**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 2: TOWER OF HANOI**

**Rules of Tower of Hanoi:**

[](https://www.codewithc.com/wp-content/uploads/2014/07/tower-of-hanoi3.png)

* Only a single disc is allowed to be transferred at a time.
* Each transfer or move should consists of taking the upper disk from one of the stack and then placing it on the top of another stack i.e. only a top most disk on the stack can be moved.
* Larger disk cannot be placed over smaller disk; placing of disks should be in increasing order.

With three disks, the puzzle can be solved in seven moves. The minimum number of moves required to solve a Tower of Hanoi puzzle is 2n-1, where n is the number of disks.

**Towers of Hanoi implementation in C using Recursion**

/\*

\* C program for Tower of Hanoi using Recursion

\*/

#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 Hanoi are :\n");

towers(num, 'A', 'C', 'B');

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);

}

**Assignment:**

1. Implement the towers of Hanoi problem in a non- recursive form (you may use iteration or any other technique).

**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)

**EXPERIMENT 5: TIME COMPLEXITY OF ALGORITHMS**

**Algorithms used:**

A comparison of the running times of the following algorithms is done:

**Insertion sort:** The traditional algorithm with no modifications/optimisation. It performs very well for smaller input sizes. And yes, it does beat merge sort

**Merge sort**: Follows the divide-and-conquer approach. For input sizes of the order 105 this algorithm is of the right choice. It renders insertion sort impractical for such large input sizes.

**Combined version of insertion sort and merge sort**: The logic of merge sort has been tweaked a little bit to achieve a considerably better running time for smaller input sizes. As we know, merge sort splits its input into two halves until it is trivial enough to sort the elements. But here, when the input size falls below a threshold such as ’n’ < 40 then this hybrid algorithm makes a call to traditional insertion sort procedure. From the fact that insertion sort runs faster on smaller inputs and merge sort runs faster on larger inputs, this algorithm makes best use both the worlds.

**Quick sort:** The library function qsort() which is available is implemented. This algorithm was considered in order to know the significance of implementation. It requires a great deal of programming expertise to minimize the number of steps and make at most use of the underlying language primitives to implement an algorithm in the best way possible. This is the main reason why it is recommended to use library functions.

//C++ code to compare performance of sorting algorithms

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <time.h>

#define MAX\_ELEMENT\_IN\_ARRAY 1000000001

int cmpfunc (const void \* a, const void \* b)

{

// Compare function used by qsort

return ( \*(int\*)a - \*(int\*)b );

}

int\* generate\_random\_array(int n)

{

srand(time(NULL));

int \*a = malloc(sizeof(int) \* n), i;

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

a[i] = rand() % MAX\_ELEMENT\_IN\_ARRAY;

return a;

}

int\* copy\_array(int a[], int n)

{

int \*arr = malloc(sizeof(int) \* n);

int i;

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

arr[i] = a[i];

return arr;

}

//Code for Insertion Sort

void insertion\_sort\_asc(int a[], int start, int end)

{

int i;

for(i = start + 1; i <= end ; ++i)

{

int key = a[i];

int j = i - 1;

while(j >= start && a[j] > key)

{

a[j + 1] = a[j];

--j;

}

a[j + 1] = key;

}

}

//Code for Merge Sort

void merge(int a[], int start, int end, int mid)

{

int i = start, j = mid + 1, k = 0;

int \*aux = malloc(sizeof(int) \* (end - start + 1));

while(i <= mid && j <= end)

{

if(a[i] <= a[j])

aux[k++] = a[i++];

else

aux[k++] = a[j++];

}

while(i <= mid)

aux[k++] = a[i++];

while(j <= end)

aux[k++] = a[j++];

j = 0;

for(i = start;i <= end;++i)

a[i] = aux[j++];

free(aux);

}

void \_merge\_sort(int a[],int start,int end)

{

if(start < end)

{

int mid = start + (end - start) / 2;

\_merge\_sort(a,start,mid);

\_merge\_sort(a,mid + 1,end);

merge(a,start,end,mid);

}

}

void merge\_sort(int a[],int n)

{

return \_merge\_sort(a,0,n - 1);

}

void insertion\_and\_merge\_sort\_combine(int a[], int start, int end, int k)

{

// Performs insertion sort if size of array is less than or equal to k

// Otherwise, uses mergesort

if(start < end)

{

int size = end - start + 1;

if(size <= k)

{

//printf("Performed insertion sort- start = %d and end = %d\n", start, end);

return insertion\_sort\_asc(a,start,end);

}

int mid = start + (end - start) / 2;

insertion\_and\_merge\_sort\_combine(a,start,mid,k);

insertion\_and\_merge\_sort\_combine(a,mid + 1,end,k);

merge(a,start,end,mid);

}

}

void test\_sorting\_runtimes(int size,int num\_of\_times)

{

// Measuring the runtime of the sorting algorithms

int number\_of\_times = num\_of\_times;

int t = number\_of\_times;

int n = size;

double insertion\_sort\_time = 0, merge\_sort\_time = 0;

double merge\_sort\_and\_insertion\_sort\_mix\_time = 0, qsort\_time = 0;

while(t--)

{

clock\_t start, end;

int \*a = generate\_random\_array(n);

int \*b = copy\_array(a,n);

start = clock();

insertion\_sort\_asc(b,0,n-1);

end = clock();

insertion\_sort\_time += ((double) (end - start)) / CLOCKS\_PER\_SEC;

free(b);

int \*c = copy\_array(a,n);

start = clock();

merge\_sort(c,n);

end = clock();

merge\_sort\_time += ((double) (end - start)) / CLOCKS\_PER\_SEC;

free(c);

int \*d = copy\_array(a,n);

start = clock();

insertion\_and\_merge\_sort\_combine(d,0,n-1,40);

end = clock();

merge\_sort\_and\_insertion\_sort\_mix\_time+=((double) (end - start))/CLOCKS\_PER\_SEC;

free(d);

start = clock();

qsort(a,n,sizeof(int),cmpfunc);

end = clock();

qsort\_time += ((double) (end - start)) / CLOCKS\_PER\_SEC;

free(a);

}

insertion\_sort\_time /= number\_of\_times;

merge\_sort\_time /= number\_of\_times;

merge\_sort\_and\_insertion\_sort\_mix\_time /= number\_of\_times;

qsort\_time /= number\_of\_times;

printf("\nTime taken to sort:\n"

"%-35s %f\n"

"%-35s %f\n"

"%-35s %f\n"

"%-35s %f\n\n",

"(i)Insertion sort: ",

insertion\_sort\_time,

"(ii)Merge sort: ",

merge\_sort\_time,

"(iii)Insertion-mergesort-hybrid: ",

merge\_sort\_and\_insertion\_sort\_mix\_time,

"(iv)Qsort library function: ",

qsort\_time);

}

int main(int argc, char const \*argv[])

{

int t;

scanf("%d", &t);

while(t--)

{

int size, num\_of\_times;

scanf("%d %d", &size, &num\_of\_times);

test\_sorting\_runtimes(size,num\_of\_times);

}

return 0;

}

**EXPERIMENT 6: HEAP SORT**

Heapsort algorithm builds a heap from the unsorted elements, and then removes the largest element and places it at the end of a sorted array. The heap is reconstructed after each removal until there is no element left in the head and the sorted array is full.

In the worst case, the complexity of the heapsort algorithm is O(nlogn) where n is the number of elements in the unsorted list.

**C heapsort implementation**

#include<stdio.h>

void create(int []);

void down\_adjust(int [],int);

int main()

{

int heap[30],n,i,last,temp;

printf("Enter no. of elements:");

scanf("%d",&n);

printf("\nEnter elements:");

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

scanf("%d",&heap[i]);

//create a heap

heap[0]=n;

create(heap);

//sorting

while(heap[0] > 1)

{

//swap heap[1] and heap[last]

last=heap[0];

temp=heap[1];

heap[1]=heap[last];

heap[last]=temp;

heap[0]--;

down\_adjust(heap,1);

}

//print sorted data

printf("\nArray after sorting:\n");

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

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

return 0;

}

void create(int heap[])

{

int i,n;

n=heap[0]; //no. of elements

for(i=n/2;i>=1;i--)

down\_adjust(heap,i);

}

void down\_adjust(int heap[],int i)

{

int j,temp,n,flag=1;

n=heap[0];

while(2\*i<=n && flag==1)

{

j=2\*i; //j points to left child

if(j+1<=n && heap[j+1] > heap[j])

j=j+1;

if(heap[i] > heap[j])

flag=0;

else

{

temp=heap[i];

heap[i]=heap[j];

heap[j]=temp;

i=j;

}

}

}