Greedy Algorithms

- Greedy algorithms apply to problems that exhibit:
  - The **greedy choice property**, and
  - **optimal substructure**.
- We have seen that **optimal substructure** means that optimal solutions contain optimal subsolutions.
- The **greedy choice property** means that an optimal solution can be obtained by making the “greedy” choice at every step.
- We don’t need solutions to subproblems in order to make a choice.
- This is the major difference between dynamic programming and greedy algorithms.
- Since there can be many optimal subsolutions, greedy algorithms can be more efficient than dynamic programming solutions.

A Greedy Problem

- A thief breaks into a cookie store.
- He has a bag that can hold up to W pounds of cookies.
- There are n cookies in the store.
- The ith cookie weighs w_i pounds and is worth v_i dollars.
- He is allowed to break the cookies and take fractions of them.
- The thief wants to maximize the value of the cookies he steals.
- How much of each cookie should he steal?
- This is called the **fractional knapsack problem** because he can take fractions of each item.

A Greedy Solution

- Notice that the ith cookie is worth $p_i = v_i/w_i$ dollars per pound.
- The item with the largest p_i has the most "bang for the buck," so it seems obvious that the thief should take as much of it as he can.
- If the thief takes x pounds of a cookie with $p_j < p_i$ instead of cookie i, his profit will obviously be smaller for the same weight.

A Greedy Solution

- The following algorithm solves the fractional knapsack problem

```
Fractional_Knapsack(Array v,Array w, int W)
   For i=1 to Size(v) Do
       p[i]=v[i]/w[i]
   Sort_Descending(p)
   i=1
   While (W>0) Do
         amount=Min(W,w[i])
         solution[i]=amount
         W=W-amount
         i=i+1
   Return solution
```

- There is a subtle concern here. What is it?
- What is the complexity of the algorithm?

A Greedy Example

- Let's assume that the thief's knapsack holds 15 pounds.
- The cookie store has the following cookies.

i	1	2	3	4	5	6	7	8
v_i	12	4	5	3	8	8	12	1
w_i	4	3	5	6	1	4	10	4
p_i	3	1.33	1	.5	8	2	1.2	.25

- Notice that we computed the value per pound above.
- Now we can sort according to p_i.

i	5	1	6	2	7	3	4	8
v_i	8	12	8	4	12	5	3	1
w_i	1	4	4	3	10	5	6	4
p_i	8	3	2	1.33	1.2	1	.5	.25

A Greedy Example

- The thief takes 1 pound of cookie 5, 4 pounds of cookie 1, 4 pounds of cookie 6, 3 pounds of cookie 2, and 3 pounds of cookie 7:

i	5	1	6	2	7	3	4	8
v_i	8	12	8	4	12	5	3	1
w_i	**1**	**4**	**4**	**3**	**3**/10	5	6	4
p_i	8	3	2	1.33	1.2	1	.5	.25

- The thief's profit is 8+12+8+4+3.6=35.6

Example 2: Encoding data

- Alphabetic characters are often stored in *ASCII* on computers, which requires 7 bits per character.
- *ASCII* is a **fixed-length code**, since each character requires the same number of bits to store.
- Notice that some characters, (e.g. **q**, **x**, **z**, **v**) are rare, and others (e.g. **e**, **s**, **t**, **a**) are common.
- It might make more sense to use less bits to store the common characters, and more bits to store the rare characters.
- An encoding that does this is called a **variable-length code**.
- A code is called **optimal** if the space required to store data with the given distribution is a minimum.
- Optimal codes are important for many applications.

Variable Length Code Example

- Assume a file has the following distribution of characters.

Letter	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13

- One good encoding might be the following:

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	10	11	101	1	01	001	110	011

- Thus, the string **TREAT** is encoded as **110111011**
- But notice that it can also be **KTVE**.
- We have to somehow keep track of which letter is which.
  (e.g. **11_01_1_10_11**?)
- This may make the code take more space than a fixed-length code.
- Another solution is to use a **prefix code**.

Prefix Codes

- A **prefix code** is a code in which no word is a prefix of another word.
- To encode a string of data, concatenate the codewords together.
- To decode, just read the bits until a codeword is recognized.
- Since no codeword is a prefix of another, this works.
- Here is a prefix code for the previous example.

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

- Notice that no codeword is the prefix of another.
- Now, we can encode **TREAT** as **1001010100100**, and it is uniquely decodable.

Decoding

- Given a code, we need to decode efficiently.
- **Example:** Decode **01111011010010100100** based on the following

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

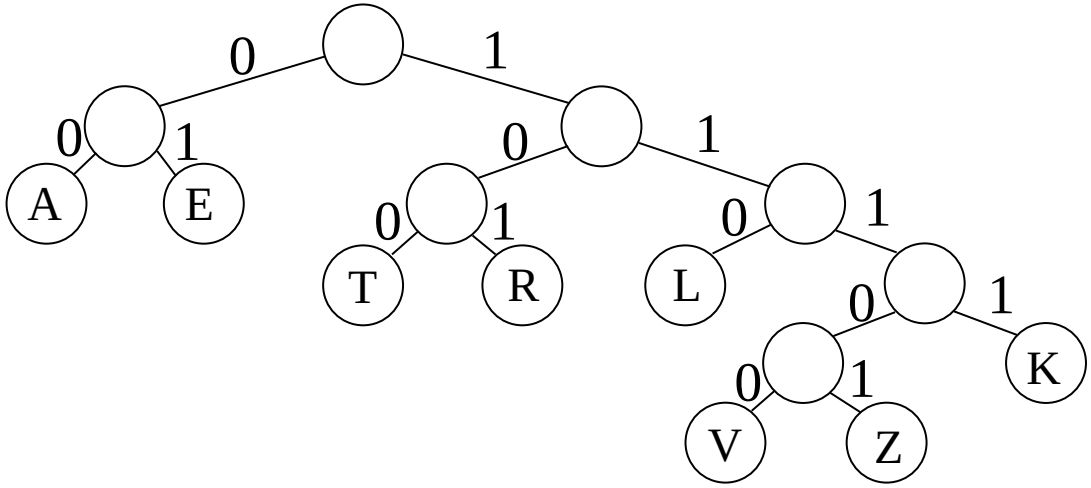
- The most obvious way is to read one bit, see if it is a character, read another, see if the two are a character, etc. (not very efficient)
- A better way is to represent the encoding with a binary tree.
  - Each leaf node stores a character.
  - A ‘0’ means go to the left child
  - A ‘1’ means go to the right child
  - The process continues until a leaf is found.
- Any code represented this way is a prefix code. Why?

More Decoding

- **Example:** Decode **01111011010010100100** based on the following encoding

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

- We will represent the data by the following binary tree:



- It is now not too hard to see that the answer is **EZRARAT**.

Constructing Codes

- We have seen that prefix codes make encoding and decoding data very easy.
- It can be shown that optimal data compression using a character code can be obtained using a prefix code.
- It might be nice to know how to construct such a code.
- We will now see how to construct one such code, called a **Huffman code**
- A Huffman code can be constructed by building the encoding tree from the bottom-up in a greedy fashion.
- Since the less frequent nodes should end up near the bottom of the tree, it makes sense that we should consider these first.
- We'll see an example, then the algorithm.

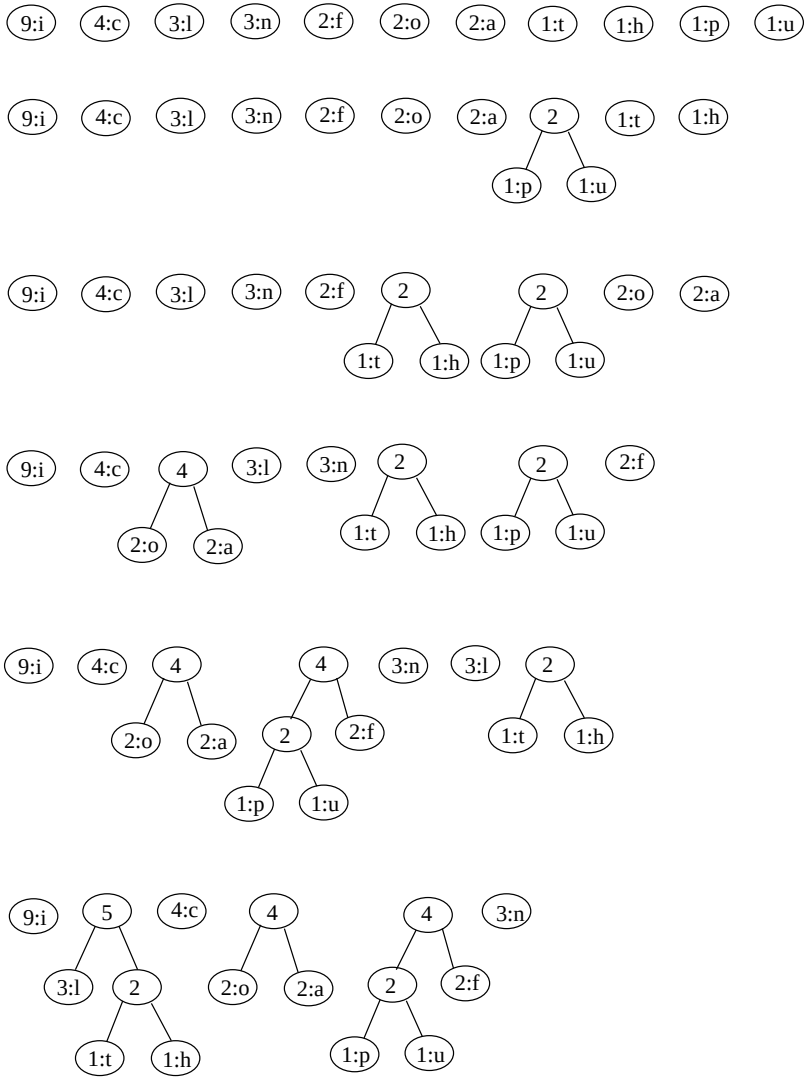
Huffman Code Example

- I want to store: **floccinaucinihilipilification**.
- I want to store it as efficiently as possible.
- The frequency of letters, sorted descending, is as follows:

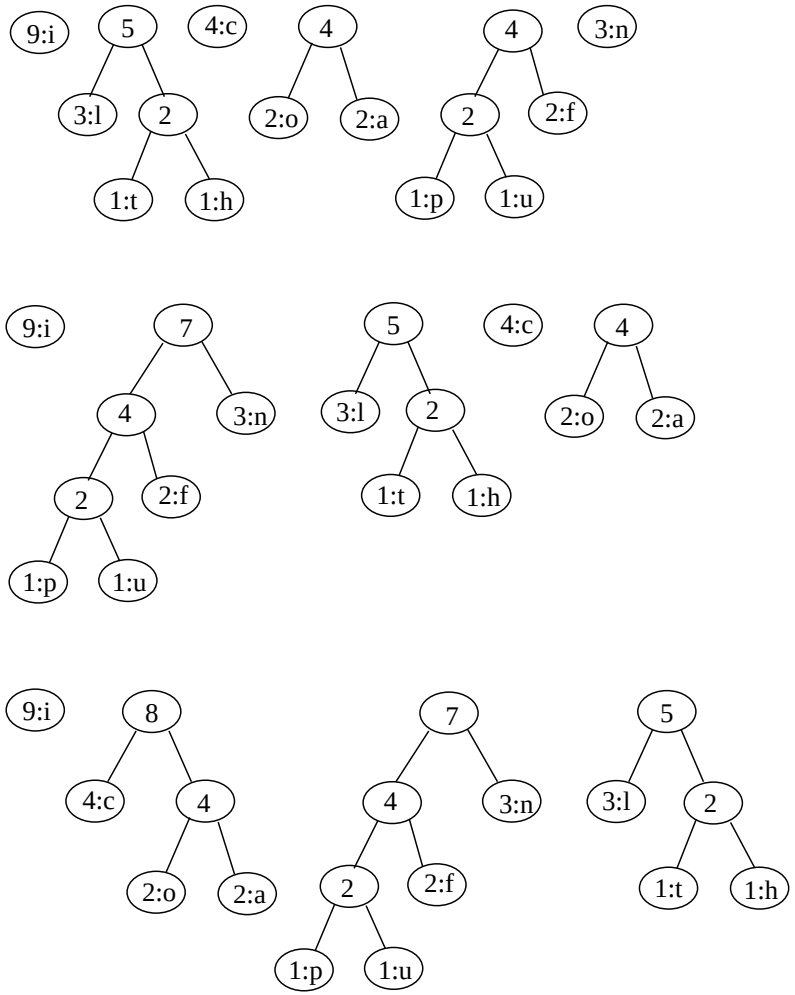
i	c	l	n	f	o	a	t	h	p	u
9	4	3	3	2	2	2	1	1	1	1

- The algorithm works sort of like this:
  - Consider each letter as a node in a not-yet-constructed tree.
  - Label each node with its letter and frequency.
  - Pick two nodes x and y of least frequency.
  - Insert a new node, and let x and y be its children. Let its frequency be the combined frequency of x and y.
  - Take x and y off the list.
  - Continue until only 1 node is left on the list.

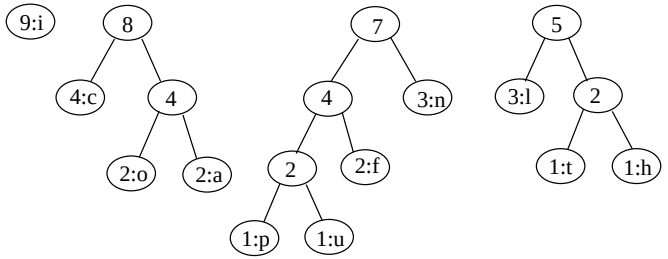
Huffman Example

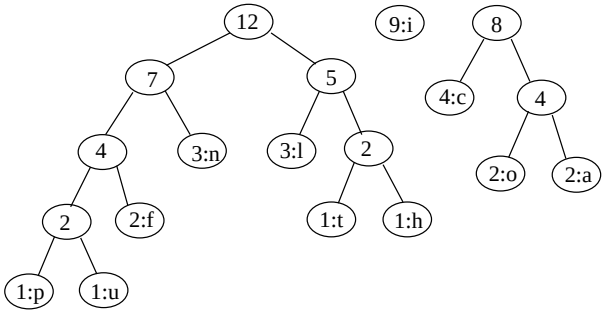


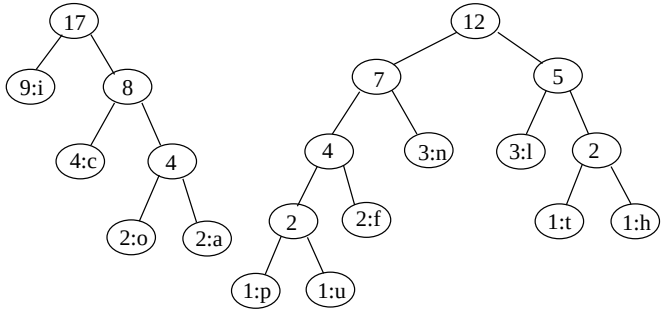
Huffman Example Continued



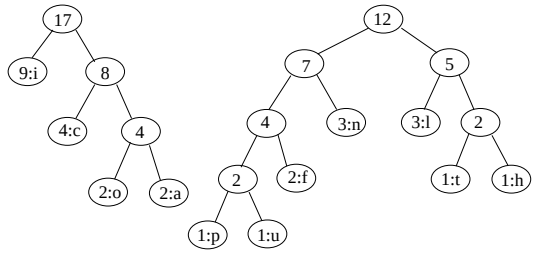
Huffman Example Continued

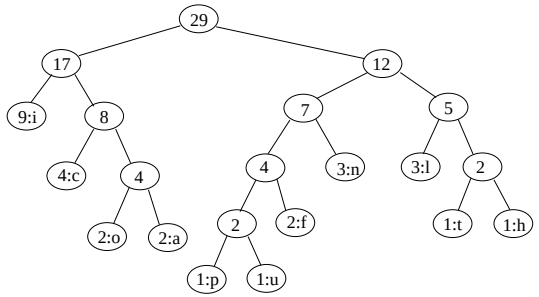


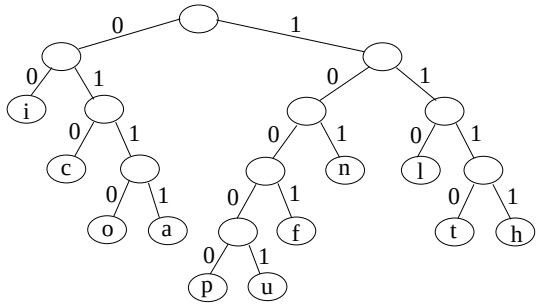




Huffman Example Continued

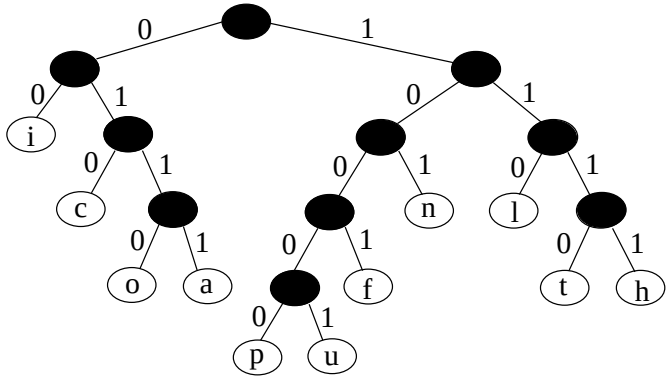






Huffman Example Continued

- We can now list the code:



i (9)	00	f (2)	1001
c (4)	010	t (1)	1110
n (3)	101	h (1)	1111
l (3)	110	p (1)	10000
o (2)	0110	u (1)	10001
a (2)	0111		

Huffman Encoding

- We have a list of n characters, each with some frequency.
- For each character, we will define a **Node** containing the *character*, *frequency*, *left*, and *right*.
- We will store the nodes with a data structure that supports the operations **Insert** and **Extract_Min**.
- A **priority queues**, which can be implemented with a heap, is a good choice.
- For this application, heaps aren't quite right. Why? What can we do about it?
- The algorithm for Huffman encoding will build a tree from the nodes in a bottom-up fashion.

Huffman Encoding Algorithm

```
Huffman_Encoding(The_List)
   While(The_List.size > 1)
       z=New Tree_Node
       x=Extract_Min(The_List)
       y=Extract_Min(The_List)
       left[z]=x
       right[z]=y
       f[z]=f[x]+f[y]
       Insert(The_List,z)
   return Extract_Min(The_List)
```

- The algorithm returns the last node in the list–the root of the tree.
- We can implement this in $O(n \log n)$ time with the proper data structure choice. (Which one?)
- Does this algorithm produce an optimal encoding?

Cost of a Huffman Code

- Assume we have a file containing n different characters, $c_1, \ldots, c_n$.
- Each character c_i has frequency $f(c_i)$.
- Let T be a tree corresponding to a Huffman code.
- Let $d_T(c_i)$ denote the depth of the leaf containing c_i.
- Note that $d_T(c_i)$ is the number of bits required to store c_i using the code.
- The **cost** of storing the file with the given code is

$$B(T) = \sum_{i=1}^{n} f(c_i) d_T(c_i).$$

- An optimal code is a code whose tree T has the minimum $B(T)$.

Huffman Code is Optimal

- Let C be the set of character in a file.
- We can imagine that the Huffman algorithm combines two characters x and y into a new character z with frequency $f(x) + f(y)$, and tries to find an optimal prefix code for $C' = C \cup \{z\} - \{x, y\}$.
- To prove that the Huffman code is optimal, we would need to show:
  - Making the greedy choice will result in an optimal prefix code. In other words, if x and y are characters of minimum frequency, then there exists some optimal prefix code in which x and y are siblings. (**greedy property**)
  - If T is an optimal code for C containing siblings x and y with parent z, then $T - \{x, y\}$ is an optimal code for $C' = C \cup \{z\} - \{x, y\}$. (**optimal substructure**)
- We will not prove these here.

Greedy Algorithm Conclusions

- An optimization problem can be solved efficiently using a greedy algorithm if it exhibits the following properties:
  - **greedy-choice property**
  - **optimal substructure**
- The most important difference between greedy algorithms and dynamic programming is that we don't solve every optimal subproblem with greedy algorithms.
- In some cases, greedy algorithms can be used to produce sub-optimal solutions. That is, solutions which aren't necessarily optimal, but are perhaps very close.