Chapter 3: Greedy Algorithm Design Technique - II

Devesh C Jinwala,
Professor, SVNIT and Adjunct Professor, IITJammu

September 13, 2022

Algorithms & Computational Complexity @ MTech - I,
Sardar Vallabhbhai National Institute of Technology, Surat

Huffman Coding

- What is Huffman Coding ?

Huffman Coding

- What is Huffman Coding ?
- An algorithm to solve the file compression algorithm when storing files that contain characters with an aim to reduce the size of the file stored.

- What is Huffman Coding ?
- An algorithm to solve the file compression algorithm when storing files that contain characters with an aim to reduce the size of the file stored.
- Huffman coding yields 25 percent of saving on the size of typical large files, but can get as much as 50 to 60 percentage saving on many large files.

Huffman Coding

- What is Huffman Coding ?
- An algorithm to solve the file compression algorithm when storing files that contain characters with an aim to reduce the size of the file stored.
- Huffman coding yields 25 percent of saving on the size of typical large files, but can get as much as 50 to 60 percentage saving on many large files.
- What is the rational or the principle on which the Huffman Coding algorithm rests ?

Huffman Coding

- What is Huffman Coding ?
- An algorithm to solve the file compression algorithm when storing files that contain characters with an aim to reduce the size of the file stored.
- Huffman coding yields 25 percent of saving on the size of typical large files, but can get as much as 50 to 60 percentage saving on many large files.
- What is the rational or the principle on which the Huffman Coding algorithm rests ?
- Let us attempt to understand the rationale theoretically.....

Rationale of Huffman Coding, Fixed Length Codes

- Consider first Fixed length codes like ASCII.

Rationale of Huffman Coding, Fixed Length Codes

- Consider first Fixed length codes like ASCII.
  - How many *printable* characters ASCII code supports ?

Rationale of Huffman Coding, Fixed Length Codes

- Consider first Fixed length codes like ASCII.
  - How many *printable* characters ASCII code supports ?
  - How many bits are required to encode C distinct characters ?

Rationale of Huffman Coding, Fixed Length Codes

- Consider first Fixed length codes like ASCII.
  - How many *printable* characters ASCII code supports ?
  - How many bits are required to encode C distinct characters ?
  - Can we not attempt to reduce this number - because that can lead to savings of bits in storage.

Rationale of Huffman Coding, Fixed Length Codes

- Consider first Fixed length codes like ASCII.
  - How many *printable* characters ASCII code supports ?
  - How many bits are required to encode C distinct characters ?
  - Can we not attempt to reduce this number - because that can lead to savings of bits in storage.
  - Let us take an example - say we have a 1000 character string that consists of letters only from an alphabet that consists of only <a, u, x, z>.....

Rationale of Huffman Coding, Fixed Length Codes

- Consider first Fixed length codes like ASCII.
  - How many *printable* characters ASCII code supports ?
  - How many bits are required to encode C distinct characters ?
  - Can we not attempt to reduce this number - because that can lead to savings of bits in storage.
  - Let us take an example - say we have a 1000 character string that consists of letters only from an alphabet that consists of only <a, u, x, z>.....
  - How many bits are required to store a string of say 1000 characters in the ASCII ?

Rationale of Huffman Coding, Fixed Length Codes

- Consider first Fixed length codes like ASCII.
  - How many *printable* characters ASCII code supports ?
  - How many bits are required to encode C distinct characters ?
  - Can we not attempt to reduce this number - because that can lead to savings of bits in storage.
  - Let us take an example - say we have a 1000 character string that consists of letters only from an alphabet that consists of only $<$a, u, x, z$>$.....
  - How many bits are required to store a string of say 1000 characters in the ASCII ?
  - But well, the alphabet consists of only two letters - so can we not design a new encoding scheme?

Rationale of Huffman Coding, Fixed Length Codes

- Consider first Fixed length codes like ASCII.
  - How many *printable* characters ASCII code supports ?
  - How many bits are required to encode C distinct characters ?
  - Can we not attempt to reduce this number - because that can lead to savings of bits in storage.
  - Let us take an example - say we have a 1000 character string that consists of letters only from an alphabet that consists of only <a, u, x, z>.....
  - How many bits are required to store a string of say 1000 characters in the ASCII ?
  - But well, the alphabet consists of only two letters - so can we not design a new encoding scheme?
    - But, do we have a machine that can support such a scheme ?

Rationale of Huffman Coding, Fixed Length Codes

- Consider first Fixed length codes like ASCII.
  - How many *printable* characters ASCII code supports ?
  - How many bits are required to encode C distinct characters ?
  - Can we not attempt to reduce this number - because that can lead to savings of bits in storage.
  - Let us take an example - say we have a 1000 character string that consists of letters only from an alphabet that consists of only <a, u, x, z>.....
  - How many bits are required to store a string of say 1000 characters in the ASCII ?
  - But well, the alphabet consists of only two letters - so can we not design a new encoding scheme?
    - But, do we have a machine that can support such a scheme ?
    - Hence, resort to ASCII code for implementing our 2-bit code, too.

Rationale of Huffman Coding, Fixed Length Codes

- Consider first Fixed length codes like ASCII.
  - How many *printable* characters ASCII code supports ?
  - How many bits are required to encode C distinct characters ?
  - Can we not attempt to reduce this number - because that can lead to savings of bits in storage.
  - Let us take an example - say we have a 1000 character string that consists of letters only from an alphabet that consists of only <a, u, x, z>.....
  - How many bits are required to store a string of say 1000 characters in the ASCII ?
  - But well, the alphabet consists of only two letters - so can we not design a new encoding scheme?
    - But, do we have a machine that can support such a scheme ?
    - Hence, resort to ASCII code for implementing our 2-bit code, too.
    - Then, how many bits are required to encode each character in our restricted alphabet?

- Consider now a specific string viz. **aaxauxz** and compute the no of bits required in the ASCII nd in our 2-bit fixed length encoding scheme.

- Consider now a specific string viz. **aaxauxz** and compute the no of bits required in the ASCII nd in our 2-bit fixed length encoding scheme.
- What is the characteristic of this string ?

- Consider now a specific string viz. **aaxauxz** and compute the no of bits required in the ASCII nd in our 2-bit fixed length encoding scheme.
- What is the characteristic of this string ?
- Are we exploiting the information about the frequency of occurrence of the characters in this scheme ?

- Consider now a specific string viz. **aaxauxz** and compute the no of bits required in the ASCII nd in our 2-bit fixed length encoding scheme.
- What is the characteristic of this string ?
- Are we exploiting the information about the frequency of occurrence of the characters in this scheme ?
- Let us design a new encoding viz. a=0, x=10, u=110, z=111. Then how many bits are required to store the string viz. **aaxauxz** ?

- Consider now a specific string viz. **aaxauxz** and compute the no of bits required in the ASCII nd in our 2-bit fixed length encoding scheme.
- What is the characteristic of this string ?
- Are we exploiting the information about the frequency of occurrence of the characters in this scheme ?
- Let us design a new encoding viz. a=0, x=10, u=110, z=111. Then how many bits are required to store the string viz. **aaxauxz** ?
- Now lets go back to our 1000 character string - say a occurs 996 times, x occurs twice and u and z occur once each making up. 1000 characters.

- Consider now a specific string viz. **aaxauxz** and compute the no of bits required in the ASCII nd in our 2-bit fixed length encoding scheme.
- What is the characteristic of this string ?
- Are we exploiting the information about the frequency of occurrence of the characters in this scheme ?
- Let us design a new encoding viz. a=0, x=10, u=110, z=111. Then how many bits are required to store the string viz. **aaxauxz** ?
- Now lets go back to our 1000 character string - say a occurs 996 times, x occurs twice and u and z occur once each making up. 1000 characters.
- What are the bits required using ASII and that using our variable length encoding scheme as mentioned above ?

- Consider now a specific string viz. **aaxauxz** and compute the no of bits required in the ASCII nd in our 2-bit fixed length encoding scheme.
- What is the characteristic of this string ?
- Are we exploiting the information about the frequency of occurrence of the characters in this scheme ?
- Let us design a new encoding viz. a=0, x=10, u=110, z=111. Then how many bits are required to store the string viz. **aaxauxz** ?
- Now lets go back to our 1000 character string - say a occurs 996 times, x occurs twice and u and z occur once each making up. 1000 characters.
- What are the bits required using ASII and that using our variable length encoding scheme as mentioned above ?

- Consider now a specific string viz. **aaxauxz** and compute the no of bits required in the ASCII nd in our 2-bit fixed length encoding scheme.
- What is the characteristic of this string ?
- Are we exploiting the information about the frequency of occurrence of the characters in this scheme ?
- Let us design a new encoding viz. a=0, x=10, u=110, z=111. Then how many bits are required to store the string viz. **aaxauxz** ?
- Now lets go back to our 1000 character string - say a occurs 996 times, x occurs twice and u and z occur once each making up. 1000 characters.
- What are the bits required using ASII and that using our variable length encoding scheme as mentioned above ?

Obviously the variable length code desired for longer strings.

How to decode such an encoding ?

- But, how to decode a string of variable length ??

How to decode such an encoding ?

- But, how to decode a string of variable length ??
- Consider a specially encoded pattern as follows....

How to decode such an encoding ?

- But, how to decode a string of variable length ??
- Consider a specially encoded pattern as follows....
    - e.g. a=1 and b=01

How to decode such an encoding ?

- But, how to decode a string of variable length ??
- Consider a specially encoded pattern as follows....
    - e.g. a=1 and b=01
    - e.g. a=1, x=01, u=001, z=000

How to decode such an encoding ?

- But, how to decode a string of variable length ??
- Consider a specially encoded pattern as follows....
  - e.g. a=1 and b=01
  - e.g. a=1, x=01, u=001, z=000
- IS the encoding defined earlier viz. a=0, x=10, u=110, z=111 a prefix code ?

How to decode such an encoding ?

- But, how to decode a string of variable length ??
- Consider a specially encoded pattern as follows....
  - e.g. a=1 and b=01
  - e.g. a=1, x=01, u=001, z=000
- IS the encoding defined earlier viz. a=0, x=10, u=110, z=111 a prefix code ?
- Prefix code first defined by Huffman

How to decode such an encoding ?

- But, how to decode a string of variable length ??
- Consider a specially encoded pattern as follows....
  - e.g. a=1 and b=01
  - e.g. a=1, x=01, u=001, z=000
- IS the encoding defined earlier viz. a=0, x=10, u=110, z=111 a prefix code ?
- Prefix code first defined by Huffman
- definition

How to decode such an encoding ?

- But, how to decode a string of variable length ??
- Consider a specially encoded pattern as follows....
  - e.g. a=1 and b=01
  - e.g. a=1, x=01, u=001, z=000
- IS the encoding defined earlier viz. a=0, x=10, u=110, z=111 a prefix code ?
- Prefix code first defined by Huffman
- definition

How to decode such an encoding ?

- But, how to decode a string of variable length ??
- Consider a specially encoded pattern as follows....
  - e.g. a=1 and b=01
  - e.g. a=1, x=01, u=001, z=000
- IS the encoding defined earlier viz. a=0, x=10, u=110, z=111 a prefix code ?
- Prefix code first defined by Huffman
- definition

Steps for decoding a variable length encoded string

- Scan the bit sequence from left to right.

How to decode such an encoding ?

- But, how to decode a string of variable length ??
- Consider a specially encoded pattern as follows....
  - e.g. a=1 and b=01
  - e.g. a=1, x=01, u=001, z=000
- IS the encoding defined earlier viz. a=0, x=10, u=110, z=111 a prefix code ?
- Prefix code first defined by Huffman
- definition

Steps for decoding a variable length encoded string

- Scan the bit sequence from left to right.
- As soon as you have seen enough bits to match the encoding of some letter, output this as the first letter of the text.

How to decode such an encoding ?

- But, how to decode a string of variable length ??
- Consider a specially encoded pattern as follows....
  - e.g. a=1 and b=01
  - e.g. a=1, x=01, u=001, z=000
- IS the encoding defined earlier viz. a=0, x=10, u=110, z=111 a prefix code ?
- Prefix code first defined by Huffman
- definition

Steps for decoding a variable length encoded string

- Scan the bit sequence from left to right.
- As soon as you have seen enough bits to match the encoding of some letter, output this as the first letter of the text.
- This must be the correct first letter, since no shorter or longer prefix of the bit sequence could encode any other letter.

How to decode such an encoding ?

- But, how to decode a string of variable length ??
- Consider a specially encoded pattern as follows....
  - e.g. a=1 and b=01
  - e.g. a=1, x=01, u=001, z=000
- IS the encoding defined earlier viz. a=0, x=10, u=110, z=111 a prefix code ?
- Prefix code first defined by Huffman
- definition

Steps for decoding a variable length encoded string

- Scan the bit sequence from left to right.
- As soon as you have seen enough bits to match the encoding of some letter, output this as the first letter of the text.
- This must be the correct first letter, since no shorter or longer prefix of the bit sequence could encode any other letter.
- Now delete the corresponding set of bits from the front of the message and iterate.

Decoding using a prefix codes: Examples

- Consider the alphabet S = a, b, c, d, e and the encoding $\gamma(a) = 11$ $\gamma(b) = 01$ $\gamma(c) = 001$ $\gamma(d) = 10$ and $\gamma(e) = 000$. Is this code a prefix code ? IF so, show to what the encoding represented by 0010000011101 decodes to ?

Decoding using a prefix codes: Examples

- Consider the alphabet S = a, b, c, d, e and the encoding $\gamma(a) = 11$ $\gamma(b) = 01$ $\gamma(c) = 001$ $\gamma(d) = 10$ and $\gamma(e) = 000$. Is this code a prefix code ? IF so, show to what the encoding represented by 0010000011101 decodes to ?
- Consider the alphabet S = a, b, c, d and the encoding $\gamma(a) = 0$ $\gamma(b) = 10$ $\gamma(c) = 110$ $\gamma(d) = 111$. How would the string 1101001101000 be decoded ?

Prefix codes and its TBL

- How to compute the total bits required to encode a string using prefix code ?
- The total bits of a prefix code encoded string c is
  - the sum over all symbols of frequency of each character c, times the number of bits of encoding of c.
- Given that C is the alphabet, c is the prefix code function i.e. c(x) is the encoding of any character x ϵC, f(x) is the frequency of occurrence of a character c ϵC, we have

Prefix codes and its TBL

- How to compute the total bits required to encode a string using prefix code ?
- The total bits of a prefix code encoded string c is
  - the sum over all symbols of frequency of each character c, times the number of bits of encoding of c.
- Given that C is the alphabet, c is the prefix code function i.e. c(x) is the encoding of any character x ϵC, f(x) is the frequency of occurrence of a character c ϵC, we have

Total Bit Length

$\sum_{x \epsilon C} f(x)|c(x)|$ for every $x \epsilon C$

Prefix codes : Data structure to implement?

- For our goal i.e. to find a prefix code representation that has the lowest possible average bits per letter.
- We anyway need suitable data structure to represent variable length code.

Data Structure for prefix codes

- Let us represent a prefix code as a binary tree

Data Structure for prefix codes

- Let us represent a prefix code as a binary tree
- Consider the alphabet as : $c(a) = 11$ $c(e) = 01$ $c(k) = 001$ $c(l) = 10$ $c(u) = 000$.

Data Structure for prefix codes

- Let us represent a prefix code as a binary tree
- Consider the alphabet as : $c(a) = 11$ $c(e) = 01$ $c(k) = 001$ $c(l) = 10$ $c(u) = 000$.
- The binary tree representing this prefix code would be as:

Data Structure for prefix codes

- Let us represent a prefix code as a binary tree
- Consider the alphabet as : c(a) $= 11$ c(e) $= 01$ c(k) $= 001$ c(l) $=$ 10 c(u) $= 000$.
- The binary tree representing this prefix code would be as:
- Note that only the leaves of this tree have the label, the internal nodes do not have.

Data Structure for prefix codes

- Let us represent a prefix code as a binary tree
- Consider the alphabet as : $c(a) = 11$ $c(e) = 01$ $c(k) = 001$ $c(l) = 10$ $c(u) = 000$.
- The binary tree representing this prefix code would be as:
- Note that only the leaves of this tree have the label, the internal nodes do not have.
- An encoding of x is a prefix of an encoding of y if and only if the path of x is a prefix of the path of y.

Data Structure for prefix codes

- Let us represent a prefix code as a binary tree
- Consider the alphabet as : c(a) = 11 c(e) = 01 c(k) = 001 c(l) = 10 c(u) = 000.
- The binary tree representing this prefix code would be as:
- Note that only the leaves of this tree have the label, the internal nodes do not have.
- An encoding of x is a prefix of an encoding of y if and only if the path of x is a prefix of the path of y.
- Can a prefix code be represented by a binary tree ?

Data Structure for prefix codes

- Let us represent a prefix code as a binary tree
- Consider the alphabet as : c(a) = 11 c(e) = 01 c(k) = 001 c(l) = 10 c(u) = 000.
- The binary tree representing this prefix code would be as:
- Note that only the leaves of this tree have the label, the internal nodes do not have.
- An encoding of x is a prefix of an encoding of y if and only if the path of x is a prefix of the path of y.
- Can a prefix code be represented by a binary tree ?
- A binary tree representation can give optimal code. But let us try to delve deeper and formalize....

External Weighted Path length of a tree

- Let C be the alphabet, $|C|$ be the no of characters in the alphabet. How many leaves the corresponding binary tree would have ?

External Weighted Path length of a tree

- Let C be the alphabet, $|C|$ be the no of characters in the alphabet. How many leaves the corresponding binary tree would have ?
- Given an extended binary tree T, associate weights with each external node.

External Weighted Path length of a tree

- Let C be the alphabet, $|C|$ be the no of characters in the alphabet. How many leaves the corresponding binary tree would have ?
- Given an extended binary tree T, associate weights with each external node.
- Then, no of bits required to encode a file is given by, for the tree T as,
  TBL(T) $= \sum_{x \epsilon C} f(x) d_{\tau}(x)$ for every $x \epsilon C$

External Weighted Path length of a tree

- Let C be the alphabet, $|C|$ be the no of characters in the alphabet. How many leaves the corresponding binary tree would have ?
- Given an extended binary tree T, associate weights with each external node.
- Then, no of bits required to encode a file is given by, for the tree T as,
  TBL(T) $= \sum_{x \epsilon C} f(x) d_{\tau}(x)$ for every $x \epsilon C$
- def: Weighted External path length of a binary tree :

External Weighted Path length of a tree

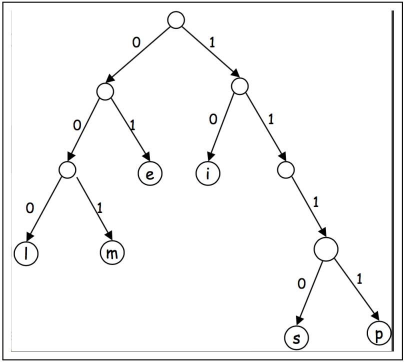
- Let C be the alphabet, $|C|$ be the no of characters in the alphabet. How many leaves the corresponding binary tree would have ?
- Given an extended binary tree T, associate weights with each external node.
- Then, no of bits required to encode a file is given by, for the tree T as,
  TBL(T) $= \sum_{x \epsilon C} f(x) d_{\tau}(x)$ for every $x \epsilon C$
- def: Weighted External path length of a binary tree :
  - The weighted path length of T is the sum of the product of the weight and path length of each external node, over all external nodes.

External Weighted Path length of a tree

- Let C be the alphabet, $|C|$ be the no of characters in the alphabet. How many leaves the corresponding binary tree would have ?
- Given an extended binary tree T, associate weights with each external node.
- Then, no of bits required to encode a file is given by, for the tree T as,
  TBL(T) $= \sum_{x \epsilon C} f(x) d_{\tau}(x)$ for every $x \epsilon C$
- def: Weighted External path length of a binary tree :
  - The weighted path length of T is the sum of the product of the weight and path length of each external node, over all external nodes.
- TBL(T) = Cost of the tree T

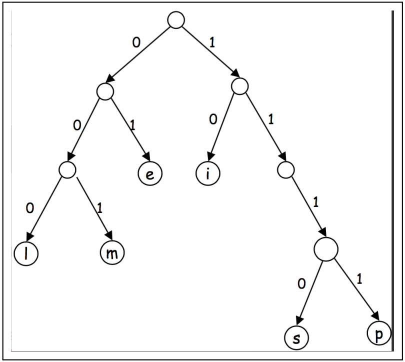
Prefix codes and Full binary trees

- Consider now another tree as shown for the code viz.



Prefix codes and Full binary trees

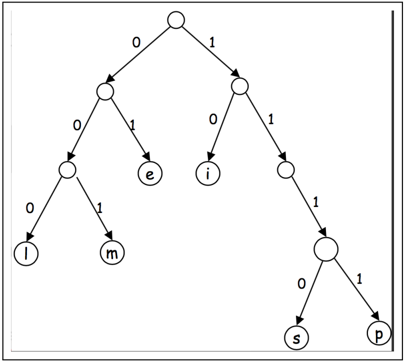
- Consider now another tree as shown for the code viz.



- Is the code used in this a prefix code ?

Prefix codes and Full binary trees

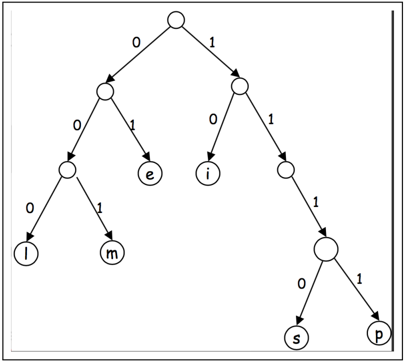
- Consider now another tree as shown for the code viz.



- Is the code used in this a prefix code ?
- Assuming each character occurs only once in a string, what is the TBL(T) = Cost of the tree T ?

Prefix codes and Full binary trees

- Consider now another tree as shown for the code viz.



- Is the code used in this a prefix code ?
- Assuming each character occurs only once in a string, what is the TBL(T) = Cost of the tree T ?
- *Can this prefix code be made more efficient*? Why is it not efficient ? How ?

Prefix codes and Full binary trees...

- The issue is *how can this prefix code be made more efficient*?

Prefix codes and Full binary trees...

- The issue is *how can this prefix code be made more efficient*?
- Assuming each character occurs only once in a string, now what is the TBL(T) = Cost of the tree T ?

Prefix codes and Full binary trees...

- The issue is *how can this prefix code be made more efficient*?
- Assuming each character occurs only once in a string, now what is the TBL(T) = Cost of the tree T ?
- Use and extended binary tree for representing a prefix code.

Prefix codes and Full binary trees...

- The issue is *how can this prefix code be made more efficient*?
- Assuming each character occurs only once in a string, now what is the TBL(T) = Cost of the tree T ?
- Use and extended binary tree for representing a prefix code.
- The concept of an Extended Binary tree, aka a full binary tree aka trie. What is an extended binary tree ?

Prefix codes and Full binary trees...

- The issue is *how can this prefix code be made more efficient*?
- Assuming each character occurs only once in a string, now what is the TBL(T) = Cost of the tree T ?
- Use and extended binary tree for representing a prefix code.
- The concept of an Extended Binary tree, aka a full binary tree aka trie. What is an extended binary tree ?
- A prefix code can always be represented as a full binary tree/trie

Prefix codes and Full binary trees...

- The issue is *how can this prefix code be made more efficient*?
- Assuming each character occurs only once in a string, now what is the TBL(T) = Cost of the tree T ?
- Use and extended binary tree for representing a prefix code.
- The concept of an Extended Binary tree, aka a full binary tree aka trie. What is an extended binary tree ?
- A prefix code can always be represented as a full binary tree/trie
- Exercise 16-3.2: A tree is full if every node that is not a leaf has two children.

Prefix codes and Full binary trees...

- The issue is *how can this prefix code be made more efficient*?
- Assuming each character occurs only once in a string, now what is the TBL(T) = Cost of the tree T ?
- Use and extended binary tree for representing a prefix code.
- The concept of an Extended Binary tree, aka a full binary tree aka trie. What is an extended binary tree ?
- A prefix code can always be represented as a full binary tree/trie
- Exercise 16-3.2: A tree is full if every node that is not a leaf has two children.
- Prove that, the binary tree corresponding to the optimal prefix code is full. OR Show that a prefix code can always be represented as a full binary tree.

Prefix codes and Full binary trees...

- The issue is *how can this prefix code be made more efficient*?
- Assuming each character occurs only once in a string, now what is the TBL(T) = Cost of the tree T ?
- Use and extended binary tree for representing a prefix code.
- The concept of an Extended Binary tree, aka a full binary tree aka trie. What is an extended binary tree ?
- A prefix code can always be represented as a full binary tree/trie
- Exercise 16-3.2: A tree is full if every node that is not a leaf has two children.
- Prove that, the binary tree corresponding to the optimal prefix code is full. OR Show that a prefix code can always be represented as a full binary tree.
    - Proof to be worked out by contradiction...

Prefix codes and Full binary trees...

- The issue is *how can this prefix code be made more efficient*?
- Assuming each character occurs only once in a string, now what is the TBL(T) = Cost of the tree T ?
- Use and extended binary tree for representing a prefix code.
- The concept of an Extended Binary tree, aka a full binary tree aka trie. What is an extended binary tree ?
- A prefix code can always be represented as a full binary tree/trie
- Exercise 16-3.2: A tree is full if every node that is not a leaf has two children.
- Prove that, the binary tree corresponding to the optimal prefix code is full. OR Show that a prefix code can always be represented as a full binary tree.
  - Proof to be worked out by contradiction...
- Where in the tree of an optimal prefix code should letters be placed with a high frequency?

Huffman Tree construction illustrations

- Construct the full binary tree for the variable length prefix codes:

Huffman Tree construction illustrations

- Construct the full binary tree for the variable length prefix codes:
  - a = 00, x = 01, u = 10, z = 11 and the frequency being 996,2,1,1 respectively

Huffman Tree construction illustrations

- Construct the full binary tree for the variable length prefix codes:
  - a = 00, x = 01, u = 10, z = 11 and the frequency being 996,2,1,1 respectively
  - a = 0, x = 10, u = 110, z = 111 with the same frequency as above

Constructing the tree: Huffman Encoding

- Defining Huffman tree
- Construct the Huffman tree for the following

a	b	c	d	e	f
6	2	3	3	4	9

The Huffman Tree Construction pseudocode

```
The Algorithm Huffman-Tree(C)
1. n = |C|
2. for i = 1 to n-1
3.        do z = ALLOCATE_NODE()
4.        x = left(z) = EXTRACT_MIN(Q)
5.        y = right(z) = EXTRACT_MIN(Q)
6.        f[z] = f[x] + f[y]
7.        INSERT(Q, z)
8. return EXTRACT_MIN(Q)
```

Why (n-1) in line 2?
Why the last line ?

The Huffman Tree Construction pseudocode

```
The Algorithm HEAP–EXTRACT–MIN(A)
1. if heap_size[A] < 1
2.    then return error heap underflow
3. min = [A]
4. A[1] = A[heap−size[A]]
5. heap−size[A]=heap−size[A]−1
5. heap−size[A] = heap−size[A] − 1
6. HEAPIFY(A, 1)
7. return min
```

Time Complexity

- HEAP-EXTRACT-MIN & INSERT take O(lg n) time on Q with n objects
- Therefore, $T(n) = \sum_{i=1}^{n} lg\ i = O(\lg(n!)) = O(n \lg n)$

Optimal Merge Patterns

Problem definition

To merge n sorted files in such a way that the fewest number of comparisons are required while merging them.

Say Given files x_1, x_2, x_3, x_4, x_5 with $|x1|$=20, $|x2|$=30, $|x3|$=10, $|x4|$=5, $|x5|$=30. Show haw can they be optimally merged.

What are the various record moves required merging these files in different patterns ?

- Consider the game of guessing a chosen object from n possibilities (say, an integer between 1 and n) by asking questions answerable by yes or no.
- Different strategies for playing this game can be modeled by decision trees

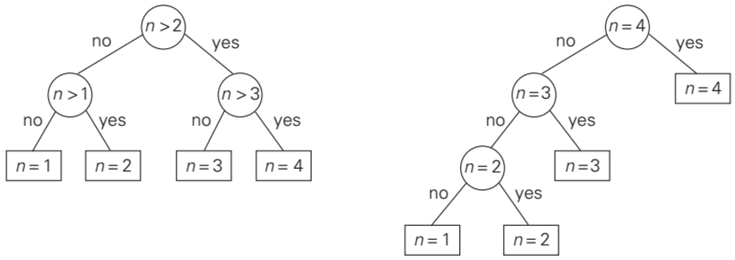


FIGURE 9.13 Two decision trees for guessing an integer between 1 and 4.

Prefix codes and Full binary trees

- Exercise 16-3.4: Lemma1: Prove that we can also express the total cost of a tree for a code as the sum, over all internal nodes, of the combined frequencies of the two children of the node.

Prefix codes and Full binary trees

- Exercise 16-3.4: Lemma1: Prove that we can also express the total cost of a tree for a code as the sum, over all internal nodes, of the combined frequencies of the two children of the node.
- Theorem2: Prove that the optimal data compression that is achievable by any character code can always be achieved with a prefix code.

Correctness Proof: Huffman coding...

The Huffman's algorithm is correct

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

- Step 1: Show that this problem satisfies the greedy choice property, that is, if a greedy choice is made by Huffman's algorithm, an optimal solution remains possible.
- Step 2: Show that this problem has an optimal substructure property, that is, an optimal solution to Huffman's algorithm contains optimal solution to subproblems.
- Step 3: Conclude correctness of Huffman's algorithm using step 1 and step 2.

Correctness Proof: Huffman coding...

The Huffman's algorithm is correct

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

- Step 1: Lemma1: Show that this problem satisfies the greedy choice property, that is, if a greedy choice is made by Huffman's algorithm, an optimal solution remains possible.

The Huffman's algorithm is correct

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

- Step 1: Lemma1: Show that this problem satisfies the greedy choice property, that is, if a greedy choice is made by Huffman's algorithm, an optimal solution remains possible.
  - Greedy Choice Property: Let c be an alphabet in which each character c has frequency f[c]. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Correctness Proof: Huffman coding...

The Huffman's algorithm is correct

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

- Step 1: Lemma1: Show that this problem satisfies the greedy choice property, that is, if a greedy choice is made by Huffman's algorithm, an optimal solution remains possible.
  - Greedy Choice Property: Let c be an alphabet in which each character c has frequency f[c]. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.
- Lemma 1 implies that process of building an optimal tree by mergers can begin with the greedy choice of merging those two characters with the lowest frequency.

Correctness Proof: Huffman coding...

The Huffman's algorithm is correct

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

- Step 1: Lemma1: Show that this problem satisfies the greedy choice property, that is, if a greedy choice is made by Huffman's algorithm, an optimal solution remains possible.
  - Greedy Choice Property: Let c be an alphabet in which each character c has frequency f[c]. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.
- Lemma 1 implies that process of building an optimal tree by mergers can begin with the greedy choice of merging those two characters with the lowest frequency.
- We have already proved that TBL(T) i.e. the total cost of the tree constructed is the sum of the costs of its mergers (internal nodes) of all possible mergers.

Correctness Proof: Huffman coding...

The Huffman's algorithm is correct

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

- Step 1: Lemma1: Show that this problem satisfies the greedy choice property, that is, if a greedy choice is made by Huffman's algorithm, an optimal solution remains possible.
  - Greedy Choice Property: Let c be an alphabet in which each character c has frequency f[c]. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.
- Lemma 1 implies that process of building an optimal tree by mergers can begin with the greedy choice of merging those two characters with the lowest frequency.
- We have already proved that TBL(T) i.e. the total cost of the tree constructed is the sum of the costs of its mergers (internal nodes) of all possible mergers.
- At each step Huffman chooses the merger that incurs the least cost

Correctness Proof: Huffman coding

Lemma 1

Let C be an alphabet in which each character c ϵ C has frequency $f(c)$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

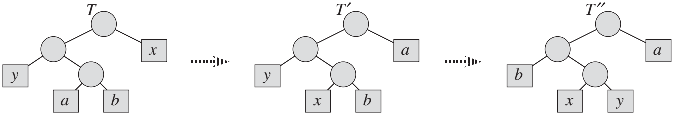


Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree T, leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies; they appear in arbitrary positions in T. Assuming that $x \neq b$, swapping leaves a and x produces tree T', and then swapping leaves b and y produces tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

Correctness Proof: Huffman coding...

The Huffman's algorithm is correct

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

- Step 2: Show that this problem has an optimal substructure property, that is, an optimal solution to Huffman's algorithm contains optimal solution to subproblems.
  - Let T be a full binary tree representing an optimal prefix code over an alphabet C, where frequency f[c] is define for each character c belongs to set C. Consider any two characters x and y that appear as sibling leaves in the tree T and let z be their parent. Then, considering character z with frequency f[z] = f[x] + f[y], tree T‘ = T - x, y represents an optimal code for the alphabet C‘ = C - x, yUz.
- Step 3: Conclude correctness of Huffman's algorithm using step 1 and step 2.

Lemma 2

A Tutorial Problem

- A long string consists of the four characters A, C, G, T; they appear with frequencies 31%, 20%, 9% and 40% respectively. Run the Huffman coding procedure from class to derive an optimal code for these characters. What is the average number of bits per character in your code?
  - Prove that in the Huffman coding scheme, if some character occurs with frequency more than 2/5, then there is guaranteed to be a codeword of length 1.
  - Prove also that if all characters occur with frequency less than 1/3, then there is guaranteed to be no codeword of length 1.
  - Suppose the frequencies of letters in an n-letter alphabet are $f_1, f_2, f_3, ...f_n$ what relationship between these frequencies leads to the longest possible codeword? You should give an intuitive justification for your answer, but you need not provide a formal proof.

- IT is necessary to include the coding table into the encoded text to make its decoding possible.
- This drawback can be overcome by using dynamic Huffman encoding in which the coding tree is updated each time a new symbol is read from the source text.
- Modern schemes like Lempel-Ziv algorithms assign codewords not to individual symbols but to strings of symbols, allowing them to achieve better and more robust compressions in many applications.

Blank

Blank

Blank