CC371:3.- Practica Calificada

Divide & Vencerás

Brando Miguel Palacios Mogollon
bpalaciosm@uni.pe

Universidad Nacional de Ingenieria
Escuela Profecional de Ciencias de la Computacion

July 9, 2020

Overview

Pregunta 1

Pregunta 2

Contenido

Solución:

Pregunta 1

Diseñe la solución aplicando divide y vencerás, determine la complejidad y programe dicho ordenamiento.

Inserción binaria

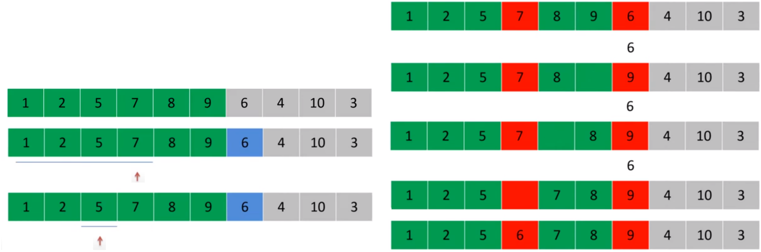
La inserción binaria es el método de la inserción obtiene una mejora al emplear un búsqueda binaria para determinar la ubicación para insertar elementos.

Ventaja sobre inserción directa:

- La posición de inserción se encuentra rápidamente, pero el desplazamiento de los elementos sigue existiendo.
- Menor numero de comparaciones.

Este método se llama inserción binaria; fue mencionado por John Mauchly ya en 1946, en la primera discusión publicada sobre clasificación de computadoras

Ilustración



(a) Búsqueda binaria (b) Inserción

Figure 1: Algoritmos de Insercion Binaria

Algoritmo Pascal

```
bin_search(arr,x,start,end)
BEGIN
 mid:=(start+end)/2;
 if (arr[mid]==x)
   return mid;
 if (start==end)
   return -1;
 if (arr[mid]> x)
   return bin_search(arr,x,start,mid-1);
 if (arr[mid]< x)
   return bin_search(arr,x,mid+1,end);

END;{bin_search}
```

Algoritmo Pascal

```
BEGIN
 for i:=1 to N do
    begin
        key:=v[i]
        j:=i-1
        loc:= binarySearch(v[],key,0,j);
        while (j>=loc) do
        begin
            v[j+1]:=v[j];
            j:=j-1
        end;{while}

        v[i+1]:=key

    end;{for}

end;{Insercion}
```

Costo	#num ejec
c_2	n
c_4	n - 1
c_5	n - 1
c_6	$k + c\log(n)$
c_7	$\sum_{j=1}^{n}(t_j)$
c_9	$\sum_{j=1}^{n}(t_j - 1)$
c_10	$\sum_{j=1}^{n}(t_j - 1)$
c_13	n - 1

Figure 3: Algoritmo Inserción Binaria

FACULTAD DE CIENCIAS

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

Obteniendo el patron:

$$T(n) = T(\frac{n}{2^i}) + i \cdot c$$

$$\Rightarrow \frac{n}{2^i} = 1$$

$$n = 2^i \rightarrow \log(n) = i$$

$$T(n) = T(\frac{n}{2^{\log(n)}}) + c \cdot \log n$$

$$T(n) = T(1) + c \log(n) = k + c \log(n) \longrightarrow T(n) \in \mathcal{O}(log(n))$$

Entre la linea 2 a 15 se realiza $n-1$ iteraciones de un bucle externo. En cada una de estas iteraciones, se realiza una búsqueda binaria para determinar la posición en la que se realiza la insercion (bin_search-diapositiva 7). En la $i^t h$ iteracion del bucle externo, la busqueda binaria considera un tiempo de ejecucion de $\mathcal{O}(\log(i))$ una vez encontrada la posicion correcta, necesitara permutar para insertar el elemento en su lugar.
El numero total de comparaciones es:

$$\sum_{i=1}^{n} \log i = (n+1)(\log(n+1)) + 2^{\log(n+1)+1} + 2 \tag{1}$$

$$= \mathcal{O}(n \log n)$$

Dado que la rutina de clasificación de inserción binaria siempre realiza la búsqueda binaria, su mejor tiempo de ejecución es $\mathcal{O}(n \log n)$

Para un elemento j, harias $\log j$ comparaciones y en un peor caso, j turnos.

$$\sum_{j=1}^{n}(j + \log j) = \frac{n(n+1)}{2} + \log(n!) = O(n^2 + n\log n) = O(n^2)$$

El trabajo lineal de desplazamiento triunfa sobre el trabajo logaritmico de comparacion. Y aunque este termine haciendo menos comparaciones, sigue siendo una cantidad lineal de iteracion y por consiguiente esta complejidad no cambia.

Algoritmo e Implementación

```
int binarySearch(int arr[], int item, int low, int high) {
    if (high <= low)
        return (item > arr[low])? (low + 1): low;
        int mid = (low + high)/2;
    if(item == arr[mid])
        return mid+1;
    if(item > arr[mid])
        return binarySearch(arr, item, mid+1, high);
        return binarySearch(arr, item, low, mid-1);
}
void BinaryInsertionSort(int arr[], int n) {
    int i, loc, j, k, selected;
    for (i = 1; i < n; ++i) {
        j = i - 1;
        selected = arr[i];
        loc = binarySearch(arr, selected, 0, j);
        while (j >= loc) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = selected;
    }
}
```

Complejidad

- Peor Caso $\mathcal{O}(n^2)$
- Caso Medio $\mathcal{O}(n^2)$
- Mejor Caso $\mathcal{O}(n\log(n))$

Contenido

EL problema del torneo de tenis

Objetivos:

Organizar un torneo con n participantes. Cada participante tiene que competir exactamente una vez con todos los posibles $n-1$ oponentes. Además, cada participante tiene que jugar exactamente un partido cada dia, con la excepción de un solo dia en el cual pueda tomar descanso del torneo.

Por simple análisis:

- Puede suponerse que la competición se realiza en dias sucesivos y cada participante compite 1 vez por dia.
- El torneo se debe completar en el menor número posible de dias.
- Si n es potencia de 2 ($n = 2^k$), el algoritmo a implementar se podra realizar en $n-1$ dias, en caso sea impar en n.
- Se planifica la pareja de cada dia, de modo que todos jueguen contra todos sin repetir, ni descansar innecesariamente.

Análisis:

Inicialmente, suponemos n par.y buscando el caso mas simple ($n = 2$), tenemos 2 jugadores en la que solo da opción a al juego de ambos. De ser un valor mayor ($n > 2$) utilizaremos "**Divide y vencerás**" para realizar la tabla de encuentros, pensando en que se tiene ya una solución para la midad de los jugadores, dividiendo la tarea por cuadrantes. Podemos ensayarlo de la siguiente forma:

- El cuadrante inferior debe enfrentar a los jugadores de numero superior entre ellos, por lo que se obtiene sumando $n/2$ a los valores del cuadrante superior.

- El cuadrante superior derecho enfrenta a los jugadores con menores y mayores números, y se puede obtener enfrentando a los jugadores numerados 1 a $\frac{n}{2}$ contra $\frac{n}{2}+1$ a n respectivamente en el dia $\frac{n}{2}$, y después rotando los valores $(\frac{n}{2})+1$ a n cada dia.
- El cuadrante inferior derecho enfrenta a los jugadores, de mayor número contra los de menor número, enfrentando a los de $[(n/2)+1, n]$ contra $[1, n/2]$ en el dia $n/2$, para luego rotar los valores de $[1, n]$ cada dia en el sentido contrario del superior derecho.

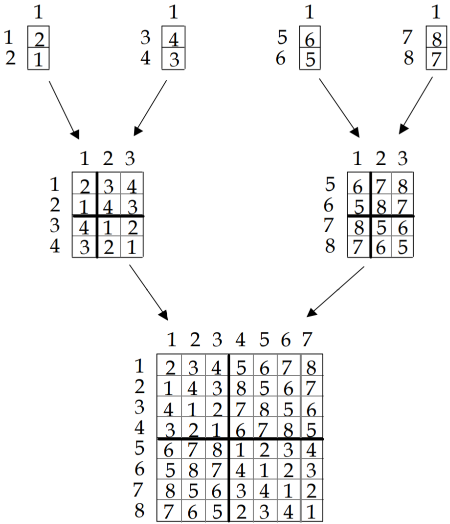


Figure 5: Procedimientos del torneo aplicado al divide y vencerás

Casos:

De momento hemos analizado los casos, en el que el valor de n es potencia de 2 es decir 2^k, por lo que los cuadrantes serán del mismo tipo, no obstante existen variantes necesarias de aclarar.

- Si n es impar $(n > 1)$, no sera suficiente tener $n - 1$ dias, sino es necesario n. Este problema se ve solucionado al añadir un participante **ficticio** $(n + 1)$. Al ser $n + 1$ par, los dias constaran de n. Los dias donde el participante i-ésimo le toque con el jugador n+1 en el j-ésimo dia, este significa un dia de descanso para el participante.
- Si n es par pero n *div* 2 es impar, debemos emplear lo siguiente:
  - Dado que en en alguna parte de la numero existira un cuadrante impar, a este se le añade un participante ficticio.
  - El dia j-esimo, $1 \leq\leq n$ div 2, se hace jugar entre si a los dos participantes, uno de numeracion inferior y otro superior, a los que les habia tocado el mismo dia de descanso.
  - Se realiza los mismo pasos que un n potencia de 2.

Ilustración

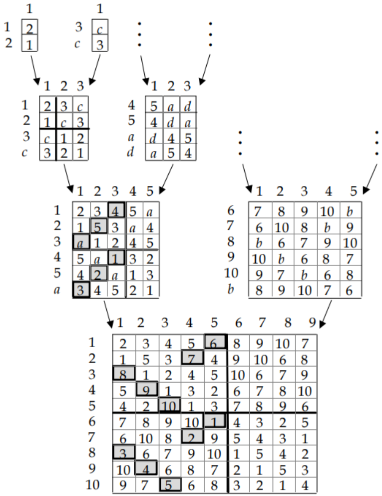


Figure 6: Procedimientos del torneo aplicado al divide y vencerás

Algoritmo

```
 CONST MAXJUG =...; (* numero maximo de jugadores *)
TYPE cuadrante = ARRAY [1..MAXJUG],[1..MAXJUG] OF
CARDINAL;
PROCEDURE Torneo(n:CARDINAL;VAR tabla:cuadrante);
VAR jug,dia:CARDINAL;
BEGIN
  IF n=2 THEN
   tabla[1,1]:=2;
   tabla[2,1]:=1;
  ELSE    Torneo(n DIV 2,tabla); (*llamada recursiva*)
   (* despues el cuadrante inferior izquierdo *)
   FOR jug:=(n DIV 2)+1 TO n DO
    FOR dia:=1 TO (n DIV 2)-1 DO
     tabla[jug,dia]:=tabla[jug-(n DIV 2),dia]+(n DIV 2);
    END;
   END;
   (* luego el cuadrante superior derecho *)
```

Algoritmo

```
    FOR jug:=1 TO (n DIV 2) DO
    FOR dia:=(n DIV 2) TO n-1 DO
     IF (jug+dia)<=n THEN tabla[jug,dia]:=jug+dia
     ELSE tabla[jug,dia]:=jug+dia-(n DIV 2)
     END;
    END;
   END;
   (* y finalmente el cuadrante inferior derecho *)
   FOR jug:=(n DIV 2)+1 TO n DO
    FOR dia:=(n DIV 2) TO n-1 DO
     IF jug>dia THEN tabla[jug,dia]:=jug-dia
     ELSE tabla[jug,dia]:=(jug+(n DIV 2))-dia
     END;
    END;
   END;
  END;
 END; (*IF*)
END Torneo;
```

La función, concurre en la función de recursividad, y como cada, una de las 3 partes tiene una complejidad $\mathcal{O}(n^2)$:

$$T(n) = \begin{cases} 1, & \text{si } n = 2 \\ 2 \cdot T(n/2) + n^2/4, & \text{si } n \geq 2 \end{cases}$$

De donde:

$$T(n) = 2 \cdot T(n/2) + n^2$$

Para el caso recursivo y comando el caso recursivo
$n = 2, k = 2, a = 2.$
Por el teorema maestro, podemos decir que:

$$T(n) = 2 \cdot T(\frac{n}{2}) + f(n^2) \longrightarrow T(n) \in \theta(n^2)$$