LAB MANUAL

For

COMPUTER LABORATORY-III
Subject :DAA

Subject Code: **410241**

Course Objectives:

● Learn effect of data preprocessing on the performance of machine learning algorithms

● Develop in depth understanding for implementation of the regression models.

● Implement and evaluate supervised and unsupervised machine learning algorithms.

● Analyze performance of an algorithm.

● Learn how to implement algorithms that follow algorithm design strategies namely divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

● Understand and explore the working of Blockchain technology and its applications.

Course Outcomes:

After completion of the course, students will be able to :

CO1: Apply preprocessing techniques on datasets.

CO2: Implement and evaluate linear regression and random forest regression models.

CO3: Apply and evaluate classification and clustering techniques.

CO4: Analyze performance of an algorithm.

CO5: Implement an algorithm that follows one of the following algorithm design strategies: divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

CO6: Interpret the basic concepts in Blockchain technology and its application.

INDEX

Group A: DESIGN AND ANALYSIS OF ALGORITHMS	
Any 5 assignments and 1 Mini project are mandatory.	
1	Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity
2	Write a program to implement Huffman Encoding using a greedy strategy.
3	Write a program to solve a fractional Knapsack problem using a greedy method
4	Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.
5	Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen‘s matrix
6	Write a program for analysis of quick sort by using deterministic and randomized variant.
	MINI PROJECTS
	7 Mini Project - Write a program to implement matrix multiplication. Also implement multithreaded matrix multiplication with either one thread per row or one thread per cell. Analyze and compare their performance.
	8- Mini Project - Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyze the performance of each algorithm for the best case and the worst case.
	9- Mini Project - Implement the Naive string matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input.
	10. Mini Project - Different exact and approximation algorithms for Travelling-Sales-Person

Assignment No: 1

Title: Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity

Aim:

To implement recursive and non recursive pgm to calculate Fibonacci numbers and analyze their time and space complexity Using python.

Prerequisites:

- Ubuntu
- PYTHON

Objective:

Analyze performance of an algorithm.
Learn how to implement algorithms that follow algorithm design strategies namely divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

Theory:

Fibonacci

In mathematics, the Fibonacci numbers are the numbers in the following integer sequence, called the Fibonacci sequence, and characterized by the fact that every number after the first two is the sum of the two preceding ones:

1,1,2,3,5,8,13,21,34,55,89,144,...

The fibonacci number fib(n) is calculated as follows -
Using the preceding definition, fib(0) = 1, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)
This is a recursive definition

```
include <iostream>

using namespace std;


int fibonacci(int n) {

 if (n < 2)

   return 1;

 else

   return fibonacci(n - 1) + fibonacci(n - 2);

}


int main()

{

   int n;
```

```
    cout << "Enter an integer "<<endl;

    cin >> n;

    cout << "Fib returned "<<fibonacci(n)<<endl;

}
```

Calculating the time complexity of the recursive approach

```
fib(n):
   if n <= 1
      return 1
   return fib(n - 1) + fib(n - 2)
```

for n > 1:

T(n) = T(n-1) + T(n-2) + 4 (1 comparison, 2 subtractions, 1 addition)

```
T(n) = T(n-1) + T(n-2) + c
    = 2T(n-1) + c    //from the approximation T(n-1) ~ T(n-2)
    = 2*(2T(n-2) + c) + c
    = 4T(n-2) + 3c
    = 8T(n-3) + 7c
    = 2^k * T(n - k) + (2^k - 1)*cLet's find the value of k for which: n - k = 0
k = nT(n) = 2^n * T(0) + (2^n - 1)*c
    = 2^n * (1 + c) - ci.e. T(n) ~ 2^n
```

Hence the time taken by recursive Fibonacci is O(2^n) or exponential. If we look at the total computations performed then every internal fib(x) requires two other fib computations to calculate.
So the total number of function calls is the number of nodes in a binary tree or O(2^n) computations.

Iterative approach.

```
#include <iostream>

#include <ctime>

#include <stdlib.h>

#include <math.h>

using namespace std;




int fibonacci(int n) {
  if (n < 2)
    return 1;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int fib_iter(int n) {
  int result = 0;
  int prev = 1;
  int pprev = 1;

  if (n < 2)
    return 1;

  for (int i = 1; i < n; i++) {
    result = prev + pprev;
    pprev = prev;
    prev = result;
  }
  return result;
}

int main()
{
    clock_t oldtime, newtime;
    double seconds;
    int n;



    cout << "Enter an integer "<<endl;
    cin >> n;
    oldtime = clock();
    int rval = fib_iter(n);
    newtime = clock();
```

```
seconds = (double)(newtime-oldtime)/CLOCKS_PER_SEC;
cout << "Fib iter "<<n<<" = "<<rval<<"  took "<<seconds<<endl;
oldtime = clock();
rval = fibonacci(n);
newtime = clock();
seconds = (double)(newtime-oldtime)/CLOCKS_PER_SEC;
```

```
cout << "Fib "<<n<<" = "<<rval<<"  took "<<seconds<<endl
```

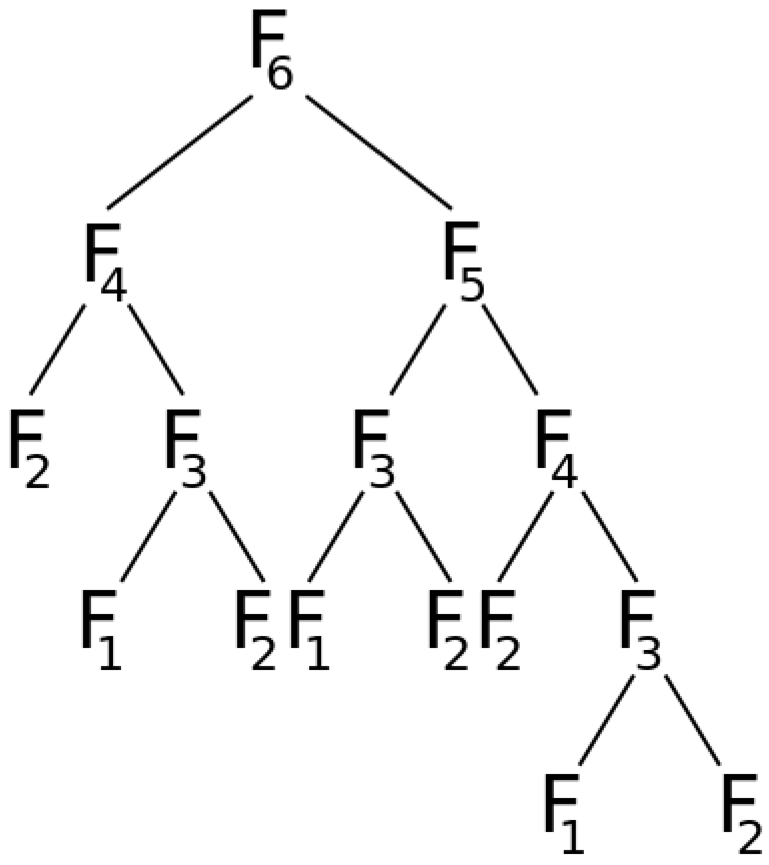
The time complexity of the iterative code is linear, as the loop runs from 2 to n, i.e. it runs in O(n) time

Space Complexity:

For the iterative approach, the amount of space required is the same for fib(6) and fib(100), i.e. as N changes the space/memory used remains the same. Hence it's space complexity is O(1) or constant.

For Fibonacci recursive implementation or any recursive algorithm, the space required is proportional to the maximum depth of the recursion tree.

Below is a diagrammatic representation of the Fibonacci recursion tree for fib(6):



The maximum depth is proportional to the N, hence the space complexity of Fibonacci recursive is O(N).

Conclusion:
Hence, we successfully implemented a recursive and non recursive pgm to calculate Fibonacci numbers and analyzed their time and space complexity Using C++.

Assignment No: 2

Title:
Write a program to implement Huffman Encoding using a greedy strategy
Aim:
Implement Huffman Encoding using a greedy strategy.

Prerequisites :

- PYTHON
- Ubuntu

Objectives:

1. Understand the importance Greedy Strategy
2. To learn Huffman Coding.

Theory:

What is Greedy Method?

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are the best fit for Greedy.

Greedy algorithms produce good solutions on some mathematical problems, but not on others. Most problems for which they work will have following components:

Components of Greedy Algorithm

The components that can be used in the greedy algorithm are:

- **Candidate set:** A solution that is created from the set is known as a candidate set.
- **Selection function:** This function is used to choose the candidate or subset which can be added in the solution.
- **Feasibility function:** A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.
- **Objective function:** A function is used to assign the value to the solution or the partial solution.
- **Solution function:** This function is used to intimate whether the complete function has been reached or not.

Disadvantages of Greedy Method
Greedy algorithm may fail to achieve the optimal solution. and may even produce the unique worst possible solution.
Greedy algorithms typically (but not always) fail to find the globally optimal solution because they usually do not operate exhaustively on all the data. They can make

commitments to certain choices too early, preventing them from finding the best overall solution later.

Applications of Greedy Method

Greedy algorithms appear in the network routing as well. Using greedy routing, a message is forwarded to the neighbouring node which is "closest" to the destination. The notion of a node's location (and hence "closeness") may be determined by its physical location.

- Kruskal's algorithm and Prim's algorithm are greedy algorithms for constructing minimum spanning trees of a given connected graph. They always find an optimal solution, which may not be unique in general.
  - It is used in a job sequencing with a deadline.
  - This algorithm is also used to solve the fractional knapsack problem.

Pseudo code of Greedy Algorithm

```
Algorithm Greedy (a, n)
{
  Solution : = 0;
 for i = 0 to n do
 {
     x: = select(a);
    if feasible(solution, x)
    {
       Solution: = union(solution , x)
    }
      return solution;
 } }
```

The above is the greedy algorithm. Initially, the solution is assigned with zero value. We pass the array and number of elements in the greedy algorithm. Inside the for loop, we select the element one by one and checks whether the solution is feasible or not. If the solution is feasible, then we perform the union.

What are Huffman Codes?

Data can be encoded efficiently using Huffman Codes.It is a widely used and beneficial technique for compressing data. Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string. Huffman coding is a lossless data compression algorithm.

There are mainly two parts. First one to create a Huffman tree, and another one to traverse the tree to find codes.

For an example, consider some strings “YYYZXXYYX”, the frequency of character Y is larger than X and the character Z has the least frequency. So the length of the code for Y is smaller than X, and code for X will be smaller than Z.

Complexity for assigning the code for each character according to their frequency is **O(n log n).**

Algorithm

huffmanCoding(string)

Input: A string with different characters.

Output: The codes for each individual characters.

```
Begin
  define a node with character, frequency, left and right child of the node for Huffman tree.
  create a list ‘freq’ to store frequency of each character, initially, all are 0
  for each character c in the string do
    increase the frequency for character ch in freq list.
  done
for all type of character ch do
    if the frequency of ch is non zero then
      add ch and its frequency as a node of priority queue Q.
  done

  while Q is not empty do
    remove item from Q and assign it to left child of node
    remove item from Q and assign to the right child of node
    traverse the node to find the assigned code
  done
End
```

traverseNode(n: node, code)

Input: The node n of the Huffman tree, and the code assigned from the previous call

Output: Code assigned with each character

```
if a left child of node n ≠φ then
```

```
  traverseNode(leftChild(n), code+’0’)    //traverse through the left child
  traverseNode(rightChild(n), code+’1’)   //traverse through the right child
else
display the character and data of current node.
```

Programme Snippet:

```
using namespace std;
struct node {
  int freq;

  char data;

  const node *child0, *child1;

   node(char d, int f = -1) { //assign values in the node
    data = d;
    freq = f;
    child0 = NULL;
    child1 = NULL;
  }

  node(const node *c0, const node *c1) {
data = 0;
    freq = c0->freq + c1->freq;
    child0=c0;
    child1=c1;
  }
bool operator<( const node &a ) const { //< operator performs to find priority in queue
    return freq >a.freq;
  }

  void traverse(string code = "")const {
    if(child0!=NULL) {
child0->traverse(code+'0'); //add 0 with the code as left child

          child1->traverse(code+'1'); //add 1 with the code as right child
```

```
      }else {
        cout << "Data: " << data<< ", Frequency: "< qu;
  int frequency[256];
for(int i = 0; i<256; i++)
      frequency[i] = 0; //clear all frequency

  for(int i = 0; i1) {
    node *c0 = new node(qu.top()); //get left child and remove from queue
    qu.pop();
    node *c1 = new node(qu.top()); //get right child and remove from queue
    qu.pop();
    qu.push(node(c0, c1)); //add freq of two child and add again in the queue
  }
```

```
  cout << "The Huffman Code: "<
```

Output

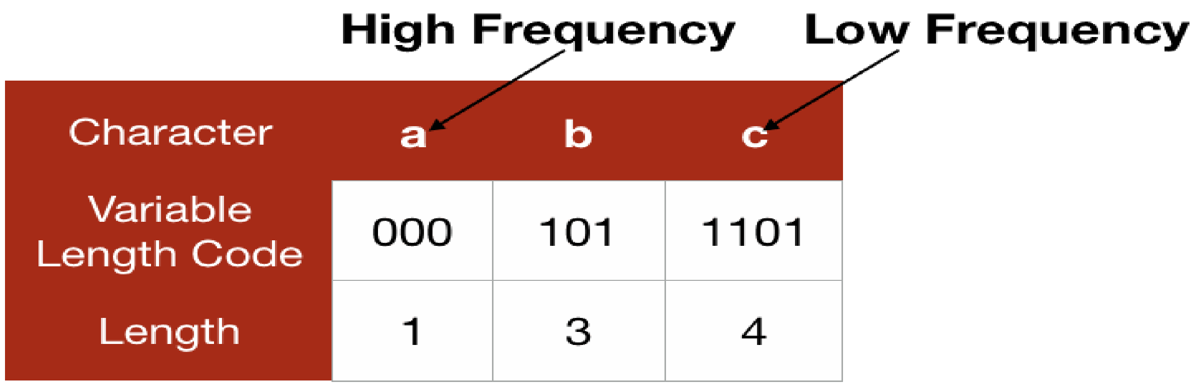
```
The Huffman Code:
Data: K, Frequency: 1, Code: 0000
Data: L, Frequency: 1, Code: 0001
Data: E, Frequency: 2, Code: 001
Data: F, Frequency: 4, Code: 01
Data: B, Frequency: 2, Code: 100
Data: C, Frequency: 2, Code: 101
Data: X, Frequency: 2, Code: 110
Data: A, Frequency: 3, Code: 111
```

Example to illustrate Huffman coding

We know that our files are stored as binary code in a computer and each character of the file is assigned a binary character code and normally, these character codes are of fixed length for different characters. For example, if we assign 'a' as 000 and 'b' as 001, the length of the codeword for both the characters are fixed i.e., both 'a' and 'b' are taking 3 bits.

Character	a	b	c
Code	000	001	010
Length	3	3	3

Huffman code doesn't use fixed length codeword for each character and assigns code words according to the frequency of the character appearing in the file. Huffman code assigns a shorter length codeword for a character which is used more number of time (or has a high frequency) and a longer length codeword for a character which is used less number of times (or has a less frequency).


High Frequency
Low Frequency

Character	a	b	c
Variable Length Code	000	101	1101
Length	1	3	4

Since characters which have high frequency has lower length, they take less space and save the space required to store the file. Let's take an example.

Character	a	b	c	d	e	f
Fixed Length Code	000	001	010	011	100	101
Frequency	51	20	2	3	9	15
Variable Length Code	0	111	1100	1101	100	101

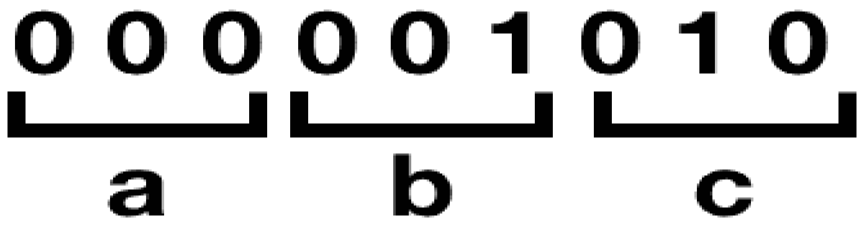
In this example, 'a' is appearing 51 out of 100 times and has the highest frequency, 'c' is appearing only 2 out of 100 times and has the least frequency. Thus, we are assigning 'a' the codeword of the shortest length i.e., 0 and 'c' a longer one i.e., 1100.

Now if we use characters of fixed length, we need 100*3 = 300 bits (each character is taking 3 bit) to represent 100 characters of the file. But to represent 100 characters with the variable length character, we need 51*1 + 20*3 + 2*4 + 3*4 + 9*3 + 15*3 = 203 bits (51*1 as 'a' is appearing 51 out of 100 times and has length 1 and so on). Thus, we can save 32% of space by using the codeword for variable length in this case.

So we have not lost any information, we are just using a different way to represent each character.

Storing, Encoding and Decoding

We basically concatenate characters while storing them into a file. For example, to store 'abc', we would use 000.001.010 i.e., 000001010 using a fixed character codeword. Now for the decoding, we know that all of our characters are 3 bits long, so we would break the code for every 3 bits and we can easily get 000, 001 and 010 which can be translated back to 'abc'.



Now, we can't use any specific length which can separate our character if we are using variable length codeword and to simplify decoding the codeword back to characters, we use prefix codes.

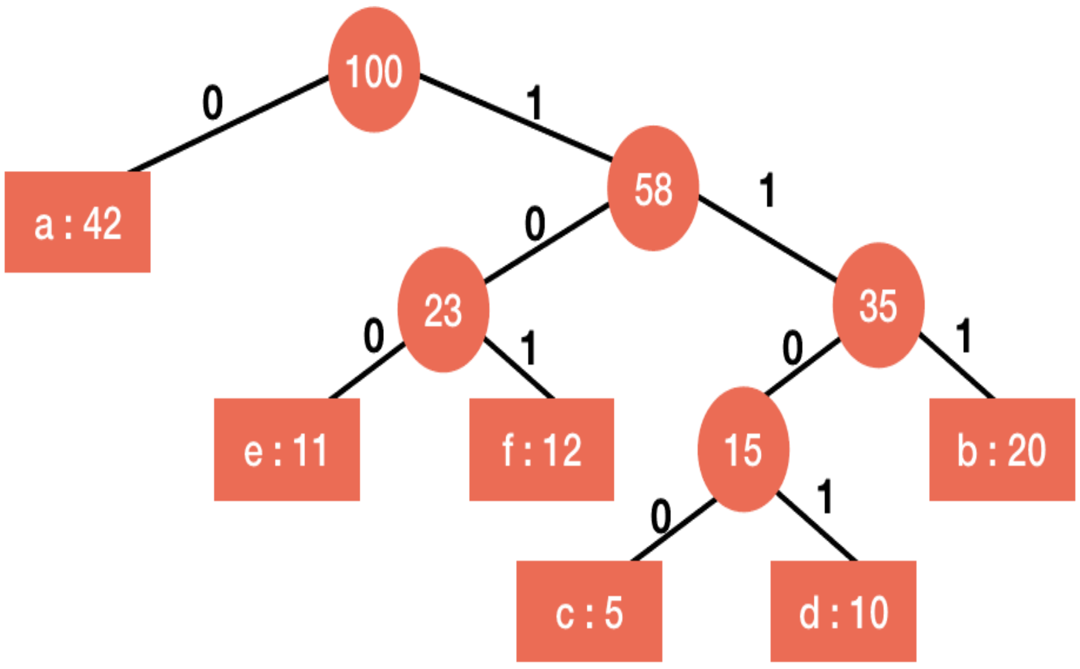
Prefix Codes

In variable length codeword, we only use such code which are not the prefix of any other character and these codes is known as **prefix codes**. For example, if we use 0, 1, 01, 10 to represent 'a', 'b', 'c' and 'd' respectively (0 is prefix of 01 and 1 is prefix of 10), then the code 00101 can be translated into 'aabab', 'acc', 'aadb', 'acab' or 'aabc'. To avoid this kind of ambiguity, prefix codes are used.

Using prefix codes made decoding unambiguous. In the above example, we have used 0, 111 and 1100 for 'a', 'b' and 'c' respectively. None of the code is the prefix of any other codes and thus any combination of these codes will decode into unique value. For example, if we write 01111100, it will uniquely decode into 'abc' only. Give it a try and try to decode it into something else.

Implementing Huffman Code

Let's first look at the binary tree given below.

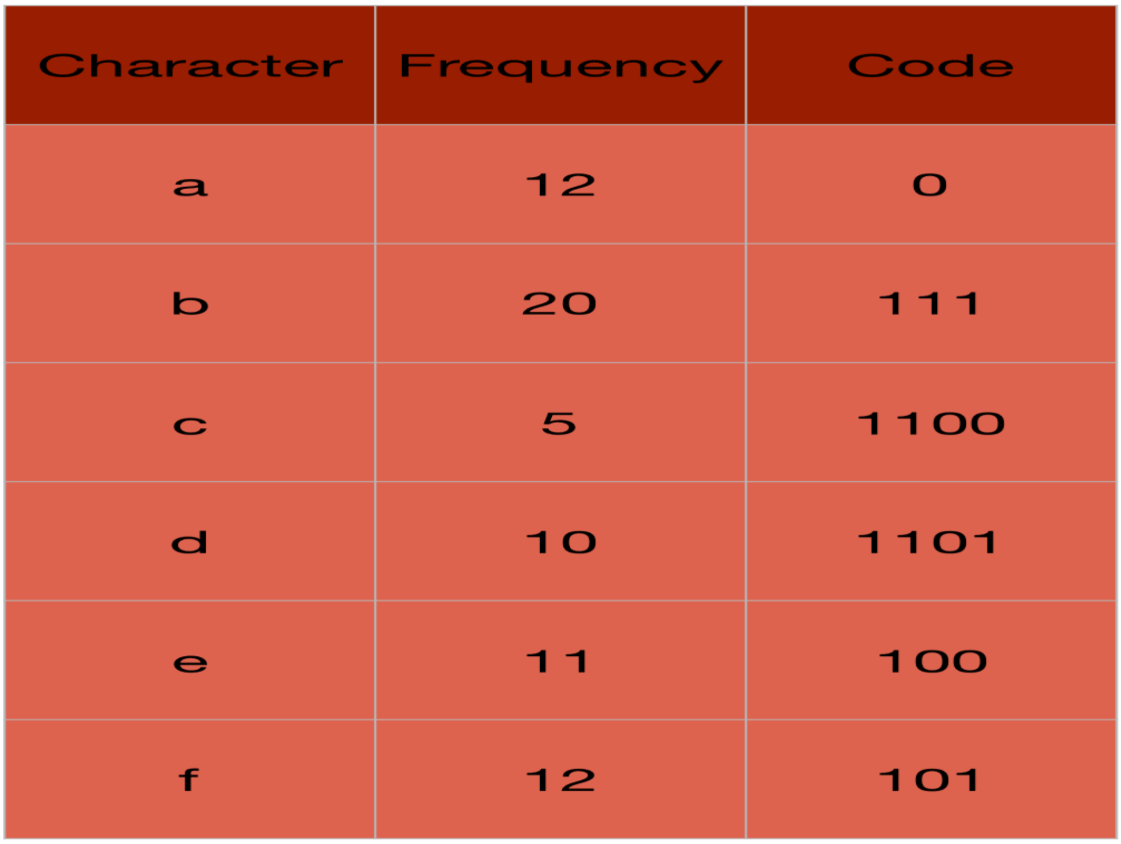


We construct this type of binary tree from the frequencies of the characters given to us.

Decoding

All the characters are on the leaves of the tree and to get the codeword for any character, we start from the root of the tree and proceed to that character. Now, if we move right from any node, we interpret that movement as 1 and if left, then 0. This is also indicated on the branch of the binary tree in the picture given above. So, we move from root to the leaf containing that character and combining the 0s and 1s of each movement, we get the codeword of the character. This is described in the picture given below.

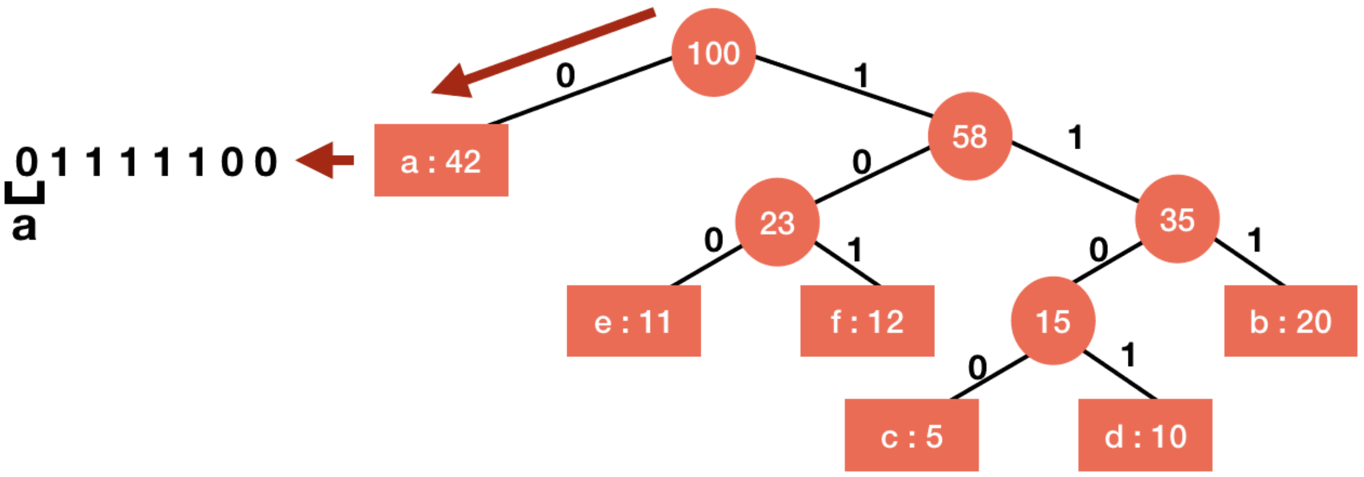
In this way, we can get the code of each character.



Character	Frequency	Code
a	12	0
b	20	111
c	5	1100
d	10	1101
e	11	100
f	12	101

We can proceed in a similar way with any code given to us to decode it. For example, let's take a case of 01111100.

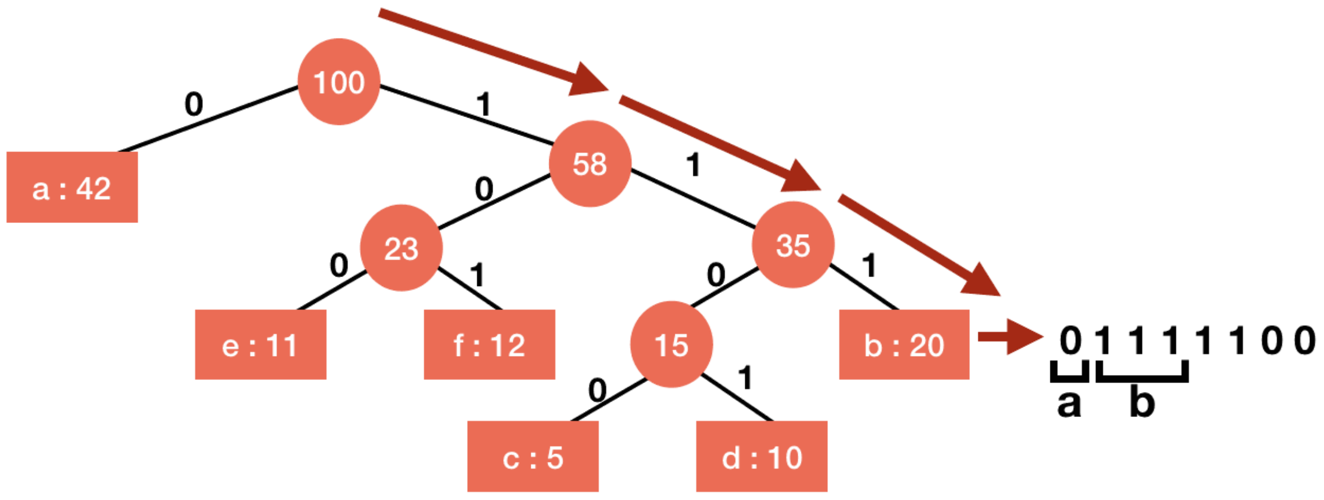
We will start from the root and since the first number is 0, so we will move left. By moving left, we encountered a character 'a' and thus the first character is a.



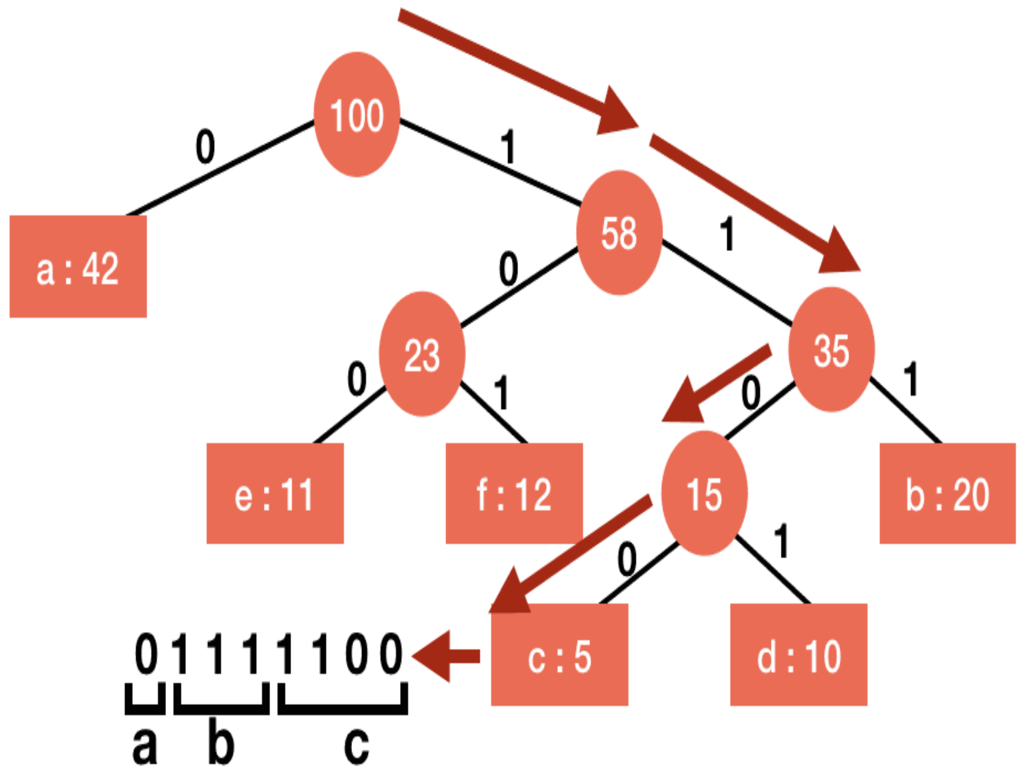
Now, we will again start from the root, we have 1 in the code, so we will move right.

Since we have not reached any of the leaves, we will continue it. The next number is also 1 and so we will again move right.

Still, we have not reached the leaf, so continuing, the next number is also 1. Moving right this time will give us the character 'b'. Thus, 'ab' is the string we have decoded till now.



Similarly, again starting from the root and moving for the next numbers will make us reach 'c'.



Thus, we have decoded 01111100 into 'abc'.

Encoding

We just have to concatenate the code of the characters to encode them.

For example, to encode 'abc', we will just concatenate 0, 111 and 1100 i.e., 01111100**.**

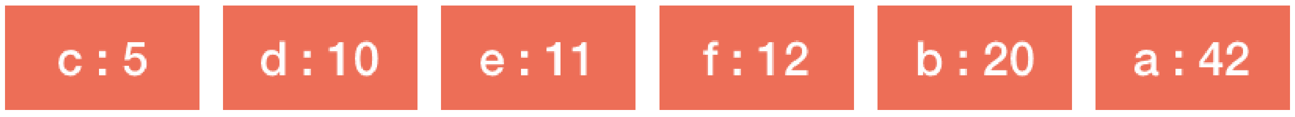
Now, our next task is to create this binary tree for the frequencies we have.

Construction of Binary Tree for Huffman Code

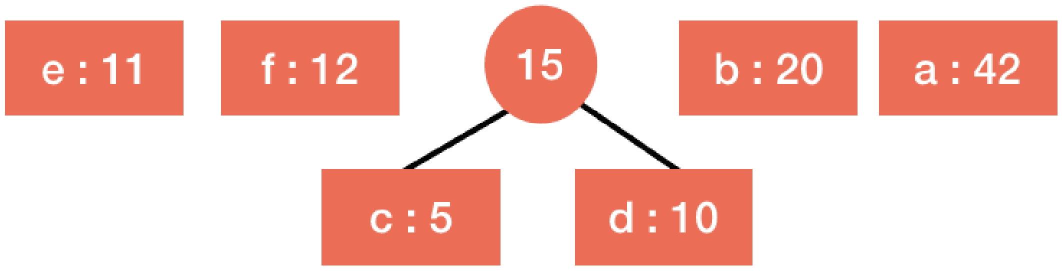
This is the part where we use the greedy strategy. Basically, we have to assign shorter code to the character with higher frequency and vice-versa. We can do this in different ways and that will result in different trees, but a full binary tree (a tree in which every node has 2 children, except the leaves) gives us the optimal code i.e., using that code will save the maximum space in storing the file.

To construct this tree for optimal prefix code, Huffman invented a greedy algorithm which we are going to use for the construction of the tree.

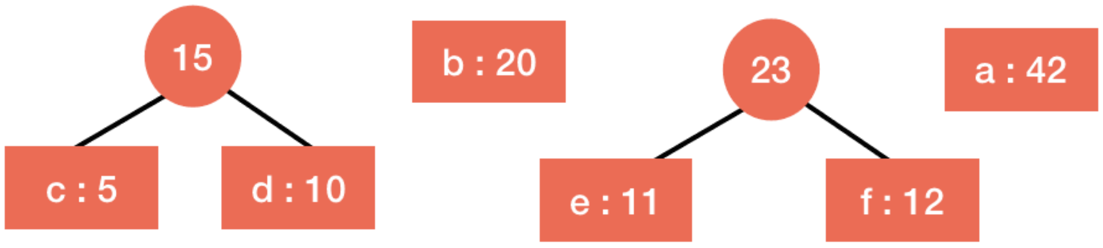
We start by sorting the characters according to their frequencies.

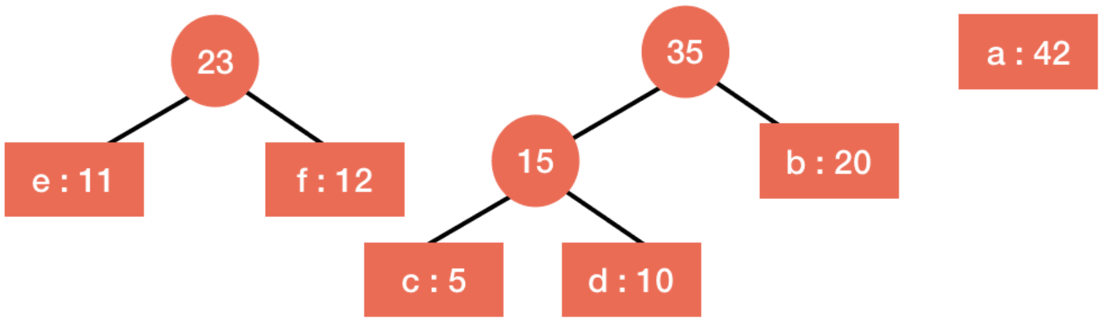


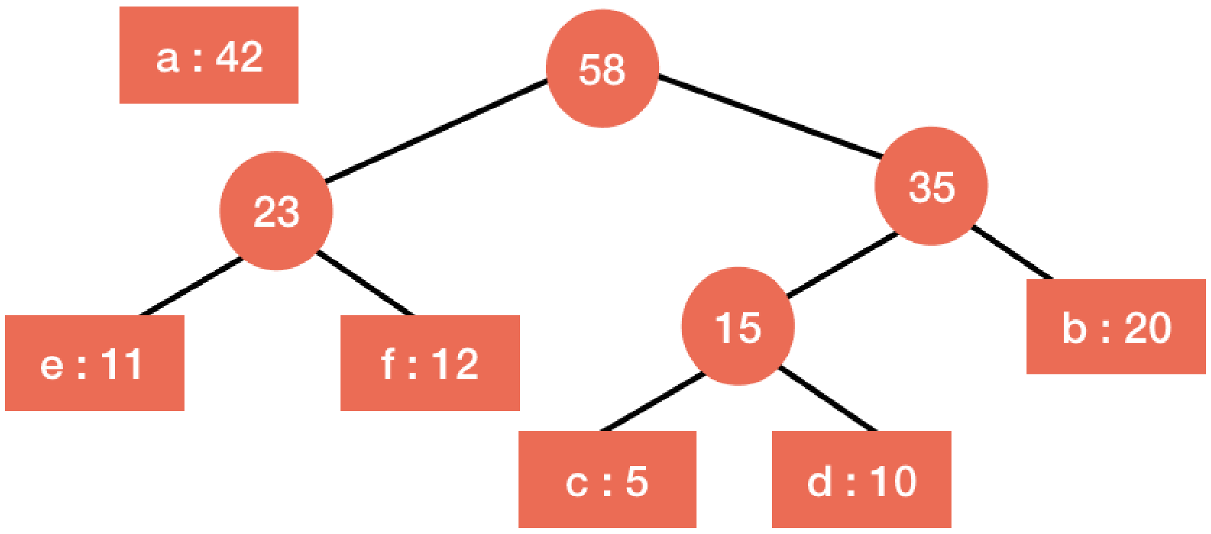
Now, we make a new node and then greedily pick the first two nodes from the sorted character and make the first node left child of the new node and second node as the right child of the new node. The value of the new node is the summation of the values of the children nodes.

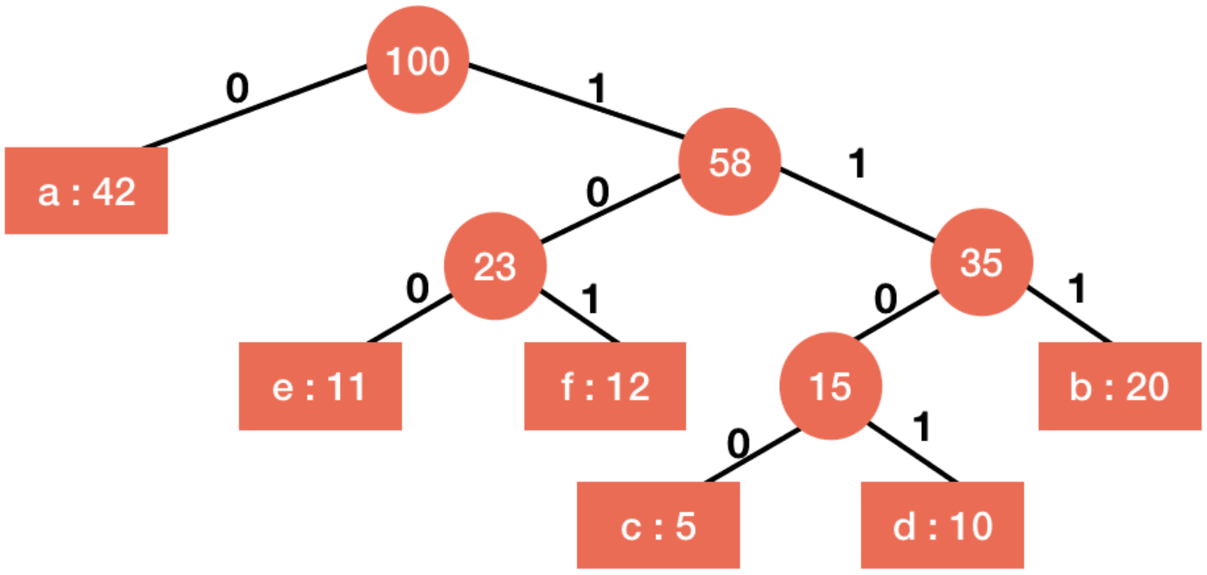


We again sort the nodes according to the values and repeat the same process with the first two nodes.





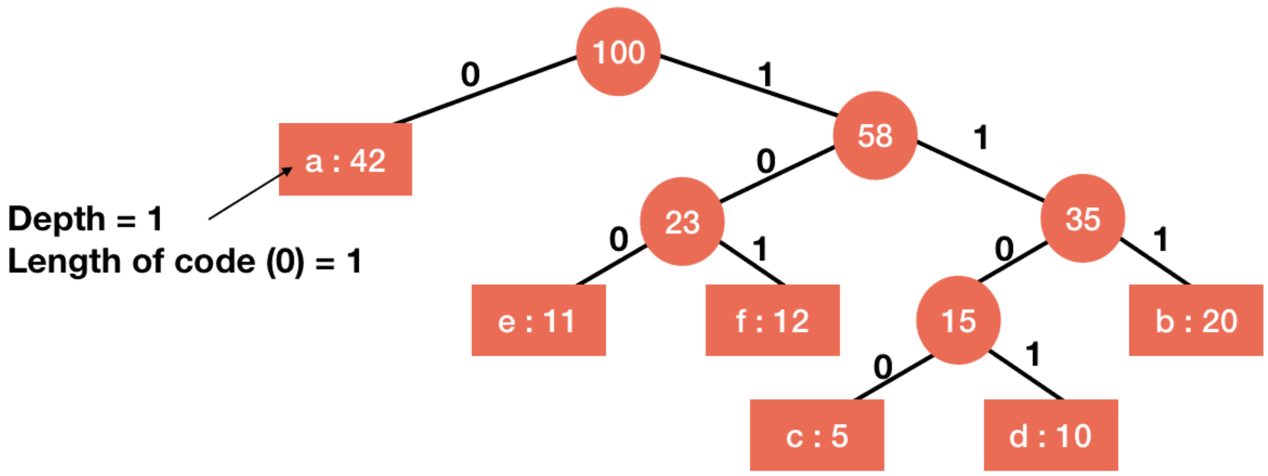




c : 5
d : 10
e : 11
f : 12
b : 20
a : 42
MIN-QUEUE

In this way, we construct the tree for optimal prefix code.

The depth of a leaf is also the length of the prefix code of the character. So, the depth $d_i$di of leaf i is the length of the prefix code of the character at leaf i. If we multiply it by its frequency i.e., $d_i * i.freq$di∗i.freq,
then this is the number of bits used by this character in the entire file.



We can sum all these for each character to get the total number of bits used to store the file.Total bits=∑idi∗i.freqTotal bits=∑idi∗i.freq

Analysis of Huffman Code

By using the min-heap to implement the queue, we can perform each operation of the queue in O(logn) time. Also, the initialization of the queue is going to take O(n) time i.e., time for building a heap.
Now, we know that each operation is taking O(logn) time. Thus, the total running time of the algorithm will be O(nlogn)

Conclusion:

Thus we have studied and implemented Huffman Encoding and also Greedy strategy.

Assignment No: 3

Title: Write a program to solve a fractional Knapsack problem using a greedy method.

Aim:
Aim of this assignment is to solve a fractional Knapsack problem using a greedy method.

Prerequisites:

- Ubuntu 14.04 (64 bit preferred),
- python

Objective:

1. To understand Greedy Method
2. To understand fractional Knapsack problem.

Theory:

The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can.

In this method, the filling of the Knapsack is done in a way that the maximum capacity of the knapsack is utilized so that maximum profit can be earned from it. The knapsack problem using the Greedy Method is referred to as:
Given a list of n objects, say {I_1, I_2,……, I_n) and a knapsack (or bag).
The capacity of the knapsack is M.
Each object I_j has a weight w_j and a profit of p_j
If a fraction x_j (where $x \in \{0...., 1)$) of an object I_j is placed into a knapsack, then a profit of $p_j x_j$ is earned.
The problem (or Objective) is to fill the knapsack (up to its maximum capacity M), maximizing the total profit earned.
Mathematically:

$$\text{Maximize (the profit)} = \sum_{j=1}^{n} p_j x_j$$

$$= \sum_{j=1}^{n} w_j x_j \le M \text{ and } x_j \in \{0,....,1\}, 1 \le j \le n$$

the value of x_j will be any value between 0 and 1 (inclusive). If any object I_j is completely placed into a knapsack, its value is 1 ($x_j = 1$). If we do not pick (or select) that object to fill into a knapsack, its value is 0 ($x_j = 0$). Otherwise, if we take a fraction of any object, then its value will be any value between 0 and 1.

Algorithm for Knapsack Problem Using Greedy Method

A pseudo-code for solving knapsack problems using the greedy method is;
greedy fractional-knapsack (P[1...n], W[1...n], X[1..n]. M)
/*P[1...n] and W[1...n] contains the profit and weight of the n-objects ordered such that X[1...n] is a solution set and M is the capacity of knapsack*/

```
{
    For j ← 1 to n do
        X[j]← 0
    profit ← 0 // Total profit of item filled in the knapsack
    weight ← 0 // Total weight of items packed in knapsacks
    j ← 1
    While (Weight < M) // M is the knapsack capacit
{
        if (weight + W[j] =< M)
            X[j] = 1
            weight = weight + W[j]
        else{
            X[j] = (M - weight)/w[j]
            weight = M
        }
        Profit = profit + p[j] * X[j]
        j++;
    } // end of while
} // end of Algorithm
```

- Sort the given array of items according to **weight / value(W /V)** ratio in descending order.
- Start adding the item with the maximum **W / V** ratio.
- Add the whole item, if the current weight is less than the capacity, else, add a portion of the item to the knapsack.
- Stop, when all the items have been considered and the total weight becomes equal to the weight of the given knapsack.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are **n** items in the store
- Weight of **i**th item $w_i>0$wi>0
- Profit for **i**th item $p_i>0$pi>0 and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of **i**th item.

$$0\leqslant x_i\leqslant 1 0\leqslant xi\leqslant 1$$

The **i**th item contributes the weight $x_i.w_i$xi.wi to the total weight in the knapsack and profit $x_i.p_i$xi.pi to the total profit.

Hence, the objective of this algorithm is to

$$maximize\sum_{n=1}^{n}(x_i.pi)$$

subject to constraint,

$$\sum_{n=1}^{n}(x_i.wi)\leqslant W\sum n=1n(xi.wi)\leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{n=1}^{n}(x_i.wi)=W$$

In this context, first we need to sort those items according to the value of $\frac{p_i}{w_i}$piwi, so that $\frac{p_{i+1}}{w_{i+1}}$pi+1wi+1 ≤ $\frac{p_i}{w_i}$piwi . Here, ***x*** is an array to store the fraction of items.

Algorithm: Greedy-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
```

Analysis

If the provided items are already sorted into a decreasing order of $\frac{p_i}{w_i}$piwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Example

Let us consider that the capacity of the knapsack ***W*** = **60** and the list of provided items are shown in the following table −

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio $(p_i w_i)$(piwi)	7	10	6	5

As the provided items are not sorted based on $p_i w_i$piwi. After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio $(p_i w_i)$(piwi)	10	7	6	5

Solution

After sorting all the items according to $p_i w_i$piwi. First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Conclusion:

Hence we implemented Fractional Knapsack problem using Greedy method.

Assignment No: A4

Title:
Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

Aim:
Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy

Prerequisites:

- Ubuntu 14.04 (64 bit preferred)

Objective:
To understand Branch and Bound Strategy.
To understand Dynamic Programming.
To understand 0-1 Knapsack problem.

Objective:

Problem Statement

We are a given a set of *n* objects which have each have a value v_i and a weight w_i. The objective of the 0/1 Knapsack problem is to find a subset of objects such that the total value is maximized, and the sum of weights of the objects does not exceed a given threshold *W*. An important condition here is that one can either take the entire object or leave it. It is not possible to take a fraction of the object

The first idea that comes to mind as soon as we look at the problem would be to look at all possible combinations of objects, calculate their total weight, and if the total weight is less than the threshold, to calculate the total value .This approach is known as the *Brute Force*. Although this approach would give us the solution, it is of exponential time complexity. Hence, we look at the other possible methods.
We can use the *Dynamic Programming* approach to solve this problem as well. Although this method is far more efficient than the Brute Force method, it does not work in scenarios where the item weights are non-integer values.

Branch and Bound Method

The goal of a branch-and-bound algorithm is to find a value x that maximizes or minimizes the value of a real-valued function $f(x)$, called an objective function, among some set S of admissible, or candidate solutions. The set S is called the search space, or feasible region. The rest of this section assumes that minimization of $f(x)$ is desired; this assumption comes without loss of generality, since one can find the maximum value of $f(x)$ by finding the minimum of $g(x) = -f(x)$. A Branch and Bound (B&B) algorithm operates according to two principles:

- It recursively splits the search space into smaller spaces, then minimizing $f(x)$ on these smaller spaces; the splitting is called *branching*.

- Branching alone would amount to brute-force enumeration of candidate solutions and testing them all. To improve on the performance of brute-force search, a B&B algorithm keeps track of *bounds* on the minimum that it is trying to find, and uses these bounds to "prune" the search space, eliminating candidate solutions that it can prove will not contain an optimal solution.

Many problems include several possible scenarios, and we need to find the optimal one. The branch and bound algorithm comes in handy where combinatorial optimization is required. Typically, these problems require all possible permutations in the worst-case scenario. Branch and bound algorithm create branches and bound to the best solution

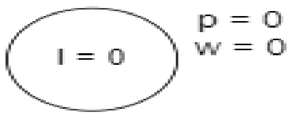
Let's see the solution to the knapsack problem using the branch and bound approach. Here we have five items, and the profit and weight for each item are mentioned in the table.

EG The knapsack has a capacity of a total weight 15:

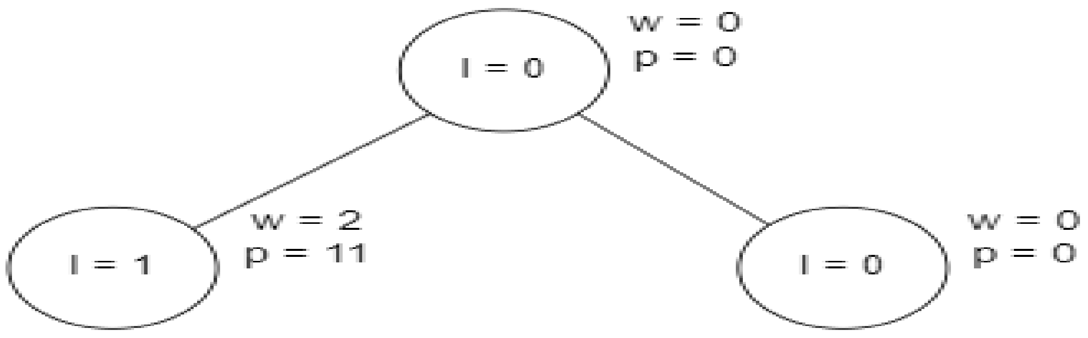
Items	Profit	Weight
0	0	0
1	11	2
2	21	5
3	31	13
4	33	10
5	43	33

Our goal is to maximize the profit while the total weight does not exceed 15.

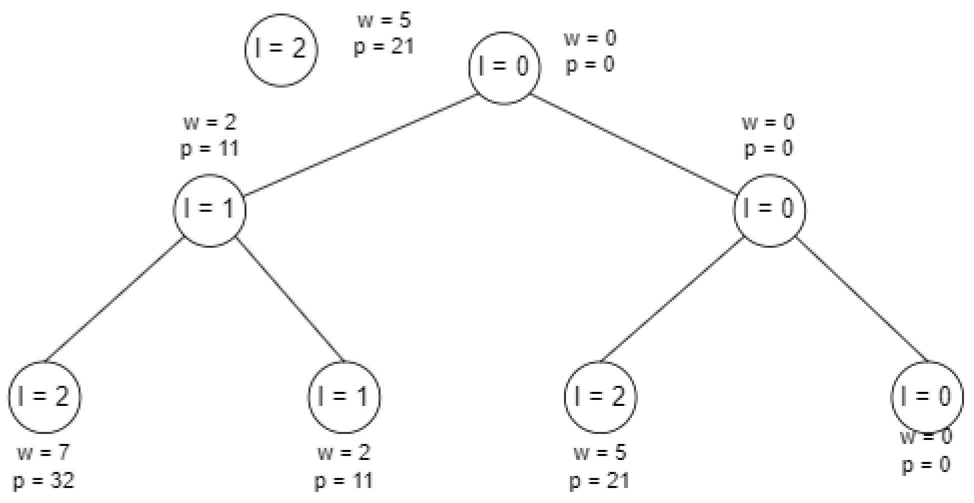
STEP 1



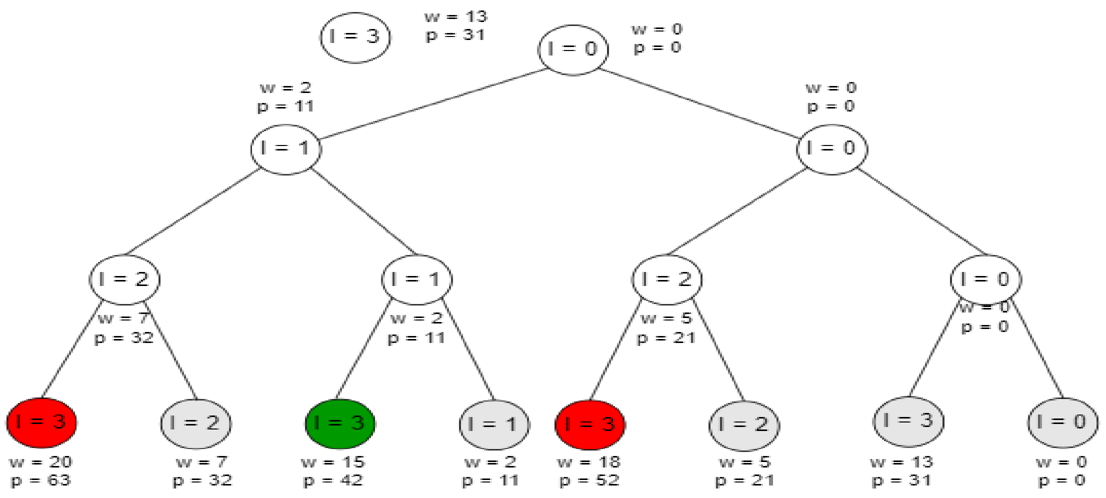
Step 2



STEP 3



STEP 4



Now let's add item 3 and see the updated values.

PROGRAM

```
public class Solution {

   int getMaximumKnapsackValue(int w, int weight[], int values[], int n, int[][] arr)
   {
 // Base Case
```

```
        if (n == 0)
        {
           return 0;
        }

        // Base Case
        if (w == 0)
        {
           return 0;
        }

        if (arr[n][w] != -1)
        {
           return arr[n][w];
        }

        // If weight is more than current knapsack weight then we have to exclude this item
        if (w < weight[n - 1])
        {
         int maxValue = getMaximumKnapsackValue(w, weight, values, n - 1,arr);
         arr[n][w] = maxValue;
         return maxValue;
        }
        else
        {
         // Pick Nth item and add its value in result
         int valueWithNthItem = getMaximumKnapsackValue(w - weight[n - 1], weight, values, n - 1, arr) +
values[n - 1];

         // Exclude Nth item and go for n-1 items
         int valueWithoutNthItem = getMaximumKnapsackValue(w, weight, values, n - 1, arr);

         // Get the maximum of both values
         int maxValue = Math.max(valueWithNthItem, valueWithoutNthItem);

         arr[n][w] = maxValue;

         return maxValue;
        }
    }

    int getMaximumKnapsack(int w, int weight[], int values[], int n)
    {
      // Initialize 2D array of size N + 1 and w + 1 with value -1
        int arr[][] = new int[n + 1][w + 1];

        for(int i = 0; i < n + 1; i++)
        {
         for(int j = 0; j < w + 1; j++)
         {
         arr[i][j] = -1;
         }
        }
        return getMaximumKnapsackValue(w, weight, values, n, arr);
    }

public static void main(String[] args)
{
        Solution solution = new Solution();
        int val[] = new int[] { 80, 110, 135 };
        int wt[] = new int[] { 15, 20, 35 };
        int W = 60;
        int n = val.length;

        int maxValue = solution.getMaximumKnapsack(W, wt, val, n);

        System.out.println("Maximum value that Knapsack can have is : " + maxValue);

}
}
```

```
Output :
Maximum value that Knapsack can have is : 245
```

Algorithm Complexity:

Time Complexity: O(N * W*)*

There will be 'N * W' calls in a recursive function as we are using a 2D array, and if the value is present in this array, then we are not making duplicate calls, so there will be utmost N * W combinations. Therefore the overall time complexity is O(N * W).

Space Complexity: O(N * W)

As we are using a 2D array of size 'N * W'. Therefore the overall space complexity is O(N * W)

Dynamic Programming

Dynamic programming amounts to breaking down an optimization problem into simpler sub-problems, and storing the solution to each sub-problem so that each sub-problem is only solved once.
DP is a useful technique for optimization problems, those problems that seek the maximum or minimum solution given certain constraints, because it looks through all possible sub-problems and never recomputes the solution to any sub-problem. This guarantees correctness and efficiency, which we cannot say of most techniques used to solve or approximate algorithms. This alone makes DP special.

Sub-problems are smaller versions of the original problem**.**

Dynamic Programming Steps

Step 1: Identify the sub-problem

Step 2: Write out the sub-problem as a recurring mathematical decision.

Step 3: Solve the original problem using Steps 1 and 2

Step 5: Code it!

Runtime Analysis of Dynamic Programs

Generally, a dynamic program's runtime is composed of the following features:

- Pre-processing
- How many times the for loop runs
- How much time it takes the recurrence to run in one for loop iteration
- Post-processing
    - Pre-processing + Loop * Recurrence + Post-processing

Recursion vs Dynamic Programming

Dynamic programming is mostly applied to recursive algorithms.Most optimization problems requires dynamic programming .

But not all problems that use recursion can use Dynamic Programming. Unless there is a presence of overlapping subproblems like in the fibonacci sequence problem. Recursion can only reach the solution using a divide and conquer approach.

That is the reason why a recursive algorithm like Merge Sort cannot use Dynamic Programming, because the subproblems are not overlapping in any way.

Greedy Algorithms vs Dynamic Programming

Greedy Algorithms are similar to dynamic programming in the sense that they are both tools for optimization.
However, greedy algorithms look for locally optimum solutions or in other words, a greedy choice, hopes in finding a global optimum. Hence greedy algorithms can make a guess that looks optimum at the time but becomes costly down the line and do not guarantee a globally optimum.

Dynamic programming, on the other hand, finds the optimal solution to subproblems and then makes an informed choice to combine the results of those subproblems to find the most optimum solution.

Conclusion:
We have implemented a 0-1 Knapsack problem using branch and bound strategy.

Assignment No: A5

Title:
Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen‘s matrix.

Aim:

Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen‘s matrix.

Prerequisites:

Objective:-

1.To understand concept of Backtracking .
2. Understand the 8-queen matrix
3. To generate 8-queen matrix using backtracking

Theory:

The 8-queens puzzle is the problem of placing eight chess queens on an 8X8 chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The 8- queens puzzle is an example of the more general n-queens problem of placing n queens on an nXn chessboard, where solutions exist for all natural numbers n with the exception of n=2 and n=3.

The 8-queens puzzle has 92 distinct solutions. If solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 fundamental solutions.

Description:

Backtracking:

Backtracking is a general algorithm for finding solutions to some computational problems, notably constraint satisfaction problems that incrementally builds candidates to the solutions, and abandons each partial candidate c (backtracks) as soon as it determines that c cannot possibly be completed to a valid solution. One example of backtracking is the eight queens Puzzle that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of k queens in the first k rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned, since it cannot possibly be completed to a valid solution. Backtracking can be applied only for problems which admit the concept of a partial candidate solution and a relatively quick test of whether it can possibly be completed to a valid solution.
Backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate a large number of candidates with a single test.

Solving 8 Queen Problem by backtracking:

The 8 queen problem is a case of more general set of problems namely n queen problem. The basic idea here is How to place n queen on n by n board, such that they don't attack each other. The complexity of solving the problem increases with n.
For example: Q1 attacks some positions, therefore Q2 has to comply with these constraints and take place, not directly attacked by Q1. Placing Q3 is harder, since we have to satisfy constraints of Q1 and Q2. Going the same way we may reach point, where the constraints make the placement of the next queen impossible. Therefore we need to relax the constraints and find new solution. To do this we are going backwards and then finding new admissible solution.

Advantages of Backtracking:

- It is a step-by-step representation of a solution to a given problem ,which is very easy to understand
- It has got a definite procedure.
- It easy to first develop an algorithm, &then convert it into a flowchart &then into a computer program.
- It is independent of programming language.
- It is easy to debug as every step is got its own logical sequence.

Disadvantages of Backtracking:

- It is time consuming.

PROGRAM SNIPPETS

Input and Output

Input:

The size of a chess board. Generally, it is 8. as (8 x 8 is the size of a normal chess board.)

Output:

The matrix that represents in which row and column the N Queens can be placed.

If the solution does not exist, it will return false.

In this output, the value 1 indicates the correct place for the queens.

The 0 denotes the blank spaces on the chess board.

Algorithm

isValid(board, row, col)

Input: The chess board, row and the column of the board.

Output : True when placing a queen in row and place position is a valid or not.

```
Begin
   if there is a queen at the left of current col, then
      return false
   if there is a queen at the left upper diagonal, then
      return false
   if there is a queen at the left lower diagonal, then
      return false;
   return true //otherwise it is valid place
End
```

solveNQueen(board, col)

Input – The chess board, the col where the queen is trying to be placed.

Output – The position matrix where queens are placed.

```
Begin
   if all columns are filled, then
      return true
   for each row of the board, do
      if isValid(board, i, col), then
         set queen at place (i, col) in the board
         if solveNQueen(board, col+1) = true, then
            return true
         otherwise remove queen from place (i, col) from board.
   done
   return false
End
```

```
#include<iostream>
using namespace std;
#define N 8

void printBoard(int board[N][N]) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++)
         cout << board[i][j] << " ";
      cout << endl;
   }
}

bool isValid(int board[N][N], int row, int col) {
   for (int i = 0; i < col; i++)    //check whether there is queen in the left or not
      if (board[row][i])
         return false;
   for (int i=row, j=col; i>=0 && j>=0; i--, j—
if (board[i][j])      //check whether there is queen in the left upper diagonal or not
         return false;
   for (int i=row, j=col; j>=0 && i<N; i++, j--)
      if (board[i][j])      //check whether there is queen in the left lower diagonal or not
         return false;
   return true;
}
```

```
bool solveNQueen(int board[N][N], int col) {
  if (col >= N)          //when N queens are placed successfully
    return true;
  for (int i = 0; i < N; i++) {     //for each row, check placing of queen is possible or not
    if (isValid(board, i, col) ) {
      board[i][col] = 1;      //if validate, place the queen at place (i, col)
      if ( solveNQueen(board, col + 1))    //Go for the other columns recursively
        return true;

      board[i][col] = 0;        //When no place is vacant remove that queen
    }
  }
  return false;       //when no possible order is found
}

bool checkSolution() {
  int board[N][N];
  for(int i = 0; i<N; i++)
    for(int j = 0; j<N; j++)
      board[i][j] = 0;      //set all elements to 0

  if ( solveNQueen(board, 0) == false ) {     //starting from 0th column
    cout << "Solution does not exist";
    return false;
  }
  printBoard(board);
  return true;
}

int main() {
  checkSolution();
}
```

OUTPUT

```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
```

```
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

Algorithm Explaination:

1. Place the queens column wise, start from the left most column.
2. If all queens are placed.
    1. return true and print the solution matrix.
3. Else
    1. Try all the rows in the current column.
    2. Check if queen can be placed here safely if yes mark the current cell in solution matrix as 1 and try to solve the rest of the problem recursively.
    3. If placing the queen in above step leads to the solution return true.
    4. If placing the queen in above step does not lead to the solution ,BACKTRACK, mark the current cell in solution matrix as 0 and return false.
4. If all the rows are tried and nothing worked, return false and print NOSOLUTION.

Complexity OF N-Queen

It has time complexity: O(n^n), As NQueen function is recursively calling.

Conclusion

We have Designed n-Queens matrix Using backtracking.

.