GREEDY ALGORITHMS :-

1. Fractional knapsack :-

```
#include<bits/stdc++.h>

using namespace std;

struct Item{
        int value;
        int weight;
};

static bool compare(struct Item a, struct Item b)
{       double r1 = (double)a.value / (double)a.weight;
        double r2 = (double) b.value / (double).b.weight;

        return r1> r2;
}

double fractionalKnapsack(int w, Item arr[], intn)
{       sort(arr, arr+n, compare);

        double gain=0;

        for(int i=0; i<n; i++){
                if(w>= arr[i].weight){

                        w-= arr[i].weight;
```

```
            gain+= arr[i].value;
        }
        else {
            gain+=(double)w/(double)arr[i].weight)
                                    * arr[i].value
            w=0;
            break;
        }
    }
    return gain;
}
int main(){
    int n; cin>>n;
    Item a[n];
    for(int i=0; i<n; i++) cin>>a[i].weight;
    for(int i=0; i<n; i++) cin>>a[i].value;
    int w; cin>>w;
    cout<< fractionalKnapsack(w,a,n)<<endl;
}
```

2. Prims Algorithm:-

```
#include<bits/stdc++.h>
using namespace std;
int minspanningTree(int V, vector<vector<int>> adj[])
{
    priority_queue<pair<int,int>, vector<pair<int,int>>,
                                    greater<pair<int,int>>> pq;

    pq.push({0,0});
    int sum=0;
    vector<int> visited(V,0);
    while(!pq.empty()){

        pair<int,int> p= pq.top();
        int wt= p.first;
        int x = p.second;

        pq.pop();
        if(visited[x]==1) continue;

        visited[x]=1;

        sum+=wt;

        for(auto it: adj[x])  pq.push({it[1], it[0]});

    }

    return sum;
}
```

```
int main() {
    int v; cin>>v;
    vector<vector<int>> adj[v];
    int edges; cin edges;
    for (int i=0; i<edges; i++) {
        int x, y, wt; cin>>x>>y>>wt;
        vector<int> temp;
        temp.push_back(y);
        temp.push_back(wt);
        adj[x].push_back(temp);
        temp.clear();
        temp.push_back(x);
        temp.push_back(wt);
        adj[y].push_back(temp);
    }
cout<< spanningTree(v, adj) <<endl;
}
```

3. Kruskals Algorithm :-

```
# include <bits/stdc++.h>
using namespace std;
int find_parent(vector<int> &p, int i) {
        while(p[i]!=-1)  i=p[i];
        return i;
}
int find_min (vector<vector<int>> &arr, int n, int &u,
                                                int &v)
{       int min = INT_MAX;
    for (int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
            if (min> arr[i][j] and arr[i][j] !=0)
            {     min = arr[i][j];
                  u=i;
                  v=j;
            }
        }
    }
    return min;
}
```

```
void krushkals(vector<vector<int>>& arr, vector<int>& p,
                                                int n)
{
    int u,v, min_cost=0;
    for(int i=0; i<n; i++){
        int min= find_min(arr, n, u, v);
        int src_par = find_parent (p, u);
        int dest_par = find_parent (p, v);
        if (scr_par != dest_par or
            (soc_par==-1 and dest_par==-1))
        {
            p[dest_par] = src_par;
            cout<<u<<" -> "<< v<<"  "<< min<<endl;
            min_cost += min;
        }
        arr[u][v]=0;
        arr[v][u]=0;
    }
    cout<<"Min cost = "<< min_cost <<endl;
}

int main(){
    int v; cin>>v;
    vector<vector<int>> arr;
    for(int i=0; i<v; i++) {
        vector<int> temp;
        for(int j=0; j<v; j++){
            int ew;
            cin>>ew;
            temp.push_back(ew);
        }
        arr.push_back(temp);
    }
    vector<int> parent_arr(v, -1);
    cout<<endl;
    krushkals(arr, parent_arr, v);
}
```

4) Optimal Merge Pattern:-

```
# include<bits/stdc++.h>
using namespace std;

int optimalMergePattern(int n, vector<int> arr){
    priority_queue<int, vector<int>, greater<int>> pq;
    for (int i=0; i<n; i++) pq.push(arr[i]);
    int sum=0;
    while (pq.size()!=1){
        int x= pq.top();
        pq.pop();
        int y= pq.top();
        pq.pop();
        pq.push(x+y);
        sum += (x+y);
    }
    return sum;
}

int main(){
    int n; cin>>n;
    vector<int> arr;
    for (int i=0; i<n; i++){
        int x; cin>>x;
        arr.push_back(x);
    }
    cout<< optimalMergePattern(n,arr) <<endl;
```