación (en su caso).
4. **(2 puntos)** Análisis de eficiencia de los métodos básico y Divide y Vencerás. Justificación de si el diseño Divide y Vencerás realizado mejora al algoritmo básico
5. **(1 punto)** Implementación de los métodos básico y Divide y Vencerás. Pruebas de ejecución para resolver instancias de ejemplo propuestas por el estudiante.

La valoración de la práctica se dará como una calificación numérica entre 0 y 10. **Todos los problemas propuestos tendrán la misma calificación máxima de 10/3**.

5. Entrega y presentación de la práctica

Se deberá entregar un documento (memoria de prácticas) realizado en equipos de 3 personas, conteniendo los siguientes apartados:

1. Solución a los problemas propuestos, conteniendo los apartados descritos en el apartado 4 de este guión, para cada problema propuesto.

La práctica deberá ser entregada por PRADO, en la fecha y hora límite explicada en clase por el profesor. No se aceptarán, bajo ningún concepto, prácticas entregadas con posterioridad a la fecha límite indicada. La entrega de PRADO permanecerá abierta con, al menos, una semana de antelación antes de la fecha límite, por lo que todo alumno tendrá tiempo suficiente para entregarla. **La práctica deberá ser enviada por todos los integrantes del equipo de trabajo**.

El profesor, en clase de prácticas, realizará controles de las prácticas a discreción, que consistirán en presentaciones de los estudiantes de cada equipo (powerpoint) y/o entrevistas individuales con el fin de asegurar de que los estudiantes alcanzan las competencias deseadas. Estas entrevistas y/o presentaciones se realizarán en las sesiones de evaluación de prácticas, previamente anunciadas en clase por el profesor.

La **no asistencia** a una sesión de evaluación de prácticas por un estudiante supondrá la **calificación de 0 (no presentado) a la práctica que deba presentar, independientemente de la calificación obtenida en la memoria de prácticas**.

La presentación/entrevista de cada práctica podrá modificar la calificación final en un rango de calificaciones [-1, +1] con respecto a la calificación obtenida en la memoria de prácticas.

IMPORTANTE: Antes de las sesiones de evaluación, cada estudiante deberá prepararse para:

- Realizar una presentación powerpoint de la práctica de **máximo 5 minutos**.
- Conocer a fondo todos los algoritmos resueltos, así como los pasos para diseñar e implementar los algoritmos básico y DyV.

- Conocer las respuestas a las preguntas requeridas en el apartado 4 para cada problema.

El desconocimiento o la ausencia del material indicado podrá suponer la calificación de 0 en la práctica.