pioneerpaper.in
PAGE NO. 1

LAB:1

Wap to implement tower of hanoi Problem in c++.

```
#include <iostream>
using namespace std;
void Hanoi(int n, char A, char B, char c)
{
  if (n==0){
    return ;
}
Hanoi (n-1, A, B, C);
cout <<"move disk " <<n <<" from rod" <<A <<" to rod"
   <<C <<endl;
Hanoi (n-1, B, C, A);
}
int main (){
    int N;
    Cout <<"Enter number of disks: ";
    cin >>N;
    Hanoi (N, 'a', 'C', 'b' );
    return 0;
}
```

Algorithm:

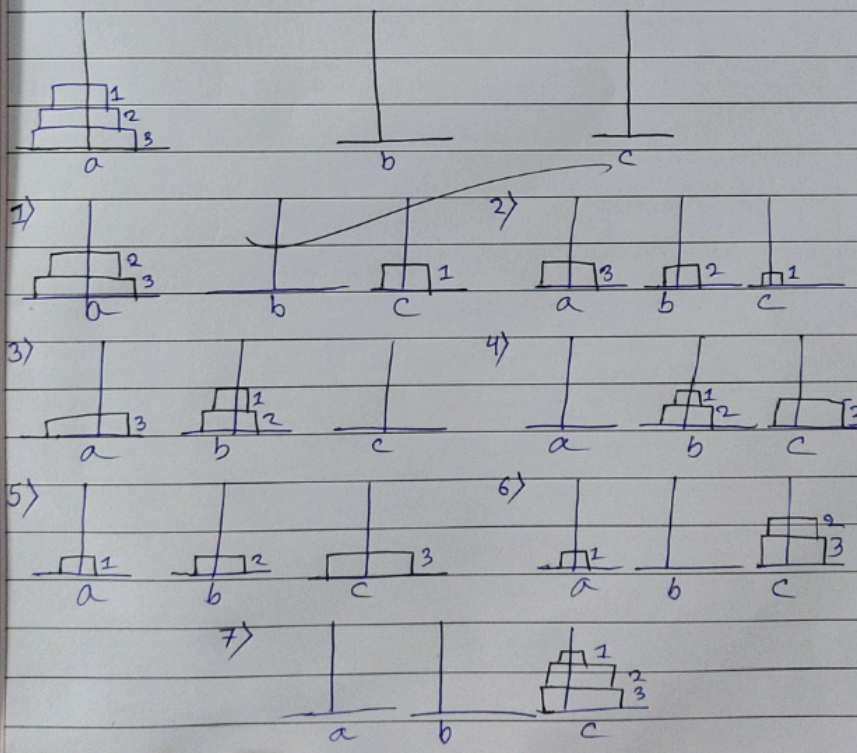
Step 1: Create function tower_of_hanoi and pass N (no. of disk), best_rod, aux_rod, end_rod.

Step 2: Make a function call for N-1 th disk.

Step 3: Print the current disk along with best_rod, end_rod.

Step 4: Repeat step 2 and step 3 until the entire stack names to end_rod.

Step 5: End.



LAB: 2

Write a Program to implement Linear Search in c++.

```
#include <iostream>
using namespace std;
int main() {
int arr[10], i, num, inden;
cout <<" Enter 10 Numbers :";
for(i = 0 ; i<10 ; i++){
    cin >> arr[i];
    }
cout<<"\n Enter a Number to Search : ";
cin >> num ;
for (i =0 ; i <10 ;i++){
  if (arr[i] == num){
     inden = i ;
     break;
  }
}
cout <<"\n Found at Index No :" <<inden;
cout <<endl;
return 0;
}
```

Algorithm:-

step 1: start

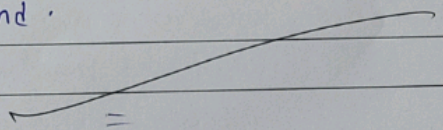
step 2: create an array.

step 3: Input the element to be searched.

step 4: Run a loop to check is the search element equals the first element of the array.

step 5: Repeat step 4 for the rest of the elements, if the search element equals any key, then print the search is sucessful, else print that it is not successful.

step 6: End.



Merge sort :-

```
#include <iostream>
using namespace std;
void merge (vector <int> & arr, int low, int mid,
            int high) {
 vector <int> temp;
 int left = low;
 int right = mid + 1;
 while (left <= mid && right <= high) {
   if (arr[left] <= arr[right]) {
      temp.push_back (arr [left]);
        left ++;
    }
   else {
      temp.push_back (arr [right]);
      right++;
    }
 }
while (left <= mid) {
   temp.push_back (arr [left]);
   left ++;
}
while (right <= high) {
   temp.push_back (arr [right]);
```

```
        right++;
    }
}
void ms(vector<int>& arr, int low, int high){
    if(low == high){
        return;
    }
    int mid = (low+high)/2;
    ms(arr, low, mid);
    ms(arr, mid+1, high);
    merge(arr, low, mid, high);
}
void mergesort(vector<int>& arr, int n){
    ms(arr, 0, n-1);
}
int main(){
    vector<int> arr;
    int n;
    cout<<"Enter array size:";
    cin>>n;
    for(int i=0; i<n; i++){
        arr.push_back(i);
    }
    cout<<"Unsorted Array:";
    for(int i=0; i<n; i++){
```

```
    cout << arr[i];
    }
    cout << endl;
    mergesort (arr, n);
    cout << "sorted Array";
    for (int i=0; i<n; i++){
        cout << arr [i];
    }
    cout << endl;
    return 0;
}
```

Algorithm :-

step 1 : create an unsorted array

step 2 : split the array from the middle into two part.

step 3 : Divide the subarrays until the ~~array~~ is unit length.

step 4 : sort the unit length subarrays,

step 5 : merge the unit length sorted array.

step 6 : Print the entire ~~sorted~~ array.

step 7 : End.

26/09/23

LAB-3

pioneerpaper.in
PAGE NO. 8

~~Knap~~ Fractional Knapsack Problem :-

```
#include <bits/stdc++.h>
using namespace std;
struct Item {
    int profit, weight;
  Item (int profit, int weight){
    this-> profit = profit;
    this-> weight = weight;
  }
};
static bool cmp (struct Item a, struct Item b){
double r1 = (double) a.profit/(double)a.weight;
double r2 = (double) b.profit/(double)b.weight;
return r1>r2;
}
double fractionalKnapsack (int W, struct Item arr[],
                           int N){
sort (arr, arr+N, cmp);
double finalvalue =0.0;
for (int i=0; i<N; i++){
  if (arr[i].weight <= W){
    W -= arr[i].weight;
    finalvalue += arr[i].profit;
  }
```

```
    else {
        finalvalue += arr[i].Profit * ((double)W/(double) arr[i].
                      weight);
        break;
    }
}
return finalvalue;
}
int main() {
    int W = 50;
    Item arr[] = {{60, 10},{100, 20},{120, 30}};
    int N = sizeof(arr) / sizeof(arr[0]);
    cout << fractionalKnapsack(W, arr, N);
    return 0;
}
```

Algorithm:-

Step 1: Sort the given array of items according to weight/value (w/v) ratio in descending order.

Step 2: Start adding the item with the maximum w/v ratio.

Step 3: Add the whole item, if the current weight is less than the capacity, else add a portion of the item to the knapsack.

Step 4: Stop when the weight is equal to the given knapsack weight and count total values.

03/10/2023