Estructura de Datos y Algoritmos

Algoritmos Iterativos de Búsqueda y Ordenación y sus tiempos

1. Algorimos de ordenación

Discutiremos el problema de ordenar un array de elementos. A los efectos de simplificar asumiremos que los arrays contienen solo enteros aunque obviamente estructuras más complicadas son posibles. Asumiremos también que el ordenamiento completo se puede realizar en memoria principal o sea la cantidad de elementos es relativamente pequeño (menos de un millón).

Ordenamientos que no se pueden realizar en memoria deben realizarse en disco. Se encuentran ejemplos en la bibliografía.

1.1. Preliminares

Los algoritmos que describiremos reciben como argumentos un array que pasa los elementos y un entero que representa la cantidad de elementos. Asumiremos que N (el número de elementos) es un número legal. Para alguno de los programas que veremos será conveniente utilizar un centinela en posición 0, por lo cual nuestros arreglo irán del 0 al N. Los datos irán del 1 al N.

Asumiremos la existencia de operadores de comparación $<$ y $>$. Llamaremos al ordenamiento que se basa en el uso de estos operadores *ordenamiento basado en comparaciones*.

1.2. Ordenamiento por inserción (Insertion Sort)

Es uno de los algoritmos más simples. Consiste en $N-1$ pasadas. En las pasadas 2 a N se cumplirá que los elementos de las posiciones 1 a N están ordenados. En la pasada P movemos el elemento P-esimo a su lugar correcto, este lugar es encontrado en las posiciones de los elementos 1 a P. El programa es el siguiente:

```
void Sort_por_insercion (int A[], int N)
{
        int j,P,tmp;
        {
            A[0]=Min_int;
            for(P=2;P<=N;P++)
                {
                 j=P;
                 tmp=A[P];
                 while(tmp < A[j-1])
```

```
                {
                        A[j] = A[j-1];
                        j-=1;
                }
                A[j]=tmp;
               }
        }
}
```

La idea del programa es la siguiente: coloco un centinela en la primer posición (posición 0) por lo cual el elemento a insertar será colocado en caso de que sea el mínimo en la posición 1. Guardamos el valor del elemento a insertar en una variable auxiliar tmp. Si tmp es menor que A[j-1] su posición será anterior o igual a (j-1), luego corro el elemento de la posición (j-1) a la posición j para hacer lugar para tmp. Repito esto para todos los elementos anteriores al tmp. Cuando encuentre una posición tal qué tmp no es menor indica que los elementos anteriores están ordenados (estaban ordenados del 0 al $P-1$). Luego lo unico que resta es colocar tmp en la posición j.

1.2.1. Análisis del tiempo del ordenamiento por inserción

El while interior se ejecuta P veces en el peor caso (su complejidad es P). Esto se hace para P de 2 a N (tiempo del for), luego el tiempo del for es:

$\Sigma_{P=2}^{N} P = 2 + 3 + 4 + \ldots + N = (N)(N+1)/2 - 1$

luego el tiempo es $O(N^2)$.

Si la entrada estaba ordenada en forma creciente cuando se llamó el programa, estoy en el mejor caso cuyo tiempo es $O(N)$ ya que la comparación del while da siempre falso, mientras que si estabá ordenada en forma decreciente y quiero ordenarlo en creciente tengo el peor caso de $O(n^2)$.

También se cumple que los tiempos son exactos, o sea el tiempo es $\Omega(n^2)$ en el peor caso y $\Omega(n)$ en el mejor caso.

1.3. Ordenamiento por Selección (Selection Sort)

La idea del selection sort es la siguiente : en la pasada i-esima seleccionamos el elemento menor entre $A[i], \ldots, A[n]$ y lo intercambiamos con el $A[i]$. Como resultado, luego de i pasadas los menores i elementos ocuparán las posiciones $A[1], \ldots, A[i]$ y además los elementos de dichas posiciones estarán ordenados. El programa es el siguiente:

```
void Sort_por_seleccion (int A[], int N)
{
        int i,j,sel,clave_sel;
        {

            for(i=1;i<N;i++)
```

```
                {
                 sel=i;
                 clave_sel=A[i];
                 for(j=i+1;j<=N;j++)
                 {
                    if (A[j]<clave_sel)
                      { clave_sel=A[j];
                        sel=j;
                      }
                 }
                 intercambio(A[i],A[sel]);
                }
        }
}
```

1.3.1. Análisis del tiempo del ordenamiento por selección

El programa ejecuta siempre la misma cantidad de instrucciones. El tiempo del for interior es $(N - i)$, la suma de este tiempo cuando varía i es

$$\Sigma_{i=1}^{N-1}(N-i) = \Sigma_{i=1}^{N-1}N - \Sigma_{i=1}^{N-1}i = (N-1).N - (N-1).N/2 = (N-1).N/2$$

luego el tiempo es $O(N^2)$ y $\Omega(N^2)$.

Veremos otros ejemplos de Ordenamiento en los cuales los programas en cuestión son *recursivos*. Veremos dichos programas una vez que hayamos estudiado recurrencias.

2. Algoritmos de Búsqueda

Con frecuencia el programador trabajará con grandes cantidades de información almacenada en arreglos. Podría ser necesario determinar si algún arreglo contiene un valor que sea igual a cierto valor clave.

El proceso para encontrar un elemento particular en un arreglo se llama búsqueda. Estudiaremos dos técnicas de búsqueda: una técnica simple llamada busqueda lineal y una más eficiente llamada busqueda binaría.

Ambos programas se pueden implementar recursivamente o no. En este capítulo veremos la implementación no recursiva.

2.1. Búsqueda lineal

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)
{
        for(int i=0;i<n;i++)
            if (A[i]==clave) return i;
        return -1;
}
```

Este método funciona bien con arreglos pequeños y con los no ordenados. En arreglos grandes u ordenados conviene aplicar la búsqueda binaria que es más eficiente.

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 el siguiente tiempo

$\Sigma_{i=0}^{n-1} 1 = n$

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 consiste en comparar X con el elemento del medio del array, si es igual entonces encontramos el elemento, sino, cuando X es menor que el elemento del medio aplicamos la misma estrategia al array a la izquierda del elemento del medio y si X es mayor que el elemento del medio aplicamos la misma estrategia al array a la derecha de dicho elemento.

Para simplificar el código definimos A[0]=X.

```
int busqueda_binaria(int A[],int X,int n)
{
        int izq=1,medio,der=n;

        A[0]=X;
        do
        {
            medio = (izq+der) / 2;
            if(izq > der)
              medio=0;
            else if A[medio] < X
                    izq = medio +1;
                 else der=medio-1;
        }
```

```
            while (A[medio] != X);
            return medio;
}
```

2.2.1. Análisis del algoritmo de búsqueda binaria

El tiempo del programa dentro del loop es O(1), luego el análisis requiere determinar cuantas veces se ejecuta el loop. El loop comienza con $der - izq = n - 1$ y termina con $izq > der$ (en el peor caso). A lo largo del loop el valor $der - izq$ se divide al menos a la mitad de su valor anterior luego la cantidad de iteraciones será de la forma $log_2 n$. Veremos esto

Podemos escribir una recurrencia para calcular el tiempo de ejecución.

T(n) = T(n/2) + c
T(n/2) = T(n/4) + c
⋮
T(2) = T(1) + c
T(1) = c
T(0) = c

consideremos $c = 1$ ya que las operaciones dentro del loop son $O(1)$. Simplificamos y obtenemos $T(n) = T(0) + k = (k + 1)$. Tenemos que hallar el valor de k que coincide con la cantidad de veces que dividimos entre 2 comenzando en n y terminando en 1.

Supongamos n es de la forma 2^k (para hallar el tiempo consideramos aquel k tal que 2^k es una cota superior de n). Luego dividiremos entre 2 a lo más k veces. Pero si $n = 2^k$, $log_2 n = k$. Luego el tiempo de ejecución es de la forma $log_2 n + 1$.

Para demostrar es $O(log_2 n)$ tenemos que encontrar c_1 tal que $log_2 n + 1 \leq c_1.log_2 n$. Tomo $c_1 = 2$ luego $log_2 n + 1 \leq log_2 n + log_2 n$ cuando $1 \leq log_2 n$ que es cierto para $n \geq 2$.

Mejor caso = elemento en el medio del array. Peor caso = elemento no se encuentra. El tiempo que calculamos corresponde a la cota superior del peor caso. La cota inferior en el peor caso coincide (realizamos las mismas operaciones).