Greedy Algorithms
Chapter 17

Elements of Greedy Algorithms

What makes an algorithm greedy?
1. Greedy choice property
2. Optimal substructure (ideally)

Greedy choice property: Globally optimal solution can be arrived by making a locally optimal solution (greedy). The greedy choice property is preferred since then the greedy algorithm will lead to the optimal, but this is not always the case – the greedy algorithm may lead to a suboptimal solution. Similar to dynamic programming, but does not solve subproblems. Greedy strategy more top-down, making one greedy choice after another without regard to subsolutions.

Optimal substructure: Optimal solution to the problem contains within it optimal solutions to subproblems. This implies we can solve subproblems and build up the solutions to solve larger problems.

Activity Selection Problem

Problem: Schedule an exclusive resource in competition with other entities. For example, scheduling the use of a room (only one entity can use it at a time) when several groups want to use it. Or, renting out some piece of equipment to different people.

Definition: Set S={1,2, … n} of activities. Each activity has a start time s_i and a finish time f_i, where $s_i<f_i$. Activities i and j are compatible if they do not overlap. The activity selection problem is to select a maximum-size set of mutually compatible activities.

A simple greedy algorithm solves this problem optimally:
```
1. Sort input activities in order by increasing finishing time
2. n ← length[s]
3. A ← 1
4. j ← 1
5. for i ← 2 to n
      6. if $s_i \geq f_j$ then
            7. A ← A ∪ {i}
            8. j ← i
9. return A
```

Just marches through each activity in terms of the finishing time, and schedules it if possible.

Example:

I	start	finish
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12

Schedule job 1:

```
 1111
   222
```

Job two does not fit, so don’t add it. Try job 3:

```
 1111
0000000
```

Job three does not fit, don’t add it. Try job 4:

```
 1111
     444
```

Fits, so leave it in. Try job 5, 6 , 7:

```
 1111
     444
   555555
     66666
       77777
```

None of these fit, try job 8:

```
 1111
     444
         8888
```

Job 9 does not fit.

This is the final, optimal schedule that maximizes the number of people that want to use of the room. The runtime is simple O(nlgn) to sort, and then O(n) to run through the finishing times, making this algorithm O(nlgn) overall.

Greedy Algorithms vs. Dynamic Programming

Greedy algorithms and dynamic programming are similar; both generally work under the same circumstances although dynamic programming solves subproblems first. Often both may be used to solve a problem although this is not always the case.

Consider the 0-1 knapsack problem. A thief is robbing a store that has items 1..n. Each item is worth v_i dollars and weighs w_i pounds. The thief wants to take the most amount of loot but his knapsack can only hold weight W. What items should he take?

This problem has optimal substructure.

Dynamic programming: We showed that we can solve this in O(nW) time, gives optimal value.
Greedy algorithm: Take as much of the most valuable item first. Does not necessarily give optimal value! (Homework problem to show this).

A simpler version of the knapsack problem is solved optimally by this greedy algorithm:

Consider the fractional knapsack problem. This time the thief can take any fraction of the objects. For example, the gold may be gold dust instead of gold bars. In this case, it will behoove the thief to take as much of the most valuable item per weight (value/weight) he can carry, then as much of the next valuable item, until he can carry no more weight.

Total value using this strategy and the above example is 8 of item 1 and 2 of item 2, for a total of $124.

Moral: Greedy algorithm sometimes gives optimal solution, sometimes not, depending on the problem. Dynamic programming, when applicable, will typically give optimal solutions, but are usually tricker to come up with and sometimes trickier to implement.

Huffman Codes

Huffman codes are frequently used for data compression. Huffman encoding is one of the earliest data compression algorithms; popular programs like Pkzip and Stuffit use their own techniques but are based on the original schemes such as Huffman or LZW. Compression is useful for archival purposes and for data transmission, when not much bandwidth is available.

Idea: Let's say you want to compress your file, and your file only contains 6 characters, ABCDEF. If you store these using an 8-bit ascii code, you will need space 8N bits, where n is the numbers of characters in your file. If n=1000, this is 8000 bits.

One way to do better: Since you only have six characters, you can represent these in fewer bits. You really only need three bits to represent these characters:

000	A
001	B
010	C
011	D
100	E
101	F

Immediately, we can reduce the storage necessary to 3N bits. If n=1000, this is 3000 bits.

What if we count the frequency of each letter and have something like the following?

A: 45%	B:10%	C:10%	D:20%	E:10%	F:5%

Now if we assign the following codes:

0	A	45%
100	B	10%
101	C	10%
111	D	20%
1100	E	10%
1101	F	5%

Notice we need 4 bits to represent F now, but the most common character, A, is represented by just 1 bit.
Also note that since we are using a variable number of bits, this messes up the counting somewhat. I can't use 110 to represent D, since then if a 110 popped up we can't tell if this is referring to D or E, since both start with 110 – we need unique prefixes.

For example to store ABFA is: 010011010

Now, to store 1000 characters with this frequency and encoding scheme requires:
450*1 + 3*100 + 3*100 + 3*200 + 4*100 + 4*50 = 2250 bits. 25% improvement over before.

Question: We can find frequencies easily in O(n) time by linearly scanning and counting up the number of occurences of each token. How do we determine what codes should be assigned each character?

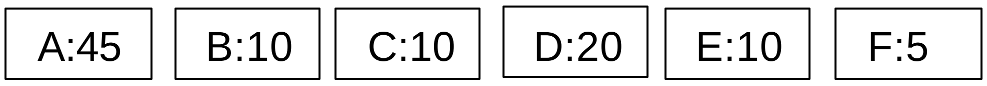
Idea: Count up frequencies, and build up trees by extracting minimum.

```
Huffman(S,f)              ; S = string of characters to encode.
                          ; F=frequences of each char
    n← |S|                ; Make each character a 'node'
    Q ← S                 ; Priority queue using the frequency as key
    for j ← 1 to n-1 do
        z← Allocate-Node()
        x← left[z] ← Extract-Min(Q)
        y← right[z] ← Extract-Min(Q)
        f[z] ← f[x]+f[y]                  ; update frequencies
        Insert(Q,z)
    return Extract-Min(Q)
```

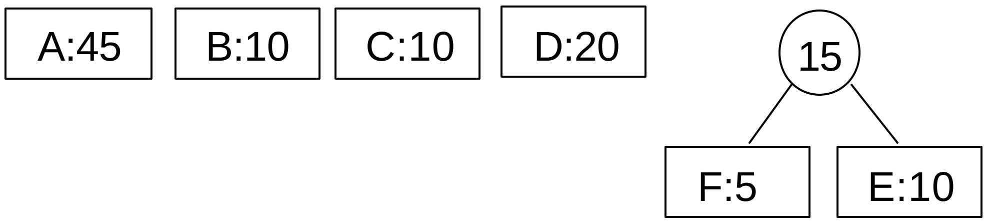
Example:

First make each character a node by itself.

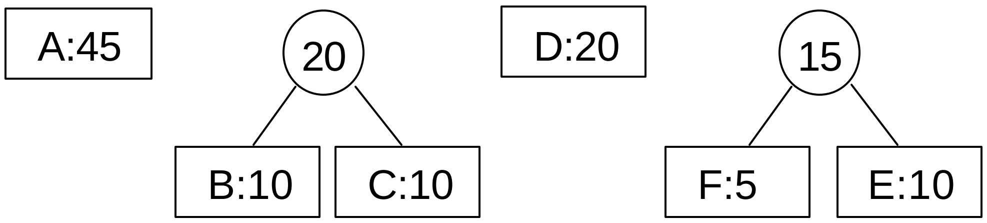
A: 45%	B:10%	C:10%	D:20%	E:10%	F:5%



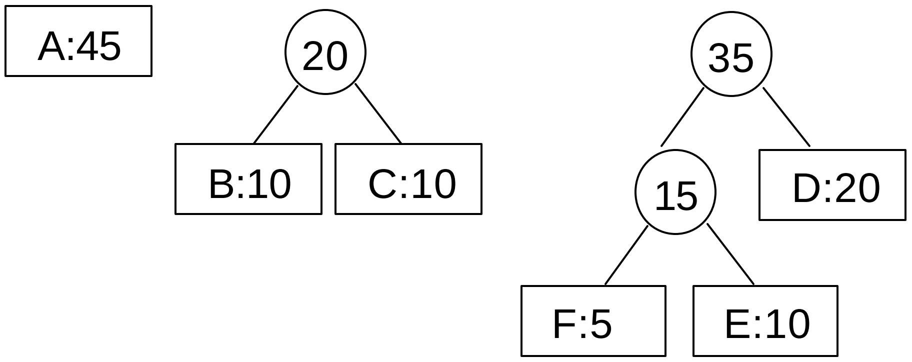
Extract the minimum and join together. These will end up as leaves farther down the tree. Mins=F and E. Put sum as the new frequency. Put minimum to the left.



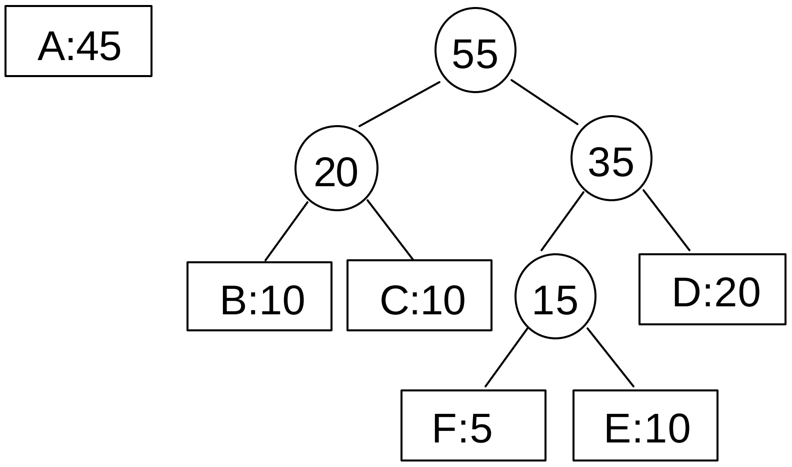
Repeat process: Extract min, B and C, and put sum as new frequency:



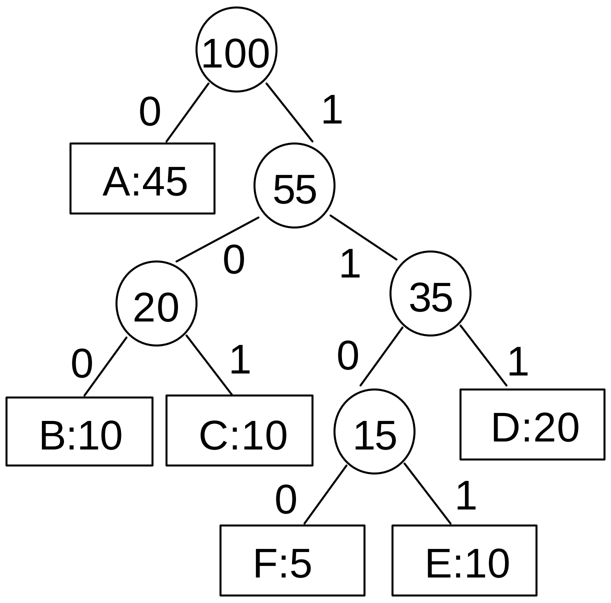
Repeat process: Extract D and 15 as min, put sum as new frequency:



Repeat process: Extract 20 and 35 as min, put sum as new frequency:



Finally, combine with A and assign 0,1 to edges:



Travel down tree to get the code:

A	0
B	100
C	101
D	111
E	1100
F	1101

We're done!

Correctness: This we will not prove, but the idea is to put the rarest characters at the bottom of the tree, and build the tree so we will not have any prefixes that are identical (guaranteed in tree merging step). By putting the rarest characters at the bottom, they will have the longest codes and the most frequent codes will be at the top. This algorithm produces an optimal prefix code, but not necessarily the most optimal compression possible.

Runtime of Huffman's algorithm: If Q implemented as a binary heap then the extract operation takes lgn time. This is inside a for-loop that loops n times, resulting in O(nlgn) runtime.