

Module 5

Basic Algorithms

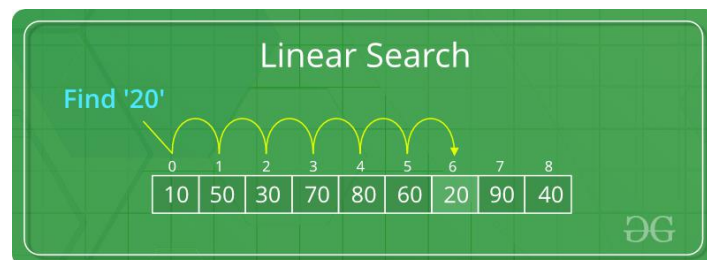
5.1 Searching

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. **Sequential Search:** In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.
2. **Interval Search:** These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

5.1.1 Linear search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Algorithm

Linear Search (Array A, Value x)

- Step 1: Set i to 1
- Step 2: if $i > n$ then go to step 7
- Step 3: if $A[i] = x$ then go to step 6
- Step 4: Set i to $i + 1$
- Step 5: Go to Step 2
- Step 6: Print Element x Found at index i and go to step 8
- Step 7: Print element not found
- Step 8: Exit

Pseudocode

```
procedure linear_search (list, value)
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for
end procedure
```

Program

```
#include <stdio.h>
int main()
{
  int array[100], search, c, n;

  printf("Enter number of elements in array\n");
  scanf("%d", &n);
```

```

printf("Enter %d integer(s)\n", n);

for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

printf("Enter a number to search\n");
scanf("%d", &search);

for (c = 0; c < n; c++)
{
    if (array[c] == search)    /* If required element is found */
    {
        printf("%d is present at location %d.\n", search, c+1);
        break;
    }
}
if (c == n)
    printf("%d isn't present in the array.\n", search);

return 0;
}

```

5.1.2 Binary search

Binary search is a fast search algorithm with run-time complexity of $O(\log n)$. This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.



Pseudocode

The pseudocode of binary search algorithms should look like this –

Procedure binary_search

A ← sorted array

n ← size of array

x ← value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound < lowerBound

EXIT: x does not exist.

```

    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

    if A[midPoint] < x
        set lowerBound = midPoint + 1

    if A[midPoint] > x
        set upperBound = midPoint - 1

    if A[midPoint] = x
        EXIT: x found at location midPoint
    end while

end procedure

```

Program

```

#include <stdio.h>

// Binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present"
                          " in array")
                  : printf("Element is present at "
                          "index %d",
                          result);

    return 0;
}

```

5.2 Basic Sorting Algorithm

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

5.2.1 Bubble sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
	1	1	2						

Algorithm

We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
begin BubbleSort(list)
```

```
    for all elements of list
```

```
        if list[i] > list[i+1]
```

```
            swap(list[i], list[i+1])
```

```
        end if
```

```
    end for
```

```
    return list
```

```
end BubbleSort
```

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable swapped which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows –
procedure bubbleSort(list : array of items)

```
    loop = list.count;

    for i = 0 to loop-1 do:
        swapped = false

        for j = 0 to loop-1 do:

            /* compare the adjacent elements */
            if list[j] > list[j+1] then
                /* swap them */
                swap( list[j], list[j+1] )
                swapped = true
            end if

        end for

        /*if no number was swapped that means
        array is sorted now, break the loop.*/

        if(not swapped) then
            break
        end if

    end for

end procedure return list
```

Program

```
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)

        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
```

```

    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}

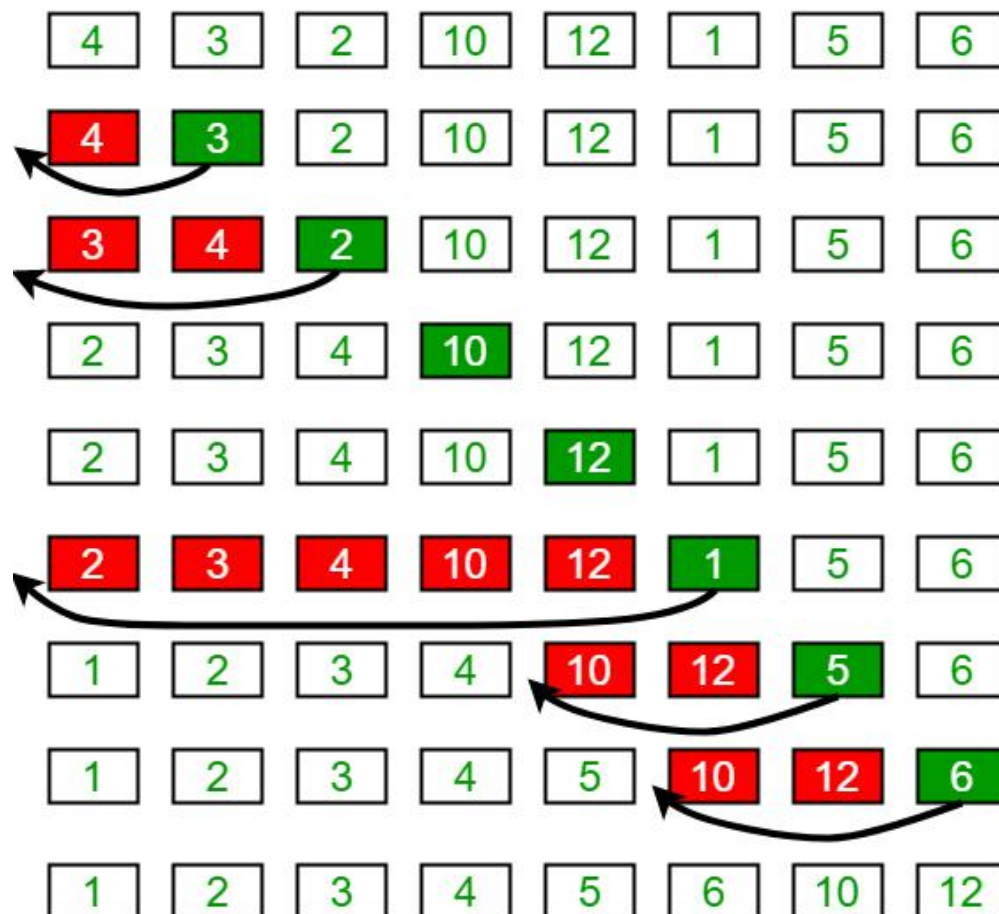
```

5.2.2 Insertion sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

Insertion Sort Execution Example



Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

- Step 1 – If it is the first element, it is already sorted. return 1;
- Step 2 – Pick next element
- Step 3 – Compare with all elements in the sorted sub-list
- Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted
- Step 5 – Insert the value
- Step 6 – Repeat until list is sorted

Pseudocode

```
procedure insertionSort( A : array of items )
    int holePosition
    int valueToInsert

    for i = 1 to length(A) inclusive do:

        /* select value to be inserted */
        valueToInsert = A[i]
        holePosition = i

        /*locate hole position for the element to be inserted */

        while holePosition > 0 and A[holePosition-1] > valueToInsert do:
            A[holePosition] = A[holePosition-1]
            holePosition = holePosition - 1
        end while

        /* insert the number at hole position */
        A[holePosition] = valueToInsert

    end for

end procedure
```

Program

```
#include <math.h>
#include <stdio.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
    }
}
```

```

    }
    arr[j + 1] = key;
}
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

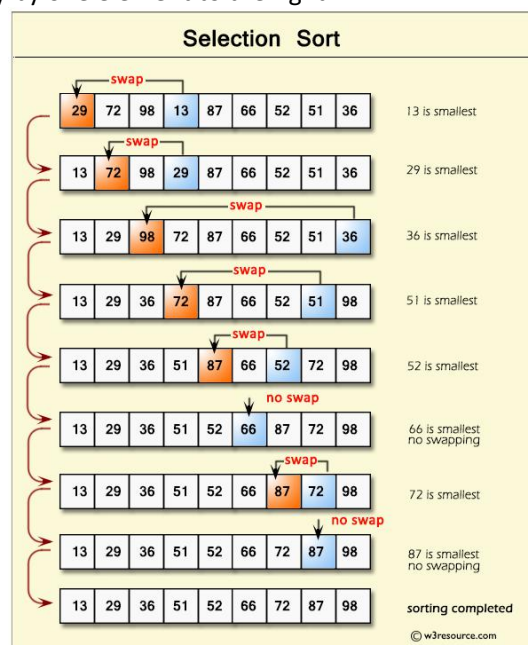
    return 0;
}

```

5.2.3 Selection sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.



Algorithm

Step 1 – Set MIN to location 0
Step 2 – Search the minimum element in the list
Step 3 – Swap with value at location MIN
Step 4 – Increment MIN to point to next element
Step 5 – Repeat until list is sorted

Pseudocode

```
procedure selection sort
    list : array of items
    n     : size of list

    for i = 1 to n - 1
        /* set current element as minimum */
        min = i

        /* check the element to be minimum */

        for j = i+1 to n
            if list[j] < list[min] then
                min = j;
            end if
        end for

        /* swap the minimum element with the current element */
        if indexMin != i then
            swap list[min] and list[i]
        end if
    end for

end procedure
```

Program

```
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
    }
}
```

```

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}

```

5.3 Finding Roots of a Equation

Bisection Method

Bisection method is an iterative implementation of the ‘Intermediate Value Theorem’ to find the real roots of a nonlinear function. According to the theorem “If a function $f(x)=0$ is continuous in an interval (a,b) , such that $f(a)$ and $f(b)$ are of opposite nature or opposite signs, then there exists at least one or an odd number of roots between a and b .”

Program

```

#include<stdio.h>
#include<conio.h>
#include<math.h>
// #define f(x) x*x-10

float f(float x)
{
    float z;
    z=x*x-10;
    return z;
}

void main()
{
    int count=1,n;
    float a,b,m;
    clrscr();
    printf("ENTER THE INTERVALS A and B :- ");
    xyz:
    scanf("%f%f",&a,&b);
    printf("No.of Iteration.");
    scanf("%d",&n);
    if(f(a)*f(b)<0);
    {

```

```

    m=(a+b)/2;
    while(count<=n)
    {
        if(f(a)*f(m)<0)
        {
            b=m;
        }
        else
        {
            a=m;
        }
        count++;
        m=(a+b)/2;
    }
    printf("Answer is : %f",m);
}

getch();
}

```

5.4 Notion of Order Complexity through example Programs

Imagine a classroom of 100 students in which you gave your pen to one person. Now, you want that pen. Here are some ways to find the pen and what the O order is.

$O(n^2)$: You go and ask the first person of the class, if he has the pen. Also, you ask this person about other 99 people in the classroom if they have that pen and so on, This is what we call $O(n^2)$.

$O(n)$: Going and asking each student individually is $O(N)$.

$O(\log n)$: Now I divide the class into two groups, then ask: "Is it on the left side, or the right side of the classroom?" Then I take that group and divide it into two and ask again, and so on. Repeat the process till you are left with one student who has your pen. This is what you mean by $O(\log n)$.

I might need to do the $O(n^2)$ search if only one student knows on which student the pen is hidden. I'd use the $O(n)$ if one student had the pen and only they knew it. I'd use the $O(\log n)$ search if all the students knew, but would only tell me if I guessed the right side.

Time Complexity of algorithm/code is not equal to the actual time required to execute a particular code but the number of times a statement executes.

Example:

```

#include <stdio.h>
int main()
{
    printf("Hello World");
}

```

In above code "Hello World!!!" print only once on a screen. So, time complexity is constant: $O(1)$ i.e. every time constant amount of time require to execute code, no matter which operating system or which machine configurations you are using.

```

#include <stdio.h>
void main()
{
    int i, n = 8;
    for (i = 1; i <= n; i++) {
        printf("Hello Word !!!");
    }
}

```

```
}
```

In above code "Hello World!!!" will print N times. So, time complexity of above code is $O(N)$.

```
#include <stdio.h>
```

```
void main()
```

```
{
```

```
    int i, n = 8;
```

```
    for (i = 1; i <= n; i++) {
```

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

```
            printf("Hello Word !!!");
```

```
    }
```

```
}
```

In above code "Hello World!!!" will print N^2 times. So, time complexity of above code is $O(N^2)$.

```
#include <stdio.h>
```

```
void main()
```

```
{
```

```
    int i, n = 8;
```

```
    for (i = 1; i <= n; i++) {
```

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

```
            printf("Hello Word !!!");
```

```
    }
```

```
}
```

In above code "Hello World!!!" will print $(N^2 + N)/2$ times. So, time complexity of above code is $O(N^2)$.