Name: Riya Vaid

Reg no: 21BCI0014

Subject: BCSE202P

Data Structures and Algorithms Lab

```c
/*
Algorithm linear search(data[],length,value)
{
	int i = 0; while(i<length)
	{
		if(value==data[i])
		return i;
		i++;
	}
}

*/


#include<stdio.h>
int search(int arr[], int N, int x)
{
	int i;
	for (i= 0; i<N;i++)
		if(arr[i]==x)
			return i;
	return -1;

}
int main(void)
{
	int arr[]={1,2,3,4,5};
	int x = 4;
	int N = sizeof(arr)/sizeof(arr[0]); int result
	= search(arr,N,x); (result == -1)
		? printf("Element is not present in array")
		: printf("Element is present at index %d",result); return 0;
}
```

1.) Linear Search

Algorithm and Source Code

Input/Output

```
Element is present at index 3
PS C:\Users\gauta_\OneDrive\Desktop\C>
```

Time Complexity

linear search is O(n).

2.) Binary Search

```
/*
Algorithm Binary_Search(A,low,high)
{
    if(low>high)
        return  False;
    mid = (low+high)/2
    if x+A[mid]
        return  True;
    if(x<A[mid])
        Binary_Search  (A,low,mid-1,x)
    if(x>A[mid])
        Binary_Search(A,mid+1,high,x)
}
*/

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[]  =  {  1,2,3,4,5};
        int  n  =  sizeof(arr)  /  sizeof(arr[0]);
        int  x  =  3;
        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;

    }
```

Output

```
Element is present at index 2
PS C:\Users\gauta_\OneDrive\Desktop\C>
```

Time Complexity

O(log n)

3.) Bubble Sort

```
/*
Algorithm Bubble_Sort
// A is Array of integers
// N is Number of Elements
{
      For  i=1  to  n  do
      {
            For  j=  1  to  n-i  do
            {
                  if(a[j]>a[j+1])//  Checking  Adjacent  Elements
                  {
                        temp=a[j];
                        a[j]=a[j+1];
                        a[j+1]=temp;
                  }
            }  //  End  of  j  loop
      } // End of i loop
}  //  End  of  the  Program

*/
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;
}
```

Output

```
Sorted array:
11 12 22 25 34 64 90
PS C:\Users\gauta_\OneDrive\Desktop\C>
```

Time Complexity

Best Case : O(n)

Average Case : O(n^2)

Worst Case : O(n^2)

4.) Insertion Sort

```
/*
Algorithm Insertion_Sort A[0].key
:= -∞;
for i := 2 to n do begin j := i;
    while A[j] < A[j-1] do begin
            swap(A[j], A[j-1]);
                j := j-1 end
      end
end
*/
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");
}

/* Driver program to test insertion sort */ int main()
{
      int arr[] = { 12, 11, 13, 5, 6 };
      int n = sizeof(arr) / sizeof(arr[0]);

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

```
    return 0;
}
```

Output

```
5 6 11 12 13
PS C:\Users\gauta_\OneDrive\Desktop\C>
```

Time Complexity

Best Case : O(n)

Average Case : O(n^2)

Worst Case : O(n^2)

5.) Quick Sort

```
/*
begin
        l := i;
        r := j; repeat
                swap(A[l],  A[r]);
        { now the scan phase begins } while
        A[l].key < pivot do
                l  :=  l  +  1;
        while A[r].key > = pivot do r  :=  r
                -  1
        until
                l > r;
        return (l)
end; { partition }
*/

#include<stdio.h>

void  quicksort(int  number[25],int  first,int  last){

int  i,  j,  pivot,  temp;

if(first<last){

pivot=first;

i=first;

j=last;

while(i<j){

while(number[i]<=number[pivot]&&i<last)

i++;

while(number[j]>number[pivot])

j--;

if(i<j){

temp=number[i];

number[i]=number[j];
```

```
number[j]=temp;
}
}
temp=number[pivot];
number[pivot]=number[j];
number[j]=temp;
quicksort(number,first,j-1);
quicksort(number,j+1,last);
}
}
int main(){
int i, count, number[25];
printf("Enter some elements (Max. - 25): ");
scanf("%d",&count);
printf("Enter %d elements: ", count);
for(i=0;i<count;i++)
scanf("%d",&number[i]);
quicksort(number,0,count-1);
printf("The Sorted Order is: ");
for(i=0;i<count;i++)
printf(" %d",number[i]);
return 0;
}
```

Output

```
64\mingw64\bin\gdb.exe' '--interpreter=mi'
Enter some elements (Max. - 25): 5
Enter 5 elements: 12
3
45
6
2
The Sorted Order is:  2 3 6 12 45
PS C:\Users\gauta_\OneDrive\Desktop\C>
```

Time Complexity

Best Case : O(n log n)

Average Case : O(n log n)

Worst Case : O(n^2)

6.) Selection Sort

```
/*
begin
    for i := 1 to n-1 do begin
        { select the lowest among A[i], . . . , A[n] and swap I t with A[i] } lowindex := i;
        lowkey := A[i].key; for j :=
        i + 1 to n do
            { compare each key with current lowkey } if
            A[j].key < lowkey then begin
                lowkey := A[j].key;
                lowindex := j
            end;
        swap(A[i], A[lowindex])
    end
end;
*/

#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;

    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx]) min_idx
            = j;

           if(min_idx != i) swap(&arr[min_idx],
            &arr[i]);
    }
}

void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
```

```c
            printf("%d  ", arr[i]);
        printf("\n");
}

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;
}
```

Output:

```
Sorted array:
11 12 22 25 64
PS C:\Users\gauta_\OneDrive\Desktop\C>
```

Time Complexity

Best Case : O(n^2)

Average Case : O(n^2)

Worst Case : O(n^2)

7.) Merge Sort

```
/*
Algorithm MergeSort(low,high)
{
	If (Low<high) then
	{
		Mid:=[(low + high)/2]
		MergeSort(low,mid);
		MergeSort(mid +1,high);
		Merge(low,mid,high);
	}
}
Algorithm Merge(low,mid,high)
{
	H:= low, i:=low; j:=mid+1;
	While ((h<=mid) and (j<=high)) do
	{
		If(a[h]<=a[j]) then
		{
			B[i] := a[h]; h:= h+1;
		}
		Else
		{
			B[i]:=a[j] ; j:= j+1;
		}
	}
	If (h>mid) then
		For k:=j to high do
		{
			B[i] := a[j]; j:=j+1;
		}
	Else
		For k:=h to mid do
		{
			B[i] := a[k] ; I := I +1;
		}
	For k:=low to high do a[k]:=b[k]
}
*/

#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays ofarr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
```

```
{
      int i, j, k;
      int n1 = m - l + 1; int
      n2 = r - m;

      /* create temp arrays */ int
      L[n1], R[n2];

      /* Copy data to temp arrays L[] and R[] */ for (i
      = 0; i < n1; i++)
            L[i] = arr[l + i]; for (j =
      0; j < n2; j++)
            R[j] = arr[m + 1 + j];

      /* Merge the temp arrays back into arr[l..r]*/ i = 0; //
      Initial index of first subarray
      j = 0; // Initial index of second subarray k =
      l; // Initial index of merged subarray while (i < n1
      && j < n2) {
            if (L[i] <= R[j]) {
                  arr[k] = L[i]; i++;
            }
            else {
                  arr[k] = R[j]; j++;
            } k+
            +;
      }

      /* Copy the remaining elements of L[], if there are any */
      while (i < n1) { arr[k] =
            L[i]; i++;
            k++;
      }

      /* Copy the remaining elements of R[], if there are any */
      while (j < n2) { arr[k] =
            R[j]; j++;
            k++;
      }
}

/* l is for left index and r is right index of the
```

```
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
	if (l < r) {
		// Same as (l+r)/2, but avoids overflow for
		// large l and h
		int m = l + (r - l) / 2;

		// Sort first and second halves
		mergeSort(arr, l, m); mergeSort(arr, m +
		1, r);

		merge(arr, l, m, r);
	}
}

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

/* Driver code */ int
main()
{
	int arr[] = { 12, 11, 13, 5, 6, 7 };
	int arr_size = sizeof(arr) / sizeof(arr[0]);

	printf("Given array is \n");
	printArray(arr, arr_size);

	mergeSort(arr, 0, arr_size - 1);

	printf("\nSorted array is \n");
	printArray(arr, arr_size); return 0;
}
```

Output

```
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13
PS C:\Users\gauta_\OneDrive\Desktop\C>
```

Time Complexity

Best Case : O(n log n)

Average Case : O(n log n)

Worst Case : O(n log n)

8.) Count Sort

```
/*
Algorithm Count_Sort
Let C[0..k] be a new array For I =
        0 to k
                C[i]=0
        For j = 1 to A.length C[A[j]]=C[A[j]]+1

        For I = t to k C[i]
                +C[i-1]

        For j =A.length downto 1 B[C[A[j]]]  =
                A[j]
                C[A[j]]  =  C[A[j]]  -1
        }
}
*/
#include  <stdio.h> #include
<string.h> #define RANGE 255

void  countSort(char  arr[])
{
        char  output[strlen(arr)];

        int  count[RANGE  +  1],  i; memset(count,
        0,  sizeof(count));

              for  (i  =  0;  arr[i];  ++i)
                    ++count[arr[i]];

        for  (i  =  1;  i  <=  RANGE;  ++i)
                count[i]  +=  count[i  -  1];

        for  (i  =  0;  arr[i];  ++i)  { output[count[arr[i]]  -  1]
                =  arr[i];
                --count[arr[i]];
        }
        for  (i  =  0;  arr[i];  ++i) arr[i]  =
                output[i];
}

int main()
{
        char  arr[]  =  "qwerty";
```

```
    countSort(arr);

    printf("Sorted character array is %sn", arr); return 0;
}
```

Output

```
Sorted character array is eqrtwyn
PS C:\Users\gauta_\OneDrive\Desktop\C>
```

Time Complexity

Best Case : O(n+k)

Average Case : O(n+k)

Worst Case : O(n+k)

9.) Queues

```
/* Algorithm
Peek()
Begin procedure peek Return
        queue [front]
End  procedure

Isfull()
Begin procedure isfull
          If  rear  equals  to  Max  size Return
                true
          Else
                Return  false
          End if
End  procedure

Isempty()
Begin procedure isempty
        If front is less MIN or front is greater than rear Return  true
        Else
              Return  false End
          if
End  procedure

Enqueue Operation Procedure
enqueue(data)
          If  queue  is  full Return
                overflow
          Endif
          Rear ⇓ rear +1 Queue
          [rear]  ⇓  data Return
          true
End  procedure

Dequeue  Opreation procedure
dequeue

     if  queue  is  empty
          return  underflow
     end if

     data  =  queue[front]
     front ← front + 1 return
     true
```

```
end procedure
*/
#include <limits.h> #include
<stdio.h> #include <stdlib.h>
struct Queue {
	int front, rear, size;
	unsigned capacity; int*
	array;
};

struct Queue* createQueue(unsigned capacity)
{
	struct Queue* queue = (struct Queue*)malloc( sizeof(struct
		Queue));
	queue->capacity = capacity; queue-
	>front = queue->size = 0;

	queue->rear = capacity - 1;
	queue->array = (int*)malloc(
		queue->capacity * sizeof(int)); return
	queue;
}


int isFull(struct Queue* queue)
{
	return (queue->size == queue->capacity);
}

int isEmpty(struct Queue* queue)
{
	return (queue->size == 0);
}

void enqueue(struct Queue* queue, int item)
{
	if (isFull(queue))
		return;
	queue->rear = (queue->rear + 1)
				% queue->capacity;
	queue->array[queue->rear] = item;
	queue->size = queue->size + 1;
	printf("%d enqueued to queue\n", item);
}


int dequeue(struct Queue* queue)
```

```
{
	if (isEmpty(queue))
		return INT_MIN;
	int item = queue->array[queue->front]; queue-
>front = (queue->front + 1)
				% queue->capacity;
	queue->size = queue->size - 1; return
	item;
}


int front(struct Queue* queue)
{
	if (isEmpty(queue))
		return INT_MIN;
	return queue->array[queue->front];
}


int rear(struct Queue* queue)
{
	if (isEmpty(queue))
		return INT_MIN;
	return queue->array[queue->rear];
}


int main()
{
	struct Queue* queue = createQueue(1000);

	enqueue(queue, 10);
	enqueue(queue, 20);
	enqueue(queue, 30);
	enqueue(queue, 40);

	printf("%d dequeued from queue\n\n", dequeue(queue));

	printf("Front item is %d\n", front(queue)); printf("Rear item is
	%d\n", rear(queue));

	return 0;
}
```

Output

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
40 enqueued to queue
10 dequeued from queue

Front item is 20
Rear item is 40
PS C:\Users\gauta_\OneDrive\Desktop\C>
```

10.) Stacks

```
/*
Peek()
Begin procedure peek Return
        stack[top]
End  procedure

Isfull()
Begin procedure isfull
        If top equal to Maxsize
                Return  true
        Else
                Return  false
          Endif End
procedure

Isempty()
Begin procedure isempty If
        top  less  than  1
              Return  true Else
                Return  false Endif
End  procedure

Push()
Begin procedure push: stack,data If
stack is full
      Return  null
Endif
Top ⇓ top +1
Stack  [top]  ⇓  data
end procedure

Pop()
Begin  procedure  pop:stack If
      stack is empty
            Return  null Endif
        Data  ⇓  stack[top]
        Top ⇓ top -1 Return
        data
End  procedure

*/

#include  <limits.h>
```

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

// A structure to represent a stack struct
Stack {
	int top;
	unsigned capacity; int*
	array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
	struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack)); stack-
	>capacity = capacity;
	stack->top = -1;
	stack->array = (int*)malloc(stack->capacity * sizeof(int)); return stack;
}

// Stack is full when top is equal to the last index int
isFull(struct Stack* stack)
{
	return stack->top == stack->capacity - 1;
}

// Stack is empty when top is equal to -1 int
isEmpty(struct Stack* stack)
{
	return stack->top == -1;
}

// Function to add an item to stack.	It increases top by 1
void push(struct Stack* stack, int item)
{
	if (isFull(stack))
		return;
	stack->array[++stack->top] = item; printf("%d
	pushed to stack\n", item);
}

// Function to remove an item from stack.	It decreases top by 1 int
pop(struct Stack* stack)
{
	if (isEmpty(stack))
		return INT_MIN;
	return stack->array[stack->top--];
```

```
}

// Function to return the top from stack without removing it int
peek(struct Stack* stack)
{
	if (isEmpty(stack))
		return INT_MIN;
	return stack->array[stack->top];
}

// Driver program to test above functions int main()
{
	struct Stack* stack = createStack(100);

	push(stack, 10);
	push(stack, 20);
	push(stack, 30);

	printf("%d popped from stack\n", pop(stack)); return 0;
}
```

Output

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popped from stack
PS C:\Users\gauta_\OneDrive\Desktop\C>
```

```
/*
Algorithem Tower_Of_Hanoi
START
Procedure Hanoi(disk, source, dest, aux)

    IF disk == 1, THEN
        move disk from source to dest ELSE
        Hanoi(disk - 1, source, aux, dest) move
        disk from source to dest Hanoi(disk - 1,      // Step 1
        aux, dest, source)                            // Step 2
    END IF                                            // Step 3


END Procedure
STOP
*/

#include <stdio.h>
void towers(int,char,char,char); int main()
{
      int num;
      printf("Enter the number of disks: ");
      scanf("%d",&num);
      printf("The sequence of moves involved in the tower of hannoi are: \n");
      towers(num,'A','C','B'); //A=source C= Destination B= temp
      return 0;
}
void towers(int num,char frompeg,char topeg,char auxpeg)
{
      if (num==1)
      {
      printf("\n Move disk 1 from peg %c to peg %c",frompeg,topeg);

      return;
      }

      towers(num-1,frompeg,auxpeg,topeg);
      printf("\n Move disk %d from peg %c to peg %c", num,frompeg,topeg);
      towers(num-1,auxpeg,topeg,frompeg);
}
```

11.) Tower Of Hanoi

Output

```
Enter the number of disks: 3
The sequence of moves involved in the tower of hannoi are:

 Move disk 1 from peg A to peg C
 Move disk 2 from peg A to peg B
 Move disk 1 from peg C to peg B
 Move disk 3 from peg A to peg C
 Move disk 1 from peg B to peg A
 Move disk 2 from peg B to peg C
 Move disk 1 from peg A to peg C
PS C:\Users\gauta_\OneDrive\Desktop\C>
```