Greedy Algorithms II

Greedy algorithms tend to be difficult to teach since different observations lead to correct greedy algorithms in different situations. Pedagogically, it’s somewhat difficult to clearly group different greedy problems. Thus, when teaching greedy algorithms, we tend to simply look at examples and try to point out hallmarks from those examples that may show up in other places. If you want to study the theory of greedy algorithms, look at Matroids (http://en.wikipedia.org/wiki/Matroid). In this lecture, I won’t get so formal. We’ll just look at a few greedy algorithms and try to make observations about them.

One common observation amongst many greedy algorithms that I’ll make here is the “replacement strategy.” The proof of correctness of many greedy algorithms goes along these lines: “the solution found by the greedy algorithm is at least as good as anything a competing solution might find because…” Basically, if you try to replace a portion of the greedy algorithm with an alternate choice, the resulting solution can be no better than the one the greedy algorithm found. Typically, this sort of observation along with mathematical induction is used to prove that at no point in time does a greedy algorithm “fall behind” any possible competing choice. It’s worthwhile to try to sketch out this sort of argument for a greedy solution you may come up with.

Huffman Coding

The idea behind Huffman coding is to find a way to compress the storage of data using variable length codes. Our standard model of storing data uses fixed length codes. For example, each character in a text file is stored using 8 bits. There are certain advantages to this system. When reading a file, we know to ALWAYS read 8 bits at a time to read a single character. But as you might imagine, this coding scheme is inefficient. The reason for this is that some characters are more frequently used than other characters. Let's say that the character 'e' is used 10 times more frequently than the character 'q'. It would then be advantageous for us to use a 7 bit code for e and a 9 bit code for q instead because that could shorten our overall message length.

Huffman coding finds the optimal way to take advantage of varying character frequencies in a particular file. On average, using Huffman coding on standard files can shrink them anywhere from 10% to 30% depending to the character distribution. (The more skewed the distribution, the better Huffman coding will do.)

The idea behind the coding is to give less frequent characters and groups of characters longer codes. Also, the coding is constructed in such a way that no two constructed codes are prefixes of each other. This property about the code is crucial with respect to easily deciphering the code.

Building a Huffman Tree

The easiest way to see how this algorithm works is to work through an example. Let's assume that after scanning a file we find the following character frequencies:

Character	Frequency
'a'	12
'b'	2
'c'	7
'd'	13
'e'	14
'f'	85

Now, create a binary tree for each character that also stores the frequency with which it occurs.

The algorithm is as follows: Find the two binary trees in the list that store minimum frequencies at their nodes. Connect these two nodes at a newly created common node that will store NO character but will store the sum of the frequencies of all the nodes connected below it. So our picture looks like follows:

```
       9              12 'a'   13 'd'   14 'e'          85 'f'
    /    \
  2 'b'  7 'c'
```

Now, repeat this process until only one tree is left:

```
        21
      /    \
     9     12 'a'        13 'd'         14 'e'          85 'f'
    /    \
  2 'b'  7 'c'
```

```
        21                 27
      /    \             /      \
     9     12 'a'     13 'd'    14 'e'       85 'f'
    /    \
  2 'b'  7 'c'
```

```
                48
            /           \
        21                27
      /    \             /      \
     9     12 'a'     13 'd'    14 'e'                 85 'f'
    /    \
  2 'b'  7 'c'
```

```
                        133
                      /       \
                 48              85 'f'
             /          \
          21               27
        /   \            /    \
      9    12 'a'    13 'd'   14 'e'
     /   \
  2 'b'  7 'c'
```

Once the tree is built, each leaf node corresponds to a letter with a code. To determine the code for a particular node, walk a standard search path from the root to the leaf node in question. For each step to the left, append a 0 to the code and for each step right append a 1. Thus for the tree above we get the following codes:

Letter	Code
'a'	001
'b'	0000
'c'	0001
'd'	010
'e'	011
'f'	1

Why are we guaranteed that one code is NOT the prefix of another? (Because for a string and any of its prefixes, the prefix can not be a leaf node in the tree - but, letters are only assigned to leaf nodes in this scheme.)

As an exercise, find a set of valid Huffman codes for a file with the given character frequencies:

Character	Frequency
'a'	15
'b'	7
'c'	5
'd'	23
'e'	17
'f'	19

Calculating Bits Saved

All we need to do for this calculation is figure out how many bits are originally used to store the data and subtract from that how many bits are used to store the data using the Huffman code.

In the first example given, since we have six characters, let's assume each is stored with a three bit code. Since there are 133 such characters, the total number of bits used is 3*133 = 399.

Now, using the Huffman coding frequencies we can calculate the new total number of bits used:

Letter	Code	Frequency	Total Bits
'a'	001	12	36
'b'	0000	2	8
'c'	0001	7	28
'd'	010	1	39
'e'	011	14	42
'f'	1	85	85
Total			238

Thus, we saved 399 - 238 = 161 bits, or nearly 40% storage space. Of course there is a small detail we haven't taken into account here. What is that?

Huffman Coding is an Optimal Prefix Code

Of all prefix codes for a file, Huffman coding produces an optimal one. In all of our examples from class on Monday, we found that Huffman coding saved us a fair percentage of storage space. But, we can show that no other prefix code can do better than Huffman coding.

First, we will show the following:

Let x and y be two characters with the least frequencies in a file. 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.

Here is how we will prove this:

Assume that a tree T stores an optimal prefix code. Let and characters a and b be sibling nodes stored at the maximum depth of the tree. We will show that we can create T' with x and y as siblings at the lowest depth of the tree such that the number of bits used for the coding with T' is the same as with T. (Let f(a) denote the frequency of the character a. Without loss of generality, assume $f(x) \leq f(y)$ and $f(a) \leq f(b)$. It also follows that $f(x) \leq f(a)$ and $f(y) \leq f(b)$. Let h be the height of the tree T. Let x have a depth of d_x in T and y have a depth of d_x in T.)

Create T' as follows: swap the nodes storing a and x, and then swap the nodes storing b and y. Now, we have that the depth of x and y in T' is h, the depth of a is d_x and the depth of b is d_y in T'.

Now, let's calculate the change in the number of bits used for the coding with tree T' with the coding in tree T. (Note: Since all other codes remain unchanged, we only need to analyze the total number of bits it takes to code a, b, x and y.)

# bits for tree T (for a,b,x and y) = $hf(a) + hf(b) + d_xf(x)\ \ d_yf(y)$

# bits for tree T' (for a, b, x, and y) = $d_xf(a) + d_yf(b) + hf(x) + hf(y)$.

Difference =

$hf(a) + hf(b) + d_xf(x)\ \ d_yf(y) - (d_xf(a) + d_yf(b) + hf(x) + hf(y)) =$
$hf(a) + hf(b) + d_xf(x)\ \ d_yf(y) - d_xf(a) - d_yf{*}b) - hf(x) - hf(y) =$
$h(f(a) - f(x)) + h(f(b)-f(y)) + d_x(f(x) - f(a)) + d_y(f(y) - f(b)) =$
$h(f(a) - f(x)) + h(f(b)-f(y)) - d_x(f(a) - f(x)) - d_y(f(b) - f(y)) =$
$(h - d_x)(f(a) - f(x)) + (h - d_y)(f(b) - f(y))$

Notice that all four of the terms above must be non-negative since we know that $h \geq d_x$, $h \geq d_y$, $f(a) \geq f(x)$, and $f(b) \geq f(y)$. Thus, it follows that this difference must be 0. Thus, the number of bits to used in a code where x and y (the two characters with lowest frequency) are siblings at maximum depth of the coding tree is optimal.

In layman's terms, give me what you think is an optimal coding tree, and I can create a new one from it with the two nodes corresponding to low frequencies at the bottom of the tree.

To complete the proof, you'll notice that by construction, Huffman coding ALWAYS makes sure that the nodes with the lowest frequencies are at the bottom of the coding tree, all the way through the construction. (You can't find any pair of nodes for which this isn't true.) Technically, to carry out the proof, you'd use induction, but we'll skip that for now...

Up and Down – 2014 Google Code Jam Round 2

The problem is as follows:

You are given a sequence of distinct integers **A** = [$\mathbf{A_1}$, $\mathbf{A_2}$, ..., $\mathbf{A_N}$], and would like to rearrange it into an *up and down* sequence (one where $\mathbf{A_1} < \mathbf{A_2} < ... < \mathbf{A_m} > \mathbf{A_{m+1}} > ... > \mathbf{A_N}$ for some index **m**, with m between 1 and **N** inclusive).

The rearrangement is accomplished by swapping two ***adjacent*** elements of the sequence at a time. Predictably, you are particularly interested in the minimum number of such swaps needed to reach an *up and down* sequence.

When studying sorting algorithms, we prove that the average case run-time of algorithms like bubble sort must be $\theta(n^2)$ for sorting n items because each swap removes at most one inversion from the array and the average array has about $\frac{1}{4}n^2$ inversion. Thus, our analysis for this problem should focus on a similar count – the number of inversions in the array. Note: an inversion in an array is a pair of items such that i < j but a[i] > a[j]. Namely, when looking at the pair of items in relation to each other, they are "out of order." We can count inversions by looping through each pair of elements in an array and adding 1 if a pair is inverted. In a random array, we expect about half of all pairs to be inverted.

The difference in this problem is that there is some split point (we don't know where) and to the left of that split point there are no inversions and to the right of it, if we consider the list in reverse order, there are no inversions. Given the type of move suggested, this means that we can remove at most one inversion per swap.

Thus, our proof of correctness looks like this:

1) We must have the array ordering be one of these possible sequences. (UUUDD or UDDDD, etc., where U represents a pair going up and D a pair going down.)

2) If we consider all of these possibilities and look at the minimum number of inversions (with respect to this specific problem), then it'll be impossible to have any competing set of swaps beat that number of inversions.

3) Finally, if we know the number of inversions in a subarray there always exists a way to remove one inversion per swap (otherwise the array would be sorted!) until no inversions exist.

Once we sketch this out, our algorithm is fairly easy to come up with.

1) Try each possible location as the place for the maximum element.
2) For each of these locations, count the number of inversions in the appropriate subarrays.
3) Take the minimum of all values calculated in step 2.

Greedy Algorithm Principle: Be Lazy!!!

A common pattern that appears in some problems with greedy solutions is the "be lazy" principle. For some types of problems, if you are trying to minimize a certain type of movement, the best strategy is not to move at all, until you are FORCED to do so.

We will provide three problems that can be solved using the "be lazy" principle:

1) The Toilet Seat Dilemma
2) Inoculating Bunnies (courtesy of Matt Fontaine)
3) Missing Parentheses (from Top Coder)

<u>The Toilet Seat Dilemma</u>

As everyone knows, the toilet seat can take two positions: up or down. Often times, some individuals complain if the seat isn't in their desired position when they go to use the toilet. Some people believe that all users of the toilet should "fix the seat to where they want it, use the toilet, and then put the seat down." Others believe that the algorithm ought to be "fix the seat to where they want to it and then use the toilet." Essentially, the latter algorithm specifies no further action after using the toilet. It uses the principle "be lazy" and only specifies a movement of the toilet seat when absolutely necessary. A question we might ask is which of these two algorithms is optimal? Of course, we must now define optimality. A clear quantitative measure of optimality might be attempting to minimize the number of times the seat position is changed.

Consider that we have a sequence of users of the toilet who need the following configurations(D = down, U = up):

U, D, D, U, U, U, U, D, U, D, U

Assume that the seat starts down.

If we use the first algorithm, we would have the following number of movements:

2 + 0 + 0 + 2 + 2 + 2 + 2 + 0 + 2 + 0 + 2 = 14

If we use the second algorithm we would have the following number of movements:

1 + 1 + 0 + 1 + 0 + 0 + 0 + 1 + 1 + 1 + 1 = 7

The second algorithm is twice as good as the first for this example!

Ideally, what we'd like to do is prove that the second algorithm always produces the optimal result. We do it as follows:

Consider any competing schedule for changing the seat. At the point in time where our greedy schedule changes the seat for the first time, the competing schedule must have flipped the seat at least once because we only move the seat when we're forced to. Thus, our schedule is at least as good as any competing schedule at first. Using mathematical induction, we can show that the greedy schedule of toilet flips will always be at least as good as any competing schedule. Basically, the greedy algorithm proposed takes advantage of the fact that if the toilet seat is used in the same position for several uses in a row, no movement is made. The first algorithm, or any other algorithm can't beat this.

Another way to prove the optimality is to note that every time the substring DU or UD is seen in the sequence of uses of the toilet, at least one seat change is required. Our greedy algorithm uses exactly this many seat changes, thus, it must be optimal. This is a similar argument to the one made about the run-time of any sort of n elements that only swaps adjacent elements being $\Omega(n^2)$ since at least one step must occur for each inversion of the array.

<u>Inoculating Bunnies</u>

This is a problem UCF coach Matt Fontaine came up with that also uses the "be lazy" principle. (Matt's version is slightly different; I changed it due to a preference of my wife's.) Imagine a train going from west to east, starting at $x = 0$, and moving in the positive x direction. Bunnies get on and off the train at various spots, with the i^{th} bunny boarding the train at $x = a_i$ and getting off the train at $x = b_i$, with $a_i < b_i$. The World Bunny Organization (WBO) would like to inoculate all of the bunnies from a terrible virus and knows the schedules of all the bunnies getting on the train, which can disburse vaccinations instantaneously in the air throughout the cabin of the train. However, using this disbursion method is expensive, thus the WBO would like to use it as infrequently as possible. Whenever it is used, all the bunnies on the train get inoculated and no side-effect is incurred when a bunny gets inoculated more than once. Determine the minimum number of times vaccinations must be disbursed in order to ensure that all of the bunnies that board the train are properly inoculated from the deadly virus.

If we are being lazy, we'll never disburse the vaccination until we're absolutely forced to do so, right before the first bunny ever gets off the train. After that point, we simply repeat the algorithm. There's simply no point in disbursing a vaccination any time earlier than necessary. Any competing schedule that tries to vaccinate earlier runs the risk of not vaccinating a bunny that hasn't yet board the train. (Imagine that the first bunny gets off the train at $x = 20$. If we vaccinate when $x = 15$, and there was a bunny that boarded the train at $x = 18$, then we haven't vaccinated that bunny with a single disbursion. If we waited until $x = 19$ (or whatever location is defined as the latest possible time that we could disburse the vaccination before that first bunny got off the train), then we vaccinate the most number of bunnies possible while still making sure we vaccinate that first bunny. In a similar manner as the previous problems, we see that no competing schedule of inoculating bunnies can beat this greedy algorithm of "being lazy."

Missing Parentheses

Problem: Given a set of parentheses, determine the minimum number of parentheses that have to be inserted into the sequence to make it well formed.

The rule for parentheses is: every close must match to a previously seen open. Thus, using the "being lazy" principle, we get the following greedy algorithm:

Reading through our string left to right, don't add a parenthesis unless we are forced to do so.

So, we keep a running tally of total open parentheses seen minus closed parentheses seen as we read in the expression. If this count ever goes BELOW 0, we must add an open parentheses right BEFORE the count hits -1.

If we get to the end of our string and our count is positive, then add the necessary remaining close parentheses.

Now, we must prove that this algorithm works. Note that any competition solution necessarily adds in an open parenthesis for each open parenthesis mentioned in this algorithm. If it doesn't, then there will be a prefix of the string that has more closes than opens. Thus, any competing solution has as many opens as this solution does. Similarly, this solution only adds close parentheses when absolutely necessary, any fewer added would result in a string with more opens than closes. This proves that our algorithm is optimal.