**CCS 3102 DESIGN AND ANALYSIS OF ALGORITHMS**

**EXPERIMENT 1: INSERTION SORT**

**Pseudo-code**

InsertionSort (A, n)

for i = 2 to n   
do key = A[i]  
 //Insert A[ j ] into the sorted sequence A[1 . . j -1]

j = i - 1;  
 while (j > 0) and (A[j] > key)   
 do A[j+1] = A[j]  
 j = j -1   
 A[j+1] = key

**Insertion sort algorithm implementation in C**

Insertion sort in C: C program for insertion sort to sort numbers. This code implements insertion sort algorithm to arrange numbers of an array in ascending order. With a little modification, it will arrange numbers in descending order. Best case complexity of insertion sort is O(n), average and the worst case complexity is O(n2).

*/\* Insertion sort ascending order \*/*

#include <stdio.h>

int main()

{

  int n, array[1000], c, d, t;

  printf("Enter number of elements**\n**");

  scanf("%d", &n);

  printf("Enter %d integers**\n**", n);

  for (c = 0; c < n; c++)

    scanf("%d", &array[c]);

  for (c = 1 ; c <= n - 1; c++) {

    d = c;

    while ( d > 0 && array[d-1] > array[d]) {

      t          = array[d];

      array[d]   = array[d-1];

      array[d-1] = t;

      d--;

    }

  }

  printf("Sorted list in ascending order:**\n**");

  for (c = 0; c <= n - 1; c++) {

    printf("%d**\n**", array[c]);

  }

  return 0;

}

**// Java program for implementation of Insertion Sort**

class InsertionSort {

    /\*Function to sort array using insertion sort\*/

    void sort(int arr[])

    {

        int n = arr.length;

        for (int i = 1; i < n; ++i) {

            int key = arr[i];

            int 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 array of size n\*/

    static void printArray(int arr[])

    {

        int n = arr.length;

        for (int i = 0; i < n; ++i)

            System.out.print(arr[i] + " ");

        System.out.println();

    }

    // Driver method

    public static void main(String args[])

    {

        int arr[] = { 12, 11, 13, 5, 6 };

        InsertionSort ob = new InsertionSort();

        ob.sort(arr);

        printArray(arr);

    }

**Time Complexity**: O(n2)

**Auxiliary Space**: O(1)

Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

**EXPERIMENT 3: QUICK SORT**

**3.1 Objective:**

Sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of *n*, the number of elements in the list to be sorted and plot a graph of the time taken versus *n*. The elements can be read from a file or can be generated using a random number generator.

**3.2 Program Logic:**

QuickSort is a Divide and Conquer algorithm. Quick sort is an algorithm of choice in many situations because it is not difficult to implement, it is a good "general purpose" sort and it consumes relatively fewer resources during execution. It picks an element as pivot and partitions the given array around the picked pivot.

There are many different versions of QuickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

**Good points**

* It is in-place since it uses only a small auxiliary stack.
* It requires only *n*log*(n)* time to sort n items.
* It has an extremely short inner loop
* This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

**Bad Points**

* It is recursive. Especially if recursion is not available, the implementation is extremely complicated.
* It requires quadratic (*i.e., n*2) time in the worst-case.
* It is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

 Quick sort works by partitioning a given array *A*[*p* . . *r*] into two non-empty sub array *A*[*p* . . *q*] and *A*[*q*+1 . . *r*] such that every key in *A*[*p* . . *q*] is less than or equal to every key in *A*[*q*+1 . . *r*]. Then the two subarrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

**QuickSort**

1. If *p* < *r* then
2. *q* Partition (*A*, *p*,*r*)
3. Recursive call to Quick Sort (*A*, *p*, *q*)
4. Recursive call to Quick Sort (*A*, *q*+*r*, *r*)

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is then partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

**3.3 Source code:**

**// Implementation of QuickSort to sort elements in descending order**

#include <stdio.h>

#include <stdlib.h>

void swap(int Array[], int one, int two) {

int temp = Array[one];

Array[one] = Array[two];

Array[two] = temp;

}

int partition(int Array[], int left, int right) {

int pivot = Array[right];

int leftPointer = left - 1;

int rightPointer = right;

for (;;) {

while (Array[++leftPointer] > pivot) {

}

while (rightPointer > 0 && Array[--rightPointer] < pivot) {

}

if (leftPointer >= rightPointer) {

break;

} else {

swap(Array, leftPointer, rightPointer);

}

}

/\* move pivot to partition point \*/

swap(Array, leftPointer, right);

return leftPointer;

}

void Quicksort(int Array[], int left, int right) {

if (left < right) {

int PartitionPoint = partition(Array, left, right);

Quicksort(Array, left, PartitionPoint - 1);

Quicksort(Array, PartitionPoint + 1, right);

}

}

#define MAX\_SIZE 10

int main(int argc, char \*\*argv) {

int i, n;

int Array[MAX\_SIZE];

if (argc > 1) {

for (n = 0; n < MAX\_SIZE && n < argc - 1; n++) {

Array[n] = strtol(argv[n + 1], NULL, 0);

}

} else {

printf("Give %d values: ",MAX\_SIZE );

for (n = 0; n < MAX\_SIZE; n++) {

if (scanf("%d", &Array[n]) != 1)

break;

}

}

Quicksort(Array, 0, n - 1);

printf("\nOutput: ");

for (i = 0; i < n; i++) {

printf("%d ", Array[i]);

}

printf("\n");

return 0;

}

**EXPERIMENT 4: MERGE SORT**

**/\* C program for Merge Sort \*/**

#include <stdio.h>

#include <stdlib.h>

// Merges two subarrays of arr[].

// 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;

}

**Python program for implementation of MergeSort**

def mergeSort(arr):

if len(arr) > 1:

# Finding the mid of the array

mid = len(arr)//2

# Dividing the array elements

L = arr[:mid]

# into 2 halves

R = arr[mid:]

# Sorting the first half

mergeSort(L)

# Sorting the second half

mergeSort(R)

i = j = k = 0

# Copy data to temp arrays L[] and R[]

while i < len(L) and j < len(R):

if L[i] <= R[j]:

arr[k] = L[i]

i += 1

else:

arr[k] = R[j]

j += 1

k += 1

# Checking if any element was left

while i < len(L):

arr[k] = L[i]

i += 1

k += 1

while j < len(R):

arr[k] = R[j]

j += 1

k += 1

# Code to print the list

def printList(arr):

for i in range(len(arr)):

print(arr[i], end=" ")

print()

# Driver Code

if \_\_name\_\_ == '\_\_main\_\_':

arr = [12, 11, 13, 5, 6, 7]

print("Given array is", end="\n")

printList(arr)

mergeSort(arr)

print("Sorted array is: ", end="\n")

printList(arr)