



EXPERIMENT - 5

Algorithms Design and Analysis Lab

Aim

To implement Algorithms using Greedy Approach and analyse its time complexity.

Syeda Reeha Quasar

14114802719

4C7

EXPERIMENT – 5

Aim:

To implement Algorithms using Greedy Approach and analyse its time complexity.

Theory:

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.[1] In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

To solve a problem based on the greedy approach, there are two stages

1. Scanning the list of items
2. Optimization

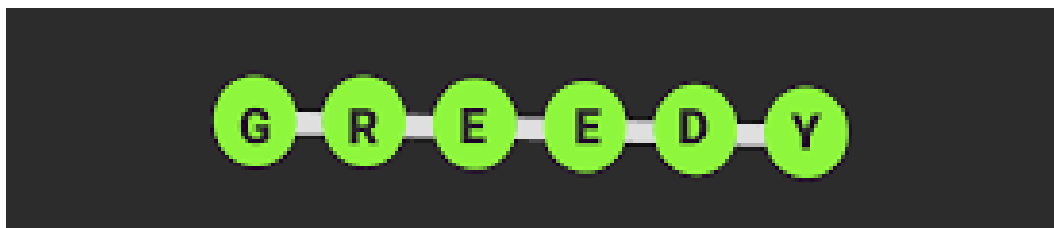
These stages are covered parallelly in this Greedy algorithm tutorial, on course of division of the array.

To understand the greedy approach, you will need to have a working knowledge of recursion and context switching. This helps you to understand how to trace the code. You can define the greedy paradigm in terms of your own necessary and sufficient statements.

Two conditions define the greedy paradigm.

- Each stepwise solution must structure a problem towards its best-accepted solution.
- It is sufficient if the structuring of the problem can halt in a finite number of greedy steps.

With the theorizing continued, let us describe the history associated with the Greedy search approach.



Knapsack problem

Instructions:

To solve 0-1 Knapsack, Dynamic Programming approach is required as Greedy approach gives an optimal solution for Fractional Knapsack. In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack. Hence, in case of 0-1 Knapsack, the value of x_i can be either 0 or 1

Problem Statement:

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items and weight of i^{th} item is w_i and the profit of selecting this item is p_i . What items should the thief take?

Dynamic-Programming Approach:

Let i be the highest-numbered item in an optimal solution S for W dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution S is V_i plus the value of the sub problem. We can express this fact in the following formula: define $c[i, w]$ to be the solution for items $1, 2, \dots, i$ and the maximum weight w .

The algorithm takes the following inputs

- The maximum weight W
- The number of items n
- The two sequences $v = \langle v_1, v_2, \dots, v_n \rangle$ and $w = \langle w_1, w_2, \dots, w_n \rangle$
- Let i be the highest-numbered item in an optimal solution S for W pounds. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ pounds and the value to the solution S is V_i plus the value of the sub problem.
- We can express this fact in the following formula: define $c[i, w]$ to be the solution for items $1, 2, \dots, i$ and maximum weight w . Then

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1, w] & \text{if } w_i \geq 0 \\ \max [v_i + c[i-1, w-w_i], c[i-1, w]] & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$

This says that the value of the solution to i items either include i^{th} item, in which case it is v_i plus a sub problem solution for $(i - 1)$ items and the weight excluding w_i , or does not include i^{th} item, in which case it is a sub problem's solution for $(i - 1)$ items and the same weight. That is, if the thief picks item i , thief takes v_i value, and thief can choose from items $w - w_i$, and get $c[i -$

1, $w - w_i$] additional value. On other hand, if thief decides not to take item i , thief can choose from item 1, 2, \dots , $i - 1$ upto the weight limit w , and get $c[i - 1, w]$ value. The better of these two choices should be made.

Pseudo code:

The first row of c is filled in from left to right, then the second row, and so on. At the end of the computation, $c[n, w]$ contains the maximum value that can be picked into the knapsack.

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
    if  $w_i \leq w$  then
      if  $v_i + c[i-1, w-w_i]$  then
         $c[i, w] = v_i + c[i-1, w-w_i]$ 
      else  $c[i, w] = c[i-1, w]$ 
    else
       $c[i, w] = c[i-1, w]$ 

```

The set of items to take can be deduced from the table, starting at $c[n, w]$ and tracing backwards where the optimal values came from. If $c[i, w] = c[i-1, w]$, then item i is not part of the solution, and we continue tracing with $c[i-1, w]$. Otherwise, item i is part of the solution, and we continue tracing with $c[i-1, w-W]$.

Sample Example:

Assume that we have a knapsack with max weight capacity $W = 5$. Our objective is to fill the knapsack with items such that the benefit (value or profit) is maximum.

Following table contains the items along with their value and weight.

Item	i	1	2	3	4
value	val	100	20	60	40
weight	wt.	3	2	4	1

Total items $n = 4$

Total capacity of the knapsack $W = 5$

Now we create a value table $V[i, w]$ where, i denotes number of items and w denotes the weight of the items. Rows denote the items and columns denote the weight. As there are 4 items so, we have 5 rows from 0 to 4.

And the weight limit of the knapsack is $W = 5$ so, we have 6 columns from 0 to 5

After calculation, the value table V

$V[i, w]$	$w = 0$	1	2	3	4	5
$i = 0$	0	0	0	0	0	0
1	0	0	0	100	100	100
2	0	0	20	100	100	120
3	0	0	20	100	100	120
4	0	40	40	100	140	140

Maximum value earned

Max Value = $V[n, W] = V[4, 5] = 140$

Result and Analysis:

This algorithm takes $\theta(n, w)$ times as table c has $(n + 1) \cdot (w + 1)$ entries, where each entry requires $\theta(1)$ time to compute.

Source Code:

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

int knapSackRec(int W, int wt[], int val[], int i, int** dp)
{
    if (i < 0)
```

```
        return 0;
    if (dp[i][W] != -1)
        return dp[i][W];
    if (wt[i] > W) {

        dp[i][W] = knapSackRec(W, wt, val, i - 1, dp);
        return dp[i][W];
    }
    else {
        dp[i][W] = max(val[i] + knapSackRec(W - wt[i], wt, val, i - 1, dp),
knapSackRec(W, wt, val, i - 1, dp));
        return dp[i][W];
    }
}

int knapSack(int W, int wt[], int val[], int n)
{
    int** dp;
    dp = new int*[n];

    for (int i = 0; i < n; i++)
        dp[i] = new int[W + 1];

    for (int i = 0; i < n; i++)
        for (int j = 0; j < W + 1; j++)
            dp[i][j] = -1;
    return knapSackRec(W, wt, val, n - 1, dp);
}

int main()
{
    int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);

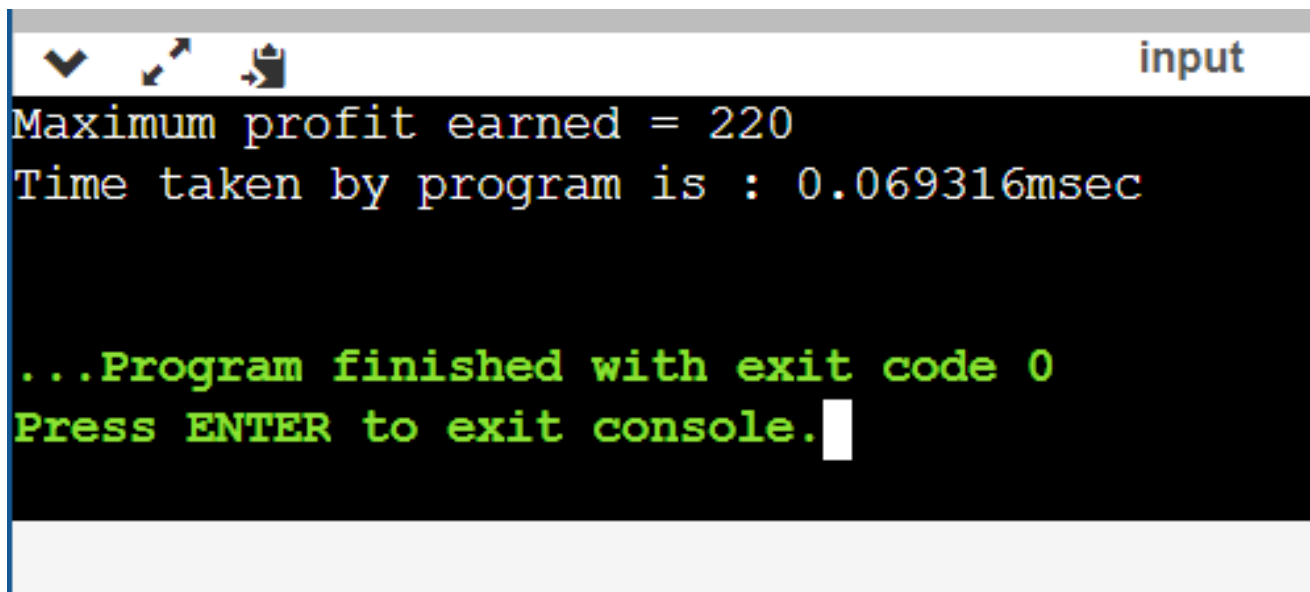
    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    cout << "Maximum profit earned = " << knapSack(W, wt, val, n);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
```

```
time_taken *= 1e-9*1000;  
  
cout << "\nTime taken by program is : " << time_taken << setprecision(6);  
cout << "msec" << endl;  
  
return 0;  
}
```

Output:

A screenshot of a console window with a dark background. The title bar is light gray and contains three icons on the left and the word 'input' on the right. The console text is as follows:
Maximum profit earned = 220
Time taken by program is : 0.069316msec

...Program finished with exit code 0
Press ENTER to exit console.
The text is in a monospaced font. The first two lines are in a light gray color, and the last three lines are in a bright green color. A white cursor is visible at the end of the last line.

```
input  
Maximum profit earned = 220  
Time taken by program is : 0.069316msec  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Fractional Knapsack problem

Source Code

```
#include <bits/stdc++.h>  
  
using namespace std;  
  
struct Item  
{  
    int value, weight;
```

```
    Item(int value, int weight) : value(value), weight(weight) {}
};

bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

double fractionalKnapsack(struct Item arr[], int N, int size)
{
    sort(arr, arr + size, cmp);
    int curWeight = 0;
    double finalvalue = 0.0;

    for (int i = 0; i < size; i++)
    {
        if (curWeight + arr[i].weight <= N)
        {
            curWeight += arr[i].weight;
            finalvalue += arr[i].value;
        }
        else
        {
            int remain = N - curWeight;
            finalvalue += arr[i].value * ((double)remain / arr[i].weight);

            break;
        }
    }
    return finalvalue;
}

int main()
{
    srand((unsigned)time(0));
    int N = 60;

    Item arr[] = {{100, 10}, {280, 40}, {120, 20}, {120, 24}};
    int size = sizeof(arr) / sizeof(arr[0]);

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);
```



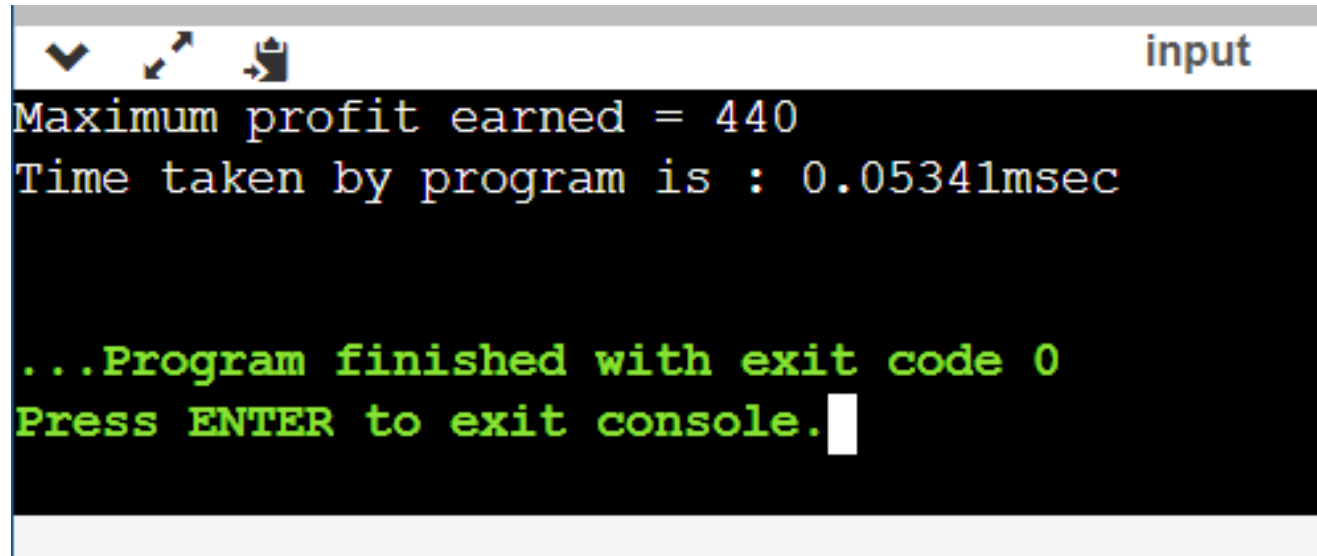
```
cout << "Maximum profit earned = " << fractionalKnapsack(arr, N, size);

auto end = chrono::high_resolution_clock::now();
double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

time_taken *= 1e-9 * 1000;

cout << "\nTime taken by program is : " << time_taken << setprecision(6) <<
"msec" << endl;
return 0;
}
```

Output



```
input
Maximum profit earned = 440
Time taken by program is : 0.05341msec

...Program finished with exit code 0
Press ENTER to exit console.
```

Viva Questions

1. Why is the greedy approach not suitable for 0-1 Knapsack Problem

Ans.

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either **0** or **1**, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

2. What is the difference between fractional and 0-1 Knapsack problem

Ans.

0-1 Knapsack problem

In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.

```
1. Input:
2.   Items as (value, weight) pairs
3.   arr[] = {{60, 10}, {100, 20}, {120, 30}}
4.   Knapsack Capacity, W = 50;
5. Output:
6.   Maximum possible value = 220
7.   by taking items of weight 20 and 30 kg
```

Fractional Knapsack

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

```
1. Input :
2.   Same as above
3. Output :
```

4. Maximum possible value = 240
5. By taking full items of 10 kg, 20 kg and
6. 2/3rd of last item of 30 kg

3. What is memorization?

Ans.

In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Memory in your C++ program is divided into two parts –

- The stack – All variables declared inside the function will take up memory from the stack.
- The heap – This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

**4. Find out maximum profit for fractional knapsack with maximum allowable weight =14 and n=5 (p1,p2,p3,p4,p5)={70,25,15,29,38};
(w1,w2,w3,w4,w5)={10,1,2,2,6}**

Ans.

Maximum profit earned = 177

Time taken by program is : 0.056174msec

5. Solve the following instance of the 0/1 knapsack problem using dynamic programming

Weight 1 2 3 2

Profit 10 15 25 12

Ans.

Maximum profit earned = 7.8

Time taken by program is : 0.064123msec

Activity Selection

To implement Activity Selection Problem.

Problem Statement:

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Example 1 : Consider the following 3 activities sorted by finish time.

`start[] = {10, 12, 20};`

`finish[] = {20, 25, 30};`

A person can perform at most **two** activities. The maximum set of activities that can be executed is {0, 2} [These are indexes in start[] and finish[]]

Example 2 : Consider the following 6 activities sorted by finish time.

`start[] = {1, 3, 0, 5, 8, 5};`

`finish[] = {2, 4, 6, 7, 9, 9};`

A person can perform at most **four** activities. The maximum set of activities that can be executed is {0, 1, 3, 4} [These are indexes in start[] and finish[]]

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

- 1) Sort the activities according to their finishing time
- 2) Select the first activity from the sorted array and print it.
- 3) Do the following for the remaining activities in the sorted array.

.....a) If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it.

Source Code

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

using namespace std;

void SelectActivities(vector<int>s,vector<int>f){
    // Vector to store results.
    vector<pair<int,int>>ans;

    // Minimum Priority Queue to sort activities in ascending order of finishing time
    (f[i]).

    priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>p;

    for(int i=0;i<s.size();i++){
        // Pushing elements in priority queue where the key is f[i]
        p.push(make_pair(f[i],s[i]));
    }

    auto it = p.top();
    int start = it.second;
    int end = it.first;
    p.pop();
    ans.push_back(make_pair(start,end));

    while(!p.empty()){
        auto itr = p.top();
        p.pop();
        if(itr.second >= end){
            start = itr.second;
```

```
        end = itr.first;
        ans.push_back(make_pair(start,end));
    }
}

cout << "Following Activities should be selected. " << endl << endl;

for(auto itr=ans.begin();itr!=ans.end();itr++){
    cout << "Activity started at: " << (*itr).first << " and ends at " <<
    (*itr).second << endl;
}
}

int main()
{
    vector<int>s = {1, 3, 0, 5, 8, 5};
    vector<int>f = {2, 4, 6, 7, 9, 9};

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    SelectActivities(s,f);

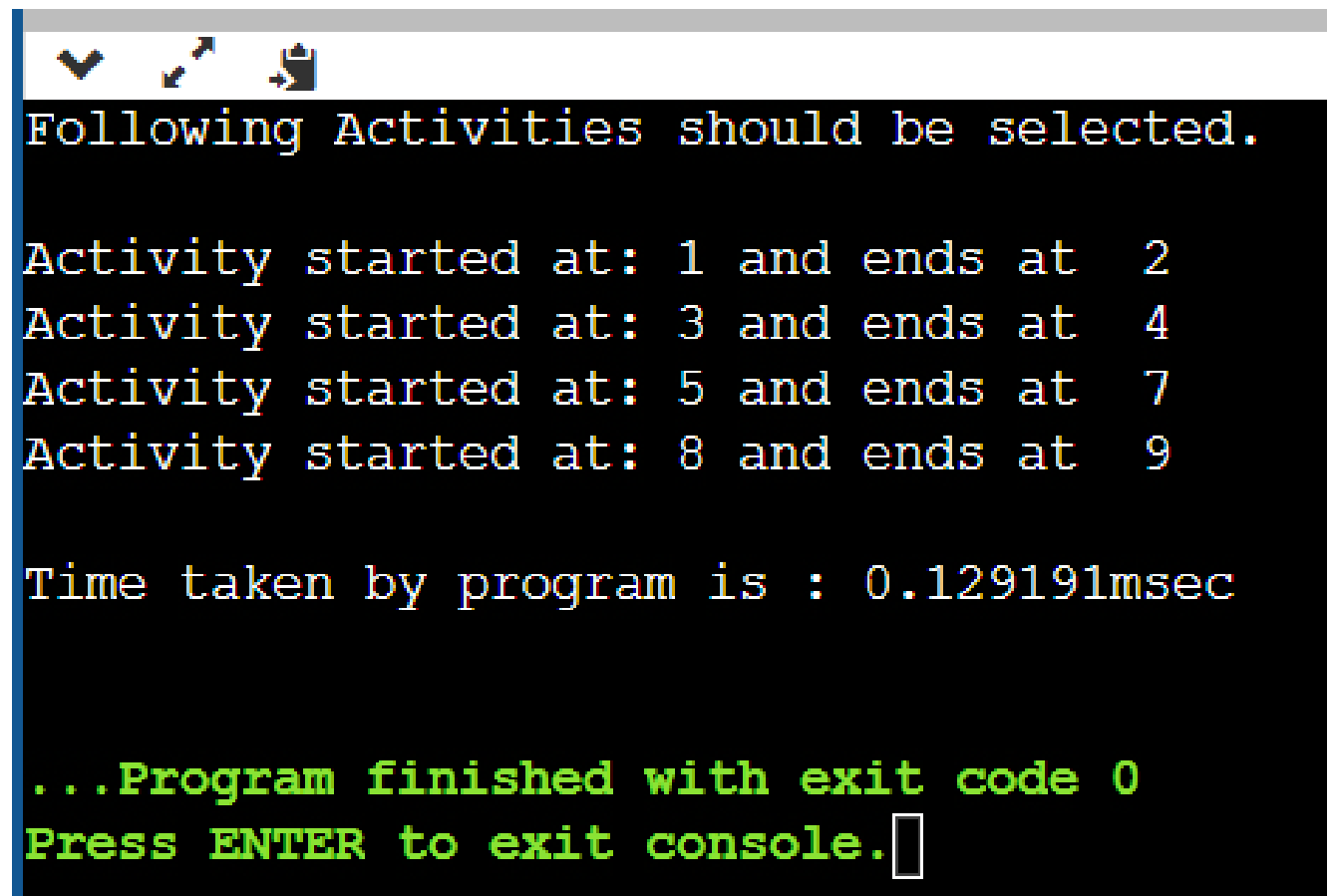
    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9*1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;
```

```
    return 0;  
}
```

Output

A screenshot of a terminal window with a dark background. The window has a title bar with three icons on the left. The output text is displayed in a monospaced font, with some lines in green and others in yellow/white. The text reads: "Following Activities should be selected." followed by four lines of activity ranges, then the execution time, and finally a green message indicating the program finished successfully with an exit code of 0, followed by a prompt to press ENTER.

```
Following Activities should be selected.  
  
Activity started at: 1 and ends at 2  
Activity started at: 3 and ends at 4  
Activity started at: 5 and ends at 7  
Activity started at: 8 and ends at 9  
  
Time taken by program is : 0.129191msec  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Huffman Encoding

To implement Huffman Coding and analyze its time complexity.

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are [Prefix Codes](#), means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be "cccd" or "ccb" or "acd" or "ab".

See [this](#) for applications of Huffman Coding.

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree

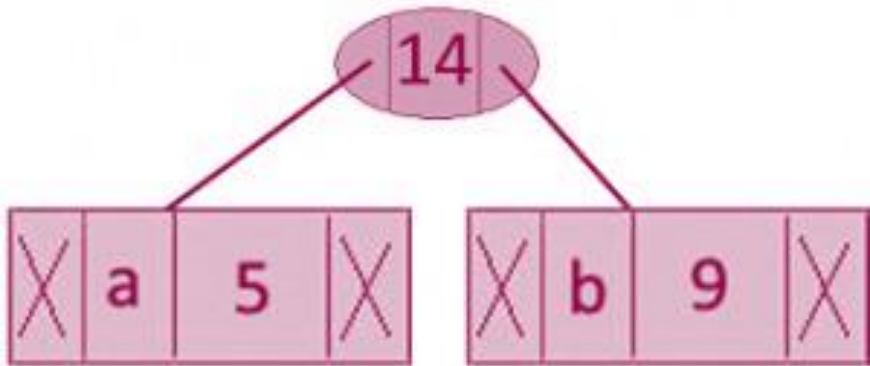
Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

character Frequency

a 5
b 9
c 12
d 13
e 16
f 45



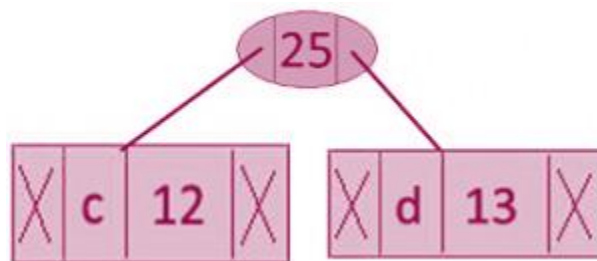
Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency $5 + 9 = 14$.

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency	Internal Node	14
c	12	e	16
d	13	f	45

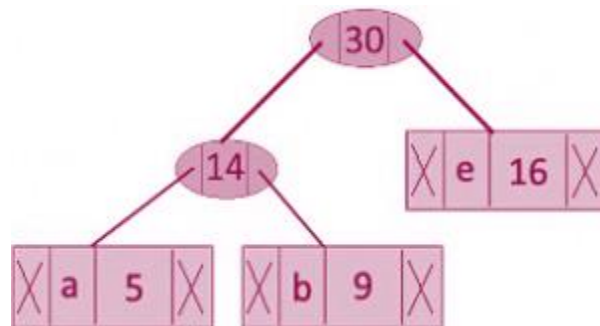
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency $12 + 13 = 25$



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

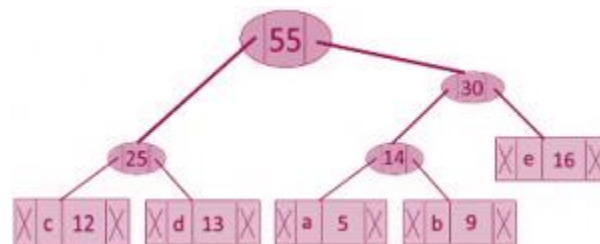
character	Frequency
Internal Node	14
E	16
Internal Node	25
f	45

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency $14 + 16 = 30$



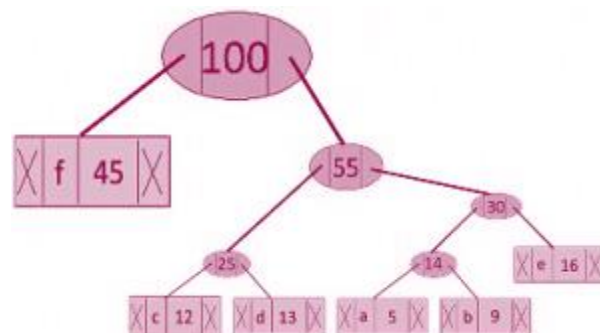
Now min heap contains 3 nodes.

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency $25 + 30 = 55$



Now min heap contains 2 nodes.

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency $45 + 55 = 100$



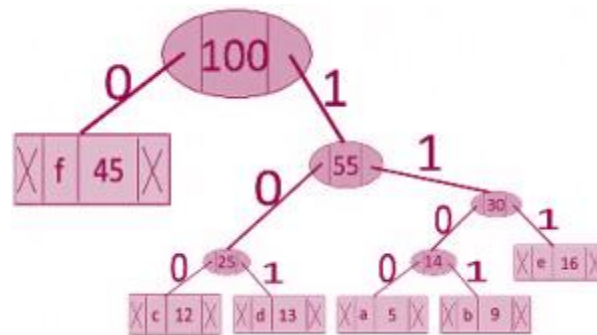
Now min heap contains only one node.

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to

the array. Print the array when a leaf node is encountered.



character code-word

f	0
c	100
d	101
a	1100
b	1101
e	111

Source Code

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

using namespace std;

struct MinHeapNode{
    char data;
    unsigned freq;
    MinHeapNode *left, *right;
    MinHeapNode(char data, unsigned freq){
        left = right = NULL;
        this->data = data;
```

```
        this->freq = freq;
    }
};

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

void printCodes(struct MinHeapNode* root, string str){
    if (!root)
        return;

    if (root->data != '#')
        cout << root->data << ": " << str << "\n";

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

void HuffmanCodes(char data[], int freq[], int size){
    struct MinHeapNode *left, *right, *top;

    priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;

    for (int i = 0; i < size; ++i)
        minHeap.push(new MinHeapNode(data[i], freq[i]));

    while (minHeap.size() != 1) {
```

```
        left = minHeap.top();
        minHeap.pop();

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

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

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

        minHeap.push(top);
    }

    printCodes(minHeap.top(), "");
}

int main(){

    char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int freq[] = { 5, 9, 12, 13, 16, 45 };

    int size = sizeof(arr) / sizeof(arr[0]);

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    HuffmanCodes(arr, freq, size);

    auto end = chrono::high_resolution_clock::now();
```

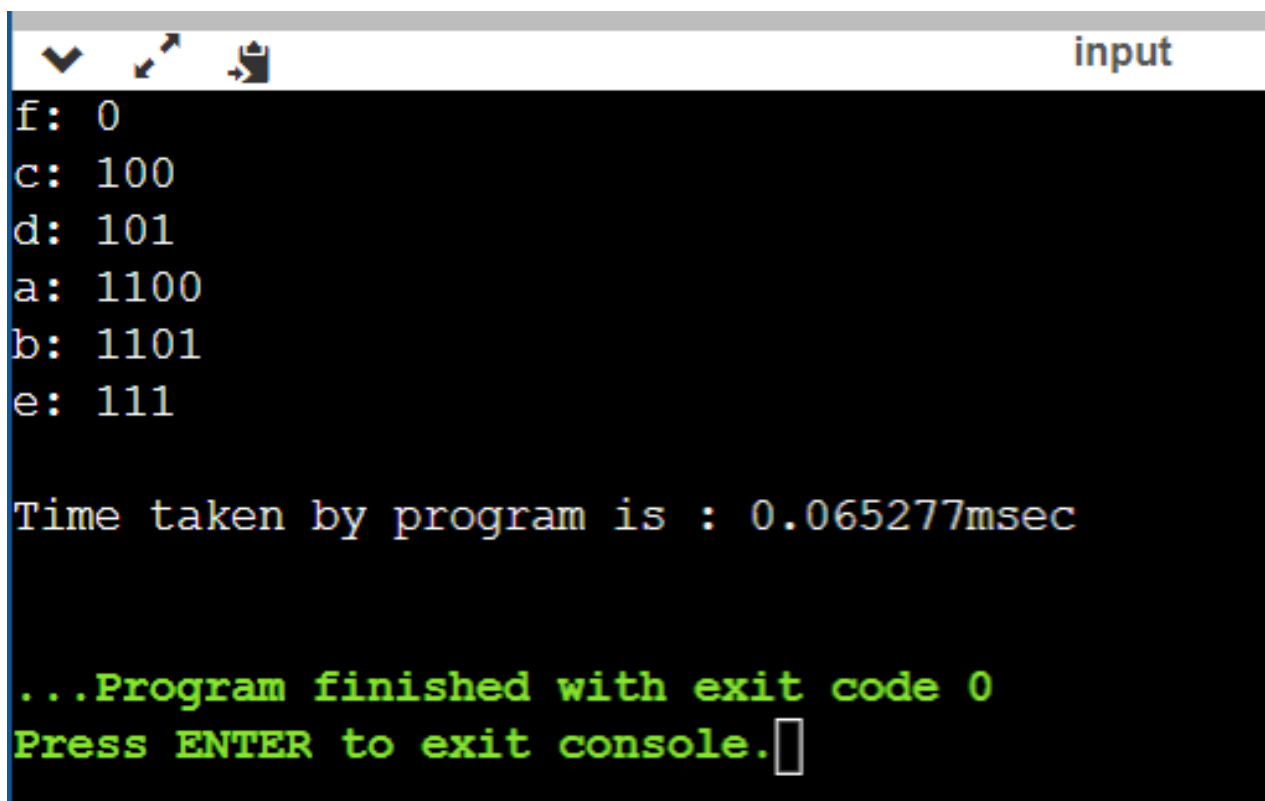
```
double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

time_taken *= 1e-9*1000;

cout << "\nTime taken by program is : " << time_taken << setprecision(6);
cout << "msec" << endl;

return 0;
}
```

Output

A screenshot of a Windows command prompt window with a title bar that says "input". The window has a black background with white and green text. The output of the program is displayed as follows:

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111

Time taken by program is : 0.065277msec

...Program finished with exit code 0
Press ENTER to exit console.
```

Task Scheduling Problem

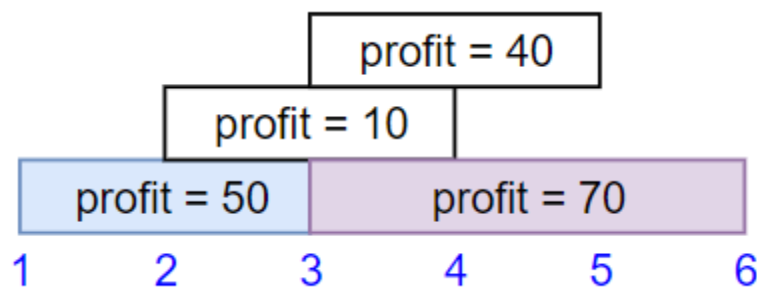
To implement Task Scheduling Problem.

We have n jobs, where every job is scheduled to be done from $startTime[i]$ to $endTime[i]$, obtaining a profit of $profit[i]$.

You're given the $startTime$, $endTime$ and $profit$ arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range.

If you choose a job that ends at time X you will be able to start another job that starts at time X .

Example 1:



Input: $startTime = [1,2,3,3]$, $endTime = [3,4,5,6]$, $profit = [50,10,40,70]$

Output: 120

Explanation: The subset chosen is the first and fourth job.

Time range $[1-3] + [3-6]$, we get profit of $120 = 50 + 70$.

Source Code

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

using namespace std;

int jobScheduling(vector<int> &startTime, vector<int> &endTime, vector<int> &profit)
{
    int n = startTime.size(), i = 0, ans = 0;
```

```
vector<pair<int, int>> v;

for (int i = 0; i < n; i++)
{
    v.push_back({startTime[i], i});
}
sort(v.begin(), v.end());
vector<int> dp(n, 0);

for (i = n - 1; i >= 0; i--)
{
    int f = v[i].first;
    int ii = v[i].second;
    int val = 0;

    val += profit[ii];

    int x = endTime[ii];

    auto it = lower_bound(v.begin(), v.end(), x, [](const pair<int, int> &p,
const int &value)
{ return p.first < value; }); // searching for next
Starttime which is greater than the current endtime

    if (it != v.end())
    {
        int j = it - v.begin();

        val += dp[j];
    }
```



```
        if (i == n - 1)
            dp[i] = max(dp[i], val);

        else
        {
            dp[i] = max(dp[i], max(val, dp[i + 1])); // either take the ith job
or skip it and take the next one
        }

        ans = max(ans, dp[i]);
    }

    return ans;
}

int main()
{
    vector<int> startTime{1, 2, 3, 3};
    vector<int> endTime{3, 4, 5, 6};
    vector<int> profit{50, 10, 40, 70};

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    cout << "Total Profit earned by job Scheduling: " << jobScheduling(startTime,
endTime, profit);

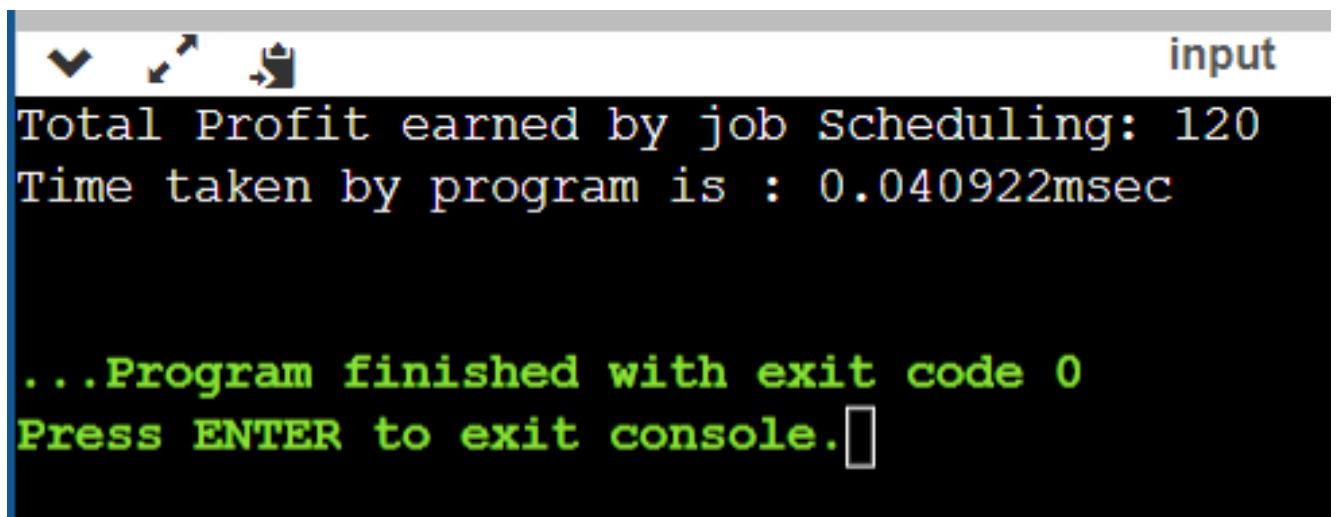
    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9 * 1000;
```

```
cout << "\nTime taken by program is : " << time_taken << setprecision(6);  
cout << "msec" << endl;  
  
return 0;  
}
```

Output

A screenshot of a console window with a dark background. The title bar is light gray and contains three icons on the left and the word 'input' on the right. The console text is as follows:
Total Profit earned by job Scheduling: 120
Time taken by program is : 0.040922msec

...Program finished with exit code 0
Press ENTER to exit console.
The last line is followed by a white cursor box.

```
input  
Total Profit earned by job Scheduling: 120  
Time taken by program is : 0.040922msec  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Viva Questions

1. What is a Greedy Algorithm?

Ans.

We call algorithms *greedy* when they utilise the greedy property. The greedy property is:

At that exact moment in time, what is the optimal choice to make?

Greedy algorithms are *greedy*. They do not look into the future to decide the global optimal solution. They are only concerned with the optimal solution *locally*. This means that **the overall optimal solution may differ from the solution the greedy algorithm chooses.**

They never look backwards at what they've done to see if they could optimise globally. This is the main difference between Greedy Algorithms and Dynamic Programming.

2. What Are Greedy Algorithms Used For?

Ans.

Greedy algorithms are quick. A lot faster than the two other alternatives (Divide & Conquer, and Dynamic Programming). They're used because they're fast.

Sometimes, Greedy algorithms give the global optimal solution every time. Some of these algorithms are:

- Dijkstra's Algorithm
- Kruskal's algorithm
- Prim's algorithm
- Huffman trees

These algorithms are Greedy, and their Greedy solution gives the optimal solution.

3. What are the various criteria used to improve the effectiveness of the algorithm?

Ans.

Input- Zero or more quantities are externally provided.

Output- At least one quantity is composed

Definiteness- Each instruction is simple and unambiguous

Finiteness- If we trace out the instructions of an algorithm, then for all step the algorithm complete after a finite number of steps

Effectiveness- Every instruction must be elementary.

4. List the advantage of Huffman's encoding?

Ans.

Huffman's encoding is one of the essential file compression techniques.

1. It is easy
2. It is flexibility
3. It implements optimal and minimum length encoding

5. Write the difference between the Dynamic programming and Greedy method.

Ans.

Dynamic programming

1. Many numbers of decisions are generated.
2. It gives an optimal solution always

Greedy method

1. Only one sequence of decision is generated.
2. It does not require to provide an optimal solution always.