GEPROC-ULA

Grupo de Entrenamiento de Programación Competitiva ULA

Universidad de Los Andes

Arreglos

- **Arreglo**: **Colección** de elementos de un mismo tipo.
- Es una estructura de datos.
- Compensaciones:
  - Acceso por posición (acceso aleatorio) a los elementos.
  - Fácil de manipular.
  - Poco flexible.
  - Tamaño estático o dinámico.
- El arreglo es una de las estructuras de datos mas comunes. Muchas otras estructuras de datos están compuestas con arreglos o especializaciones.

Usos comunes

- Recorrido ascendente/descendente.
- Recorrido con saltos.
- Ordenamiento ascendente/descendente u otro criterio.
- Intercambios de elementos internos. (swap)
- Búsqueda de elementos:
  - Búsqueda del elemento X.
  - Menor/Mayor.
  - Menor/Mayor **frecuencia**.
- Búsqueda de posición(es) dado un valor(es) o un patrón(es).
- Búsqueda de patrones.
- Búsqueda de **sub-arrays**.
- Búsqueda de **conjuntos**.
- Conteo de **patrones**.

Conceptos clave

- **Frecuencia**: Número de veces en que dicho evento se repite durante un experimento o muestra estadística.
- **Sub-array**: Parte o sección de un arreglo.
- **Conjunto** : Colección de ciertos valores, sin ningún orden concreto ni valores repetidos.

arreglo: tamaño 13

23	23	4	6	9	1	21	1	0	12	21	4	5
0	1	2	3	4	5	6	7	8	9	10	11	12

- ¿Cúal es la frecuencia de cada uno de los elementos del arreglo?
- Determine un sub-arreglo.
- Determine el conjunto de elementos del arreglo.
- ¿Cúal es el sub-arreglo de tamaño 3 cuya suma es mayor?

Algoritmos básicos sobre arreglos:

- **Búsqueda binaria**: Es un algoritmo de búsqueda que encuentra la posición de un valor en un arreglo ordenado.

```
1 int binarySearch(int arr[], int l, int r, int x){
2 while (l <= r)
3   {
4         int m = l + (r-l)/2;
5
6         // Check if x is present at mid
7         if (arr[m] == x)
8         return m;
9
10        // If x greater, ignore left half
11        if (arr[m] < x)
12        l = m + 1;
13        // If x is smaller, ignore right half
14        else
15        r = m - 1;
16  }
17        // if we reach here, then element was
18        // not present
19        return -1;
20
```

Para el siguiente arreglo aplicar el algoritmo de búsqueda binaria. X = **19**.

1	**3**	**5**	**7**	**9**	**11**	**13**	**15**	**17**	**19**	**21**	**23**
l=0	1	2	3	4	m=5	6	7	8	9	10	r=11

1	3	5	7	9	11	**13**	**15**	**17**	**19**	**21**	**23**
0	1	2	3	4	5	l=6	7	m=8	9	10	r=11

1	3	5	7	9	11	13	15	17	**19**	**21**	**23**
0	1	2	3	4	5	6	7	8	l=9	m=10	r=11

1	3	5	7	9	11	13	15	17	**19**	21	23
0	1	2	3	4	5	6	7	8	l = r = **m = 9**	10	11

- La búsqueda binaria es usada en muchos problemas de diferentes tipos.
- Esta definida en la biblioteca estándar de C. **(stdlib.h)**.
- No obstante,la mayoría de veces es necesario modificar el algoritmo para satisfacer ciertos criterios.

```c
/* bsearch example */
#include <cstdio>      /* printf */
#include <cstdlib>     /* qsort, bsearch, NULL */

int compareints (const void * a, const void * b)
{
        return ( *(int*)a - *(int*)b );
}

int values[] = { 10,20,30,40,50,60 };
int main ()
{
        int * pItem;
        int key = 40;
        pItem = (int*) bsearch (&key, values, 6, sizeof(int), compareints);
        if (pItem!=NULL)
        printf ("%d is in the array.\n",*pItem);
        else
        printf ("%d is not in the array.\n",key);
        return 0;
}
```

Algoritmo de partición de Hoare: Es un algoritmo que re-ordena una colección de elementos de tal manera que todos los elementos con valor menor o igual a un **pivote** se sitúan antes del pivote, mientras que todos los valores mayores que el pivote se sitúan después del pivote. Luego de esta operación el pivote se encuentra en su posición definitiva.

Este algoritmo es parte fundamental del algoritmo de ordenamiento **Quicksort**.

Existe una versión alternativa de este algoritmo conocido como **partición de Lomuto.**

```
1 int partition(int arr[], int low, int high) {
2 int pivot = arr[low]; //Elección del pivote.
3 int i = low - 1, j = high + 1;
4
5 while (true)
6 {         // Find leftmost element greater than or equal to pivot
7         do
8         {
9                 i++;
10        } while (arr[i] < pivot);
11        // Find rightmost element smaller than or equal to pivot
12        do
13        {
14                j--;
15        } while (arr[j] > pivot);
16        // If two pointers met.
17        if (i >= j)
18        return j;
19
20        swap(arr[i], arr[j]);
21 }
22 }
```

Pivote X = 5, pos = 3

4	2	6	**5**	3	9
l=0	1	2	3	4	r=5

4	2	**6**	**5**	**3**	9
0	1	l=2	3	r=4	5

4	2	**3**	**5**	**6**	9
0	1	l=2	3	r=4	5

4	2	3	**5**	6	9
0	1	2	l=r=3	4	5

- El algoritmo de partición es ampliamente utilizado en los problemas de arreglos.
- De acuerdo al problema, se requieren versiones del algoritmo modificadas.
- Es de gran utilidad cuando se necesita partir un arreglo bajo cierto criterio.(No necesariamente mayor, menor).
- La elección del pivote es lo mas importante de este algoritmo.

```
1 int partition(int arr[], int low, int high){
2 int pivot = arr[high];    // pivot
3 int i = (low - 1);  // Index of smaller element
4
5 for (int j = low; j <= high- 1; j++)
6 {
7         // If current element is smaller than or
8         // equal to pivot
9         if (arr[j] <= pivot)
10        {
11                i++;    // increment index of smaller element
12                swap(arr[i], arr[j]);
13        }
14 }
15        swap(arr[i + 1], arr[high]);
16        return (i + 1);
17
```

- El algoritmo de partición de Lomuto es menos complejo de entender.
- También es ampliamente utilizado en muchos problemas sobre arreglos.