Estructura de Datos y Algoritmos

Algoritmos Recursivos de Búsqueda y Ordenación y sus tiempos

1. Algoritmos de ordenación recursivos

1.1. Mergesort, Ordenamiento por fusión

Mergesort se ejecuta en un tiempo de $O(nlogn)$ en el peor caso, donde n es el tamaño del array a ordenar. La cantidad de comparaciones realizadas es cerca de optima. Es un ejemplo bueno de algoritmo recursivo.

La operación fundamental del algoritmo es la de fusionar dos arrays ordenados. Como los arrays están ordenados esto se puede hacer en una pasada a través de los arrays. La salida es puesta en un tercer array.

Tendremos dos arrays de entrada A y B y un array de salida C y tres contadores $Aptr$, $Bptr$ y $Cptr$ que están inicializados al comienzo de los arrays respectivos. El elemento menor entre $A[Aptr]$ y $B[Bptr]$ es copiado a $C[Cptr]$ y los contadores que corresponda son incrementados. Cuando uno de los arrays de entrada fue copiado totalmente, el resto del otro array se copia a C.

El tiempo de realizar el merge de dos arrays es lineal porque a lo maximo se realizan $n - 1$ comparaciones donde n es la cantidad total de elementos. Cada comparación agrega un elemento a C excepto la última que agrega dos.

El algoritmo de ordenamiento por fusión es sencillo de describir. Si $n = 1$ la repuesta es dicho elemento, sino divido el array en dos mitades. Cada mitad es ordenada recursivamente, esto me da dos mitades ordenadas que pueden ser fusionadas utilizando el programa de fusión descripto antes.

Presentamos el programa abajo:

```
void sort_por_fusion(int A[], int B[], int izq, int der)
{
      int centro;

      if (izq < der) {
        centro=(izq+der)/2;
        sort_por_fusion(A,B,izq,centro);
        sort_por_fusion(A,B,centro+1,der);
        fusion(A,B,izq,centro+1,der);
        }
}
```

```
void fusion(int A[], int B[], int izq, int centro, int der)
```

```
{
    int final_izq, nroelem, tmp;

    final_izq = centro - 1;
    tmp = izq;
    nroelem=der-izq+1;

    while ((izq <= final_izq) && (centro <= der))
    {
        if (A[izq] < A[centro])
          {
          B[tmp]=A[izq];
          izq++;}
        else
            {
             B[tmp]=A[centro];
             centro++;
            }
            tmp++;
            }
    while (izq <= final_izq)
        {
         B[tmp]=A[izq];
         tmp++;
         izq++;
        }
    while (centro <= der)
        {
         B[tmp]=A[centro];
         tmp++;
         centro++;
        }
    for(int i=1;i<=nroelem;i++)
       {
        A[der]=B[der];
        der- -;
       }
}
```

1.1.1. Análisis del ordenamiento por fusión

El ordenamiento por fusión es un ejemplo clásico de las técnicas usadas para analizar programas recursivos. Escribiremos una relación de recurrencia para el cálculo del tiempo de ejecución.

Asumimos n es potencia de 2 de modo que siempre dividimos la entrada en dos mitades. Para $n = 1$ el tiempo es constante, luego lo denotaremos por 1. En otro caso el tiempo de ordenar n números es igual al tiempo de realizar dos ordenamientos recursivos de tamaño $n/2$, más el tiempo de fusion que es lineal.

Obtenemos la siguiente recurrencia:

$T(1) = 1$
$T(n) = 2T(n/2) + n$

resolvamos la recurrencia:

$T(n) = 2T(n/2) + n$ sustituimos n por n/2, obtenemos
$T(n/2) = 2T(n/4) + n/2$ luego
$2T(n/2) = 4T(n/4) + n$ sustituimos n por n/2, obtenemos
$2T(n/4) = 4t(n/8) + n/2$ luego
$4T(n/4) = 8T(n/8) + n$
tenemos :
$T(n) = 2T(n/2) + n = 4T(n/4) + 2n = 8T(n/8) + 3n$
si continuamos de esta manera obtenemos:
$T(n) = 2^k T(n/2^k) + k.n$

como $n = 2^k$ entonces $k = log_2 n$, obtenemos:

$T(n) = nT(1) + nlog_2 n = nlog_2 n + n = O(nlog_2 n)$

1.2. Quicksort, Ordenamiento rápido

Como implica el nombre el ordenamiento rápido es el programa de ordenamiento más rápido conocido en práctica. Su tiempo de ejecución promedio es $O(nlogn)$. Tiene un peor caso de $O(n^2)$ pero este caso es muy poco probable.

El algorimto de ordenamiento rápido es simple de entender y probar correcto. El algoritmo básico consiste de los siguientes pasos

1. si el número de elementos es 0 o 1 está ordenado.
2. elija un elemento cualquiera en la entrada, este elemento es llamado *pivote*.
3. particionar el conjunto sin el pivote en dos conjuntos disjuntos: los menores o iguales a él y los mayores o iguales a él.
4. devolver la ordenaci'on del primero de los conjuntos seguido del pivote seguido de la ordenación del segundo de los conjuntos.

sort_rapido(i,j) ordena desde $A[i]$ hasta $A[j]$ en su lugar.

Comenzamos definiendo una función hallo_pivote. Si hallo_pivote no encuentra dos valores distintos retorna 0, sino retorna el indice del mayor de los primeros dos elementos distintos. La función es como sigue:

```
int hallo_pivote(int A [],int i,int j)
{
    int clave1,k;

    clave1=A[i];
    for (k=i+1;k<=j;k++)
        if (A[k]>clave1) return k;
        else if (A[k]<clave1) return i;
```

```
    return 0;
}
```

A continuación consideremos el problema de ordenar de $A[i]$ a $A[j]$ en su lugar de modo que las claves menores que el pivote aparezcan a la izquierda que las otras. Para hacer esto introducimos dos cursores izq y der inicialmente indicando los extremos izquierdos y derechos de la porción de A que estamos ordenando respectivamente. En todo momento los elementos a la izquierda de izq esto es $A[i], \ldots, A[izq-1]$ tendrán claves menores que el pivote y todos los elementos a la derecha de der, esto es $A[der+1], \ldots, A[j]$ tendrán claves iguales o mayores que el pivote.

Inicialmente $i = izq$ y $j = der$ de forma que las sentencias anteriores se cumplen pues no hay nada a la izquierda de izq ni a la derecha de der. Aplicamos repetidamente los siguientes pasos que mueven izq a la derecha y der a la izquierda hasta que se cruzan despues de lo cual $A[i], \ldots, A[izq-1]$ va a contener todas las claves menores que el pivote y $A[der+1], \ldots, A[j]$ todas las claves mayores que el pivote:

1. búsqueda: mover izq a la derecha mientras $A[izq]$ sea menor que el pivote. Mover der a la izquierda mientras $A[der]$ sea mayor o igual al pivote.

2. testeo : si $izq > der$ (que significa $izq = der + 1$) hemos particionado exitosamente la entrada.

3. cambio : si $izq < der$ intercambiar $A[izq]$ con $A[der]$. Luego de hacer esto $A[izq]$ tiene una clave menor que el pivote y $A[der]$ tiene una clave al menos igual al pivote de modo que sabemos que en la siguiente búsqueda izq se va a mover al menos una posición a la derecha y der se va a mover al menos una posición a la izquierda.

```
int particion (int A[],int i,int j,int pivote)
{
    int izq,der;

    izq=i;
    der=j;
    do
    {
        intercambio(A,izq,der);
        while(A[izq] < pivote) izq++;
        while(A[der] >= pivote) der- -;
    }
    while (izq <= der);
    return izq;
}
```

el intercambio inicial no interesa ya que no asumimos ningún orden en la entrada al comienzo del programa.

El programa final es el siguiente:

```
void sort_rapido (int A[], int i, int j)
{
    int pivote,indice,k;

    indice=hallo_pivote(A,i,j);
    if (indice! =0)
    {
      pivote=A[indice];
      k=particion(A,i,j,pivote);
      sort_rapido(A,i,k-1);
      sort_rapido(A,k,j);
    }
}
```

1.2.1. Análisis del ordenamiento rápido

Tiempo de particion:

a cada elemento asociamos un tiempo que veremos es constante. Luego el tiempo no es mayor que esa constante por el número de items.

En particion los items son los elementos de $A[i]$ a $A[j]$. La constante es el tiempo desde que izq o der apunta por primera vez al elemento hasta que deja el elemento. Hay que notar que una vez que se deja un elemento no se vuelve a el.

Dejamos un elemento ya sea aumentando izq o dismuyendo der. Cuanto tiempo puede pasar hasta que ejecutamos estas sentencias? en el peor caso ejecutamos: la inicialización de izq y der, un intercambio, podemos testear los loops sin modificar izq ni der, en una segunda pasada se ejecuta el intercambio lo que nos asegura que los loops interiores van a modificar izq y der. Esto es un tiempo constante independiente de i y j.

Tiempo de quicksort:

el tiempo de hallo_pivote es $O(j-i+1)$ y en muchos casos más pequeño. La llamada a particion lleva tiempo $O(j-i+1)$, entonces el tiempo de cada llamada individual a quicksort tiene tiempo proporcional al numero de elementos.

Dicho de otra forma, el tiempo total que toma quicksort es la suma sobre todos los elementos de la cantidad de veces que el elemento es parte de un sub-array para el cual una llamada recursiva se realizó.

En el peor caso, el pivote coincide con el elemento mayor del array, esto divide el array en uno de n-1 elementos y el otro de 1. El elemento r_i forma parte de $n-i+1$ llamadas luego el tiempo está dado por:

$\Sigma_{i=1}^{n}(n-i+1) = \Sigma_{i=1}^{n} n - \Sigma_{i=1}^{n} i + \Sigma_{i=1}^{n} 1 =$
$n^2 - n(n+1)/2 + n = 2n^2/2 - n^2/2 - n/2 + n =$
$n^2/2 + n/2$

Luego en el peor caso quicksort toma un tiempo proporcional a n^2 (ambos $O(n^2)$ y $\Omega(n^2)$).

El tiempo promedio de quicksort es $O(nlog_2n)$.

2. Algoritmos de Búsqueda

2.1. Búsqueda lineal recursiva

La búsqueda lineal compara los elementos del array con la clave de búsqueda hasta que encuentra el elemento o bien hasta que se determina que no se encuentra.

```
int busqueda_lineal (int A[], int clave, int n, int i)
{
            if (i==n+1) return -1;
            elseif (A[i]==clave) return i;
                else return busqueda_lineal(A,clave,n,i+1);
            return -1;
}
```

La función se invoca inicialmente con : busqueda_lineal(A,n,1).

2.1.1. Análisis del algoritmo de búsqueda lineal

En el peor caso el elemento se encuentra en la última posición o no se encuentra. Esto da la siguiente recurrencia: (T(j) corresponde a cuando el elemento está en la posición j)

T(1) = 1
T(2) = T(1) + 1
⋮
T(n-1) = T(n-2) + 1
T(n) = T(n-1) + 1

luego

T(n) = T(n-1) + 1 = T(n-2) + 2 = ... = T(1) + (n-1) = n

luego la búsqueda lineal es $O(n)$ y $\Omega(n)$.

2.2. Búsqueda binaria

Dados un entero X y un array A de n enteros que se encuentran ordenados y en memoria, encontrar un i talque $A[i] == X$ o retornar 0 si X no se encuentra en el array (consideramos los elementos del array de 1 a N.

La estrategia es la misma que en el caso iterativo.
Para simplificar el código definimos A[0]=X.

```
int busqueda_binaria(int A[],int X, int i, int j)
{
        int medio;

        if (i>j) return 0;
        medio = (i+j) / 2;
        if (A[medio] < X) return busqueda_binaria(A,X,medio+1,j)
        else if (A[medio] > X) return busqueda_binaria(A,X,i,medio-1)
            else return medio;
}
```

2.2.1. Análisis del algoritmo de búsqueda binaria

El tiempo del programa sin contar las llamadas recursivas es constante. Si planteamos una recurrencia para calcular el tiempo es:

$T(n) = T(n/2) + 1$
$T(n/2) = T(n/4) + 1$
$\vdots$
$T(2) = T(1) + 1$
$T(1) = 1$
$T(0) = 1$

o sea la misma que en el caso iterativo. Luego el tiempo es $O(log_2 n)$.