GREEDY ALGORITHMS AND HUFFMAN ENCODING

DR. GODFREY C. MUGANDA

1. Greedy Algorithms

You should consider a greedy algorithm when you need to solve certain types of optimization problems.

Recall that in an optimization problem, you have a set of *feasible* solutions and you are looking to pick an *optimal solution*. An optimal solution is a feasible solution that maximizes or minimizes some objective function.

In this lecture, we will assume that we are working with a minimization problem, but what we say applies to maximization problems as well.

2. When is greed appropriate?

A greedy algorithm is appropriate when each feasible solution is a sequence of parts where each part contributes to the objective function.

This means that the value of the objective function on a feasible solution is the sum of the contributions of each part of the solution.

With this type of problem, a greedy algorithm tries to build an optimal solution is stages, where at each stage, it picks a part to add to the solution under construction by picking the part that adds the least to the total cost.

3. Coin Changing Problem

When you go to the store and you pay cash for something, you may need to get change back. Say you are owed 21 cents in change.

There are many solutions, including:

(1) A feasible solution: 21 pennies = 21 coins.
(2) An optimal solution: 2 dimes and 1 penny = 3 coins.

Change-making problem can be solved using backtracking, and it can also be solved using dynamic programming.

4. A Greedy Algorithm for Change Making

In the general change-making problem, there is a system of n coins $c[1, \ldots, n]$, where $c[i]$ is the value of the ith coin and

$$c[1] > c[2] > \cdots > c[n] = 1.$$

You always assume $c[n] = 1$, because otherwise you cannot make change for some amounts.

Example: In the US monetory system, $n = 4$ and

$$c[1] = 25, \quad c[2] = 10, \quad c[3] = 5, \quad c[4] = 1$$

The greedy strategy when making change for an amount A is to always give the largest number of the biggest coin whose value $c[i] < A$.

This number is $A/c[i]$.

More specifically, if $A = 56$, then in the US system 25 is the largest coin that will fit so you give 2 quarters because

$$56/25 = 2.$$

After that, you must still make change for $56 - (25 \times 2) = 6$ cents.

Here is a formal statement of the change-making problem:

INPUT: A system of coins $c[1] > c[2] > \cdots > c[n] = 1$ and an integer A to make change for.
OUTPUT: An integer array $p[1 \ldots n]$ where $p[i]$ is the number of coins of value $c[i]$ given and $\sum_{i=1}^{n} p[i] \cdot c[i] = A$.

For example, if $A = 56$ with the US system of coins, then the output array $p[1, \ldots, n]$ will be:

$$\begin{aligned} p[1] &= 2 \qquad (2 \text{ quarters} = 50 \text{ cents}) \\ p[2] &= 0 \qquad (0 \text{ dimes} = 0 \text{ cents}) \\ p[3] &= 1 \qquad (1 \text{ nickle} = 5 \text{ cents}) \\ p[4] &= 1 \qquad (1 \text{ penny} = 1 \text{ cent}) \end{aligned}$$

Here is the greedy algorithm.

```
i = 1
while (i ≤ n)
{
  p[i] = A/c[i]
  A = A − p[i] ∗ c[i]
  i = i + 1
}
```

The greedy algorithm works for the US system of coins. However, it does not work for all systems of coins.

Here is a counter example. Let $n = 3$ and let the coin system be

$$c[1] = 10, \quad c[2] = 8, \quad c[3] = 1$$

and we want to make change for $A = 16$.

The optimal solution uses two 8-cent pieces.

The greedy algorithm uses one 10 cent piece and 6 pennies, for a total of 7 coins.

There is a proof that shows the greedy algorithm works for the US system of coins, but it is not easy to follow.

In general, proofs of correctness for greedy algorithms are not easy to follow, so we will skip the proof of correctness of greedy change making for the US system.

5. Data Representation With Fixed-length Codes

Data is normally represented using *fixed-length* codes, where each character is represented using the same number of bits.

The most common way of representing text data is a fixed-length code known as the ASCII code. In ASCII, each character is represented by an 8-bit code.

In general, if your data involves n characters and you choose to use a fixed-length code, each character will be represented by a $\log_2 n$ bits.

If you have 8 characters, you need 3 bits.

If you have 4 characters, you need 2 bits.

If you have 3 characters, you need 2 bits.

If you have 2 characters, you need just 1 bit.

Suppose your file contains only 3 characters 'a', 'b', and 'c'. Using a fixed-length code, we need 2 bits per character, say

```
a : 00        b: 01         c: 10
```

Now suppose your file has thirteen characters:

```
a b a a a a b a a a c c b
```

The representation of the file will require $13 \times 2 = 26$ bits.

Can we do any better? That is, is it possible to compress the data and use fewer bits?

6. Variable-Length Codes and Data Compression

Data compression tries to minimize the number of bits needed to represent a file of data.

One way to compress data is to use *variable-length* codes for the characters, where you look at the frequency with which characters appear in the file, and use shorter codes for characters that appear more frequently, and longer codes for characters that appear few times.

Look at our example file: `a b a a a a b a a a c c b`

The character frequencies are

`a: 8` `b: 3` `c:2`

If we use the variable-length code

`a : 0` `b: 10` `c: 11`

then the total number of bits needed to represent the file is $8(1) + 3(2) + 2(2) = 18$.

This is about 69% of the previous number of bits, which was 26.

7. Parsing and Prefix-free Codes

Suppose we use a variable-length code of

`a : 0` `b: 01` `c: 1`

Now if we have an encoded message 01, we will not know whether the message is `b` or `ac`.

This is because the code for 'a' is a prefix of the code for 'b'.

We want our variable-length codes to be *prefix-free*: no code for a character is a prefix for another character's code.

8. Prefix-Free Codes as Full Binary Trees

If you have a prefix-free, variable-length code, one that is set up nicely and has no "gaps," you can express all the codes as paths in a *full binary tree*, where at each node X, the left edge from X is labeled 0 and the right edge from X is labeled 1.

You can represent such a prefix-free code as a full binary tree where the characters being coded for are at the leaves of the tree, and the labels on the path from the root to the character specify the code of the character.

Recall our previous example `a b a a a a b a a a c c b`

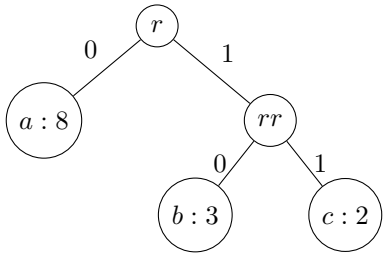
with character frequencies

`a: 8` `b: 3` `c:2`

If we use the variable-length code

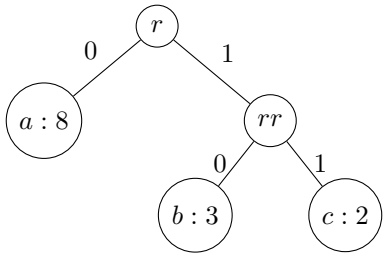
`a : 0` `b: 10` `c: 11`

We can represent it as a full-binary tree like this:



In this tree we have also shown the frequency with which each character appears in the file.

9. Weighted Path Length of Coding Trees



In a tree that represents a code for n characters

$$c_1, \ldots, c_n$$

with frequencies

$$f_1, f_2, \ldots, f_n,$$

The characters $[c_i : f_i]$ and their frequencies are at the leaves of the tree at a depth d_i which is equal to the length of the code assigned to c_i.

You can compute the number of bits in the compressed file like this

$$\sum_{i=1}^{n} f_i \cdot d_i$$

where d_i is the depth of the frequency f_i in the tree. This expression is called the *weighted path length* of the tree.

The goal of data compression is to find a coding tree with *minimum path length.*

Huffman devised a greedy algorithm for finding such trees.

10. Huffman Trees

A Huffman tree is a tree for an optimal code for a set of characters, that is, a tree for a code that minimizes the weighted path length

$$\sum_{i=1}^{n} f_i \cdot d_i$$

Huffman's greedy algorithm for finding optimal trees tries to place the smaller frequencies f_i at a greater depth, so that the larger frequencies have a smaller depth.

Let us suppose that we have n characters c_i with frequencies

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

sorted in non-decreasing order.

We will use this data as a running example, suppose we have characters and frequencies

`T:4` `S:6` `R:13` `O:16` `A:22` `E:60`

The Huffman algorithm is based on the following facts, which we state without proof. We are still assuming the frequencies are sorted in non-decreasing order.

(1) There is an optimal tree in which the two smallest frequencies, in this case

$$f_1 \text{ and } f_2$$

have the same parent (are siblings).

(2) Finding an optimal tree for n characters with frequencies

$$f_1,\, f_2,\, f_3, \ldots, f_n$$

is equivalent to finding an optimal tree for $n-1$ characters with frequencies

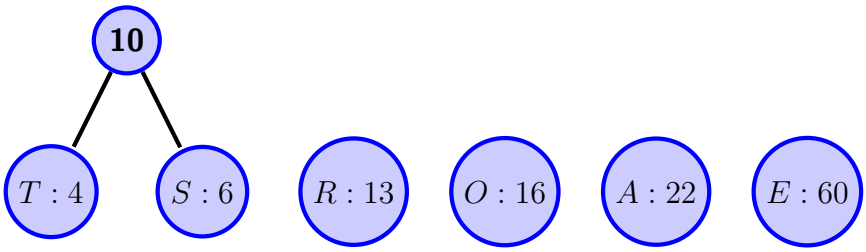
$$f_1 + f_2,\, f_3, \ldots, f_n.$$

11. Working of Huffman's Algorithm

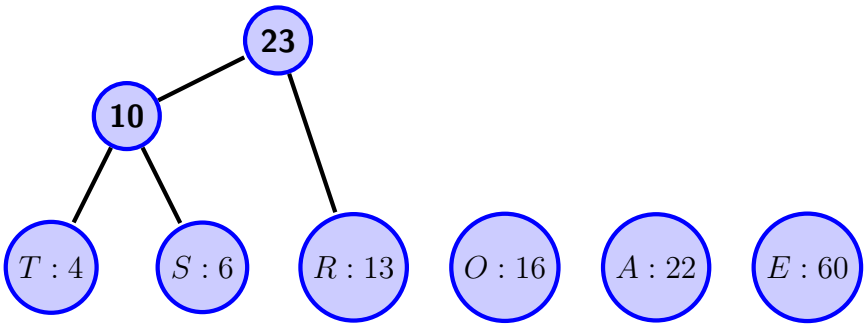
Let us use these two facts to see how Huffman's greedy algorithm finds an optimal code for

`T:4` `S:6` `R:13` `O:16` `A:22` `E:60`

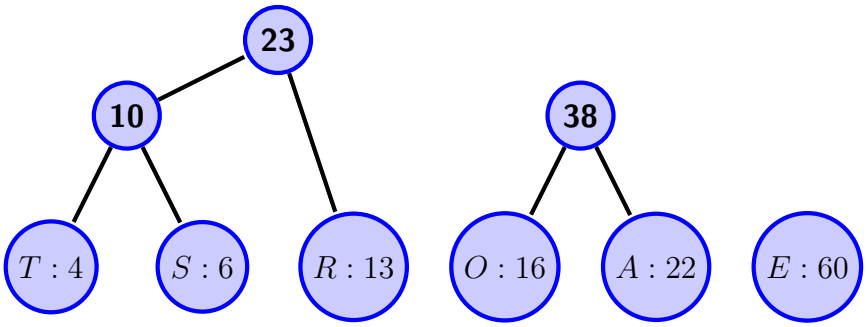
First, we find the two smallest frequencies f_1 (4) and f_2 (6) and make them siblings, replacing them with the sum $f_1 + f_2$ (10):



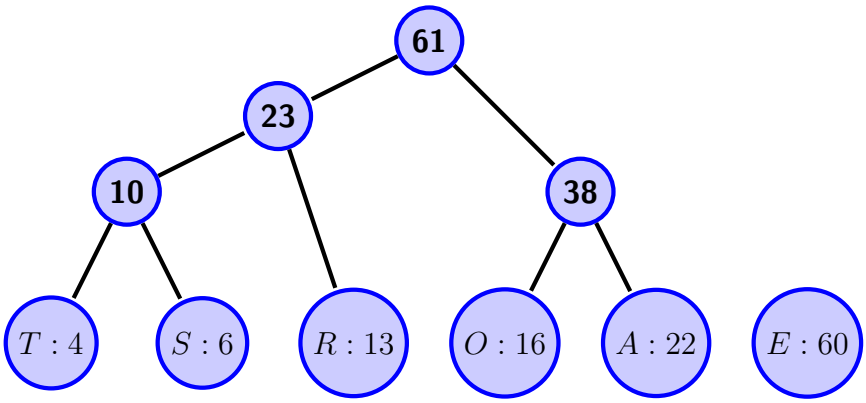
Next, pick the two smallest frequencies, 10 and 13, and make them siblings, replacing them with the frequency sum:



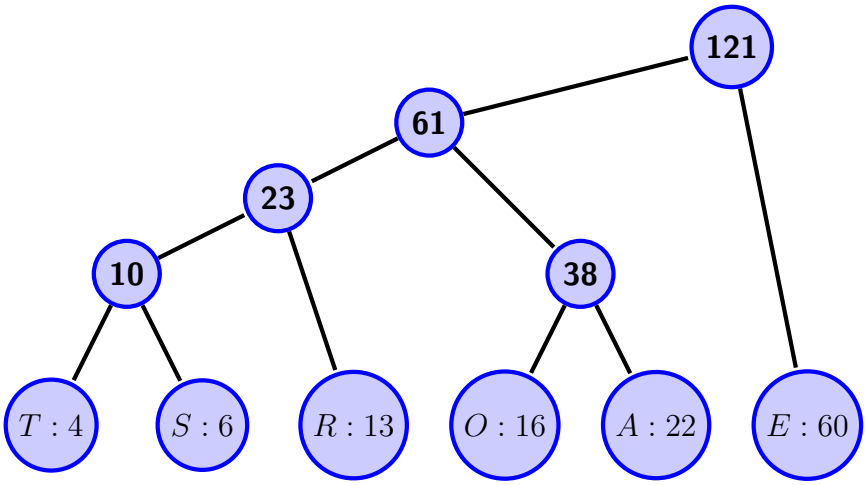
Next, make the two smallest frequencies, 16 and 22, siblings, replacing them with the sum 38:



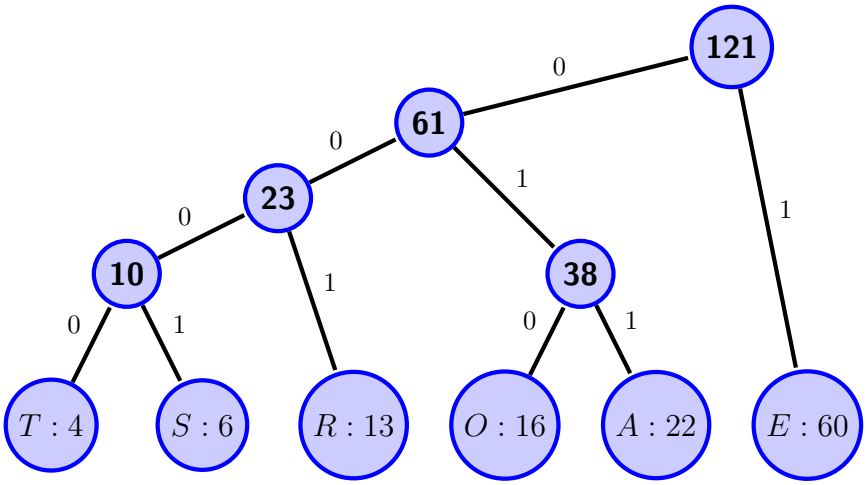
Next, locate the two smallest frequencies, 23 and 38, and make them siblings with a frequency sum of 61:



Finally, make the two smallest (and only remaining) frequencies, 61 and 60, siblings with a frequency sum of 121:



Having completed the Huffman tree, we can label the edges to determine the optimal code:



The result is the following Huffman code

Character	Frequency	Code
T	4	0000
S	6	0001
R	13	001
O	16	010
A	22	011
E	60	1

12. Implementation of Huffman's Algorithm

The algorithm builds a tree so we need some nodes.

```
class Node
{
   int freq;
   Node left;
   Node right;
   Node(int f, Node L, Node R)
   {
     this.freq = f;
     this.left = L;
     this.right = R;
   }
   Node(int f)
   {
      this.freq = f;
      this.left = null;
      this.right = null;
   }
}
```

We need a data structure H called a *heap*. The heap will store nodes ordered by according to the frequency.

This is a data structure that supports two main operations:

(1) H.remove(): removes and returns the node with the smallest frequency
(2) H.add(x): adds a new node x to the heap.

13. Huffman's Algorithm

INPUT: An array of frequencies $f[1, \ldots, n]$
OUTPUT: A Huffman tree for the frequencies $f[1, \ldots, n]$

```
for i = 1 to n do
    H.add(new Node(f[i]))
end for
while H.size() > 1 do
      Node N_1 = H.remove()
      Node N_2 = H.remove()
      H.add(new Node(N_1.freq + N_2.freq, N_1, N_2))
end while
return H.remove()
```