

## 1 Greedy algorithms

In algorithm design, a *greedy* approach is a kind of local optimization (“hill climbing”) strategy in which we try to build a good solution by adding the element (or taking the step, or making the change) that yields the largest immediate positive increase. Greedy approaches are among the most common algorithm design techniques. In general, such an approach can be used on any problem with the following structure:

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  $\text{score}(x)$  be the value assigned to that solution. 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.
2. A solution can be built up in pieces, and a partial solution can also be assigned a numerical value or score.
3. At each intermediate step in building up a solution, there are a non-trivial number of options for which piece to add next.

The greedy approach to such a problem is to always choose the option that yields the largest positive (negative if minimizing) difference in the score of the intermediate solution. Greedy approaches do not always yield optimal solutions—they can, and often do, get stuck in “local” optima. (Under what conditions on the function  $\text{score}(x)$  will a greedy approach always yield an optimal solution?) As a result, once we have defined our greedy algorithm, it remains to be proved that the greedy solution is indeed optimal.

### 1.1 Insertion sort is an optimal greedy algorithm

Recall that insertion sort (Chapter 2) is not an efficient sorting algorithm, running in  $\Theta(n^2)$  time (how much space does it use?), but it *is* an optimal greedy algorithm.

Recall that insertion sort works like this:

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

The loop invariant for insertion sort is: 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. Initially,  $j = 2$  and this is trivially true (initialization). Because we insert the  $j$ th item into the subarray  $A[1..j-1]$  such that it remains sorted, if the invariant is true at the beginning of the loop, it remains true at the end of the loop (maintenance). And, when the loop terminates,  $j = n$  and we have inserted the last element correctly into the sorted subarray (termination).

To show that insertion sort is an optimal greedy algorithm, we must first define a score function that allows us to build sorted sequences one element at a time. Clearly, if  $A$  is already sorted,  $\text{score}(A)$  must yield a maximal value, but, to facilitate the greedy step, it must also give partial credit if  $A$  is partially sorted and that credit should be larger the more sorted  $A$  is. How can we quantify how sorted a sequence is?

One score function that suffices is to count the number of sequential comparisons that violate the sorting requirement  $A[i] \leq A[i+1]$ . That is,

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

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.,  $A[1] \leq A[2] \leq \dots \leq A[n]$ , and this 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 a score between these two extremes.

With the score function selected, now we must consider what choices that an algorithm faces, while sorting. 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 sequence some where, any where else. (For any particular  $A$ , how many such local moves are there?) For instance, instead of letting  $j = 1$  to  $n$ , we could work backwards, and instead of inserting it into  $A[1..j-1]$  so that this subarray remains sorted, we could insert it anywhere in  $A$ . Clearly, most of these moves would not be favorable. But, insertion sort is. In fact, it is optimal. (Are there other choices that are also optimally greedy?)

Given the loop invariant described above, we can now show that insertion sort is an optimal greedy algorithm. 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.

## 1.2 Linear storage media

Now let's consider a related problem. 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.<sup>1</sup> 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  $i$ th file. If the files are stored on the tape in order of their indices, the cost of accessing the  $j$ th file is

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

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  $j$ th 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  $\pi$  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 1:  $E[\text{cost}(\pi)]$  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

---

<sup>1</sup>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.

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.  $\square$

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  $i$ th 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  $\pi$  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 1?) But, what if the sizes and frequencies are both non-uniform? The answer is to sort by the length-frequency ratio  $L/f$ .

*Lemma 2: total-cost( $\pi$ ) 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 1, 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.  $\square$

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

### 1.3 Huffman codes

Continuing on the theme of storing and accessing information, consider the problem of *compression*. We are given a “message” or file  $x$  that is a sequence of characters (symbols) drawn from an  $n$ -character alphabet  $\Sigma$ . The task is to identify an encoding (mapping) of the original alphabet into a *prefix-free binary code*. A binary code is one built on the binary alphabet  $\Sigma_{01} = \{0, 1\}$ , and a prefix-free code is one in which no codeword is the prefix of any other. If a (binary) code is prefix-free, then it can be represented by a (binary) tree in which codewords are only located at the

bottom of the tree, at the leaves.<sup>2,3</sup> The length of a codeword is given by the depth of its associated leaf. The goal of compression is to identify an encoding tree such that the encoded message is as small as possible. If  $f_i$  is the frequency of the  $i$ th character in the original message, we seek to minimize the function

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

In 1952 David Huffman developed a greedy algorithm that produces an encoding that minimizes this function. In a compressed form, the idea is thus: merge the two least frequent characters and recurse.

Let's use a cute "self-descriptive" example from Lee Sallows<sup>4</sup>

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, let's ignore capitalization, the spaces (44), apostrophes (19), commas (19), hyphens (3) and the one period and focus on encoding the letters alone. The frequencies of the 26 letters are

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. (Note: if there is a tie for which pair of characters to merge, there will not be a unique Huffman code.)

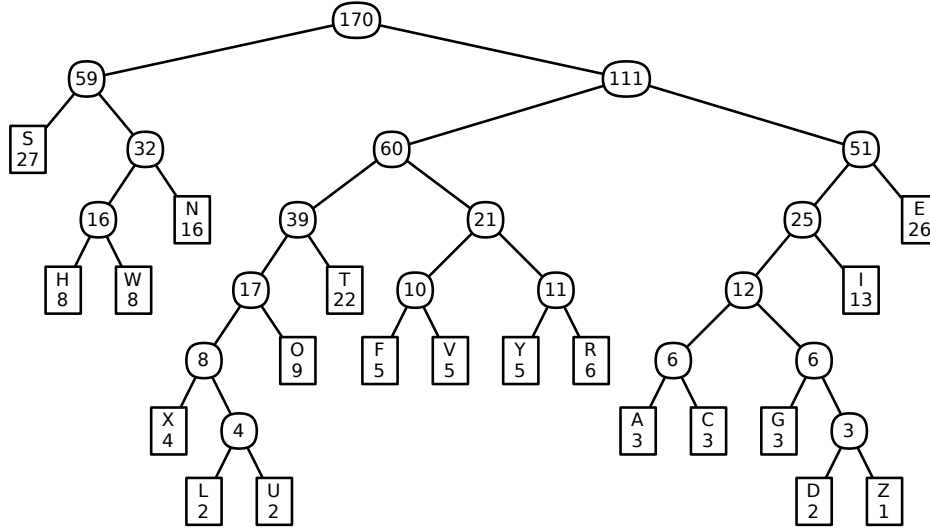
To read the encoding of a character, place a 0 on each left-branch and a 1 on each right-branch, then write the sequence of 1s and 0s backwards as you read up the tree. To decode an encoded letter, do the reverse: start at the root, and take the left-right path given by the encoding down the tree to arrive at the decoded character. For example, the encoding of the letter "a" is 110000. In our example, the encoded message is 646 bits long.

---

<sup>2</sup>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.

<sup>3</sup>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 be ordered in any way.

<sup>4</sup>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.



Now let's prove that Huffman encoding is optimal.

*Lemma 3: 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  $T'$  be the code tree if we swapped  $x$  and  $a$ . The depth of  $x$  increases by some amount at most  $\Delta$ , 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.  $\square$

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  $f_1, \dots, f_n$  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 3, we know that the optimal tree  $T$  has characters 1 and 2 as siblings.

Let  $T'$  be the Huffman tree for  $f_3, \dots, f_{n+1}$ . 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 observing that the depth  $d_T$  relative to  $T'$  is 1. Thus, 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. Thus, by induction, we have proved the theorem.  $\square$

(At home exercise: how long does it take to build a Huffman code on an  $n$ -character message? Hint: how can we efficiently find the minimum-frequency character at each step of the recursion?)

## 2 Next Time

1. Guest lecture by Prof. Ehrenfeucht on Wednesday, February 2
2. No office hours this week (out of town)
3. Reminder: Problem Set 1 due Tuesday, February 1 by 11:59pm via email