UNIVERSIDAD DE GRANADA

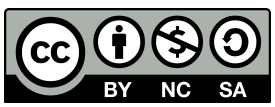
Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Práctica 2: Divide y Vencerás

Doble Grado Ingeniería Informática y Matemáticas

Autores:
Jose Alberto Hoces Castro
Javier Gómez López
Moya Martín Castaño

Abril 2022



Índice

1. Introducción

El objetivo de esta práctica es utilizar la técnica "divide y vencerás" para resolver problemas de forma más eficiente que otras alternativas más sencillas o directas. Para ello, se plantean los siguientes dos problemas:

- **Ejercicio 1:** Este problema consiste en realizar la búsqueda de un elemento en un vector ordenado con n elementos.
- **Ejercicio 2:** Este problema consiste en dados k vectores de n elementos, todos ellos ordenados de menor a mayor, combinar todos los vectores en uno único ordenado.

2. Desarrollo

Para los análsis de algoritmos que nos pedirán más adelante, hemos realizado los siguientes pasos:

1. Un **análisis teórico** de los algoritmos usando las técnicas vistas en clase.
2. Un **análisis empírico** donde hemos ejecutado los algoritmos en nuestros ordenadores bajo las misma normas y condiciones. Hemos compilado usando la compilación `-Og`. Además hemos usado como *datasets* de pruebas generadores de datos aleatorios proporcionados por la profesora. Por otro lado, para automatizar el proceso, hemos generado unos *scripts* de generación de datos de prueba y de ejecución de nuestros programas. Hemos ejecutado cada algoritmo 15 veces en cada uno de los tamaños que han sido probados, y hemos hecho la media de ellos para reducir perturbaciones que puedan alterar el resultado.
3. Un **análisis híbrido** donde hemos tomado los datos de cada uno de los alumnos del grupo y hemos hallado la K (constante oculta). Para ello hemos usado gnuplot.

2.1. Ejercicio 1

El enunciado del problema es el siguiente: *Dado un vector ordenado (de forma no decreciente) de números enteros v, todos distintos, el objetivo es determinar si existe un índice i tal que $v[i] = i$ y encontrarlo en ese caso. Diseñar e implementar un algoritmo "divide y vencerás" que permita resolver el problema. ¿Cuál es la complejidad de ese algoritmo y la del algoritmo "obvio" para realizar esta tarea? Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.*

Supóngase ahora que los enteros no tienen por qué ser todos distintos (pueden repetirse). Determinar si el algoritmo anterior sigue siendo válido, y en caso negativo proponer uno que sí lo sea. ¿Sigue siendo preferible al algoritmo obvio?

2.1.1. Algoritmo de fuerza bruta

La manera obvia de resolver este ejercicio sería mediante un algoritmo secuencial, que vaya recorriendo el vector hasta encontrar el elemento buscado. Pasemos a realizar un análisis de este algoritmo:

1. **Análisis Teórico**

```
int buscarSecuencial(int v[], int n){
    for (size_t i = 0; i < n; i++) //O(n)
    {
        if (v[i] == i){ //O(1)
            return i; //O(1)
        }
    }

    return -1;//O(1)
}
```

Tal y como se ha indicado en los comentarios del código, todas las operaciones de asignación y comprobación de los `if` son $\mathcal{O}(1)$. Estas, a sus vez, se incluyen dentro de un bucle `for`. Dicho bucle es $\mathcal{O}(n)$, obteniendo así que la función `int buscarSecuencial` es $\mathcal{O}(n)$, es decir

$$T(n) \in \mathcal{O}(n)$$

2. **Análisis Empírico**

Tras ejecutar el algoritmo para 26 tamaños distintos; desde 1000000 hasta 20000000 dando saltos de 760000, hemos obtenido los siguientes resultados:

Búsqueda secuencial	
Elementos (n)	**Tiempo (s)**

Cuadro 1: Experiencia empírica de algoritmo de Inserción sin optimizar

2.1.2. Divide y Vencerás

La oportunidad para usar la técnica divide y vencerás ocurre en el momento en el que trabajamos con vectores ordenados (en nuestro caso de manera creciente). Esto nos permite usar un algoritmo que se basa en la técnica "divide y vencerás": la búsqueda binaria.

La búsqueda binaria consiste en ir comprobando los elementos del vector comenzado por su mitad. Al estar el vector ordenado, si el valor que hemos comprobado es mayor que el buscado, podemos desechar la mitad superior de nuestro elemento. Aplicando esto de manera recursiva, obtenemos un algoritmo de características muy buenas e interesantes. Pasemos a su análisis:

1. **Análisis teórico**

```
int buscarBinaria(vector<int> v, int n){

    int inicio = 0; //O(1)
    int fin = n-1; //O(1)
    int medio = (inicio+fin)/2; //O(1)

    while(inicio <= fin){ //O(log(n))

        if(v.at(medio) > medio){ //O(1)
            fin = medio - 1; //O(1)
        }
        else if(v.at(medio) < medio){ //O(1)
            inicio = medio + 1; //O(1)
        }
        else{
            return medio; //O(1)
        }

        medio = (inicio + fin)/2; //O(1)
    }

    return -1; //O(1)
}
```

   Como podemos observar en los comentarios del código, estamos ante un bucle `while` cuyo interior es $\mathcal{O}(1)$. Puesto que en cada iteración del bucle vamos descartando una mitad del vector, el bucle en sí mismo es $\mathcal{O}(log(n))$. Por tanto, podemos afirmar que:

$$T((n) \in \mathcal{O}(log(n))$$

2. **Análisis empírico** Tras ejecutar el algoritmo para 26 tamaños distintos; desde 1000000 hasta 20000000 dando saltos de 760000, hemos obtenido los siguientes resultados:

   INSERTAR RESULTADOS BINARIA SIN REPETECIONES

3. **Análisis híbrido** El análisis híbrido nos permitirá comprobar que nuestro análisis teórico era correcto. Usando los datasets del anterior apartado, hemos usado gnuplot para graficar los puntos obtenidos junto con su función de ajuste. A continuación mostramos la gráfica con su respectiva función de ajuste:

   INSERTAR GRÇAFICA BINARIA SIN REPETICIONES

   Y las constantes ocultas son:

   INSERTAR FUNCION AJUSTADA

   Para terminar nuestro análisis de este algoritmo, terminaremos de confirmar que el ajuste logarítmico es el óptimo viendo la varianza residual que nos ha proporcionado gnuplot:

   INSERTAR VARIANZAS

   Como es muy próxima a 0, podemos asegurar que el ajuste es muy bueno.

2.1.3. ¿Elementos repetidos?

Otra parte que se nos plantea en el ejercicio es la siguiente: ¿qué pasaría si en el elemento tuviésemos elementos repetidos? En este caso, la búsqueda binaria tal y como la programamos antes dejaría de ser efectiva. Aquí un ejemplo:

$$1 \quad 2 \quad 3 \quad 4 \quad 4 \quad 5 \quad 6 \quad 7$$

En este caso, el primer elemento que comprobaría nuestro algoritmo sería $v[3] = 4$. Entonces, desecharía los siguiente elementos a partir de $v[3]$. Pero vemos que en $v[4] = 4$, que es el elemento que deseamos.

Para evitar este problema, hemos diseñado la siguiente solución:

```
int buscarBinaria(vector<int> v, int inicio,int fin){

    if(inicio == fin){
        if(v[inicio] == inicio){
                return inicio;
        }
        else{
                return -1;
        }
    }

    if(v[inicio] != inicio && v[fin-1] != fin-1){

        int medio = (inicio+fin)/2;
        int resultado = buscarBinaria(v,inicio+1,medio);
        if(resultado != -1){
                return resultado;
        }
        else{
               resultado = buscarBinaria(v,medio+1,fin-1);
               return resultado;
        }
    }
    else if(v[inicio] == inicio){

        return inicio;
    }
    else if(v[fin-1] == fin-1){

        return fin-1;
    }

    return -1;
}
```

Con este algoritmo se nos siguiente la siguiente ecuación de recurrencia:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) - 1$$

Resolviéndola mediante los métodos vistos en clase, obtenemos que:

$$T(n) \in \mathcal{O}(n)$$

Vemos que es el mismo orden de eficiencia que la búsqueda secuencial. Esto ocurre nuestro peor caso es que el elemento que buscamos este en $v[n-2]$. Puesto que vamos comprobando elementos siempre en las mitades inferiores hasta que encontramos lo deseado, para llegar a el elemento $v[n-2]$ tendríamos que hacer n iteraciones. Además, este algoritmo es incluso peor que la búsqueda secuencial, ya que en vectores muy grandes podemos llenar la pila de llamadas y hacer nuestro programa imposible de ejecutar.

2.2. Ejercicio 2

El enunciado del problema es el siguiente: *Se tienen k vectores ordenados (de menor a mayor), cada uno con n elementos, y queremos combinarlos en un único vector ordenado (con kn elementos). Una posible alternativa consiste en, utilizando un algoritmo clásico, mezclar los dos primeros vectores, posteriormente mezclar el resultado con el tercero, y así sucesivamente.*
¿Cuál sera el tiempo de ejecución de este algoritmo?
Existe un algoritmo más eficiente?. Si es así diseñe, analice la eficiencia e implemente dicho algoritmo.
Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

2.2.1. Algoritmo de fuerza bruta

Como se comenta en el enunciado, el algortimo de fuerza bruta de este ejercicio es ir mezclando los dos primeros vectores, después mezclar el resultante con el tercero y así sucesivamente.

Para ello la forma simple definir un array del tamaño final que sabemos que será k·n. Inicializamos el primer vector en las priemras n posiciones del array y vamos iterando en dicho array cada elemento de cada nuevo vector e ir comprobando si es menor que alguno de los ya insertados previamente. Si el elemento es menor se realiza una traslación hacia la derecha desde la posición en la que ha resultado ser menor hasta el último elemento insertado en el array. Si no es menor, entonces se insertará en la siguiente posición al útlimo insertado. De esta forma se va expandiendo hasta completar las k*n posiciones del array.

1. **Análisis Teórico**

```
void mergeKArrays(int nElementos, int **arr, int nVectores, int * &v_resultante)
{

    int k = 0;
    bool encontrado;
    for(int i = 0; i < nElementos; i++){
      v_resultante[i] = arr[0][i];
    }

    for(int i = 1; i < nVectores; i++){          //O(k)
      for(int j = 0; j < nElementos; j++){       //O(n)
        encontrado = false;
        k = 0;
        while(k < nElementos * i + j && !encontrado){    //O(kn + n)
          if(v_resultante[k] > arr[i][j])
            encontrado = true;
          else
            k++;
        }

        if(encontrado){
          for(int l = nElementos*i+j-1; l >= k; l--){    //O(kn)
            v_resultante[l+1] = v_resultante[l];
          }
          v_resultante[k] = arr[i][j];
        }
        else{
          v_resultante[nElementos * i + j ] = arr[i][j];
        }
      }
    }

}
```

Tal y como se ven en los comentarios del código, el bucle principal está formado por dos bucles anidados cuyos tamaños son $\mathcal{O}(k)$ y $\mathcal{O}(n)$ respectivamente y dentro de ellos tenemos dos bucles en paralelo de tamaños $\mathcal{O}(kn+n)$ y $\mathcal{O}(kn)$. Luego la eficiencia teórica del algoritmo, según lo visto en clase, es evidente que es:

$$T((n) \in \mathcal{O}(k^2n^2)$$

2.2.2. Divide y Vencerás