Experiment 9 a: List of Marks obtained by students of your class is maintained in form a linked List. Apply a sorting algorithm with complexity O (n^2) to sort the list.

**CO Attained:** CO3 and CO5

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

**Resource:** C/C++, Dev C++

**Program Logic:** Quick Sort is a Divide and Conquer algorithm. 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 the first element as a pivot.

2. Always pick last element as a pivot (implemented below)

3. Pick a random element as a pivot.

4. Pick the median as a pivot.

1 Algorithm QuickSort(p, q)

2 // Sorts the elements a[p],..., a[q] which reside in the global

3 // array a[l:n] into ascending order; a[n+1]is considered to

4 //be defined and must be >= all the elements in a[l:n].

5 {

6 if (p <q) then //If there are more than one element

7 {

8 // divide P into two sub problems.

9 j :=Partition(a,p, q + 1);

10 // j is the position of the partitioning element.

11 // Solve the sub problems.

12 QuickSort(p,j-1);

13 QuickSort(j+ l,q);

14 // There is no need for combining solutions.

15 } }

#include<bits/stdc++.h>

using namespace std;

class Node{

public:

intval;

Node \*next;

Node(){

this->val = INT\_MIN;

this->next = NULL;

}

Node(int x){

val = x;

this->next = NULL;

}

};

class LL{

Node \*head;

public:

LL(){

this->head = NULL;

}

voidinsertAtBegin(int x){

Node \*n1 = new Node(x);

n1->next = head;

head = n1;

}

voidinsertAtEnd(int x){

Node \*n1 = new Node(x);

if(head==NULL){

head=n1;

return ;

}

Node \*temp = head;

while(temp->next){

temp = temp->next;

}

temp->next = n1;

}

void sorting(){

Node \*temp1 = head, \*temp2 = head;

while(temp1->next){

temp2 = temp1->next;

while(temp2){

if(temp2->val< temp1->val){

int temp = temp2->val;

temp2->val = temp1->val;

temp1->val = temp;

}

temp2 = temp2->next;

}

temp1 = temp1->next;

}

}

voidprintList(){

if(!head){

cout<<"Underflow: No data to print.";

return ;

}

Node \*temp = head;

while(temp){

cout<<temp->val<<"->";

temp = temp->next;

}

}

};

int main(){

LL l1;

l1.insertAtEnd(11); // 11

l1.insertAtEnd(12); // 11 12

l1.insertAtEnd(13); // 11 12 13

l1.insertAtEnd(14); // 11 12 13 14

l1.insertAtBegin(90); // 90 11 12 13 14 15

l1.insertAtBegin(80); // 80 90 11 12 13 14 15

l1.sorting();

l1.printList();cout<<"\n"; // 11->13->14

return 0;

}

**Outcome:**

11,12, 13 ,14,80,90

Experiment 9b : Two arrays are maintained such that the marks are in descending order. We need to reverse the two arrays using another data structure so that the arrays are in ascending order. Hint: Use LIFO

**CO Attained:** CO1, CO3, CO4

**Objective:**

To learn about the usage of a data structure using stack (LIFO). This is one of the common questions asked in placement

**Resource:** C/C++, Dev C++

**Algorithm:**

1. Let the size of A1[] be m and the size of A2[] be n.
2. Create a temporary array temp of size m and copy the contents of A1[] to it.
3. Create another array visited[] and initialize all entries in it as false. visited[] is used to mark those elements in temp[] which are copied to A1[].
4. Sort temp[]
5. Initialize the output index ind as 0.
6. Do following for every element of A2[i] in A2[]
7. Binary search for all occurrences of A2[i] in temp[], if present then copy all occurrences to A1[ind] and increment ind. Also mark the copied elements visited[]
8. Copy all unvisited elements from temp[] to A1[]

#include<bits/stdc++.h>

using namespace std;

voidreverse\_using\_stack(vector<int>&v){

stack<int>st;

for(auto &x : v){

st.push(x);

}

intitr=0;

while(st.size()){

v[itr++] = st.top();

st.pop();

}

}

void print1d(vector<int>&v){

for(auto &x : v){

cout<<x<<"";

}

cout<<"\n";

}

int main(){

vector<int>v1 = {5,4,3,2,1};

vector<int>v2 = {9,8,7,6,5};

reverse\_using\_stack(v1);

reverse\_using\_stack(v2);

print1d(v1);

print1d(v2);

return 0;

}

Outcome:

1,2,3,4,5,6,7,8,9

Experiment 10a: A thief enters a house. He is carrying a sack of some capacity say ‘m’. He can fill the sack with objects in the house and can carry a fraction of the object. The weights of each object are given by wi. For carrying each of the object in the sack, he earns a profit of pi. Write a code to maximize his profit.

**CO Attained:** CO2, CO4, CO5

**Objective:** To learn the 0-1 knapsack using Dynamic programming

**Resource:** C/C++, Dev C++

**Algorithm:**

The two sequences v = <v1, v2, …,vn> and w = <w1, w2, …, wn>

Dynamic-0-1-knapsack (v, w, n, W)

for w = 0 to W do

c[0, w] = 0

for i = 1 to n do

c[i, 0] = 0

for w = 1 to W do

ifwi ≤ w then

if vi + c[i-1, w-wi] then

c[i, w] = vi + c[i-1, w-wi]

else c[i, w] = c[i-1, w]

else

c[i, w] = c[i-1, w]

CODE:

#include <bits/stdc++.h>

using namespace std;

// Returns the value of maximum profit

intknapSackRec(int W, intwt[], intval[], int i, int\*\* dp)

{

// base condition

if (i < 0)

return 0;

if (dp[i][W] != -1)

returndp[i][W];

if (wt[i] > W) {

// Store the value of function call

// stack in table before return

dp[i][W] = knapSackRec(W, wt, val, i - 1, dp);

returndp[i][W];

}

else {

// Store value in a table before return

dp[i][W] = max(val[i]

+ knapSackRec(W - wt[i], wt, val,

i - 1, dp),

knapSackRec(W, wt, val, i - 1, dp));

// Return value of table after storing

returndp[i][W];

}

}

intknapSack(int W, intwt[], intval[], int n)

{

// double pointer to declare the

// table dynamically

int\*\* dp;

dp = new int\*[n];

// loop to create the table dynamically

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

dp[i] = new int[W + 1];

// loop to initially filled the

// table with -1

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

for (int j = 0; j < W + 1; j++)

dp[i][j] = -1;

returnknapSackRec(W, wt, val, n - 1, dp);

}

// Driver Code

int main()

{

intval[] = { 60, 100, 120 };

intwt[] = { 10, 20, 30 };

int W = 50;

int n = sizeof(val) / sizeof(val[0]);

cout<<knapSack(W, wt, val, n);

return 0;

}

**Outcome:**

**Input:**N = 3, W = 50, profit[] = { 60, 100, 120 }, weight[] = { 10, 20, 30 }  
**Output:**240