1 Greedy algorithms

In algorithm design, a *greedy algorithm* is one that breaks a problem down into a sequence of simple steps, each of which is chosen such that some measure of the "quality" of the intermediate solution increases after each step.

A classic example of a greedy approach is navigation in a k-dimensional Euclidian space. Let $\mathbf{y}$ denote the location of our destination, and let $\mathbf{x}$ denote our current position, with $\mathbf{x}, \mathbf{y} \in \mathbb{R}^k$. The distance remaining to our destination is then simply the Euclidian distance between $\mathbf{x}$ and $\mathbf{y}$:

$$L_2(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2} \ , \tag{1}$$

and the greedy step to take at each moment in our journey is the one that minimizes $L_2(\mathbf{x}, \mathbf{y})$.[1]

Note that this approach is only *optimal*—that is, guarantees that we reach the target—when the space is smooth, with no impassable regions of space. It is not hard to construct examples in a 2-dimensional space containing impassable regions for which the greedy solution can fail. A trivial case is the following: consider traveling by car or foot from Boulder, CO to London, England. Ignoring the possibility of taking a ferry, there is no land path that connects North America to England, and thus there is no path.

Here's a more subtle case: consider traveling by car or foot from Miami, FL to Merida, Mexico (on the tip of the Yucatan Peninsula, across the Gulf of Mexico from Florida). The problem here is not that a path does not exists, because a path that first moves north along the peninsula and then follows the edge of the Gulf of Mexico will get us there. But the greedy approach cannot find this path because it first increases the distance to the destination, which is forbidden by the greedy choice. Instead, the greedy approach would take us south to the end of the Florida peninsula. Once there, we would find the part of the beach that is closest to Merida and then throw up our hands claiming that there is no new move that would not increase the remaining distance.

When a problem has these kinds of "local optima," from which no greedy move can further improve the quality of our position, we must use other means to solve it because a greedy approach can get stuck. (If there exists even a single input for which a greedy approach fails, then the greedy algorithm is not a correct algorithm.) But, many problems do have the particular kind of structure, if you can see it, that allows a correct greedy solution.

[1] I'm using L_2 to denote the Euclidian distance here because the general form of this distance measure is L_p. In many problems, it is convenient to choose $p \neq 2$, which can produce a distance metric with slightly different mathematical properties.

1.1 Structure of a greedy algorithm

Greedy algorithms are possibly the most common type of algorithm because they are so simple to formulate. But, for the greedy approach to be correct (and fulfill the promise to never fail on any input), the problem must have the following mathematical properties:

1. Every solution to the problem can be assigned or translated in a numerical value or score. Let x be a candidate solution and let score(x) be the value assigned to that solution.

2. The "best" (optimal) solution has the highest value among all solutions (in the case of maximization; lowest, in the case of minimization); i.e., it is the global optimum, and this solution contains optimal solutions to all of its subproblems ("optimal substructure" property).

3. A solution can be constructed incrementally (and a partial solution assigned a score) without reference to future decisions, past decisions or the set of possible solutions to the problem ("greedy choice" property).

4. At each intermediate step in constructing or finding a solution, there are a set of options for which piece to add next.

A greedy algorithm always chooses the incremental option that yields the largest improvement in the intermediate solution's score.

As with any algorithm, once we have formulated our greedy approach, we must prove that it is correct, i.e., always finds the correct solution given any arbitrary input. Similarly, we must also analyze its time and space usage. These three parts (correctness, time usage, space usage) are what is typically meant by "analyzing" an algorithm

1.2 Huffman codes

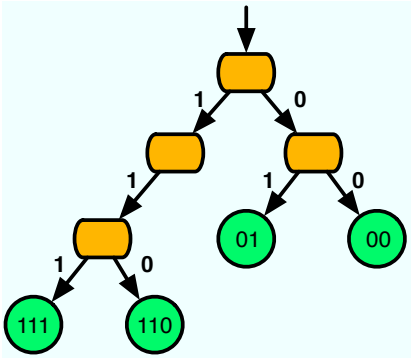
Our primary example of a greedy algorithm will be a problem from *compression*, specifically Huffman encoding.

Here is the problem description. Let Σ, with $|\Sigma| = n$, denote the input alphabet, a set of distinct "symbols" that can be used to construct a string of symbols or message. We are given a message or file x that is an arbitrary sequence of symbols drawn from Σ. The problem is to identify an encoding (mapping) f of words from Σ to codewords in a binary alphabet Γ, such that the encoding is *prefix-free* and the encoded message minimal in size. Our goal is to efficiently (computationally) identify the most efficient (compression) encoding of the message.

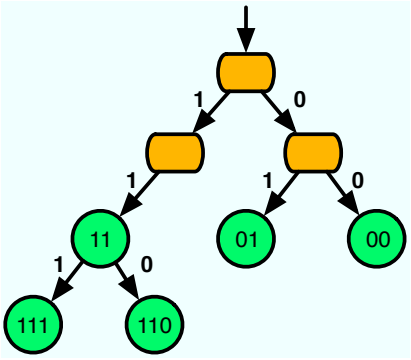
1.2.1 Binary code trees and prefix-free codes

Before moving forward, let us define a little more carefully what these requirements mean. A binary code is simply a set of strings (set of codewords) all drawn from the binary alphabet $\Sigma_{01} = \{0, 1\}$. A prefix-free code is a set of codewords such that no codeword is a prefix of any other. That is, if x is in the code, then there can exist no codeword $y = x + z$, where $+$ denotes concatenation. Otherwise, x is a prefix of y. If and only if a (binary) code is prefix-free, can it be represented by a (binary) tree in which codewords are only located at the bottom of the tree, at the leaves.[2,3]

To illustrate these ideas, below are two decoding trees, one that represents a prefix-free code, with codewords $\{111, 01, 110, 00\}$, and one that does not, which has codewords $\{11, 111, 01, 110, 00\}$. (Note that the second code is not prefix-free because 11 is a prefix of both 111 and 110.)



code $= \{111, 110, 01, 00\}$



code $= \{11, 111, 110, 01, 00\}$

In the coding trees, codewords are denoted by green circles, with the corresponding codeword contained within and non-codewords are denoted by orange oblongs. To decode a string, we treat the tree like a finite-state automaton: each character we read moves us from one state (node) to another; if we read a 1, we move to the left child; otherwise, we move to the right child. When we reach a codeword node, we know what codeword we have just read, and we can return to the top of the tree to begin reading the next codeword. Clearly, any codeword that sits at an internal node in the tree is a prefix for all codewords in the subtree below it. We want to avoid prefix codes because they induce ambiguity in the decoding.

[2]Note that this is different from the *trie* data structure, which explicitly permits words to be prefixes of each other. Tries are very handy data structures for storing character strings in a space-efficient manner that allows fast lookups based on partial (prefix) matchings.

[3]Also note that this kind of tree is very different from a search tree. In fact, the ordering of the leaves in an encoding is very unlikely to follow any particular pattern.

1.2.2 Cost of coding

The length of a codeword is given by the depth of its associated node. Thus, a good compression scheme is equivalent to a small tree, and the best compression is achieved by a minimal tree.[4] However, the total "cost" of an encoding scheme is not just a function of the size of the tree, but also a function of the frequency of the words we encode. Let f_i be the frequency in the message x of the ith codeword in the original message. A minimal code minimizes the function

$$\text{cost}(x) = \sum_{i=1}^{n} f_i \cdot \text{depth}(i) \ .$$

In 1948, Claude Shannon proved that the theoretical lower bound on the cost per word (in bits) using a binary encoding is given by the entropy $H = -\sum_{i=1}^{n} p_i \log_2 p_i$, where $p_i = f_i/n$. Notice that the definition of entropy is similar the cost function we wrote down, when $\text{depth}(i) = \log_2 f_i$. This is not an accident.

1.2.3 Huffman's algorithm

In 1952 David Huffman developed a greedy algorithm that produces an encoding that minimizes this function and gets as close to Shannon's bound as possible for a finite-sized string. The key idea of Huffman's algorithm is remarkably compact: merge the two least frequent "characters" and recurse.

Huffman encoding begins by first tabulating the frequencies f_i of each word in x; this can be done quickly by constructing a histogram. We then create the n leaves of the coding tree. At the i leaf, we store the ith word of the input alphabet Σ and its frequency f_i in the input message x. Let Γ denote the set of symbols we are currently working with. Initially $\Gamma = \Sigma$. At the kth step of the algorithm, we select the two words in Γ with smallest frequencies, f_i and f_j. We create a new word $n + k$ with frequency $f_{n+k} = f_i + f_j$, and make it the parent of i and j. We then update Γ by removing words i and j and adding word $n + k$. The algorithm halts when $|\Gamma| = 1$, i.e., when no pair of symbols remains, and returns to us the constructed tree.

Because the size of our intermediate alphabet Γ decreases by one at each step, the algorithm must terminate after exactly $n - 1$ steps. Implementing the algorithm simply requires a data structure that allows us to efficiently find the two symbols in Γ with the smallest frequencies. This is typically done using a *priority queue* or *min heap* (Chapter 6). Each find, merge, and add cycle takes $O(\log n)$ time and there are n such operations; thus, the running time is $O(n \log n)$.[5] (How much space does it take?)

[4] "A" minimal tree, rather than "the" minimal tree, as there are often multiple smallest trees for the same set of codewords.

[5] It is possible to run this sequence in $O(n)$ time using a pair of cross-liked queues; however, this version is more complicated to implement and assumes the word frequencies f_i have already been computed and sorted, which takes $O(n \log n)$ time.

1.2.4 A cute example

Here is a cute "self-descriptive" Huffman example from Lee Sallows[6]

> This sentence contains three a's, three c's, two d's, twenty-six e's, five f's, three g's, eight h's, thirteen i's, two l's, sixteen n's, nine o's, six r's, twenty-seven s's, twenty-two t's, two u's, five v's, eight w's, four x's, five y's, and only one z.

To keep things simple, we will ignore capitalization, the spaces (44), apostrophes (19), commas (19), hyphens (3) and the one period in the sentence; instead, we will focus on encoding the letters alone. The frequencies of the 26 letters are the following.

A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

Below is the encoding tree produced by applying Huffman's rule to this histogram: after 19 merges, all 20 characters have been merged and the history of merges gives the encoding tree. (Suppose there is a tie as to which pair of characters to merge; what does this tell us about the uniqueness of the Huffman code?)

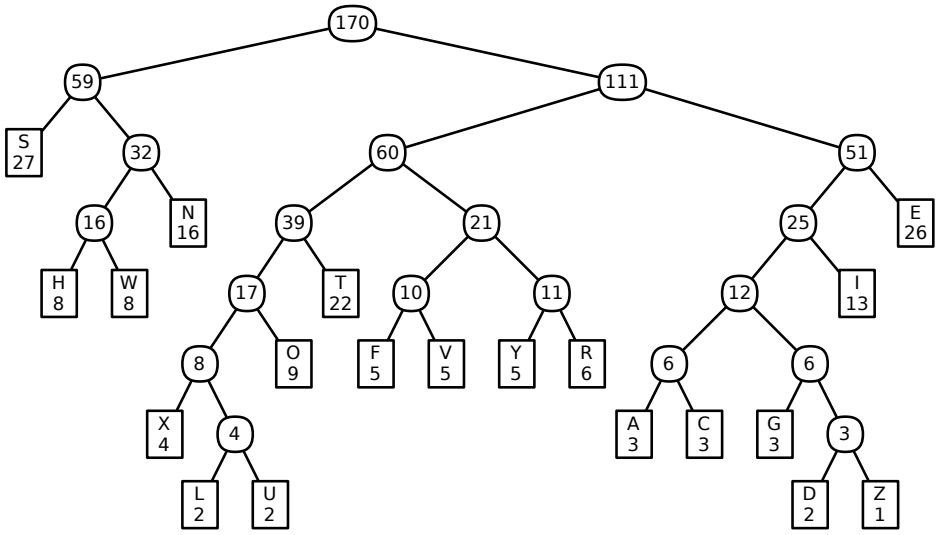
To read the encoding of a character, place a 0 on each left-branch and a 1 on each right-branch, then write the sequence of 1s and 0s backwards as you read up the tree. To decode an encoded letter, do the reverse: start at the root, and take the left-right path given by the encoding down the tree to arrive at the decoded character. For example, the encoding of the letter "a" is 110000. In our example, the encoded message is 661 bits long. Here is a table showing how we arrive at that number, with each input symbol, its frequency f_i in the input, the length of its encoding d_i and the total cost for encoding those symbols $f_i d_i$. The total message length is just $\sum_i f_i d_i$.

	A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
f_i	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
d_i	6	6	7	3	5	6	4	4	7	3	5	5	2	4	7	5	4	6	5	7
$f_i d_i$	18	18	14	78	25	18	32	54	14	48	45	30	54	88	14	25	32	24	25	7

At home exercise: verify that this tree is correct.[7]

[6] A. K. Dewdney. Computer recreations. *Scientific American*, October 1984. Other examples appeared a few years earlier in some of Douglas Hofstadter's columns. My credit goes to Jeff Erickson, who also produced the figure below.

[7] This tree is not, in fact, correct. The correct tree produces an encoding that uses only 649 bits.



1.2.5 Huffman codes are optimal

Now let's prove that Huffman encoding is optimal.

Lemma 1: Let x and y be the two least frequency characters (with ties broken arbitrarily). There is an optimal code tree in which x and y are siblings and their parent is the largest depth of any leaf.

Proof: Let T be an optimal code tree with depth d. Because T is a "full" binary tree (every tree node has either 0 or 2 children), it must have at least two leaves at depth d that are siblings. Assume that these leaves are not x and y, but rather some other pair a and b.

Let T' by the code tree if we swapped x and a. The depth of x increases by some amount at most Δ, and the depth of a decreases by the same amount. Thus,

$$\text{cost}(T') = \text{cost}(T) - (f_a - f_x)\Delta \ .$$

But, by assumption, x is one of the two least frequency characters while a is not. This implies $f_a \geq f_x$, and thus swapping x and a does not increase the total cost of the code. Since T was an optimal code tree, swapping x and a cannot decrease their cost. Thus, T' is also an optimal code tree (and $f_a = f_x$). Similarly for swapping y and b. Thus, doing both swaps yields an optimal tree with x and y as siblings and at maximum depth. □

We can now prove that the greedy encoding rule of Huffman is optimal.

Theorem 1: Huffman codes are optimal prefix-free binary codes.

Proof: If the message has only one or two different characters, the theorem is trivially true.

Otherwise, let $f_1, \ldots, f_n$ be the frequencies in the original message. Without loss of generality, let f_1 and f_2 be the smallest frequencies. When we recurse, we create a new frequency at the end of the list, e.g., $f_{n+1} = f_1 + f_2$. By Lemma 1, we know that the optimal tree T has characters 1 and 2 as siblings.

Let T' be the Huffman tree for $f_3, \ldots, f_{n+1}$. The inductive hypothesis implies that T' is an optimal code tree for this smaller set of frequencies. In the final optimal code tree T, we replace the frequency f_{n+1} with an internal node that has 1 and 2 as children.

Now, we can write the cost of T in terms of the cost of T'; let d_i be the depth of node i:

$$\begin{aligned}
\text{cost}(T) &= \sum_{i=1}^{n} f_i d_i \\
&= \left(\sum_{i=3}^{n+1} f_i d_i \right) + f_1 d_1 + f_2 d_2 - f_{n+1} d_{n+1} \\
&= \text{cost}(T') + f_1 d_1 + f_2 d_2 - f_{n+1} d_{n+1} \\
&= \text{cost}(T') + (f_1 + f_2) d_T - f_{n+1}(d_T - 1) \\
&= \text{cost}(T') + f_1 + f_2
\end{aligned}$$

Where we arrive at the last line by recalling that $f_{n+1} = f_1 + f_2$, multiplying everything out and canceling like terms. (Or, to put it another way, we recall that the depth d_T relative to T' is 1.) Thus, we find that minimizing the cost of T is the same as minimizing the cost of T', and attaching the leaves for 1 and 2 to the leaf in T' labeled $n + 1$ gives an optimal code tree for the original message. And, by induction, we have proved the theorem. □

2 Next Time

1. Read Chapter 22 (Graph Algorithms)
2. Reminder: Problem Set 1 due today, Monday February 4 by 11:59pm via email

3 Two more greedy algorithms

3.1 Insertion sort is an optimal but inefficient greedy algorithm

A greedy algorithm can be optimal, but not efficient. To illustrate this, we will consider the behavior of *insertion sort*. Recall that insertion sort takes $\Theta(n^2)$ time to sort n numbers and that we know a number of sorting algorithms that are more efficient, taking only $O(n \log n)$ time.

Insertion sort is a simple loop. It starts with the first element of the input array A. For each subsequent element j, it then inserts $A[j]$ into the sorted list $A[1..j-1]$.

```
INSERTION-SORT(A)
  for j=2 to n
     // assert: A[1..j-1] is sorted
     insert A[j] into the sorted sequence A[1..j-1]
     // assert: A[1..j] is sorted
}}
```

To see that insertion sort is correct, we observe that the following loop invariant, i.e., a property of the algorithm (either its behavior or its internal state) that is true each time we begin or end the loop.

Note that at the start of the `for` loop, the subarray $A[1..j-1]$ contains the original elements of $A[1..j-1]$ but now in sorted order. When $j = 2$ ("initialization") this fact is true because a list with one element is, by definition, sorted. When we insert the jth item into the subarray $A[1..j-1]$, we do so in a way that $A[1..j]$ is now sorted; thus, if the invariant is true at the beginning of the loop, it will also be true at the end of the loop ("maintenance").[8] Finally, when the loop terminates, $j = n$ and we have inserted the last element correctly into the sorted subarray ("termination"); thus, the entire array is sorted.

Now that we know insertion sort is correct, we can show that it is, in fact, an *optimal greedy algorithm*, meaning that (i) at any intermediate step, the algorithm always makes the choice that increases the quality of the intermediate state (greedy property) and (ii) it returns the correct answer upon termination (optimum behavior).

To begin, we first define a score function that allows us to build sorted sequences one element at a time. Clearly, if A is already sorted, score(A) must yield a maximal value, but it must also give partial credit if A is partially sorted and that credit should be larger the more sorted A is.

[8] For those of you unfamiliar with induction, this is a verbal "proof by induction" using the loop invariant. Loop invariants are nice precisely because they make proofs by induction easy.

A sufficient score function is to count the number of sequential comparisons that violate the sorting requirement, i.e.,

$$\text{score}(A) = \sum_{i=1}^{n-1} (A[i] \leq A[i+1]) \quad ,$$

where we assume that the comparison operator is a binary function that returns 1 if the comparison yields true and 0 if it yields false. By definition, this property is true for all i in a fully sorted array, i.e., $A[1] \leq A[2] \leq \cdots \leq A[n]$, and so a fully sorted sequence will receive the maximal score of $n-1$. A sequence in reverse order will receive the lowest score of 0, and every other sequence can be assigned something between these two extremes.

With the score function selected, let's analyze the behavior of insertion sort under this function. An intermediate solution here is the partially sorted array. Given such an array, our "local" move is to take one element $A[j]$ and insert it into the subarray $A[1..j-1]$. Not all choices of where within $A[1..j-1]$ we put $A[j]$ will lead to a sorted list upon termination, and most of the possible choices of where to insert $A[j]$ are suboptimal.

Recall our loop invariant from above. For any input sequence A, when the loop initializes, it has $\text{score}(A) \geq 0$, where the lower bound is achieved by a strict reverse ordering. Because the sorted subarray's size grows by 1 each time we pass through the loop, so too does the number of correct sequential comparisons. That is, our loop invariant is equivalent to $\text{score}(A) \geq j-1$ where j is the loop index. And, when the loop completes, $j = n$ and $\text{score}(A) \geq n-1$. Thus, insertion sort is a kind of optimally greedy algorithm.

3.2 Linear storage media

Here's another good example of a simple greedy algorithm. Suppose we have a set of n files and that we want to store these files on a tape, or some other kind of linear storage media.[9] Once we've written these files to the tape, the cost of accessing any particular file is proportional to the length of the files stored ahead of it on the tape. In this way, tape access is very slow and costly relative to either magnetic disks or RAM. Let $L[i]$ be the length of the ith file. If the files are stored on the tape in order of their indices, the cost of accessing the jth file is

$$\text{cost}(j) = \sum_{i=1}^{j} L[i] \quad .$$

[9] Although tape memory is not often used by individuals, it remains one of the most efficient storage media for both very large files and for archival purposes. A linked list can be used to simulate a linear storage medium if the amount of data that can be stored in each node is limited; in fact, this kind of abstraction is precisely how files are stored on magnetic media.

That is, we first have to scan past (which takes the same time as reading) the first $j-1$ files, and then we read the jth file.

If files are requested uniformly at random, then the expected cost for reading one is

$$E[\text{cost}] = \sum_{j=1}^{n} \Pr(j) \cdot \text{cost}(j) = \frac{1}{n} \sum_{j=1}^{n} \sum_{i=1}^{j} L[i] \ .$$

What if we change the ordering of the files on the tape? If not all files are the same size, this will change the cost of accessing some files versus others. For instance, if the first file is very large, then the cost of accessing every other file will be larger by an amount equal to its length. We can formalize and analyze the impact of a given ordering by letting $\pi(i)$ give the index of the file stored at location i on the tape. The expected cost of accessing a file is now simply

$$E[\text{cost}(\pi)] = \frac{1}{n} \sum_{j=1}^{n} \sum_{i=1}^{j} L[\pi(i)] \ .$$

What ordering π should we choose to minimize the expected cost? Intuitively, we should order the files in order of their size, smallest to largest. Let's prove this.

Lemma 2: $E[\text{cost}(\pi)]$ is minimized when $L[\pi(i)] \leq L[\pi(i+1)]$ for all i.

Proof: Suppose $L[\pi(i)] > L[\pi(i+1)]$ for some i. If we swapped the files at these locations, then the cost of accessing the first file $\pi(i)$ increases by $L[\pi(i+1)]$ and the cost of accessing $\pi(i+1)$ decreases by $L[\pi(i)]$. Thus, the total change to the expected cost is $(L[\pi(i+1)] - L[\pi(i)])/n$, which is negative because, by assumption, $L[\pi(i)] \leq L[\pi(i+1)]$. Thus, we can always improve the expected cost by swapping some out-of-order pair, and the globally minimum cost is achieved when the files are sorted. □

Thus, any greedy algorithm that repeatedly swaps out-of-order pairs on the tape will lead us to the globally optimal ordering. (At home exercise: can you bound the expected cost in this case?)

Suppose now that files are not accessed with equal probability, but instead the ith file will be accessed $f(i)$ times over the lifetime of the tape. Now, the total cost of these accesses is

$$\text{total-cost}(\pi) = \sum_{j=1}^{n} \sum_{i=1}^{j} f(\pi(j)) \cdot L[\pi(i)] \ .$$

What ordering π should we choose now? Just as when the access frequencies were the same but the lengths were different we would sort the files by their lengths, if the lengths are all the same but the

access frequencies different, we should sort the files in decreasing order of their access frequencies. (Can you prove this by modifying Lemma 2?) But, what if the sizes and frequencies are both non-uniform? The answer is to sort by the length-frequency ratio L/f.

Lemma 3: total-cost(π) is minimized when $\frac{L[\pi(i)]}{f(\pi(i))} \leq \frac{L[\pi(i+1)]}{f(\pi(i+1))}$ *for all* i.

Proof: Suppose $\frac{L[\pi(i)]}{f(\pi(i))} > \frac{L[\pi(i+1)]}{f(\pi(i+1))}$. The proof follows the same structure as Lemma 2, but where we observe that the proposed swap changes the total cost by $L[\pi(i+1)] \cdot f(\pi(i)) - L[\pi(i)] \cdot f(\pi(i+1))$, which is negative. □

Thus, the same class of greedy algorithms is optimal for non-uniform access frequencies.