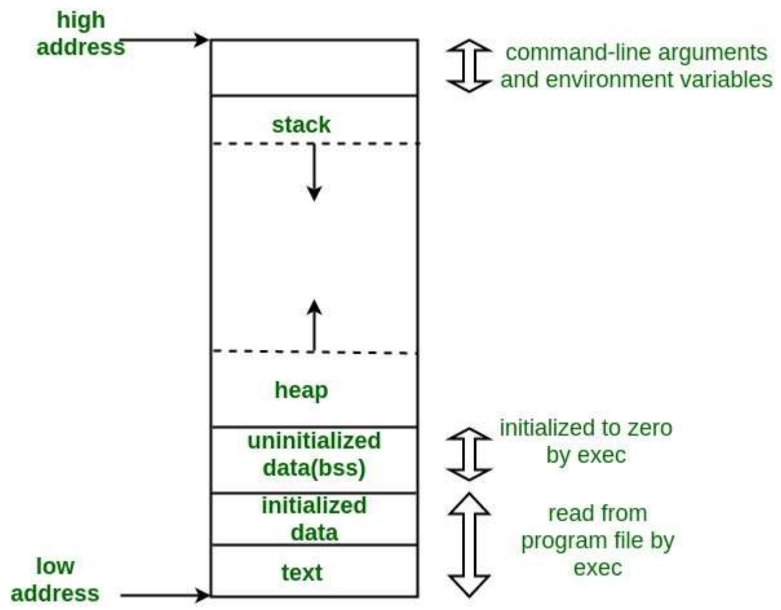
Name : Ahmed Mustafa abdellnaby NTI 4 Month → Group B

Memory Section of C Program

A typical memory representation of a C program consists of the following sections.

1. Text segment (i.e., instructions)
2. Initialized data segment
3. Uninitialized data segment (bss) {block started by symbol}
4. Heap
5. Stack

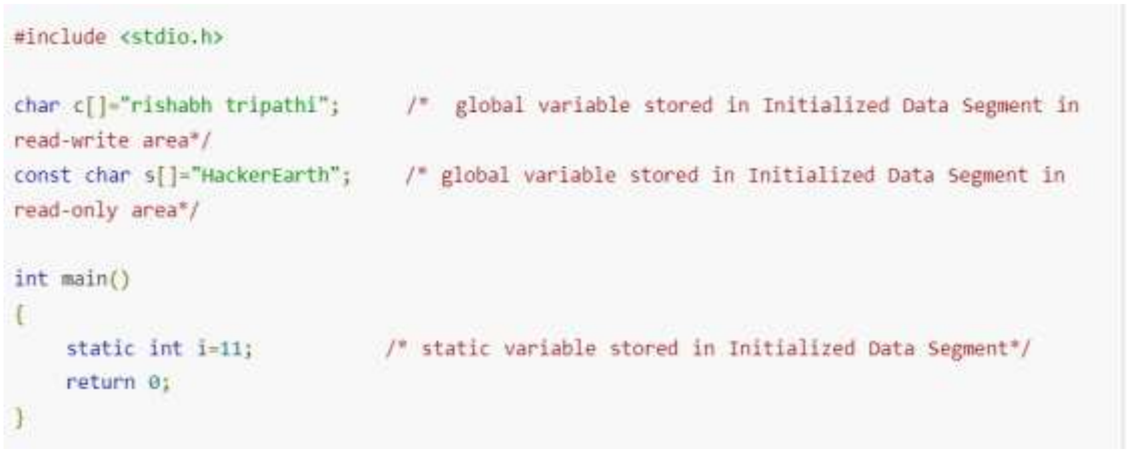

high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
low address
text

1. **Text Segment:**

is one of the sections of a program in an object file or in memory, which contains executable instructions.
As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.
Usually, the text segment is **sharable** so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, and so on. Also, the text segment is often **read-only**, to prevent a program from accidentally modifying its instructions.

2. **Initialized Data Segment:**

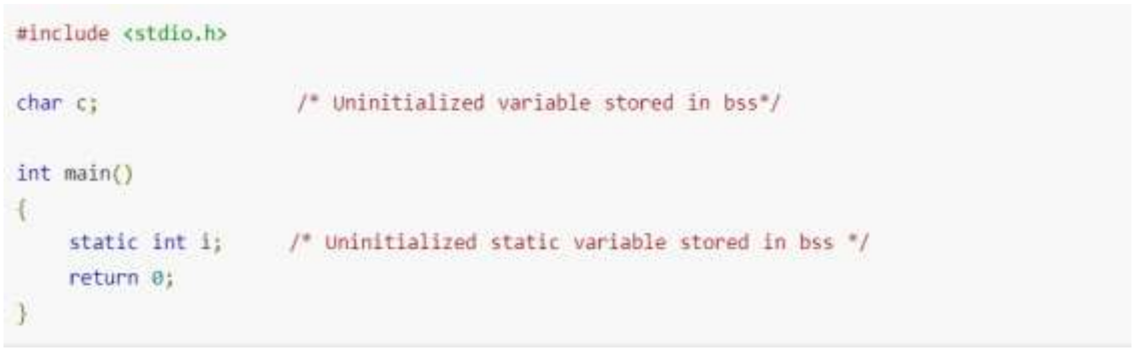
   Initialized data stores all **global, static, constant**, and external variables **(**declared with **extern** keyword) that are initialized beforehand. Data segment is **not read-only**, since the values of the variables can be altered at run time. This segment can be further classified into initialized **read-only area** and initialized **read-write area**.



```
#include <stdio.h>

char c[]="rishabh tripathi";     /*  global variable stored in Initialized Data Segment in
read-write area*/
const char s[]="HackerEarth";    /* global variable stored in Initialized Data Segment in
read-only area*/

int main()
{
    static int i=11;          /* static variable stored in Initialized Data Segment*/
    return 0;
}
```

3- Uninitialized Data Segment (bss): -

Data in this segment is initialized to arithmetic **0** before the program starts executing. Uninitialized data starts at the end of the data segment and contains all **global** variables and **static** variables that are initialized to **0** or do not have explicit initialization in source code.



```
#include <stdio.h>

char c;                 /* Uninitialized variable stored in bss*/

int main()
{
    static int i;     /* Uninitialized static variable stored in bss */
    return 0;
}
```

4- **Heap: -**

Heap is the segment where ***dynamic memory allocation*** usually takes place. When some more memory needs to be allocated using **malloc** and **calloc** function, heap grows upward. The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

```
#include <stdio.h>
int main()
{
    char *p=(char*)malloc(sizeof(char));    /* memory allocating in heap segment */
    return 0;
}
```

5- Stack: -

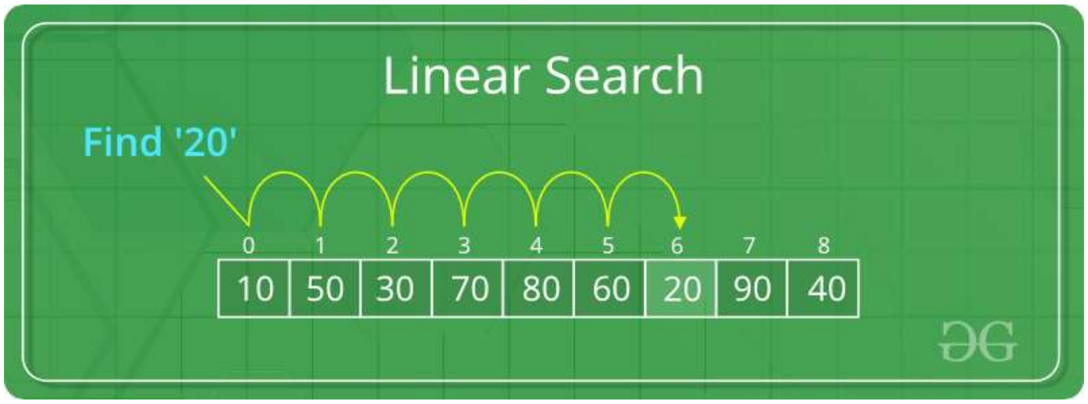
Stack segment is used to store all **local variables** and is used for **passing arguments** to the functions along with the **return** address of the instruction which is to be executed after the function call is over. Local variables have a scope to the block which they are defined in, they are created when control enters into the block.
All recursive function calls are added to stack. so be careful of stack overflow errors.

Name : Ahmed Mustafa abdellnaby   NTI 4 Month → Group B

searching algorithms

1- Linear Search Algorithm

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.



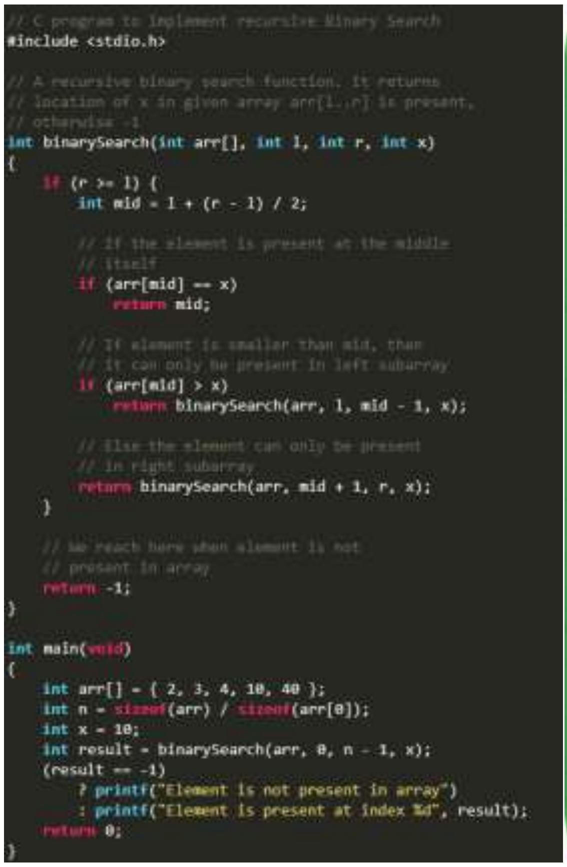
Name : Ahmed Mustafa abdellnaby NTI 4 Month → Group B

2- Binary Search Approach:

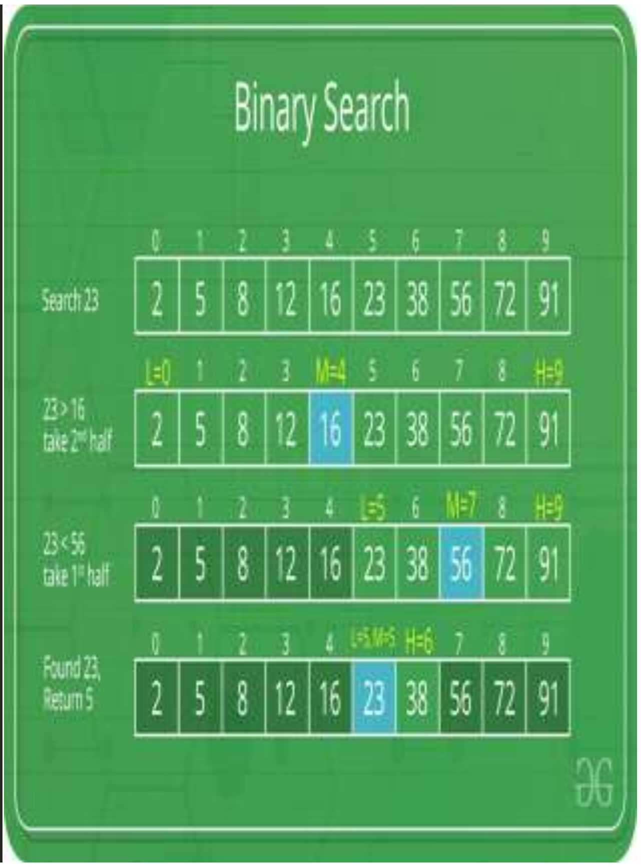
Binary Search is a searching algorithm used in a **sorted** array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

The basic steps to perform Binary Search are:

- Begin with the mid element of the whole array as a search key.
- If the value of the search key is equal to the item, then return an index of the search key.
- Or if the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
- Otherwise, narrow it to the upper half.
- Repeatedly check from the second point until the value is found or the interval is empty.



```c
// C program to implement recursive Binary Search
#include <stdio.h>

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    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;
}
```



Name : Ahmed Mustafa abdellnaby   NTI 4 Month → Group B

Sorting

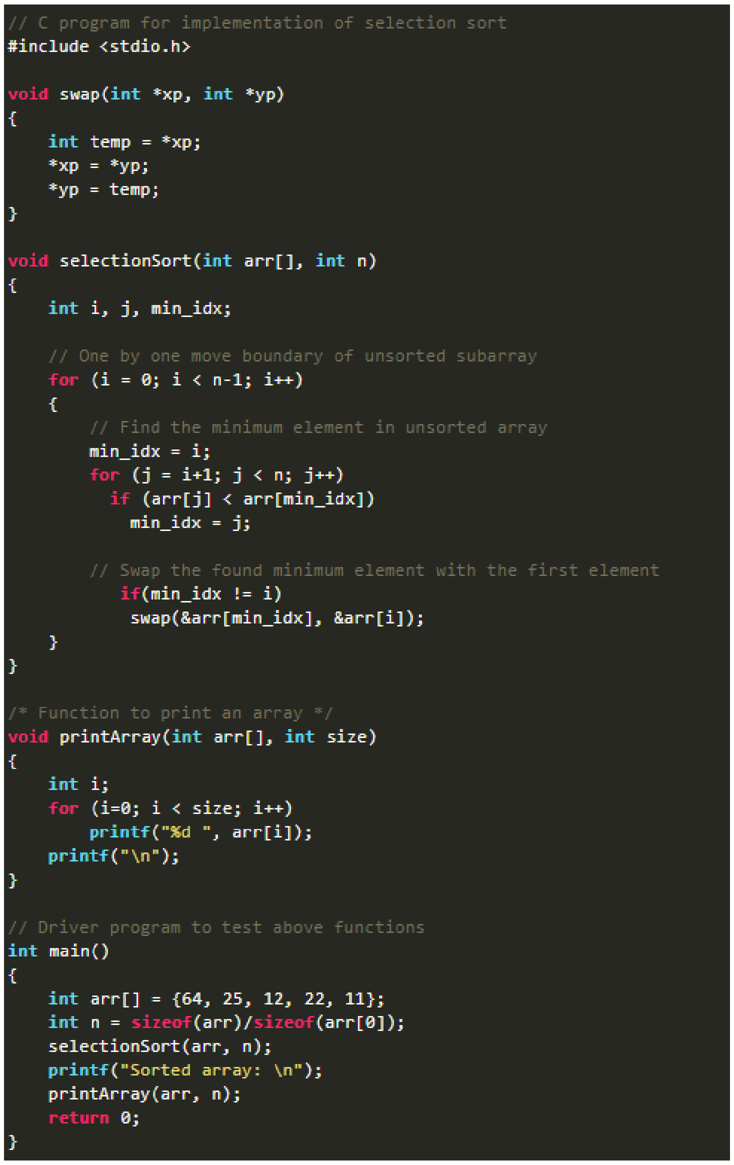
1- selection sort algorithm

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

- The subarray which already sorted.
- The remaining subarray was unsorted.

In every iteration of the selection sort, the minimum element (Considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

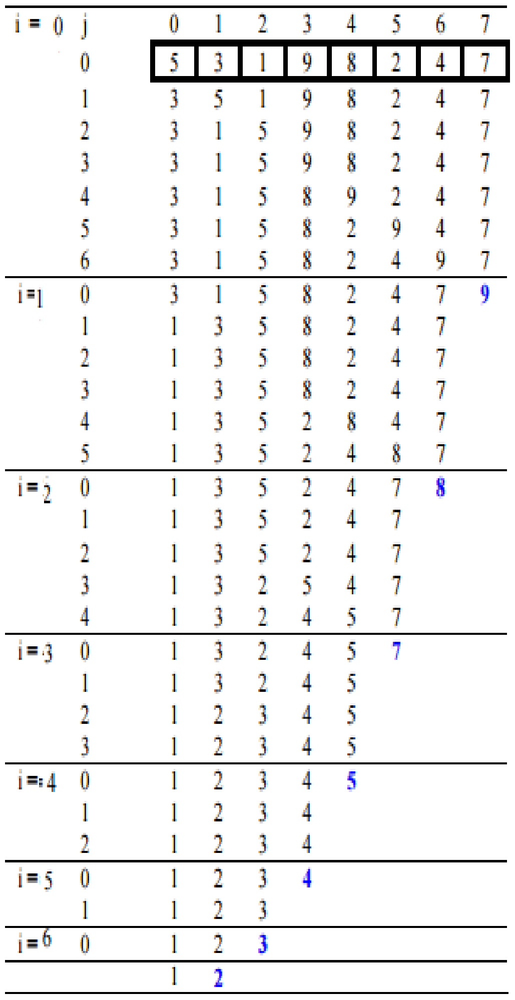


```c
// C program for implementation of selection sort
#include <stdio.h>

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
           if(min_idx != i)
            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;
}
```

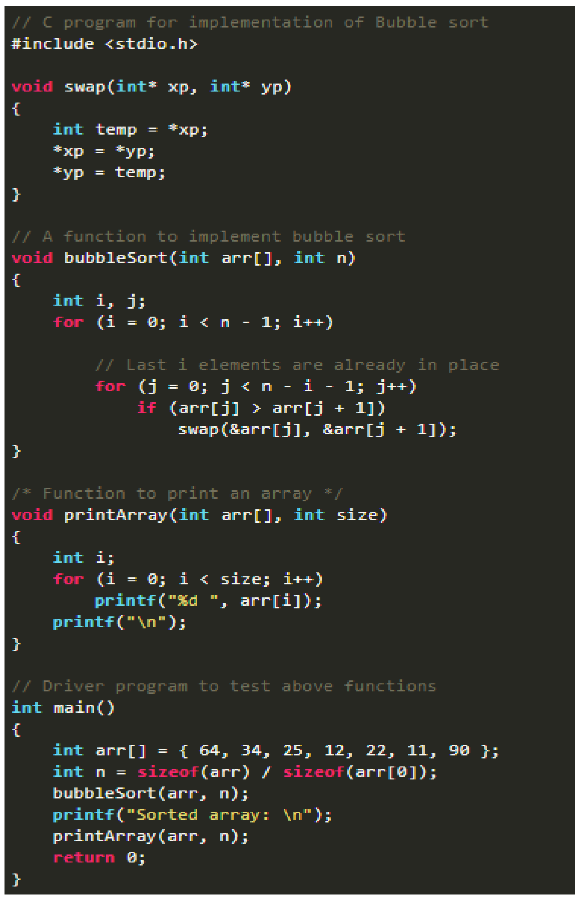
Name : Ahmed Mustafa abdellnaby NTI 4 Month → Group B

2- Bubble Sort

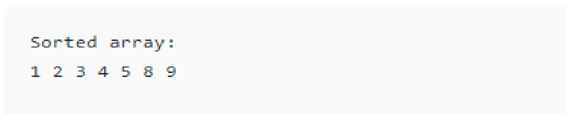
Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.



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	2						



```c
// C program for implementation of Bubble sort
#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");
}

// Driver program to test above functions
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;
}
```



```
Sorted array:
1 2 3 4 5 8 9
```