Logo, company name

Description automatically generated

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY**

**AMITY UNIVERSITY UTTAR PRADESH NOIDA**

5th SEMESTER

 Analysis and Design of Algorithms

ADA LAB FILE [CSE 303]

SUBMITTED BY:

**NAME – Shashikant Solanki**

**ENROLLMENT NO- A2305219303**

B. TECH(CSE)

BATCH 2019-2023, 5th SEMESTER, SECTION-**5CSE5X**

SUBMITTED TO

FACULTY NAME : **Dr. Garima Aggarwal**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Content** | **Page No.** |
| 1 | 1A: Linear Search  1B: Binary Search | 3 – 11 |
| 2 | 2A: Selection Sort  2B: Bubble Sort  2C: Insertion sort | 12 – 24 |
| 3 | 3A: Quick Sort  3B: Merge Sort | 25 – 36 |
| 4 | Fractional Knapsack using greedy approach | 37 – 42 |
| 5 | 5A: Kruskal’s Algorithm  5B: Prim’s Algorithm | 43 – 54 |
|  |  |  |

**PRACTICAL - 1(a)**

Aim- Write a program for Linear Search(Iterative Approach).

**Tool Used : Sublime Text 3**

1a. Linear Search : Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that item is returned, otherwise the search continues till the end of the data collection

**Algorithm for Linear Search**

Linear\_Search\_itr ( Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

**Code:**

#include <bits/stdc++.h>

using namespace std;

bool Linear\_Search(vector<int>&vec, int key)

{

for(auto i : vec)

{

if(i == key)

{

return true;

}

}

return false;

}

int main()

{

int n;

cout << "\n Enter No of elements : ";

cin >> n;

vector<int>arr(n);

cout << "\n Enter elemnets : ";

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

cin >> arr[i];

int key;

cout << "\n Enter key : ";

cin >> key;

if(Linear\_Search(arr,key))

cout << "Element present.";

else

cout << "Not present.";

return 0;

}

**Code:**

**Text

Description automatically generated**

**Analysis of Algorithm:**

Best case : the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1)

Average case : In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs.



= 

= Θ(n)

Worst Case: Θ (n)

Let T(n) be the number of comparisons (time) required for linear search on an array of size n. Note,

when n = 1,

T(1) = 1

Then, T(n) = 1 + T(n − 1)

T(n-1) = 1 + T(n − 2)

T(n-2) = 1 + T(n − 3)

T(n-k) = 1 + T(1)

Now, substitute in T(n)

T(n) = 1 + · · · + 1 + T(1)

and T(1) = 1

Therefore, T(n) = n − 1 + 1 = n

i.e., T(n) = Θ (n)

**PRACTICAL - 1(b)**

Aim- Write a program for Binary Search(Iterative Approach & Recursive Approach).

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. 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 until the value is found, or the interval is empty.

**Algorithm for Binary Search (Iterative)**

Binary\_Search\_itr(A[], min, max, d)

A : Array of Elements

min: Lowest Index of A

max: Highest index of A

d: key element

Step 1: (a) Repeat while min <=max

mid=(min+(max-min))/2

(b) if d= A[mid]

return mid

(c) Else if d < A[mid]

max=mid-1

(d) else

min = mid +1

Step2: Return -1

Step 3: Exit

**Code:**

#include <bits/stdc++.h>

using namespace std;

int Binary\_Search(vector<int>&vec, int key)

{

int low = 0, high = vec.size() - 1;

while(low <= high)

{

int mid = low + (high-low)/2;

if(vec[mid] == key)

return mid;

else if(vec[mid] < key)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

//Driver Program

int main()

{

int n;

cout << "\n Enter No of elements : ";

cin >> n;

vector<int>arr(n);

cout << "\n Enter elemnets : ";

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

cin >> arr[i];

int key;

cout << "\n Enter key : ";

cin >> key;

int res = Binary\_Search(arr,key);

if(res != -1)

cout << "Element present at index : " << res << "\n";

else

cout << "Element Not present." << "\n";

return 0;

}

**Algorithm for Binary Search (Recursive Approach)**

Binary\_Search\_rec(A[], low, high, x)

A[]: elements are in ascending order

low, high: the bounds for searching in A[]

d: the element to be searched

Step 1: (a) Repeat while min <=max

mid=(min+(max-min))/2

(b) if d= A[mid]

return mid

(c) Else if d < A[mid]

return Binary\_Search\_rec(A[], low, mid-1, d)

(d) else

return Binary\_Search\_rec(A[], mid+1, high, d)

Step2: Return -1

Step 3: Exit

**Code:**

#include <bits/stdc++.h>

using namespace std;

int Binary\_Search(vector<int>&vec,int low, int high, int key)

{

if(low <= high)

{

int mid = low + (high-low)/2;

if(vec[mid] == key)

return mid;

else if(vec[mid] < key)

return Binary\_Search(vec,mid+1,high,key);

//low = mid + 1;

else

return Binary\_Search(vec,low,mid-1,key);

//high = mid - 1;

}

return -1;

}

//Driver Program

int main()

{

int n;

cout << "\n Enter No of elements : ";

cin >> n;

vector<int>arr(n);

cout << "\n Enter elemnets : ";

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

cin >> arr[i];

int key;

cout << "\n Enter key : ";

cin >> key;

int res = Binary\_Search(arr,0,n-1,key);

if(res != -1)

cout << "Element present at index : " << res << "\n";

else

cout << "Element Not present." << "\n";

return 0;

}

Graphical user interface, text, application

Description automatically generated

**Analysis of Algorithm:**

Best case : the best case occurs when x is present at the middle location. The number of operations in the best case is constant (not dependent on n). So, time complexity in the best case would be Θ(1)

Average case : In average case analysis, we take all possible inputs and calculate computing time for all the inputs. Sum all the calculated values and divide the sum by total number of inputs.

= Θ(log2N)

Worst case : When x is not present, the search() functions compare it with all the elements of a[] one by one. Therefore, the worst-case time complexity of linear search would be Θ(log2N)

Let T(n) be the number of comparisons (time) required for linear search on an array of size n and c is constant.

Then, T(n) = T(n/2) + c

T(n/2) = T(n/4) + c

T(n/4) = T(n/8) + c

lly T(n/2k) = 1

n = 2k

k = log2n

Now, substitute in T(n)

T(n) = T(n/2k) + kc + 1

Therefore, T(n) = 1 + log2n.c + 1 = log2n

i.e., T(n) = Θ (log2n)

**PRACTICAL - 2(a)**

**Aim**- Write a program for Selection Sort.

**Tool Used : Sublime Text 3**

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty, and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

**Algorithm for selection sort:**

procedure selection sort

list : array of items

n : size of list

for i = 1 to n - 1

/\* set current element as minimum\*/

min = i

/\* check the element to be minimum \*/

for j = i+1 to n

if list[j] < list[min] then

min = j;

end if

end for

/\* swap the minimum element with the current element\*/

if indexMin != i then

swap list[min] and list[i]

end if

end for

end procedure

**Code:**

#include <bits/stdc++.h>

using namespace std;

void SelectionSort(vector<int>& vec)

{

int n = vec.size();

for(int i=0; i<n-1; i++)

{

int mid\_index = i;

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

{

if(vec[j] < vec[mid\_index])

mid\_index = j;

}

swap(vec[i],vec[mid\_index]);

}

}

void solve()

{

int n;

cout << "\n Enter no of elements : ";

cin >> n;

vector<int>arr(n);

cout << "\n Enter array elements : ";

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

cin >> arr[i];

cout << "\n Array before Sorting : ";

for(auto it : arr)

cout << it << " ";

cout << "\n";

SelectionSort(arr);

cout << "\n Array after Sorting : ";

for(auto it : arr)

cout << it << " ";

cout << "\n";

}

int main() {

solve();

}

**Output:**

Text

Description automatically generated

**Analysis of Algorithm:**

Graphical user interface

Description automatically generated with medium confidence

Using Recurrence relation

T(n) = T(n-1) + n

T(n-1) = T(n-2) + n  
T(n-2) = T(n-3) + n  
…

T(n) = T(n-3) + n + n + n

So,by back substitution, we get n + n + n + n + … - a total of n times

∑n−1 =n(n−1)/2

i=0

Hence the resultant complexity is O (n2) for all best, average and worst cases.

Best, average and worst-case time complexity: O(n2) which is independent of distribution of data.

**PRACTICAL - 2(b)**

**Aim**- Write a program for Bubble Sort.

Bubble Sort is comparison based sorting algorithm. In this algorithm adjacent elements are compared and swapped to make correct sequence. This algorithm is simpler than other algorithms, but it has some drawbacks also. This algorithm is not suitable for large number of data set. It takes much time to solve the sorting tasks.

**Algorithm:**

Begin

for i := 0 to size-1 do

flag := 0;

for j:= 0 to size –i – 1 do

if array[j] > array[j+1] then

swap array[j] with array[j+1]

flag := 1

done

if flag ≠ 1 then

break the loop.

done

End

**Code:**

#include <bits/stdc++.h>

using namespace std;

void BubbleSort(vector<int>& vec)

{

int n = vec.size();

for(int i=0; i<n-1; i++)

{

for(int j = 0; j<n-i-1; j++)

{

if(vec[j] > vec[j+1])

swap(vec[j],vec[j+1]);

}

}

}

void solve()

{

int n;

cout << "\n Enter no of elements : ";

cin >> n;

vector<int>arr(n);

cout << "\n Enter array elements : ";

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

cin >> arr[i];

cout << "\n Array before Sorting : ";

for(auto it : arr)

cout << it << " ";

cout << "\n";

BubbleSort(arr);

cout << "\n Array after Sorting : ";

for(auto it : arr)

cout << it << " ";

cout << "\n";

}

//Driver Program

int main() {

/\*

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

freopen("output.txt","w",stdout);

#endif

\*/

solve();

}

**Output:**

Text

Description automatically generated

**Analysis of Algorithm:**

Bubble sort uses the so-called ["decrease-by-one"](http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Levitin/L07-DecreaseConquer.htm#one) technique, a kind of divide-and-conquer. Its recurrence can be written as

T(n)=T(n−1)+(n−1).

Text

Description automatically generated

Now firstly, if we measure running time as the number of comparisons performed by Bubble Sort, then we observe that for each iteration of loop variable ii, there are n−in−i comparisons being made. Thus, total number of comparisons performed is: ∑n n−i = n(n−1)/2 = O(n2)

i=1

Suppose instead of measuring performance/running time by number of comparisons, we want to do so by the number of swap(x,y) operations performed. It is difficult to estimate the average running time as measured in number of swap operations without making assumptions on the nature of distribution of the input set.

However, it is simple if our interest is in the worst-case performance for any input size n, we try to choose the array A that maximises the number of swaps. Clearly, we can't have more swap operations performed than the number of comparisons we make(convince yourself of this).

Thus, total number of swap operations required by a run of Bubble Sort is ≤ n(n−1)/2.

Now all that remains is to show that this is a tight bound and to do so, consider input of the form A=[n,n−1,⋯,1] i.e., in the input A[i]=n−i+1. It is easy to verify that this input requires exactly n(n−1/)2 swap operations to produce the output. Thus, in the worst case, running time of Bubble Sort measured in terms of swap  performed is n(n−1)/2 = O(n2)

In other words, agnostic to exact input, running time is O(n2)

**Worst and Average Case Time Complexity:** O(n2). Worst case occurs when array is reverse sorted.

**Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

**PRACTICAL - 2(c)**

**Aim-** Write a program for Insertion Sort.

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially, and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst-case complexity are of Ο(n2), where **n** is the number of items.

**Algorithm for insertion sort:**

procedure insertionSort( A : array of items )

int holePosition

int valueToInsert

for i = 1 to length(A) inclusive do:

/\* select value to be inserted \*/

valueToInsert = A[i]

holePosition = i

/\*locate hole position for the element to be inserted \*/

while holePosition > 0 and A[holePosition-1] > valueToInsert do:

A[holePosition] = A[holePosition-1]

holePosition = holePosition -1

end while

/\* insert the number at hole position \*/

A[holePosition] = valueToInsert

end for

end procedure

**Code:**

#include <bits/stdc++.h>

using namespace std;

void Insertionsort(vector<int>&vec)

{

for(int i=1; i<vec.size(); i++)

{

int key = vec[i];

int hole = i;

while(hole > 0 && vec[hole-1] > key)

{

vec[hole] = vec[hole-1];

hole--;

}

vec[hole] = key;

}

}

void solve()

{

int n;

cout << "\n Enter no of elements : ";

cin >> n;

vector<int>arr(n);

cout << "\n Enter array elements : ";

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

cin >> arr[i];

cout << "\n Array before Sorting : ";

for(auto it : arr)

cout << it << " ";

cout << "\n";

Insertionsort(arr);

cout << "\n Array after Sorting : ";

for(auto it : arr)

cout << it << " ";

cout << "\n";

}

//Driver Program

int main() {

int tc = 1;

// cin >> tc;

for (int t = 1; t <= tc; t++) {

// cout << "Case #" << t << ": ";

solve();

}

}

**Output:**

Text

Description automatically generated

**Analysis of Algorithm:**

Text

Description automatically generated with medium confidence

Text, letter

Description automatically generated

We can express insertion sort as a recursive procedure as follows. To sort A[1... n], we recursively sort A[1... n-1] and then insert A[n] into the sorted array A[1... n-1]. Write a recurrence for the running time of this recursive version of insertion sort.

The recurrence I formed was

T(n)= O(1) if n=1

T(n−1)+O(n) if n>1

For all other cases the time depends on sorting the sequence A[1...n-1] and then insertion into that sequence. Hence it should be their sum, i.e., T(n−1)+Θ(n).

**Worst and Average Case Time Complexity:** O(n2). Worst case occurs when array is reverse sorted.

**Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

**PRACTICAL - 3(a)**

**Aim-** Write a program to implement quick sort and perform its complexity analysis.

**Platform Used-** Sublime Text 3

**Quick Sort :** QuickSort 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.

Always pick first element as pivot.

Always pick last element as pivot (implemented below)

Pick a random element as pivot.

Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

**Algorithm-**

PARTITION (Array arr, Value p, Value r)

x ← arr [r]

i ← p – 1

for j ← p to j ← r – 1

if arr [j] < x

then i ← i + 1

exchange arr [i] <-> arr [j]

exchange arr [i + 1] arr [r]

return i + 1

QUICKSORT (Array arr, Value p, Value r)

if p < r

then q ← PARTITION (Array arr, Value p, Value r)

QUICKSORT (Array arr, Value p, Value q)

QUICKSORT (Array arr, Value q+1, Value r)

**Code-**

#include <bits/stdc++.h>

using namespace std;

int Partition(vector<int> &vec, int start, int end)

{

int pivot = vec[end];

int PartitionIndex = start;

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

{

if (vec[i] <= pivot)

{

swap(vec[i], vec[PartitionIndex]);

PartitionIndex++;

}

}

swap(vec[PartitionIndex], vec[end]);

return PartitionIndex;

}

void Quicksort(vector<int> &vec, int start, int end)

{

if (start >= end)

return;

int Pindex = Partition(vec, start, end);

Quicksort(vec, start, Pindex - 1);

Quicksort(vec, Pindex + 1, end);

}

void solve()

{

cout << "\nEnter the number of elements : ";

int n;

cin >> n;

vector<int> vec(n);

cout << "\nEnter array elements : ";

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

{

cin >> vec[i];

}

cout << "\nArray before sorting : ";

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

cout << vec[i] << " ";

cout << "\n";

Quicksort(vec, 0, n - 1);

cout << "\nArray after sorting : ";

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

cout << vec[i] << " ";

cout << "\n";

}

int main()

{

int tc = 1;

// cin >> tc;

for (int t = 1; t <= tc; t++)

{

// cout << "Case #" << t << ": ";

solve();

}

}

**Result-**

**Text

Description automatically generated**

**Complexity Analysis-**

QUICKSORT (Array arr, Value p, Value r)

if p < r

then q ← PARTITION (Array arr, Value p, Value r)

QUICKSORT (Array arr, Value p, Value q)

QUICKSORT (Array arr, Value q+1, Value r)

Recurrence Relation –

T(n) = T(q) + T(n-q) + f(n)

Worst Case-

* One region has one element and the other has n-1 elements.
* Maximally unbalanced data structure

Recurrence q = 1

T(n) = T(1) + T(n-1) + n

T(1) = Ɵ(1)

T(n) = T(n-1) + n

Using Master’s method for decreasing function a = 1, T(n) = n\*f(n) where f(n) = n

T(n) = Ɵ()

Best Case-

* Partitioning produces two regions of n/2

q = n/2

T(n) = 2T(n/2) + Ɵ(n)

Using Masters method for dividing function, a = 2, b = 2, k = 1, p = 0, log2 2 = k, p > -1

T(n) =

Average Case-

* Consider a 9-1 proportional split for every partition for average case

Q(n) = T(9n/10) + T(n/10) + n



Using the recursion tree-

Longest path = T(n) <=

Shortest path = T(n) >=

T(n) = Ɵ(n log n)

**PRACTICAL - 3(b)**

**Aim-** Write a program to implement merge sort and perform its complexity analysis.

**Platform Used-** Sublime Text 3

**Merge Sort** : Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

**Algorithm for Merge Sort:**

MergeSort(arr[], l, r)

If l < r

1. Find the middle point to divide the array into two halves:

middle m = l+ (r-l)/2

2. Call mergeSort for first half:

Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

**Code:**

#include <bits/stdc++.h>

using namespace std;

void Merge(vector<int> &vec, vector<int> &left, vector<int> &right)

{

int i = 0, j = 0, k = 0;

while ((i < left.size()) && (j < right.size()))

{

if (left[i] <= right[j])

{

vec[k] = left[i];

i++;

k++;

}

else

{

vec[k] = right[j];

j++;

k++;

}

}

while (i < left.size())

{

vec[k] = left[i];

i++;

k++;

}

while (j < right.size())

{

vec[k] = right[j];

j++;

k++;

}

}

void Mergesort(vector<int> &vec)

{

if (vec.size() < 2)

return;

int mid = vec.size() / 2;

vector<int> left(mid);

vector<int> right((vec.size()) - mid);

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

left[i] = vec[i];

for (int i = mid; i < vec.size(); i++)

right[i - mid] = vec[i];

Mergesort(left);

Mergesort(right);

Merge(vec, left, right);

}

void solve()

{

cout << "\nEnter the number of elements : ";

int n;

cin >> n;

vector<int> vec(n);

cout << "\nEnter array elements : ";

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

{

cin >> vec[i];

}

cout << "\nArray before sorting : ";

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

cout << vec[i] << " ";

cout << "\n";

Mergesort(vec);

cout << "\nArray after sorting : ";

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

cout << vec[i] << " ";

cout << "\n";

}

int main()

{

solve();

return 0;

}

**Output :**

**Text

Description automatically generated**

**Analysis of Algorithm :**

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

MergeSort(arr[], l, r) T(n)

If l < r

middle m = l+ (r-l)/2

Call mergeSort(arr, l, m) T(n/2)

Call mergeSort(arr, m+1, r) T(n/2)

Call merge(arr, l, m, r) n

Thus, the recurrence relation obtained from above is

T(n) = 2T(n/2) + θ(n)

Solving the recurrence using master method,

T(n) = aT(n/b) + f(n) where a >= 1 and b > 1

Using Masters method for dividing function, a = 2, b = 2, k = 1, p = 0, log2 2 = k, p > -1

T(n) =

**Results :**

Merge Sort is an efficient, stable sorting algorithm with an average, best-case, and worst-case time complexity of O(n log n).

Merge Sort has an additional space complexity of O(n) in its standard implementation.

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: No

Stable: Yes

**Comparison between Quick Sort and Merge Sort:**

| Basis for comparison | Quick Sort | Merge Sort |
| --- | --- | --- |
| The partition of elements in the array | The splitting of a array of elements is in any ratio, not necessarily divided into half. | In the merge sort, the array is parted into just 2 halves (i.e. n/2). |
| Worst case complexity | O(n2) | O(nlogn) |
| Works well on | It works well on smaller array | It operates fine on any size of array |
| Speed of execution | It work faster than other sorting algorithms for small data set like Selection sort etc | It has a consistent speed on any size of data |
| Additional storage space requirement | Less(In-place) | More(not In-place) |
| Efficiency | Inefficient for larger arrays | More efficient |
| Sorting method | Internal | External |
| Stability | Not Stable | Stable |
| Preferred for | for Arrays | for Linked Lists |
| Locality of reference | Good | poor |

**PRACTICAL - 4**

**Aim-** Write a program to implement fractional knapsack using greedy approach.

**Platform Used-** Sublime Text 3

**Fractional Knapsack :** In Fractional Knapsack, we can break items for maximizing the total value of knapsack. This problem in which we can break an item is also called the fractional knapsack problem.

Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

**Algorithm:**

Fractional-Knapsack (w[1..n], p[1..n], W)

for i = 1 to n

do x[i] = 0

weight = 0 for i = 1 to n

if weight +w[i] <=W then

x[i] = 1

weight = weight + w[i] else

x[i] = (W – weight) / w[i]

weight = w break return x

**Code:**

#include <bits/stdc++.h>

using namespace std;

bool comp(pair<int,int>p1 , pair<int,int> p2)

{

double x1 = (double)p1.first / (double)p1.second;

double x2 = (double)p2.first / (double)p2.second;

if(x1 > x2)

return true;

return false;

}

void solve()

{

int N,W; cin >> N >> W;

//pair<values,weight>

vector<pair<int,int>>vec;

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

{

int x,y; cin >> x >> y;

vec.push\_back({x,y});

}

sort(vec.begin(),vec.end(),comp);

double profit = 0;

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

{

if(vec[i].second <= W)

{

profit+=vec[i].first;

W-=vec[i].second;

}

else

{

double x = (vec[i].first\*W)/(double)vec[i].second;

profit+=x;

break;

}

}

cout << "Max Profit : " << profit << "\n";

}

int main()

{

int tc = 1;

//cin >> tc;

while(tc--)

{

solve();

}

return 0;

}

**Output:**

Text

Description automatically generated

**Complexity Analysis-**

Sorting the items takes O(NlogN) time and the for loop for adding all the elements takes O(n) time. So we the time complexity for the fractional Knapsack algorithm is O(NlogN) and there is no extra space used so space complexity is O(1)

**Results-**

Hence the most optimal solution for the fractional knapsack problem is obtained using the greedy Algorithm.

**PRACTICAL – 5(a)**

**Aim-** Write a program to implement Kruskal’s Minimum Spanning Tree Algorithm

**Platform Used-** Sublime Text

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph. Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

The Kruskal's algorithm is given as follows.

**Algorithm for Kruskal’s**

Step 1: Create a forest in such a way that each graph is a separate tree.

Step 2: Create a priority queue Q that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY

Step 4: Remove an edge from Q

Step 5: IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree).

ELSE

Discard the edge

Step 6: END

**Code :**

#include <bits/stdc++.h>

using namespace std;

struct node{

int u;

int v;

int wt;

node(int f1, int f2, int w)

{

u = f1;

v = f2;

wt = w;

}

};

bool comp(node a1, node a2)

{

if(a1.wt < a2.wt)

return true;

return false;

}

int findparent(int u, vector<int> &parent)

{

if(u == parent[u])

return u;

return findparent(parent[u],parent);

}

void unionn(int u, int v, vector<int> &parent, vector<int> &rank)

{

u = findparent(u, parent);

v = findparent(v, parent);

if (rank[u] < rank[v])

{

parent[u] = v;

}

else if (rank[v] < rank[u])

{

parent[v] = u;

}

else

{

parent[v] = u;

rank[u]++;

}

}

int main()

{

int N = 1000;

int m;

cin >> m;

vector<node> edges;

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

{

int u, v, wt;

cin >> u >> v >> wt;

edges.push\_back(node(u, v, wt));

}

sort(edges.begin(), edges.end(), comp);

vector<int> parent(N);

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

parent[i] = i;

vector<int> rank(N, 0);

int cost = 0;

vector<pair<int, int>> mst;

for (auto it : edges)

{

if (findparent(it.v, parent) != findparent(it.u, parent))

{

cost += it.wt;

mst.push\_back({it.u, it.v});

unionn(it.u, it.v, parent, rank);

}

}

cout << cost << endl;

for (auto it : mst)

cout << it.first << " - " << it.second << endl;

return 0;

}

**Output :**

Graphical user interface, text

Description automatically generated

**Complexity Analysis-**

O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be at most O(V2), so O(LogV) is O(LogE) the same. Therefore, the overall time complexity is O(ElogE) or O(ElogV)

**Results-**

Hence the minimum spanning tree is obtained using the Kruskal’s minimum spanning tree greedy Algorithm.

**PRACTICAL – 5(b)**

**Aim-** Write a program to implement Prim’s Minimum Spanning Tree Algorithm

**Platform Used-** Sublime Text

Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

The algorithm is given as follows.

**Algorithm**

* Step 1: Select a starting vertex
* Step 2: Repeat Steps 3 and 4 until there are fringe vertices
* Step 3: Select an edge e connecting the tree vertex and fringe vertex that has minimum weight
* Step 4: Add the selected edge and the vertex to the minimum spanning tree T  
  [END OF LOOP]
* Step 5: EXIT

**Code :**

#include<bits/stdc++.h>

using namespace std;

int main(){

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

freopen("output.txt","w",stdout);

#endif

int N,m;

cin >> N >> m;

vector<pair<int,int> > adj[N];

int a,b,wt;

for(int i = 0; i<m ; i++){

cin >> a >> b >> wt;

adj[a].push\_back(make\_pair(b,wt));

adj[b].push\_back(make\_pair(a,wt));

}

int parent[N];

int key[N];

bool mstSet[N];

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

key[i] = INT\_MAX, mstSet[i] = false;

priority\_queue< pair<int,int>, vector <pair<int,int>> , greater<pair<int,int>> > pq;

key[0] = 0;

parent[0] = -1;

pq.push({0, 0});

while(!pq.empty())

{

int u = pq.top().second;

pq.pop();

mstSet[u] = true;

for (auto it : adj[u]) {

int v = it.first;

int weight = it.second;

if (mstSet[v] == false && weight < key[v]) {

parent[v] = u;

key[v] = weight;

pq.push({key[v], v});

}

}

}

int summ = 0;

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

{

if(mstSet[i] == true)

summ+=key[i];

}

cout << summ << endl;

for (int i = 1; i < N; i++)

cout << parent[i] << " - " << i <<" \n";

return 0;

}

**Output :**

Graphical user interface, text

Description automatically generated

**Complexity Analysis-**

* If adjacency list is used to represent the graph, then using breadth first search, all the vertices can be traversed in O(V + E) time.
* We traverse all the vertices of graph using breadth first search and use a min heap for storing the vertices not yet included in the MST.
* To get the minimum weight edge, we use min heap as a priority queue.
* Min heap operations like extracting minimum element and decreasing key value takes O(logV) time.

So, overall time complexity

= O(E + V) x O(logV)

= O((E + V)logV)

= O(ElogV)

**Comparison between Prim’s and Kruskal’s Algorithm**

|  |  |
| --- | --- |
| **Prim’s Algorithm** | **Kruskal’s Algorithm** |
| The tree that we are making or growing always remains connected. | The tree that we are making or growing usually remains disconnected. |
| Prim’s Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree. | Kruskal’s Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest. |
| Prim’s Algorithm is faster for dense graphs. | Kruskal’s Algorithm is faster for sparse graphs. |

**Results-**

Hence the minimum spanning tree is obtained using the Prim’s minimum spanning tree greedy Algorithm.