Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Week 11

❖ Greedy Algorithm

A **greedy algorithm** is a simple algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's Algorithm, which is used to find the shortest path through a graph.

Structure of a Greedy Algorithm

Greedy algorithms take all of the data in a particular problem, and then set a rule for which elements to add to the solution at each step of the algorithm.

Greedy choice property

A global (overall) optimal solution can be reached by choosing the optimal choice at each step.

Optimal substructure

A problem has an optimal substructure if an optimal solution to the entire problem contains the optimal solutions to the sub-problems.

In other words, greedy algorithms work on problems for which it is true that at every step, there is a choice that is optimal for the problem up to that step, and after the last step, the algorithm produces the optimal solution of the complete problem.

Greedy Algorithm

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
    - You take the best you can get right now, without regard for future consequences
    - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

❖ Huffman Coding

Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Overview:

Huffman Coding is a popular algorithm used for lossless data compression. It's named after its inventor, David A. Huffman, who introduced it in 1952. This algorithm is particularly effective because it assigns shorter codes to more frequent characters and longer codes to less frequent characters.

Key Concepts

Lossless Compression: Huffman Coding is a type of lossless compression, meaning no data is lost during the process. This makes it ideal for text and other data where exact reconstruction is critical.

Frequency Analysis: The algorithm uses the frequency of each character in the input data to build a binary tree. Characters that appear more frequently are given shorter binary codes.

Binary Tree: Huffman Coding constructs a binary tree, also known as a Huffman Tree, based on the frequency of characters. Each leaf node represents a character, and the path from the root to the leaf gives the character's code.

How its Works..?

- Calculate Frequency:
  i. First we calculate the frequency (count) of each character in the data.
  ii. Create a list of nodes, where each node represents a character and its frequency.
- Build the Priority Queue:
  i. Insert all nodes into a priority queue (min-heap), which is sorted by frequency.
  ii.Nodes with lower frequencies have higher priority.
- Construct the Huffman Tree:
  While there is more than one node in the priority queue:
  i. Remove the two nodes with the lowest frequency.
  ii. Create a new node with these two nodes as children and the sum of their frequencies as the new node's frequency.
  iii. Insert the new node back into the priority queue.

Time Complexity

O(nlogn)

Example:

We have a Message which has 100 Characters.
And in this message we have:

a=50 ; b=10
c=30 ; d=5
e=3 ; f=2

Solution:

Message = 100 characters
Ffrequencies of char that are present in message also given in question…

1st of all if we use ASCII to send the message

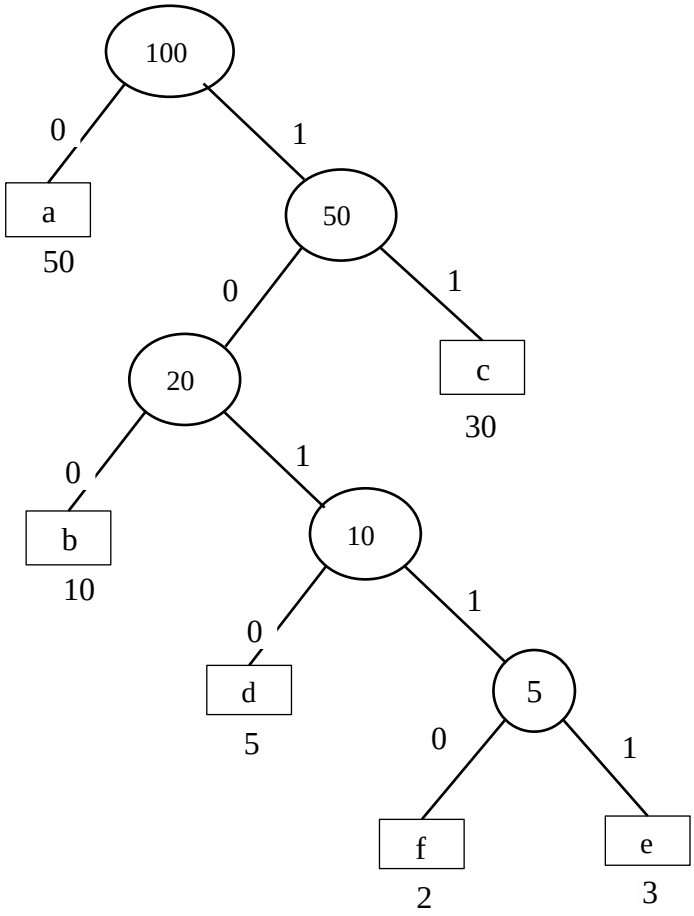
Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

For character ACSII (0-127) -->(7-bits)
Message = 100 (character) * 7bits
Total cost = 700bits

2nd if see we have 6char in message
And 6 character

a-000
b-001
c-010
d-011
e-100
f-101

Now
We use “Huffman Coding Technique”
First of all, we make huffman tree



Now we make a table for code of each character
As we see
There are 3bits are use for 6 char
So

3 * 100 = 300bits

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Data	Frequency	Variable bits code	Total Bits
a	50	0	50
b	10	100	30
c	30	11	60
d	05	1010	20
e	03	10111	15
f	02	10110	10
		Total	**185**

Example

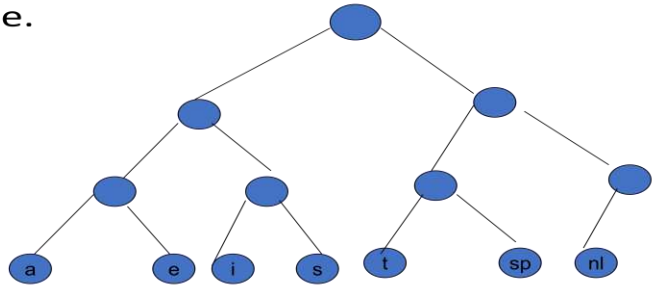
- The normal ASCII character set consists of roughly 100 "printable" characters.
- In order to distinguish these characters, $\lceil \log 100 \rceil = 7$ bits are required.
- Seven bits allow the representation of 128 characters, so
- The ASCII character set adds some other "nonprintable" characters.
- An eighth bit is added as a parity check.
- The important point, however, is that
- If the size of the character set is C, then $\lceil \log C \rceil$ bits are needed in a standard encoding.
- Suppose we have a file that contains only the characters a, e, i, s, t, plus blank spaces and newlines.
- The following table shows the frequency of each character and also the total numbers of bits required to store these characters.

Character	Code	Frequency	Total Bits
A	000	10	30
E	001	15	45
I	010	12	36
S	011	3	9
T	100	4	12
SPACE	101	13	39
NEWLINE	110	1	3

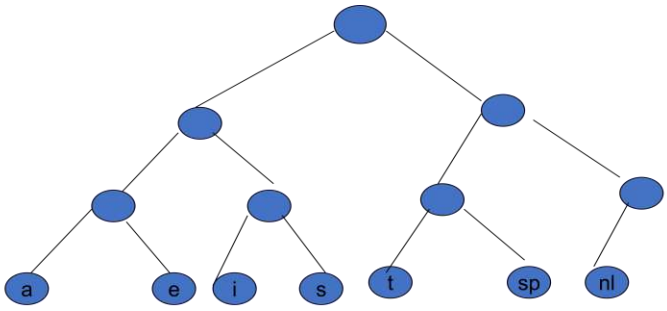
TOTAL		**174**

- Since, disk space is precious on virtually every machine, therefore the question is
- **Is it possible to provide a better code and reduce the total number of bits required?**
- The answer is, this is possible, and a simple strategy achieves 25 percent savings on typical large files and 50 to 60 percent savings on many large data files.
- The general strategy is
  - To allow the code length to vary from character to character and to ensure that the frequently occurring characters have short code.
- If all the characters occur with the same frequency, then
  - There is not likely to be any savings.

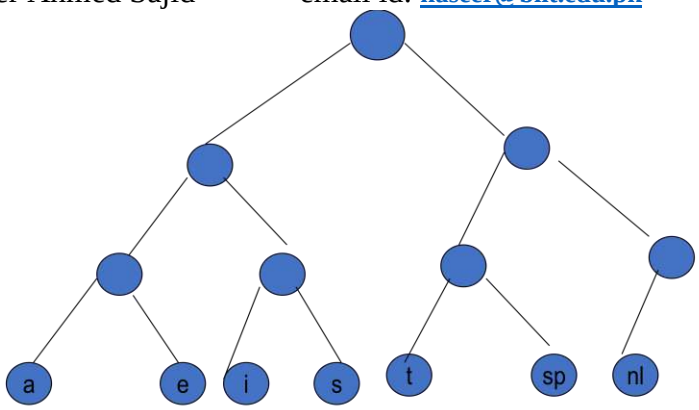
- The binary code that represents the alphabet can be represented by the binary tree.



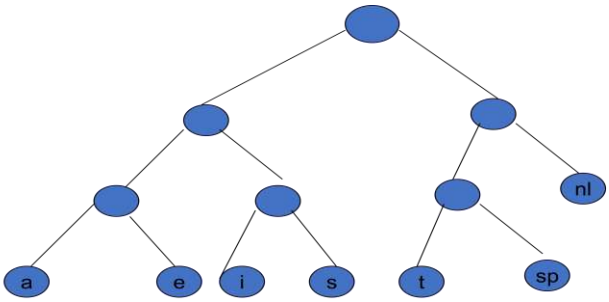
- The tree has data only at the leaves



- The representation of each character can be found by starting at the root and recording the path, using a 0 to indicate the left branch and a 1 to indicate the right branch.
- E.g. s is encoded as 011.
- If character c_i is at depth d_i and occurs f_i times, then the *cost* of the code is equal to $d_i f_i$ and if there are n characters, then $\sum d_i f_{i \text{ for all } i}$



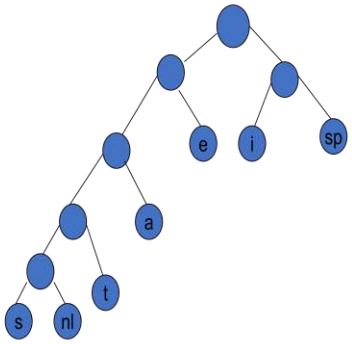
- A better tree can be obtained that by noticing that the *newline* is an only child.
- By placing the *newline* symbol one level higher, we obtain the new tree



- This new tree has a cost of 173, but is still far more optimal.
- This tree is a strictly binary tree.

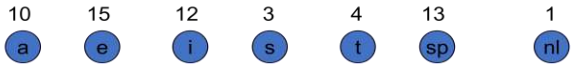
- If the characters are placed only at the leaves, any sequence of bits can always be decoded unambiguously.
- E.g. suppose the encoded string is 01001111000101100010001111.
- 0 is not a character code, 01 is not a character code but 010 represents i.
- So the fist character is i.
- Then 011 follows, giving a t.
- Then 11 follows, which is a *newline.*
- The remainder of a code is *a, space, t, i, e,* and *newline.*
- Thus, it does not matter if
- **The character codes are different lengths, as long as No character code is a prefix of another character code.**
- Such an encoding is known as **prefix code.**
- Conversely, if a character is contained in a non-leaf node, it is no longer possible to guarantee that the decoding will be unambiguous.

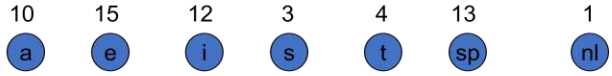
- Hence our basic problem is
- **Find the full binary tree of minimal total cost, in which all characters are contained in the leaves.**
- The following tree is an optimal tree for our sample alphabet.



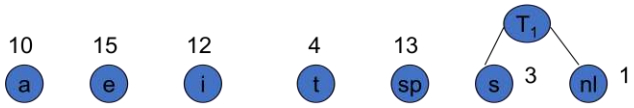
Character	Code	Frequency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
space	11	13	26
newline	00001	1	5
Total			146

- There are many optimal code that can be obtained by swapping children in the encoding tree.
- The main unresolved question is
- **How the coding tree is constructed?**
- Maintain a forest of trees with the weight of each tree.
- The weight of the tree is equal to the sum of the frequencies of its leaves.
- If there are c leaves (different characters), at the beginning there will be c single node trees.





- Select the two trees of smallest weights (T_1, T_2) and from a new tree with sub-trees T_1 and T_2.
- Assign the weight to the root of the new tree.



- Repeat this process. At the end, there will be one tree which is the optimal Huffman code tree

- It begins with a set of |c| leaves and performs a sequence of |c| - 1 merging operations to create a final tree.
- In the pseudo code, assume that C is a set of n characters and that each character c ∈ C is an object with a defined frequency f|c|.
- A priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge together.
- The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged

```
Huffman ( c )
{
1.      n = |c|    //initialize priority queue with character in c
2.      Q = c
3.      for i = 1 to n-1
4.      do      z = Allocate _ Node()
5.              x = left[z] = Extract_Min(Q)
6.              y = right[z] = Extract_Min(Q)
7.              f[z] = f[x] + f[y]
8.              Insert(Q, Z);
9.      return Extract_Min(Q)
}
```

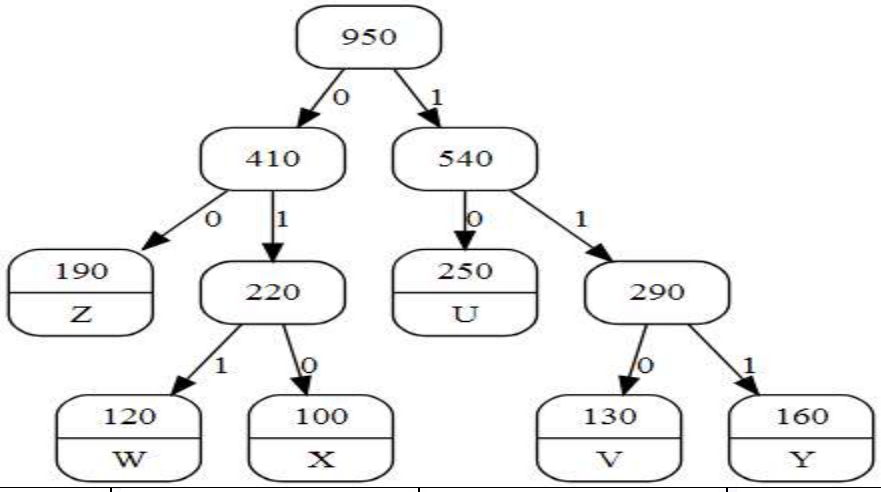
- The for loop in an algorithm repeatedly extracts the two nodes x and y of lowest frequency from the queue, and replaces them in the queue with a new node z representing their merger.

- The frequency of z is computed as the sum of the frequencies of x and y in line 7.
- After n-1 merges, the one node left in the queue. The root of the code tree is returned.

Analysis of Huffman Coding Algorithm

- Assume that queue is implemented as a binary heap.
- For a set C of n character, the initialization of Q in line 2 can be performed in O(n) time.
- The for loop is executed exactly n-1 times.
- Within the loop each heap operation requires time O(log n).
- The loop contributes to O(nlogn).
- Thus the total running time of the Huffman algorithm is O(nlogn) for a set of n characters.

Example
Data: U(250) , V(130) , W(120) , X(100) , Y(160) , Z(190)



Data	Frequency	Variable bits code	Total Bits
U	250	10	500
V	130	110	390
W	120	011	360
X	100	010	300
Y	160	111	480
Z	190	00	380
		Total	**2410**

Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010