Name : Sonal Dilip Gholap

UID : 2021600020

Branch : CSE-AIML

Batch : B

Experiment : 5

Aim : Experiment on Greedy Approach.

- **Objectives :**

1. To study Greedy Approach by implementing Fractional Knapsack and Huffman Coding problem.
2. To derive and analyze the time complexity of Fractional Knapsack and Huffman Coding problem.
3. To understand the basic principles of greedy algorithms and learning how to design and implement a greedy algorithm.

- **Theory :**

1. A Greedy Algorithm is an approach to solving a problem that selects the most appropriate option based on the current situation.
2. This algorithm ignores the fact that the current best result may not bring about the overall optimal result. Even if the initial decision was incorrect, the algorithm never reverses it.
3. This simple, intuitive algorithm can be applied to solve any optimization problem which requires the maximum or minimum optimum result. The best thing about this algorithm is that it is easy to understand and implement.
4. The runtime complexity associated with a greedy solution is pretty reasonable. However, you can implement a greedy solution only if the problem statement follows two properties mentioned below :
5. Greedy Choice Property: Choosing the best option at each phase can lead to a global (overall) optimal solution.

6. Optimal Substructure: If an optimal solution to the complete problem contains the optimal solutions to the subproblems, the problem has an optimal substructure.

- **Fractional Knapsack Problem**

Pseudocode for Fractional Knapsack Problem

```
Fractional Knapsack (array W, array V, int M) :

    for i = 1 to size (V)
        calculate cost[i] = V[i] / W[i]
    Sort-Descending (cost)
    i = 1
    while (i <= size(V))
        if  W[i] <= M
            M ← M – W[i]
            total ← total + V[i]
        if  W[i] > M
            fraction = (V[i] / W[i])*M
            total ← total + fraction
            M = 0
    return total
```

Derivation of Time Complexity

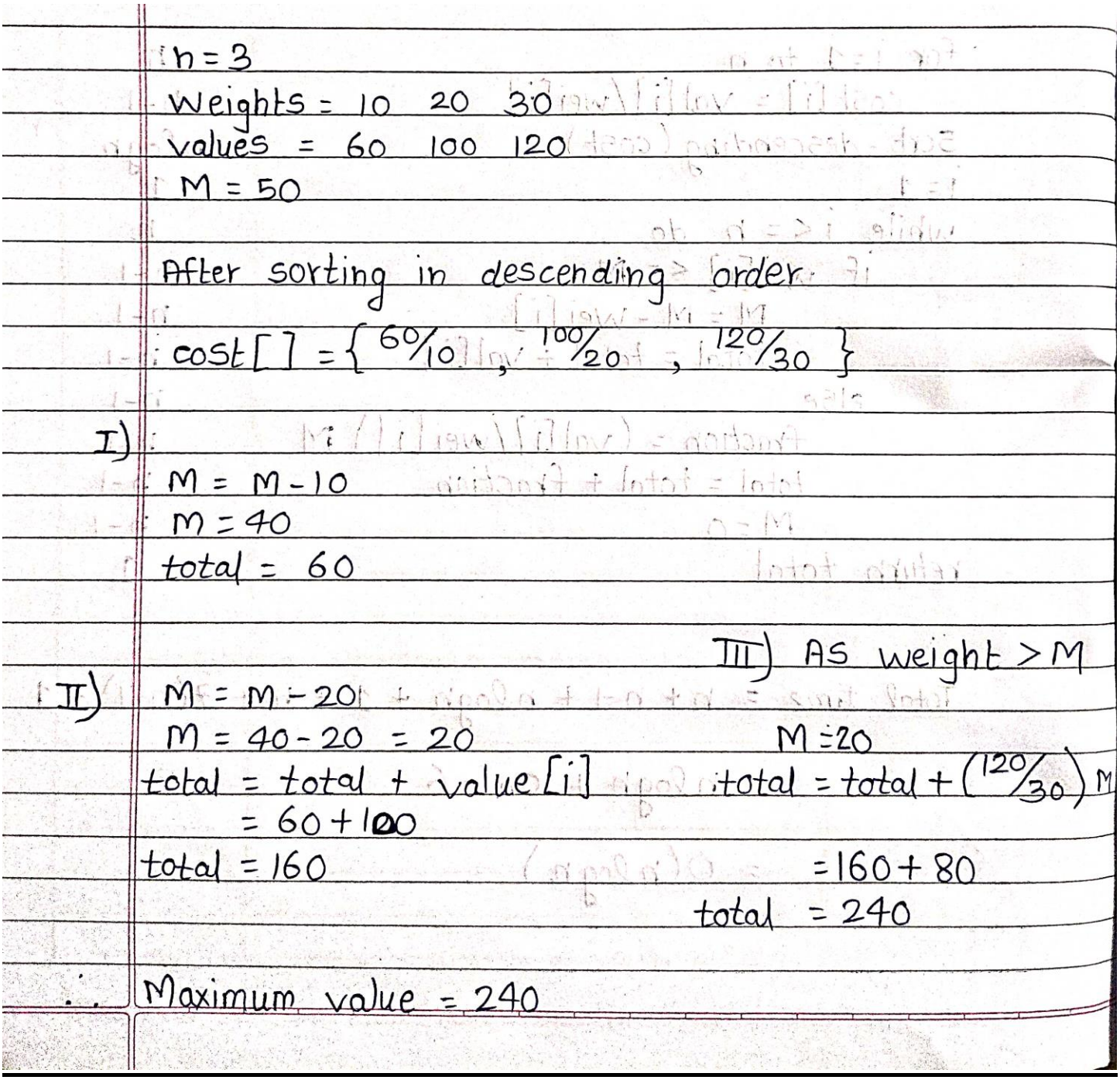
	Time taken
for i = 1 to n	n
cost[i] = val[i]/wei[i]	n-1
Sort-descending (cost)	nlogn
i = 1	1
while i <= n do	n
if wei[i] <= M	n-1
M = M - wei[i]	n-1
total = total + val[i]	n-1
else	n-1
fraction = (val[i]/wei[i]) M	n-1
total = total + fraction	n-1
M = 0	n-1
return total	1

Total time = $n + n-1 + n\log n + 1 + n + 7(n-1) + 1$

$= n\log n + 10n - 6$

$= O(n\log n)$

Solved Problem :



Code :

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

// this custom comparator function will allow
// us to compare our vector based on the
// ratio of values to weights
```

```cpp
bool compare(pair <float, int> p1, pair <float, int> p2)
{
    return p1.first > p2.first;
}

float fractional_knapsack(vector <int> weights, vector <int> values, int capacity)
{
    int len = weights.size();
    float total_value = 0;

    // vector to store the items based on their value/weight ratios
    vector <pair <float, int>> ratio(len, make_pair(0.0, 0));

    for(int i = 0; i < len; i++)
        ratio[i] = make_pair(values[i]/weights[i], i);

    // now sort the ratios in non-increasing order
    // for this purpose, we will use our custom
    // comparator function
    sort(ratio.begin(), ratio.end(), compare);

    // start selecting the items
    for(int i = 0; i < len; i++)
    {
        if(capacity == 0)
            break;

        int index = ratio[i].second;

        if(weights[index] <= capacity)
        {
            // we item can fit into the knapsack
            // hence take the whole of it
            capacity -= weights[index];

            // add the value of this item to the
            // final answer
            total_value += values[index];
        }
        else
        {
            // this item doesn't fit into the bag
            // and thus we take a fraction of it
            int value_to_consider = values[index] * (float(capacity)/float(weights[index]));
            total_value += value_to_consider;
            capacity = 0;
        }
    }

    return total_value;
```

```
}

int main()
{
    int n;
    cout << "Enter number of items: ";
    cin >> n;

    cout << "Enter the weights of the items: ";

    vector <int> weights;

    for(int i = 0; i < n; i++)
    {
        int weight;
        cin >> weight;
        weights.push_back(weight);
    }

    cout << "Enter the values of each item: ";

    vector <int> values;

    for(int i = 0; i < n; i++)
    {
        int value;
        cin >> value;
        values.push_back(value);
    }

    cout << "Enter the capacity of the knapsack: ";

    int capacity;
    cin >> capacity;

    cout << "The maximum value possible based on current list is: " <<
fractional_knapsack(weights, values, capacity) << endl;
}
```

Output :

```
PS C:\C++ Learning Course>  & 'c:\Users\Sonal Dilip Gholap\.vscode\extensions\ms-vscode.cpptoo
ls-1.15.1-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In
-ckgqd40e.rd1' '--stdout=Microsoft-MIEngine-Out-05dxo52x.1bl' '--stderr=Microsoft-MIEngine-Err
or-y0grrs4b.xxx' '--pid=Microsoft-MIEngine-Pid-f20oa3bq.jb3' '--dbgExe=C:\Program Files (x86)\
mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
Weights of the items: 1 2 3 4 5 6 7 8 9 10
Values of each item: 20 41 62 83 104 125 146 167 188 209
Capacity of the knapsack: 25
The maximum value possible based on current list is: 518
PS C:\C++ Learning Course>
```

- **Huffman Coding Problem**

Pseudocode for Huffman Coding Problem

```
Huffman(C) :

    n = C.size
    Q = priority_queue()
    for i = 1 to n
        n = node(C[i])
        Q.push(n)
    while Q.size() is not equal to 1
        Z = new node()
        Z.left = x = Q.pop
        Z.right = y = Q.pop
        Z.frequency = x.frequency + y.frequency
        Q.push(Z)
return Q
```

Derivation of Time Complexity

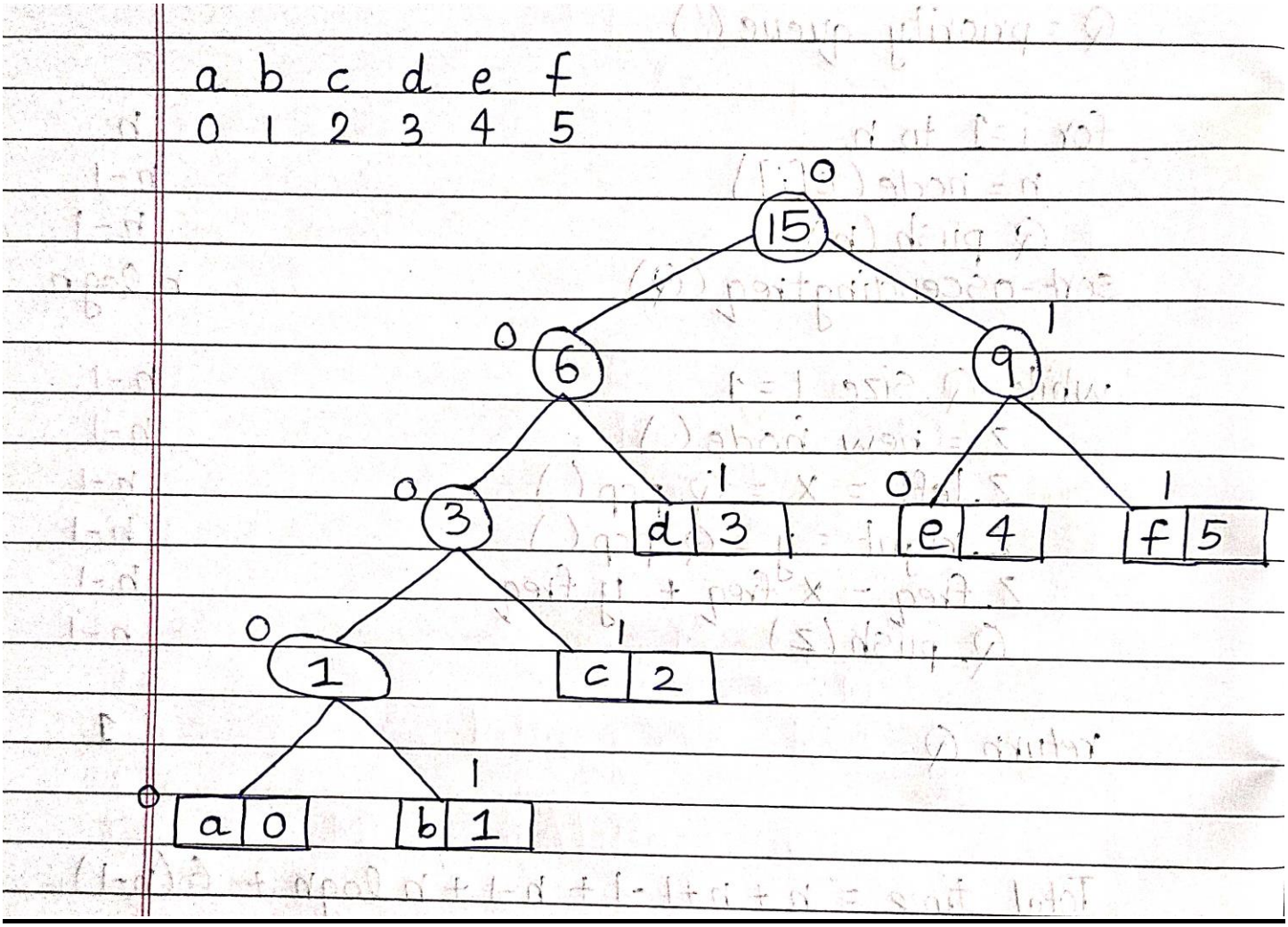
	Time taken
n = C.size	
Q = priority_queue ( )	
for i = 1 to n	n
n = node (c[i])	n-1
Q. push (n)	n-1
sort-ascendingfreq (Q)	n log n
while Q. size != 1	n-1
z = new node ( )	n-1
z. left = x = Q. pop ( )	n-1
z. right = y = Q. pop ( )	n-1
z. freq = x. freq + y. freq	n-1
Q. push (z)	n-1
return Q	1

Total time = $n + n+1 - 1 + n-1 + n\log n + 6(n-1)$

$= n\log n + 9n + (-7)$

$= O(n \log n)$

Solved Problem :



Code :

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

  //Huffman tree node
  struct MinHeapNode{
      char data;
      int freq;

      MinHeapNode *left,*right;

      MinHeapNode(char data,int freq){
          left=right=NULL;
          this->data=data;
          this->freq=freq;
      }
};

//For comparison of two nodes.
struct compare{
```

```
    bool operator()(MinHeapNode *l,MinHeapNode *r)
    {
        return (l->freq>r->freq);
    }
};

//Print Huffman Codes
void printCodes(struct MinHeapNode* root,string str){
   //If root is Null then return.
   if(!root){
       return;
   }
   //If the node's data is not '$' that means it's not an internal node and print the string.
   if(root->data!='$'){
       cout<<root->data<<": "<<str<<endl;
   }

   printCodes(root->left,str+"0");
   printCodes(root->right,str+"1");
}

//Build Huffman Tree
void HuffmanCodes(char data[],int freq[],int size){

struct MinHeapNode *left,*right,*top;

//create a min heap.
priority_queue<MinHeapNode*,vector<MinHeapNode*>,compare> minheap;

// For each character create a leaf node and insert each leaf node in the heap.
for(int i=0;i<size;i++){
    minheap.push(new MinHeapNode(data[i],freq[i]));
 }

 //Iterate while size of min heap doesn't become 1
 while(minheap.size()!=1){
     //Extract two nodes from the heap.
     left = minheap.top();
     minheap.pop();

     right = minheap.top();
     minheap.pop();

     /*Create a new internal node having frequency equal to the sum of
     two extracted nodes.Assign '$' to this node and make the two extracted
     node as left and right children of this new node.Add this node to the
     heap.*/

     top = new MinHeapNode('$',left->freq+right->freq);
```

```
    top->left = left;
    top->right = right;

    minheap.push(top);
 }
 //Calling function to print the codes.
 printCodes(minheap.top()," ");
}

int main(){

  int size=5;
  char arr[] = {'a','b','c','d','e'};
  int freq[] = {10,5,2,14,15};

  cout<<"Characters: ";
  for(int i=0;i<5;i++){
    cout<<arr[i]<<" ";
  }
  cout<<"Frequency: ";
  for(int i=0;i<5;i++){
    cout<<freq[i]<<" ";
  }

  HuffmanCodes(arr,freq,size);

return 0;
}
```

Output :

```
PS C:\C++ Learning Course>  & 'c:\Users\Sonal Dilip Gholap\.vscode\extensions\ms-vscode.cpptoo
ls-1.15.1-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In
-norhnjw1.z1l' '--stdout=Microsoft-MIEngine-Out-bthwem4j.fym' '--stderr=Microsoft-MIEngine-Err
or-svrooq1l.zry' '--pid=Microsoft-MIEngine-Pid-tgcqk4ce.pzc' '--dbgExe=C:\Program Files (x86)\
mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
Characters: a b c d e f
Frequency: 0 1 2 3 4 5
d:  00
a:  0100
b:  0101
c:  011
e:  10
f:  11
PS C:\C++ Learning Course>
```

Conclusion :

1. Time complexity of Fractional Knapsack problem – **O(n*logn)**
2. Time complexity of Huffman Coding problem – **O(n*logn)**
3. Greedy algorithms can be a simple and effective way to solve optimization problems.
4. Greedy algorithms work by making locally optimal choices at each step, without considering the global picture, in the hope of finding a global optimum.