Week 4 -Lecture 7

In this lecture we learned about the basic Greedy algorithms , how to solve Activity Selection problem, Huffman coding etc So we discussed the following points:

- How to think about Greedy Algorithms and their solutions.
- What is Activity Selection problem and how to solve it greedily.
- What are Huffman codes and how to solve it Greedily.

Greedy Algorithms:

These are the algorithms where we solve the problem by making the locally optimal choice at each stage and then prove that doing so (solving sub-problems) actually gives the best solution for the original

problems.

Basic Greedy Design:

The greedy design for such problems usually consists of two steps:

- Greedy choice Property - First we select some element or property that is guaranteed to be present in the optimal solution for the problem.
- Optimal Substructure Property : Here since we have already chosen some element or property , we decide if we can state the rest of the problem as a smaller version of the original problem.

The proof is usually done using Induction.

There are many problems which can be solved using Greedy approach like the Minimum Spanning Tree problem, Activity Selection Problem , Huffman coding problem , etc.

We discussed MST in last class , so let us look at Activity Selection Problem which were discussed in today's class .

Activity Selection Problem:

In this problem we are given an array(n elements) of pairs consisting of start time and finish time in random order.

We are required to select maximum number of activities that can be say performed by a single person , that is all chosen activities time frame should be disjoint and there shouldn't be any moment in time where a person is performing more than one job.

Greedy Choice:

Using the basic Greedy design that we discussed above, let us think how to build a greedy solution for the problem.

- **Greedy Choice** : The first observation we can make out is that the activity with the minimum finish time will be a part of the optimal solution.

Proof:

Say the activity with minimum finish time is a_0 and the current optimal solution is S . So if a_0 is present in S then we don't need to prove anything .

If a_0 is not part of S , then consider a_k in S which has the minimum finish time.

BY observation we can realise that we can replace a_k by a_0 since a_0 is the global minimum and a_k is only the local minimum.

So since finish time of a_k has to be greater than equal to that of a_0 , we can replace it and see that it does not affect the number of elements in our optimal solution.

- **Optimal substructure** : Given that we have chosen one activity in our solution , we can remove all activities from the set which are non compatible with a_0 , that is all those activities whose time frame are not disjoint with a_0 .

  And from the remaining elements , we can just repeat same algorithm by picking element with minimum finish time and then removing in-compatible elements and we can continue to do this until the set is non-empty.

  This will guarantee us the optimal solution

```
sort(according to finish time)
Set i = 0;    //pointing at first element
for j = 1 to n-1 do
```

```
if start time of j >= finish time of i then
  Print j
  Set i = j
```

So Time complexity would be n log n because of sorting.

Huffman Coding:

Huffman coding is used for solving the problem of finding the most economical way to write a given long string in binary.

Problem Statement:

Given few characters and their respective frequency in an input string , we need to find the minimum number of bits to represent the the string .

Naive method:

Say we have four letters A ,B,C and D and their are n characters in the input string with frequency of the characters as f_A,f_B,f_c and f_D.

The naive way to think would be to have four representations :

- A - 00
- B - 01
- C - 10
- D - 11

So it would take us 2*n bits to represent the given string in binary form .

But it turns out we can perform much better using Huffman coding .let us see how

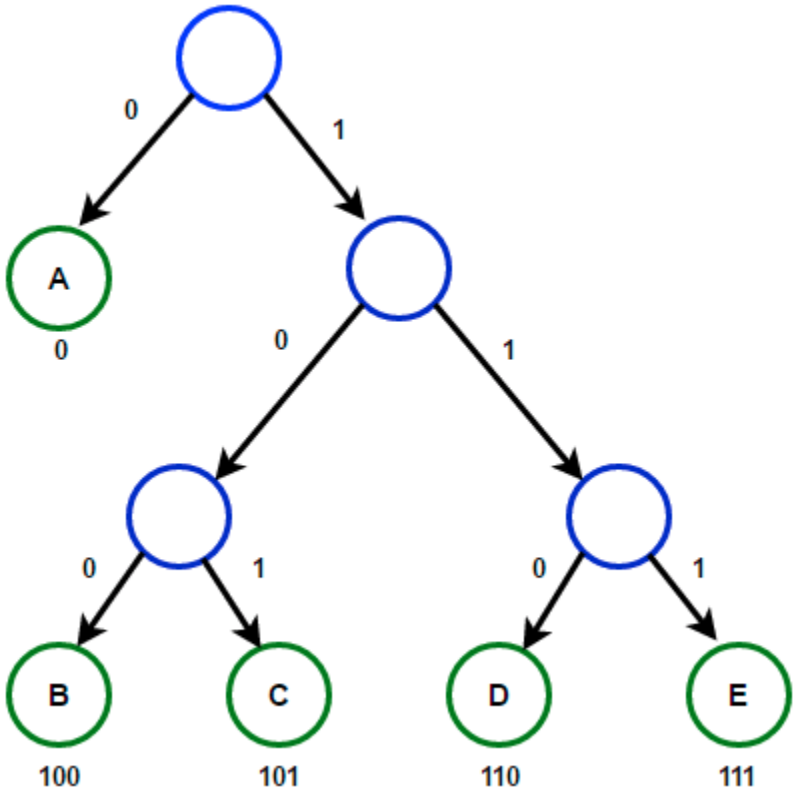
Huffman coding:

In Huffman coding , we assign variable length codes for the input characters and the length of the assigned codes are based on the frequency of the corresponding characters.

But we can't just randomly assign the codes , we will need to make sure that **no codeword is a prefix of another codeword.**

This is because we would also need to get back the original string from the binary representation.

So our codes would look like a tree called the Huffman tree whose leaf nodes would be the code-words and the tree would be complete binary tree (since if it is not , we can remove the node with one child and that will ensure a better answer).



So formally we are required to find the cost of a tree which is defined as :

Cost of tree = summation from 1 to n of (f[i] * depth[i])

Let us see how to solve this problem greedily :

Greedy Choice:

The greedy choice in this problem would be to first select the two symbols with smallest frequency as the bottom of the tree.

This is because their depth would be the maximum and so they will take most no of bits for their code-words , so it is optimal to select them in a manner so as to reduce the cost.

Hence we take the symbols with minimum frequency as the bottom of our tree.

Optimal Substructure:

Once we have taken the two leaf with the minimum frequency , we can remove both of them ,add their frequencies , and add this new node to our tree. Now we can repeat the same process again.

So the substructure here is that the cost of a tree with frequency of n leaves being f_0,f_1, .. etc is equal to cost of tree with n-1 leaves being (f_0+f_1) , f_2 ..etc .

So cost of tree would be sum of all frequencies of leaves and internal nodes except the root.

Algorithm:

- Create a min_heap and insert the frequencies of the n leaf nodes.
- Get the two nodes with the min frequency from min_heap and create a new internal node with frequency equal to sum of frequencies of extracted nodes and add it o min_heap.
- The first extracted node if left child and other is right child .
- Repeat 2 , 3 until the heap contains only one node.

```
Procedure Huffman(C):
for i = 1 to n
    n = node(C[i])
    Q.push(n)
end for
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)
end while
Return Q
```

$$Timecomplexity = nlogn$$

The nlogn complexity is because we will iterate through all n nodes and insertion and deletion from min_heap is log n .

So in total nlog n.

Entropy:

In information theory , Entropy is defined as the number of bits on average required to represent some information or data.

So more the entropy , more will be the randomness of the system .

So given a random variable X with possible outcomes x1,x2 .... which occur with probability p(x1),p(x2) .. etc.

The entropy of X is defined as

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Here log pi denotes the depth of that random variable in the binary tree.

Hence the concept of entropy allows us to understand how many questions or bits we would need to ask on average to extract any information .