The point is, ladies and gentleman, greed is good. Greed works, greed is right.
Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.
Greed in all its forms, greed for life, money, love, knowledge
has marked the upward surge in mankind.
And greed—mark my words—will save not only Teldar Paper
but the other malfunctioning corporation called the USA.

— Gordon Gekko [Michael Douglas], *Wall Street* (1987)

There is always an easy solution to every human problem—
neat, plausible, and wrong.

— H. L. Mencken, "The Divine Afflatus",
New York Evening Mail (November 16, 1917)

4 Greedy Algorithms

4.1 Storing Files on Tape

Suppose we have a set of n files that we want to store on magnetic tape.[1] In the future, users will want to read those files from the tape. Reading a file from tape isn't like reading a file from disk; first we have to fast-forward past all the other files, and that takes a significant amount of time. Let $L[1\,..\,n]$ be an array listing the lengths of each file; specifically, file i has length $L[i]$. If the files are stored in order from 1 to n, then the cost of accessing the kth file is

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

[1]Readers who are tempted to object that magnetic tape has been obsolete for decades are cordially invited to tour your nearest supercomputer facility; ask to see the tape robots. Alternatively, consider filing a sequence of books on a library bookshelf. You know, those strange brick-like objects made of dead trees and ink?

The cost reflects the fact that before we read file k we must first scan past all the earlier files on the tape. If we assume for the moment that each file is equally likely to be accessed, then the *expected* cost of searching for a random file is

$$\mathrm{E}[cost] = \sum_{k=1}^{n} \frac{cost(k)}{n} = \sum_{k=1}^{n}\sum_{i=1}^{k} \frac{L[i]}{n}.$$

If we change the order of the files on the tape, we change the cost of accessing the files; some files become more expensive to read, but others become cheaper. Different file orders are likely to result in different expected costs. Specifically, let $\pi(i)$ denote the index of the file stored at position i on the tape. Then the expected cost of the permutation π is

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

Which order should we use if we want the expected cost to be as small as possible? The answer is intuitively clear; we should store the files in order from shortest to longest. So let's prove this.

Lemma 4.1. $\mathrm{E}[cost(\pi)]$ *is minimized when* $L[\pi(i)] \le L[\pi(i+1)]$ *for all* i.

Proof: Suppose $L[\pi(i)] > L[\pi(i+1)]$ for some i. To simplify notation, let $a = \pi(i)$ and $b = \pi(i+1)$. If we swap files a and b, then the cost of accessing a increases by $L[b]$, and the cost of accessing b decreases by $L[a]$. Overall, the swap changes the expected cost by $(L[b] - L[a])/n$. But this change is an improvement, because $L[b] < L[a]$. Thus, if the files are out of order, we can improve the expected cost by swapping some mis-ordered adjacent pair. □

This example gives us our first *greedy algorithm*. To minimize the *total* expected cost of accessing the files, we put the file that is cheapest to access first, and then recursively write everything else; no backtracking, no dynamic programming, just make the best local choice and blindly plow ahead. If we use an efficient sorting algorithm, the running time is clearly $O(n \log n)$, plus the time required to actually write the files. To show that the greedy algorithm is actually correct, we prove that the output of any other algorithm can be improved by some sort of exchange

Let's generalize this idea further. Suppose we are also given an array $F[1\,..\,n]$ of *access frequencies* for each file; file i will be accessed exactly $F[i]$ times over the lifetime of the tape. Now the *total* cost of accessing all the files on the tape is

$$\Sigma cost(\pi) = \sum_{k=1}^{n}\left(F[\pi(k)] \cdot \sum_{i=1}^{k} L[\pi(i)]\right) = \sum_{k=1}^{n}\sum_{i=1}^{k}\big(F[\pi(k)] \cdot L[\pi(i)]\big).$$

Now what order should store the files if we want to minimize the total cost? (This question is similar in spirit to the optimal binary search tree problem, but the target data structure and the cost function are both different, so the algorithm must be different, too.)

We've already proved that if all the frequencies are equal, then we should sort the files by increasing size. If the frequencies are all different but the file lengths $L[i]$ are all equal, then intuitively, we should sort the files by *decreasing* access frequency, with the most-accessed file first. In fact, this is not hard to prove by modifying the proof of Lemma 4.1. But what if the sizes and the frequencies both vary? In this case, we should sort the files by the *ratio* L/F.

Lemma 4.2. *$\Sigma cost(\pi)$ is minimized when* $\dfrac{L[\pi(i)]}{F[\pi(i)]} \le \dfrac{L[\pi(i+1)]}{F[\pi(i+1)]}$ *for all i.*

Proof: Suppose $L[\pi(i)]/F[\pi(i)] > L[\pi(i+1)]/F[\pi(i+i)]$ for some i. To simplify notation, let $a = \pi(i)$ and $b = \pi(i+1)$. If we swap files a and b, then the cost of accessing a increases by $L[b]$, and the cost of accessing b decreases by $L[a]$. Overall, the swap changes the total cost by $L[b]F[a]-L[a]F[b]$. But this change is an improvement, because

$$\frac{L[a]}{F[a]} > \frac{L[b]}{F[b]} \iff L[b]F[a]-L[a]F[b] < 0.$$

Thus, if any two adjacent files are out of order, we can improve the total cost by swapping them. □

4.2 Scheduling Classes

The next example is slightly more complex. Suppose you decide to drop out of computer science and change your major to Applied Chaos. The Applied Chaos department offers all of its classes on the same day every week, called "Soberday" by the students (but interestingly, *not* by the faculty). Every class has a different start time and a different ending time: AC 101 ("Toilet Paper Landscape Architecture") starts at 10:27pm and ends at 11:51pm; AC 666 ("Immanentizing the Eschaton") starts at 4:18pm and ends at 4:22pm, and so on. In the interest of graduating as quickly as possible, you want to register for as many classes as possible. (Applied Chaos classes don't require any actual *work*.) The university's registration computer won't let you register for overlapping classes, and no one in the department knows how to override this "feature". Which classes should you take?

More formally, suppose you are given two arrays $S[1..n]$ and $F[1..n]$ listing the start and finish times of each class; to be concrete, we can assume that $0 \le S[i] < F[i] \le M$ for each i, for some value M (for example, the number of picoseconds in Soberday). Your task is to choose the largest possible subset $X \in \{1,2,\ldots,n\}$ so that for any pair $i, j \in X$, either $S[i] > F[j]$ or $S[j] > F[i]$. We can illustrate the problem by drawing each class as a rectangle whose left and right x-coordinates show the start and finish times. The goal is to find a largest subset of rectangles that do not overlap vertically.

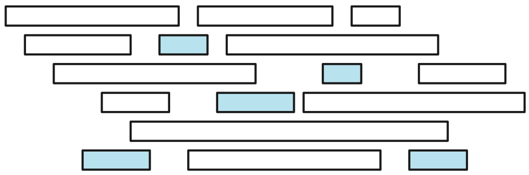


Figure 4.1. A maximum conflict-free schedule for a set of classes.

This problem has a fairly simple recursive solution, based on the observation that either you take class 1 or you don't. Let B denote the set of classes that *end before* class 1 starts, and let A denote the set of classes that *start after* class 1 ends:

$$
\begin{aligned}
B &:= \{i \mid 2 \le i \le n \text{ and } F[i] < S[1]\}\\
A &:= \{i \mid 2 \le i \le n \text{ and } S[i] > F[1]\}
\end{aligned}
$$

If class 1 is in the optimal schedule, then so are the optimal schedules for B and A, which we can find recursively. If not, we can find the optimal schedule for $\{2,3,\ldots,n\}$ recursively. So we should try both choices and take whichever one gives the better schedule. Evaluating this recursive algorithm from the bottom up gives us a dynamic programming algorithm that runs in $O(n^3)$ time. I won't bother to go through the details, because we can do better.[2]

Intuitively, we'd like the first class to finish as early as possible, because that leaves us with the largest number of remaining classes. This intuition suggests the following simple greedy algorithm. Scan through the classes in order of finish time; whenever you encounter a class that doesn't conflict with your latest class so far, take it! See Figure 4.2 for a visualization of the resulting greedy schedule.

We can write the greedy algorithm somewhat more formally as follows. (Hopefully the first line is understandable.) The algorithm clearly runs in $O(n \log n)$ time; the running time is dominated by the initial sort.

[2]But you should still work out the details yourself. The dynamic programming algorithm can be used to find the "best" schedule for several different definitions of "best", but the greedy algorithm I'm about to describe only works when "best" means "biggest". Also, you can improve the running time to $O(n^2)$ using a different recurrence.

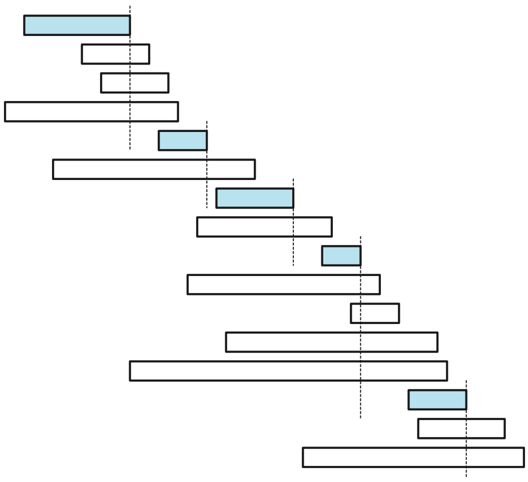


Figure 4.2. The same classes sorted by finish times and the greedy schedule.

```
GreedySchedule($S[1..n], F[1..n]$):
    sort $F$ and permute $S$ to match
    $count \leftarrow 1$
    $X[count] \leftarrow 1$
    for $i \leftarrow 2$ to $n$
        if $S[i] > F[X[count]]$
            $count \leftarrow count + 1$
            $X[count] \leftarrow i$
    return $X[1..count]$
```

To prove that GreedySchedule actually computes the largest conflict-free schedule, we use an exchange argument, similar to the one we used for tape sorting. We are not claiming that the greedy schedule is the *only* maximal schedule; there could be others. (Compare Figures 4.1 and 4.2!) All we can claim is that at least one of the optimal schedules is the one produced by the greedy algorithm.

Lemma 4.3. *At least one maximal conflict-free schedule includes the class that finishes first.*

Proof: Let f be the class that finishes first. Suppose we have a maximal conflict-free schedule X that does not include f. Let g be the first class in X to finish. Since f finishes before g does, f cannot conflict with any class in the set $S \setminus \{g\}$. Thus, the schedule $X' = X \cup \{f\} \setminus \{g\}$ is also conflict-free. Since X' has the same size as X, it is also maximal. □

To finish the proof, we call on our old friend, induction.

Theorem 4.4. *The greedy schedule is an optimal schedule.*

Proof: Let f be the class that finishes first, and let A be the subset of classes that start after f finishes. The previous lemma implies that some optimal schedule contains f, so the best schedule that contains f is an optimal schedule. The best schedule that includes f must contain an optimal schedule for the classes that do not conflict with f, that is, an optimal schedule for A. The greedy algorithm chooses f and then, by the inductive hypothesis, computes an optimal schedule of classes from A. □

The proof might be easier to understand if we unroll the induction slightly.

Proof: Let $\langle g_1, g_2, \dots, g_k\rangle$ be the sequence of classes chosen by the greedy algorithm. Suppose we have a maximal conflict-free schedule

$$S = \langle g_1, g_2, \dots, g_{j-1}, c_j, c_{j+1}, \dots, c_m\rangle,$$

where class c_j is different from the class g_j chosen by the greedy algorithm. (We could have $j = 1$, in which case this schedule starts with a non-greedy choice c_1.) By construction, the jth greedy choice g_j does not conflict with any earlier class $g_1, g_2, \dots, g_{j-1}$, and because our schedule S is conflict-free, neither does c_j. Moreover, g_j has the *earliest* finish time among all classes that don't conflict with the earlier classes; in particular, g_j finishes before c_j. It follows that g_j does not conflict with any of the later classes $c_{j+1}, \dots, c_m$. Thus, the schedule

$$S' = \langle g_1, g_2, \dots, g_{j-1}, g_j, c_{j+1}, \dots, c_m\rangle,$$

is also conflict-free. (This argument is just a generalization of Lemma 4.3, which considers the case $j = 1$.)

By induction, it now follows that there is an optimal schedule $\langle g_1, g_2, \dots, g_k, c_{k+1}, \dots, c_m\rangle$ that includes *every* class chosen by the greedy algorithm. But this is impossible unless $k = m$; if some class c_{k+1} does not conflict with any of the first k greedy classes, then the greedy algorithm would choose more than k classes! □

4.3 General Structure

The basic structure of this correctness proof is exactly the same as for the tape-sorting problem: an inductive exchange argument.

- Assume that there is an optimal solution that is different from the greedy solution.
- Find the "first" difference between the two solutions.

- Argue that we can exchange the optimal choice for the greedy choice without making the solution worse (although the exchange might not make it better).

This argument implies by induction that some optimal solution *contains* the entire greedy solution, and therefore *equals* the greedy solution. Sometimes, as in the scheduling problem, an additional step is required to show no optimal solution *strictly* improves the greedy solution.

4.4 Huffman Codes

A *binary code* assigns a string of 0s and 1s to each character in the alphabet. A binary code is *prefix-free* if no code is a prefix of any other. (Confusingly, prefix-free codes are also commonly called *prefix codes*.) 7-bit ASCII and Unicode's UTF-8 are both prefix-free binary codes. Morse code is a binary code with symbols • and —, but it is *not* prefix-free, because the code for `E` (•) is a prefix of the codes for `I` (••), `S` (•••), and `H` (••••).[3]

Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves. The code word for any symbol is given by the path from the root to the corresponding leaf; `0` for left, `1` for right. Thus, the length of any symbol's codeword is the depth of the corresponding leaf in the code tree. Although they are superficially similar, binary code trees are *not* binary search trees; we don't care at all about the order of symbols at the leaves.

Suppose we want to encode a message written in an n-character alphabet so that the encoded message is as short as possible. Specifically, given an array frequency counts $f[1..n]$, we want to compute a prefix-free binary code that minimizes the total encoded length of the message:

$$\sum_{i=1}^{n} f[i]\cdot depth(i).$$

This is exactly the same cost function we considered for optimizing binary search trees, but the optimization problem is different, because code trees are not required to keep the keys in any particular order.

[3]For this reason, Morse code is arguably better described as a prefix-free *ternary* code, with three symbols: •, —, and pause. Alternatively, Morse code can be considered a prefix-free binary code, with one beat of sound/light/current/high voltage/smoke/gas (■) and one beat of silence/darkness/no current/low voltage/air/liquid (□) as the two symbols. Then each "dit" is encoded as ■□, each "dah" as ■■■□, and each pause as □□. In standard Morse code, each letter is followed by one pause, and each word is followed by two additional pauses; however, □s at the end of the entire coded message are omitted. For example, the string "`MORSE CODE`" is unambiguously encoded as the following bit string:

■■■□■■■□□□■■■□■■■□■■■□□□■□■■■□□□■□■□■□□□■□□□□□□□■■■□■□■■■□■□□□■■■□■■■□■■■□□□■■■□■□■□□□■.

In 1951, as a PhD student at MIT, David Huffman developed the following greedy algorithm to produce such an optimal code:[4]

Huffman: Merge the two least frequent letters and recurse.

Huffman's algorithm is best illustrated through an example. Suppose we want to encode the following helpfully self-descriptive sentence, discovered by Lee Sallows:[5]

> 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 the forty-four spaces, nineteen apostrophes, nineteen commas, three hyphens, and only one period, and encode only the letters:

```
THISSENTENCECONTAINSTHREEASTHREECSTWODSTWENTYSIXESFIVEFST
HREEGSEIGHTHSTHIRTEENISTWOLSSIXTEENNSNINEOSSIXRSTWENTYSEV
ENSSTWENTYTWOTSTWOUSFIVEVSEIGHTWSFOURXSFIVEYSANDONLYONEZ
```
[6]

Here is the frequency table for Sallows' sentence:

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

Huffman's algorithm picks out the two least frequent letters, breaking ties arbitrarily—in this case, say, Z and D—and merges them together into a single

[4]Huffman was a student in an information theory class taught by Robert Fano, who was a close colleague of Claude Shannon, the father of information theory. Fano and Shannon had previously developed a different greedy algorithm for producing prefix codes—split the frequency array into two subarrays as evenly as possible, and then recursively build a code for each subarray—but these Fano-Shannon codes were known not to be optimal. Fano posed the problem of finding an optimal prefix code to his class. Huffman decided to solve the problem as a class project, instead of taking a final exam, not realizing that the problem was open, or that Fano and Shannon had already tried and failed to solve it. After several months of fruitless effort, Huffman eventually gave up and decided to take the final exam after all. As he was throwing his notes in the trash, the solution dawned on him. Huffman would later describe the epiphany as "the absolute lightning of sudden realization".

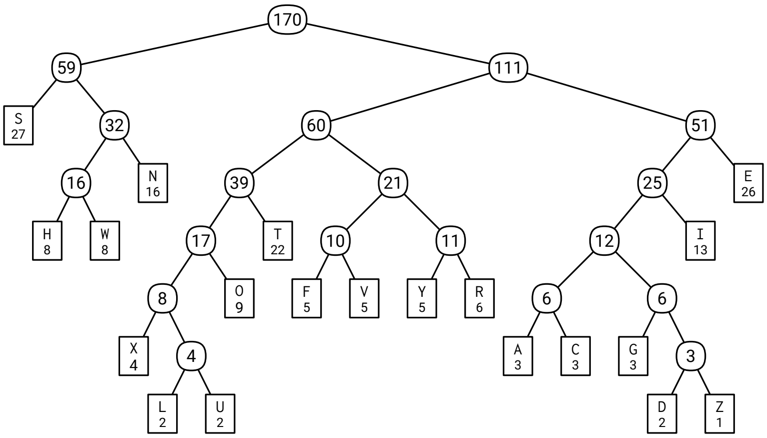
[5]This sentence was first reported by Alexander Dewdney in his October 1984 "Computer Recreations" column in *Scientific American*. Sallows himself published the remarkable story of its discovery in 1985, along with several other self-descriptive sentences; you can find Sallows' paper on his web site. Frustrated with the slow progress of his code running on a VAX 11/780, Sallows designed and built dedicated hardware to perform a brute-force search for self-descriptive sentences with various flavor text ("This pangram has...", "This sentence contains exactly...", and so on). Careful theoretical analysis limited the search space to just over six billion possibilities, which his 1-MHz Pangram Machine enumerated in just under two hours.

[6]... and he talked for forty-five minutes, and nobody understood a word that he said, but we had fun fillin' out the forms and playin' with the pencils on the bench there.

new character DZ with frequency 3. This new character becomes an internal node in the code tree we are constructing, with Z and D as its children; it doesn't matter which child is which. The algorithm then recursively constructs a Huffman code for the new frequency table

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

After 19 merges, all 20 letters have been merged together. The record of merges gives us our code tree. The algorithm makes a number of arbitrary choices; as a result, there are actually several different Huffman codes. One such Huffman code is shown below; numbers in non-leaf nodes are frequencies for merged characters. For example, the code for A is 110000, and the code for S is 00.



Encoding Sallows' sentence with this particular Huffman code would yield a bit string that starts like so:

1001	0100	1101	00	00	111	011	1001	111	011	110001	111	110001	10001	011	1001	…
T	H	I	S	S	E	N	T	E	N	C	E	C	O	N	T	

Here is the list of costs for encoding each character in Sallows' sentence, along with that character's contribution to the total length of the encoded sentence:

char	A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
freq	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
depth	6	6	7	3	5	6	4	4	7	3	4	4	2	4	7	5	4	6	5	7
total	18	18	14	78	25	18	32	52	14	48	36	24	54	88	14	25	32	24	25	7

Altogether, the encoded message is 646 bits long. Different Huffman codes encode the same characters differently, possibly with code words of different length, but the overall length of the encoded message is the same for *every* Huffman code: 646 bits.

Given the simple structure of Huffman's algorithm, it's rather surprising that it produces an *optimal* prefix-free binary code.[7] Encoding Sallows' sentence using *any* prefix-free code requires at least 646 bits! Fortunately, the recursive structure makes this claim easy to prove using an exchange argument, similar to our earlier optimality proofs. We start by proving that the algorithm's very first choice is correct.

Lemma 4.5. *Let x and y be the two least frequent characters (breaking ties between equally frequent characters arbitrarily). There is an optimal code tree in which x and y are siblings.*

Proof: I'll actually prove a stronger statement: There is an optimal code in which x and y are siblings *and* have the largest depth of any leaf.

Let T be an optimal code tree, and suppose this tree has depth d. Because T is a full binary tree, it has at least two leaves at depth d that are siblings. (Verify this by induction!) Suppose those two leaves are *not* x and y, but some other characters a and b.

Let T' be the code tree obtained by swapping x and a, and let $\Delta = d - \mathit{depth}_T(x)$. This swap increases the depth of x by Δ and decreases the depth of a by Δ, so

$$\mathit{cost}(T') = \mathit{cost}(T) + \Delta \cdot (f[x] - f[a]).$$

Our assumption that x is one of the two least frequent characters but a is not implies $f[x] \le f[a]$, and our assumption that a has maximum depth implies $\Delta \ge 0$. It follows that $\mathit{cost}(T') \le \mathit{cost}(T)$. On the other hand, T is an optimal code tree, so we must also have $\mathit{cost}(T') \ge \mathit{cost}(T)$. We conclude that T' is *also* an optimal code tree.

Similarly, swapping y and b must give yet another optimal code tree. In this final optimal code tree, x and y are maximum-depth siblings, as required. □

Now optimality is guaranteed by our dear friend the Recursion Fairy! Our recursive argument relies on the following non-standard recursive definition: *A full binary tree is either a single node, or a full binary tree where some leaf has been replaced by an internal node with two leaf children.*

Theorem 4.6. *Every Huffman code is an optimal prefix-free binary code.*

Proof: If the message has only one or two distinct characters, the theorem is trivial, so assume otherwise.

Let $f[1..n]$ be the original input frequencies, and assume without loss of generality that $f[1]$ and $f[2]$ are the two smallest frequencies. To set up the

[7]It was certainly surprising to both Huffman and Fano!

recursive subproblem, define $f[n+1] = f[1] + f[2]$. Our earlier exchange argument implies that 1 and 2 are (deepest) siblings in some optimal code for $f[1..n]$.

Let T' be the Huffman tree for $f[3..n+1]$; the inductive hypothesis implies that T' is an optimal code tree for the smaller set of frequencies. To obtain the final code tree T, we replace the leaf labeled $n+1$ with an internal node with two children, labelled 1 and 2. I claim that T is optimal for the original frequency array $f[1..n]$.

To prove this claim, we can express the cost of T in terms of the cost of T' as follows. (In these equations, $\mathit{depth}(i)$ denotes the depth of the leaf labelled i in either T or T'; each leaf that appears in both T and T' has the same depth in both trees.)

$$\begin{aligned}
\mathit{cost}(T) &= \sum_{i=1}^{n} f[i]\cdot \mathit{depth}(i)\\
&= \sum_{i=3}^{n+1} f[i]\cdot \mathit{depth}(i) \;+\; f[1]\cdot \mathit{depth}(1) + f[2]\cdot \mathit{depth}(2)\\
&\qquad\qquad\qquad\quad -\; f[n+1]\cdot \mathit{depth}(n+1)\\
&= \mathit{cost}(T') + (f[1]+f[2])\cdot \mathit{depth}(T) \;-\; f[n+1]\cdot(\mathit{depth}(T)-1)\\
&= \mathit{cost}(T') + f[1] + f[2] \;+\; (f[1]+f[2]-f[n+1])\cdot(\mathit{depth}(T)-1)\\
&= \mathit{cost}(T') + f[1] + f[2]
\end{aligned}$$

This equation implies that minimizing the cost of T is equivalent to minimizing the cost of T'; in particular, attaching leaves labeled 1 and 2 to the leaf in T' labeled $n+1$ gives an optimal code tree for the original frequencies. □

To efficiently construct a Huffman code, we keep the characters in a priority queue, using the character frequencies as priorities. We can represent the code tree as three arrays of indices, listing the Left and Right children and the Parent of each node. The leaves of the final code tree are nodes at indices 1 through n, and the root is the node with index $2n-1$.

```
BUILDHUFFMAN($f[1..n]$):
    for $i \leftarrow 1$ to $n$
        $L[i] \leftarrow 0$;  $R[i] \leftarrow 0$
        INSERT($i, f[i]$)
    for $i \leftarrow n$ to $2n-1$
        $x \leftarrow$ EXTRACTMIN( )      ⟨⟨find two rarest symbols⟩⟩
        $y \leftarrow$ EXTRACTMIN( )
        $f[i] \leftarrow f[x] + f[y]$     ⟨⟨merge into a new symbol⟩⟩
        INSERT($i, f[i]$)
        $L[i] \leftarrow x$;  $P[x] \leftarrow i$     ⟨⟨update tree pointers⟩⟩
        $R[i] \leftarrow y$;  $P[y] \leftarrow i$
    $P[2n-1] \leftarrow 0$
```

BUILDHUFFMAN performs $O(n)$ priority-queue operations: exactly $2n-1$ INSERTS and $2n-2$ EXTRACTMINS. If we implement the priority queue as a standard binary heap, each of these operations requires $O(\log n)$ time, and thus the entire algorithm runs in ***$O(n \log n)$ time***.

Finally, simple algorithms to encode and decode messages using a fixed Huffman code are shown in Figure 4.3; both algorithms run in $O(m)$ time, where m is the length of the encoded message.

```
HUFFMANENCODE($A[1..k]$):
    $m \leftarrow 1$
    for $i \leftarrow 1$ to $k$
        HUFFMANENCODEONE($A[i]$)

HUFFMANENCODEONE($x$):
    if $x < 2n-1$
        HUFFMANENCODEONE($P[x]$)
        if $x = L[P[x]]$
            $B[m] \leftarrow 0$
        else
            $B[m] \leftarrow 1$
        $m \leftarrow m+1$
```

```
HUFFMANDECODE($B[1..m]$):
    $k \leftarrow 1$
    $v \leftarrow 2n-1$
    for $i \leftarrow 1$ to $m$
        if $B[i] = 0$
            $v \leftarrow L[v]$
        else
            $v \leftarrow R[v]$
        if $L[v] = 0$
            $A[k] \leftarrow v$
            $k \leftarrow k+1$
            $v \leftarrow 2n-1$
```

Figure 4.3. Encoding and decoding algorithms for Huffman codes

4.5 Stable Matching

Every year, thousands of new doctors must obtain internships at hospitals around the United States. During the first half of the 20th century, competition among hospitals for the best doctors led to earlier and earlier offers of internships, sometimes as early as the second year of medical school, along with tighter deadlines for acceptance. In the 1940s, medical schools agreed not to release information until a common date during their students' fourth year. In response, hospitals began demanding faster decisions. By 1950, hospitals would regularly call doctors, offer them internships, and demand *immediate* responses. Interns were forced to gamble if their third-choice hospital called first—accept and risk losing a better opportunity later, or reject and risk having no position at all.[8]

Finally, a central clearinghouse for internship assignments, now called the National Resident Matching Program (NMRP), was established in the early 1950s. Each year, doctors submit a ranked list of all hospitals where they would accept an internship, and each hospital submits a ranked list of doctors they would accept as interns. The NRMP then computes an matching between doctors and hospitals that satisfies the following *stability* requirement. A matching is ***unstable*** if there is a doctor α and hospital B that would be both happier with each other than with their current match; that is,

- α is matched with some other hospital A, even though she prefers B.
- B is matched with some other doctor β, even though they prefer α.

In this case, we call (α, B) an *unstable pair* for the matching. The goal of the Resident Match is a ***stable matching***, which is a matching with no unstable pairs.

For simplicity, I'll assume from now on that there are exactly the same number of doctors and hospitals; each hospital offers exactly one internship; each doctor ranks all hospitals and vice versa; and finally, there are no ties in the doctors' or hospitals' rankings.[9]

Some Bad Ideas

At first glance, it is not clear that a stable matching always exists! Certainly not *every* matching of doctors and hospitals is stable. Suppose there are three doctors (Dr. Quincy, Dr. Rotwang, Dr. Shephard, represented by lower-case letters) and three hospitals (Arkham Asylum, Bethlem Royal Hospital, and County General Hospital, represented by upper-case letters), who rank each

[8]The American academic job market involves similar gambles, at least in computer science. Some departments start making offers in February with two-week decision deadlines; other departments don't even start interviewing until March; MIT notoriously waits until May, when all its interviews are over, before making *any* faculty offers. Needless to say, the mishmash of offer dates and decision deadlines causes tremendous stress, for candidates and departments alike. For similar reasons, since 1965, most American universities have agreed to a common April 15 deadline for prospective graduate students to accept offers of financial support (and by extension, offers of admission).

[9]In reality, most hospitals offer multiple internships, each doctor ranks only a subset of the hospitals and vice versa, and there are typically more internships than interested doctors. And then it starts getting complicated.

other as follows:

q	r	s
A	C	A
C	A	B
B	B	C

A	B	C
r	s	q
q	q	r
s	r	s

The matching $\{Aq, Br, Cs\}$ is unstable, because Arkham would rather hire Dr. Rotwang than Dr. Quincy, and Dr. Rotwang would rather work at Arkham than at Bedlam. (A, r) is an unstable pair for this matching.

One might imagine using an incremental algorithm that starts with an arbitrary matching, and then greedily performs exchanges to resolve instabilities. Unfortunately, resolving one instability can new ones; in fact, this incremental "improvement" can lead to an infinite loop. For example, if we start with our earlier unstable matching $\{Aq, Br, Cs\}$, each of the following exchanges resolves one unstable pair (indicated over the arrow), but the sequence of exchanges leads back to the original matching:[10]

$$\{Aq, Br, Cs\} \xrightarrow{Ar} \{Ar, Bq, Cs\} \xrightarrow{Cr} \{As, Bq, Cr\} \xrightarrow{Cq} \{As, Br, Cq\} \xrightarrow{Aq} \{Aq, Br, Cs\}$$

Alternatively, we might try the following multi-round greedy protocol. In each round, every unmatched hospital makes an offer to their favorite unmatched doctor, then every unmatched doctor with an offer accepts their favorite offer. It's not hard to prove that at least one new doctor-hospital pair is matched each round, so the algorithm always ends with a matching. For the previous example input, we already have a stable matching $\{Ar, Bs, Cq\}$ at the end of the first round! But consider the following input instead:

q	r	s
C	A	A
B	C	B
A	B	C

A	B	C
q	q	s
s	r	r
r	s	q

In the first round, Dr. Shephard accepts an offer from County, and Dr. Quincy accepts an offer from Bedlam (rejecting Arkham's offer), leaving only Dr. Rotwang and Arkham unmatched. Thus, the protocol ends with the matching $\{Ar, Bq, Cs\}$ after two rounds. Unfortunately, this matching is unstable; Arkham and Dr. Shephard prefer each other to their matches.

The Boston Pool and Gale-Shapley Algorithms

In 1952, the NRMP adopted the "Boston Pool" algorithm to assign interns, so named because it had been previously used by a regional clearinghouse in the

[10]This example was discovered by Donald Knuth.

Boston area. Ten years later, David Gale and Lloyd Shapley described and formally analyzed a generalization of the Boston Pool algorithm and proved that it computes a stable matching. Gale and Shapley used the metaphor of college admissions. Essentially the same algorithm was independently developed by Elliott Peranson in 1972 for use in medical school admissions. Similar algorithms have since been adopted for many other matching markets, including faculty hiring in France, hiring of new economics PhDs in the United States, university admission in Germany, public school admission in New York and Boston, billet assignments for US Navy sailors, and kidney-matching programs.

Shapley was awarded the 2012 Nobel Prize in Economics for his research on stable matchings, together with Alvin Roth, who significantly extended Shapley's work and used it to develop several real-world exchanges. (Gale did not share the prize, because he died in 2008.)

Like our last failed greedy algorithm, the Gale-Shapley algorithm proceeds in rounds until every position has been accepted. Each round has two stages:

1. An arbitrary unmatched hospital A offers its position to the best doctor α (according to A's preference list) who has not already rejected it.
2. If α is unmatched, she (tentatively) accepts A's offer. If α already has a match but prefers A, she rejects her current match and (tentatively) accepts the new offer from A. Otherwise, α rejects the new offer.

Each doctor ultimately accepts the best offer that she receives, according to her preference list.[11] In short, hospitals make offers greedily, and doctors accept offers greedily. The doctors' ability to reject their current matches in favor of better offers is the key to making this mutual greedy strategy work.

For example, suppose that there are four doctors (Dr. Quincy, Dr. Rotwang, Dr. Shephard, and Dr. Tam) and four hospitals (Arkham Asylum, Bethlem Royal Hospital, County General Hospital, and The Dharma Initiative), who rank each other as follows:

q	r	s	t
A	A	B	D
B	D	A	B
C	C	C	C
D	B	D	A

A	B	C	D
t	r	t	s
s	t	r	r
r	q	s	q
q	s	q	t

Given these preference lists as input, the Gale-Shapley algorithm might proceed as follows:

[11]The 1952 Boston Pool algorithm is a special case of the Gale-Shapley algorithm that executes offers in a particular order. Roughly speaking, each offer is made by a hospital X whose favorite doctor (among those who haven't rejected X already) ranks X highest. Because the order of offers depends on the entire set of preference lists, this algorithm *must* be executed by a central authority; in contrast, the Gale-Shapley algorithm does not even require each participant to know their own preferences in advance, as long as they behave consistently with some fixed rankings.

1. Arkham makes an offer to Dr. Tam.
2. Bedlam makes an offer to Dr. Rotwang.
3. County makes an offer to Dr. Tam, who rejects her earlier offer from Arkham.
4. Dharma makes an offer to Dr. Shephard. (From this point on, there is only one unmatched hospital, so the algorithm has no more choices.)
5. Arkham makes an offer to Dr. Shephard, who rejects her earlier offer from Dharma.
6. Dharma makes an offer to Dr. Rotwang, who rejects her earlier offer from Bedlam.
7. Bedlam makes an offer to Dr. Tam, who rejects her earlier offer from County.
8. County makes an offer to Dr. Rotwang, who rejects it.
9. County makes an offer to Dr. Shephard, who rejects it.
10. County makes an offer to Dr. Quincy.

After the tenth round, all pending offers are accepted, and the algorithm returns the matching $\{As, Bt, Cq, Dr\}$. You can (and should) verify by brute force that this matching is stable, even though no doctor was hired by her favorite hospital, and no hospital hired their favorite doctor; in fact, County ended up hiring their *least* favorite doctor. This is not the only stable matching for these preference lists; the matching $\{Ar, Bs, Cq, Dt\}$ is also stable.

Running Time

Analyzing the number of offers performed by the algorithm is relatively straightforward (which is why we're doing it first). Each hospital makes an offer to each doctor at most once, so the algorithm makes at most n^2 offers.

To analyze the actual *running time*, however, we need to specify the algorithm in more detail. How are the preference lists given to the algorithm? How does the algorithm decide whether any hospital is unmatched, and if so, how does it find an unmatched hospital? How does the algorithm store the tentative matchings? How does the algorithm decide whether a doctor prefers to new offer to her current match? Most fundamentally: *How does the algorithm actually represent doctors and hospitals?*

One possibility is to represent each doctor and hospital by a unique integer between 1 and n, and to represent preferences as two arrays $Dpref[1..n, 1..n]$ and $Hpref[1..n, 1..n]$, where $Dpref[i, r]$ represents the rth hospital in doctor i's preference list, and $HPref[j, r]$ represents the rth doctor in hospital j's preference list. With the input in this form, the Boston Pool algorithm can execute each offer in constant time, after some initial preprocessing; the overall implementation runs in $\boldsymbol{O(n^2)}$ ***time***. We leave the remaining details as a straightforward exercise.

A somewhat harder exercise is to prove that there are inputs (and choices of who makes offers when) that force $\Omega(n^2)$ offers to be made before the algorithm halts. Thus, our $O(n^2)$ upper bound on the worst-case running time is *tight*.

Correctness

But why is the algorithm *correct* at all? How do we know that it always computes a stable matching, or *any* complete matching for that matter? Gale and Shapley argued correctness as follows.

The algorithm continues as long as there is at least one unfilled position; conversely, when the algorithm terminates (after at most n^2 rounds), every position is filled. No doctor can accept more than one position, and no hospital can hire more than one doctor. So the algorithm always computes a matching. (Whew!) It remains only to prove that the resulting matching is stable.

Suppose the algorithm matches some doctor α to some hospital A, even though she prefers another hospital B. Because every doctor accepts the best offer she receives, α received no offer she liked more than A; in particular, B never made an offer to α. On the other hand, B made offers to every doctor they prefer over their final match β. It follows that B prefers β over α, which means (α, B) is *not* an unstable pair. We conclude that there are no unstable pairs; the matching is stable!

Optimality!

Surprisingly, the correctness of the Gale-Shapley algorithm does not depend on which hospital makes its offer in each round. In fact, no matter which unassigned hospital makes an offer in each round, *the algorithm always computes the same matching*! Let's say that α is a ***feasible*** doctor for A if there is a stable matching that assigns doctor α to hospital A.

Lemma 4.7. *During the Gale-Shapley algorithm, each hospital A is rejected only by doctors that are infeasible for A.*

Proof: We prove the lemma by induction on the number of rounds. Consider an arbitrary round of the algorithm, in which doctor α rejects one hospital A for another hospital B. The rejection implies that α prefers B to A. Every doctor that appears higher than α in B's preference list was already rejected B in an earlier round and therefore, by the inductive hypothesis, is infeasible for B.

Now consider an arbitrary matching (of the same doctors and hospitals) that assigns α to A. We already established that α prefers B to A. If B prefers α to its partner, the matching is unstable. On the other hand, if B prefers its partner to α, then (by our earlier argument) its partner is infeasible, and again the matching is unstable. We conclude that there is no stable matching that assigns α to A. □

Now let $\boldsymbol{best(A)}$ denote the highest-ranked *feasible* doctor on A's preference list. Lemma 4.7 implies that every doctor that A prefers to its final match is infeasible for A. On the other hand, the final matching is stable, so the doctor assigned to A must be feasible for A. The following result is now immediate:

Corollary 4.8. *The Gale-Shapley algorithm matches* $best(A)$ *with A, for every hospital A.*

In other words, the Gale-Shapley algorithm computes the *best possible* stable matching from the hospitals' point of view. It turns out that this matching is also the *worst* possible from the doctors' point of view! Let $\boldsymbol{worst(\alpha)}$ denote the lowest-ranked feasible hospital on doctor α's preference list.

Corollary 4.9. *The Gale-Shapley algorithm matches α with* $worst(\alpha)$*, for every doctor α.*

Proof: Suppose Gale and Shapley assign doctor α to hospital A; we need to show that $A = worst(\alpha)$. Consider an arbitrary stable matching where A is *not* matched with α but with another doctor β. The previous corollary implies that A prefers $\alpha = best(A)$ to β. Because the matching is stable, α must therefore prefer her assigned hospital to A. This argument works for *any* stable matching, so α prefers *every* other feasible match to A; in other words, $A = worst(\alpha)$. □

A subtle consequence of these two corollaries, discovered by Lester Dubins and David Freedman in 1981, is that a doctor can potentially improve her match by lying about her preferences, but a hospital cannot. (However, a set of hospitals can collude so that *some* of their matches improve.) Partly for this reason, the National Residency Matching Program reversed its matching algorithm in 1998, so that potential residents offer to work for hospitals, according to their preference orders, and each hospital accepts its best offer. Thus, the new algorithm computes the best possible stable matching for the doctors, and the worst possible stable matching for the hospitals. In practice, however, this reversal altered less than 1% of the residents' matches. As far as I know, the precise effect of this change on the *patients* is an open problem.

Exercises

Caveat lector: Some of these exercises *cannot* be solved using greedy algorithms! Whenever you describe and analyze a greedy algorithm, you ***must***

also include a proof that your algorithm is correct; this proof will typically take the form of an exchange argument. These proofs are especially important in classes (like mine) that do *not* normally require proofs of correctness.

1. The GREEDYSCHEDULE algorithm we described for the class scheduling problem is not the only greedy strategy we could have tried. For each of the following alternative greedy strategies, either prove that the resulting algorithm always constructs an optimal schedule, or describe a small input example for which the algorithm does *not* produce an optimal schedule. Assume that all algorithms break ties arbitrarily (that is, in a manner that is completely out of your control). *[Hint: **Three** of these algorithms are actually correct.]*

   (a) Choose the course x that *ends last*, discard classes that conflict with x, and recurse.

   (b) Choose the course x that *starts first*, discard all classes that conflict with x, and recurse.

   (c) Choose the course x that *starts last*, discard all classes that conflict with x, and recurse.

   (d) Choose the course x with *shortest duration*, discard all classes that conflict with x, and recurse.

   (e) Choose a course x that *conflicts with the fewest other courses*, discard all classes that conflict with x, and recurse.

   (f) If no classes conflict, choose them all. Otherwise, discard the course with *longest duration* and recurse.

   (g) If no classes conflict, choose them all. Otherwise, discard a course that *conflicts with the most other courses* and recurse.

   (h) Let x be the class with the *earliest start time*, and let y be the class with the *second earliest start time*.
      - If x and y are disjoint, choose x and recurse on everything but x.
      - If x completely contains y, discard x and recurse.
      - Otherwise, discard y and recurse.

   (i) If any course x completely contains another course, discard x and recurse. Otherwise, choose the course y that *ends last*, discard all classes that conflict with y, and recurse.

2. Now consider a weighted version of the class scheduling problem, where different classes offer different number of credit hours (totally unrelated to the duration of the class lectures). Your goal is now to choose a set of non-conflicting classes that give you the largest possible number of credit hours, given arrays of start times, end times, and credit hours as input.

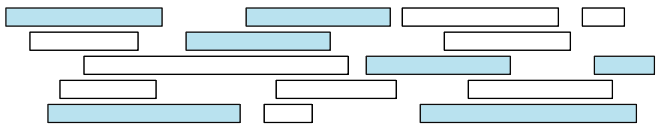
(a) Prove that the greedy algorithm described at the beginning of this chapter—Choose the class that ends first and recurse—does *not* always return an optimal schedule.

(b) Prove that ***none*** of the greedy algorithms described in Exercise 1 always return an optimal schedule. *[Hint: Solve Exercise 1 first; the algorithms that don't work there don't work here, either.]*

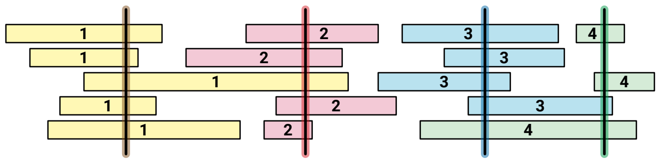
(c) Describe and analyze an algorithm that always computes an optimal schedule. *[Hint: Your algorithm will not be greedy.]*

3. Let X be a set of n intervals on the real line. We say that a subset of intervals $Y \subseteq X$ *covers* X if the union of all intervals in Y is equal to the union of all intervals in X. The *size* of a cover is just the number of intervals.

   Describe and analyze an efficient algorithm to compute the smallest cover of X. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, representing the left and right endpoints of the intervals in X. If you use a greedy algorithm, you must prove that it is correct.


A set of intervals, with a cover (shaded) of size 7.

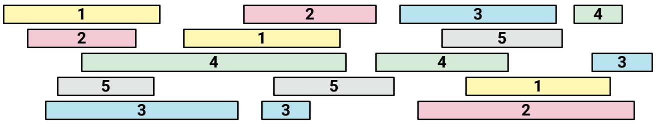
4. Let X be a set of n intervals on the real line. We say that a set P of points *stabs* X if every interval in X contains at least one point in P. Describe and analyze an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, representing the left and right endpoints of the intervals in X. As usual, If you use a greedy algorithm, you must prove that it is correct.



A set of intervals stabbed by four points (shown here as vertical segments)

5. Let X be a set of n intervals on the real line. A *proper coloring* of X assigns a color to each interval, so that any two overlapping intervals are assigned different colors. Describe and analyze an efficient algorithm to compute the minimum number of colors needed to properly color X. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, representing the left and

right endpoints of the intervals in X. As usual, if you use a greedy algorithm, you must prove that it is correct.



A proper coloring of a set of intervals using five colors.

6. (a) For every integer n, find a frequency array $f[1..n]$ whose Huffman code tree has depth $n-1$, such that the largest frequency is as small as possible.

   (b) Suppose the total length N of the *unencoded* message is bounded by a polynomial in the alphabet size n. Prove that the any Huffman tree for the frequencies $f[1..n]$ has depth $O(\log n)$.

♥7. Call a frequency array $f[1..n]$ ***α-heavy*** if it satisfies two conditions:

   - $f[1] > f[i]$ for all $i > 1$; that is, 1 is the *unique* most frequent symbol.
   - $f[1] \geq \alpha \sum_{i=1}^{n} f[i]$; that is, at least an α fraction of the symbols are 1s.

   Find the largest real number α such that in every Huffman code for every α-heavy frequency array, symbol 1 is represented by a single bit. *[Hint: First prove that* $1/3 \leq \alpha \leq 1/2$.*]*

8. Describe and analyze an algorithm to compute an optimal *ternary* prefix-free code for a given array of frequencies $f[1..n]$. Don't forget to prove that your algorithm is correct for *all* n.

9. Describe in detail how to implement the Gale-Shapley stable matching algorithm, so that the worst-case running time is $O(n^2)$, as claimed earlier in this chapter.

10. (a) Prove that it is possible for the Gale-Shapley algorithm to perform $\Omega(n^2)$ offers before termination. (You need to describe both a suitable input and a sequence of $\Omega(n^2)$ valid offers.)

    (b) Describe for any integer n a set of preferences for n doctors and n hospitals that forces the Gale-Shapley algorithm to execute $\Omega(n^2)$ rounds, *no matter which valid proposal is made in each round. [Hint: Part (b) implies part (a).]*

11. Describe and analyze an efficient algorithm to determine whether a given set of hospital and doctor preferences has to a *unique* stable matching.

12. Consider a generalization of the stable matching problem, where some doctors do not rank all hospitals and some hospitals do not rank all doctors, and a doctor can be assigned to a hospital only if each appears in the other's preference list. In this case, there are three additional unstable situations:

    - A matched hospital prefers an unmatched doctor to its assigned match.
    - A matched doctor prefers an unmatched hospital to her assigned match.
    - An unmatched doctor and an unmatched hospital appear in each other's preference lists.

    A stable matching in this setting may leave some doctors and/or hospitals unmatched, even though their preference lists are non-empty. For example, if every doctor lists Harvard as their only acceptable hospital, and every hospital lists Dr. House as their only acceptable intern, then only House and Harvard will be matched.

    Describe and analyze an efficient algorithm that computes a stable matching in this more general setting. *[Hint: Reduce to an instance where every doctor ranks every hospital and vice versa, and then invoke Gale-Shapley.]*

13. The Scandinavian furniture company Fürni has hired n drivers to deliver n identical orders to n different addresses in Wilmington, Delaware. Each driver has their own well-established delivery route through Wilmington that visits all n addresses. Assuming they follow their routes as they always do, two drivers never visit the same addresses at the same time.

    In principle, each of the n drivers can deliver their furniture to any of the n addresses, but there's a complication. One of the drivers has secretly wired proximity sensors and explosives to the Johannshamn sofas (with the Strinne green stripe pattern). If two sofas are ever at the same address at the same time, both will explode, destroying both the delivery truck and the building at that address. This can only happen if one driver delivers an order to that address, and then later another driver visits that same address *while the furniture is still on their truck*.

    Your job as the Fürni dispatcher is to assign each driver to a delivery address. Describe an algorithm to assign addresses to drivers so that each of the n addresses receives their furniture order and there are no explosions.

    For example, suppose Jack's route visits 537 Paper Street at 6pm and 1888 Franklin Street at 8pm, and Marla's route visits 537 Paper at 7pm and 1888 Franklin at 9pm. Then Jack should deliver to 1888 Franklin, and Marla should deliver to 537 Paper; otherwise, there would be an explosion at 1888 Franklin at 8pm. (Cue the Pixies.) *[Hint: Jack and Marla are a bit unstable.]*

14. Suppose you are a simple shopkeeper living in a country with n different types of coins, with values $1 = c[1] < c[2] < \cdots < c[n]$. (In the U.S., for example, $n = 6$ and the values are 1, 5, 10, 25, 50 and 100 cents.) Your beloved and benevolent dictator, El Generalissimo, has decreed that whenever you give a customer change, you must use the smallest possible number of coins, so as not to wear out the image of El Generalissimo lovingly engraved on each coin by servants of the Royal Treasury.

    (a) In the United States, there is a simple greedy algorithm that always results in the smallest number of coins: subtract the largest coin and recursively give change for the remainder. El Generalissimo does not approve of American capitalist greed. Show that there is a set of coin values for which the greedy algorithm does *not* always give the smallest possible of coins.

    (b) Now suppose El Generalissimo decides to impose a currency system where the coin denominations are consecutive powers $b^0, b^1, b^2, \ldots, b^k$ of some integer $b \geq 2$. Prove that despite El Generalissimo's disapproval, the greedy algorithm described in part (a) does make optimal change in this currency system.

    (c) Describe and analyze an efficient algorithm to determine, given a target amount T and a sorted array $c[1\,..\,n]$ of coin denominations, the smallest number of coins needed to make T cents in change. Assume that $c[1] = 1$, so that it is possible to make change for any amount T.

15. Suppose you are given an array $A[1\,..\,n]$ of integers, each of which may be positive, negative, or zero. A contiguous subarray $A[i\,..\,j]$ is called a ***positive interval*** if the sum of its entries is greater than zero. Describe and analyze an algorithm to compute the minimum number of positive intervals that cover every positive entry in A. For example, given the following array as input, your algorithm should output 3. If every entry in the input array is negative, your algorithm should output 0.

$$\boxed{\overbrace{+3 \mid -5 \mid +7 \mid -4 \mid +1}^{\text{sum}=2} \mid -8 \mid \overbrace{+3 \mid -7 \mid +5}^{\text{sum}=1} \mid -9 \mid \overbrace{+5 \mid -2 \mid +4}^{\text{sum}=7}}$$

16. Consider the following process. At all times you have a single positive integer x, which is initially equal to 1. In each step, you can either *increment* x or *double* x. Your goal is to produce a target value n. For example, you can produce the integer 10 in four steps as follows:

$$1 \xrightarrow{+1} 2 \xrightarrow{\times 2} 4 \xrightarrow{+1} 5 \xrightarrow{\times 2} 10$$

Obviously you can produce any integer n using exactly $n-1$ increments, but for almost all values of n, this is horribly inefficient. Describe and analyze an algorithm to compute the *minimum* number of steps required to produce any given integer n.

17. Suppose we have n skiers with heights given in an array $P[1..n]$, and n skis with heights given in an array $S[1..n]$. Describe an efficient algorithm to assign a ski to each skier, so that the average difference between the height of a skier and her assigned ski is as small as possible. The algorithm should compute a permutation σ such that the expression

$$\frac{1}{n}\sum_{i=1}^{n}\left|P[i]-S[\sigma(i)]\right|$$

is as small as possible.

18. Alice wants to throw a party and she is trying to decide who to invite. She has n people to choose from, and she knows which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints:

    - For each guest, there should be at least five other guests that they already know.
    - For each guest, there should be at least five other guests that they *don't* already know.

    Describe and analyze an algorithm that computes the largest possible number of guests Alice can invite, given a list of n people and the list of pairs who know each other.

19. Suppose we are given two arrays $C[1..n]$ and $R[1..n]$ of positive integers. An $n \times n$ matrix of 0s and 1s *agrees with R and C* if, for every index i, the ith row contains $R[i]$ 1s, and the ith column contains $C[i]$ 1s. Describe and analyze an algorithm that either constructs a matrix that agrees with R and C, or correctly reports that no such matrix exists.

20. You've just accepted a job from Elon Musk, delivering burritos from San Francisco to New York City. You get to drive a Burrito-Delivery Vehicle through Elon's new Transcontinental Underground Burrito-Delivery Tube, which runs in a direct line between these two cities.

    Your Burrito-Delivery Vehicle runs on single-use batteries, which must be replaced after at most 100 miles. The actual fuel is virtually free, but the batteries are expensive and fragile, and therefore must be installed only by official members of the Transcontinental Underground Burrito-Delivery Vehicle Battery-Replacement Technicians' Union.[12] Thus, even if you replace

[12]or as they call themselves in German, Die Transkontinentaluntergrundburritolieferfahrzeug-batteriewechseltechnikervereinigung.

your battery early, you must still pay full price for each new battery to be installed. Moreover, your Vehicle is too small to carry more than one battery at a time.

There are several fueling stations along the Tube; each station charges a different price for installing a new battery. Before you start your trip, you carefully print the Wikipedia page listing the locations and prices of every fueling station along the Tube. Given this information, how do you decide the best places to stop for fuel?

More formally, suppose you are given two arrays $D[1..n]$ and $C[1..n]$, where $D[i]$ is the distance from the start of the Tube to the ith station, and $C[i]$ is the cost to replace your battery at the ith station. Assume that your trip starts and ends at fueling stations (so $D[1] = 0$ and $D[n]$ is the total length of your trip), and that your car starts with an empty battery (so you must install a new battery at station 1).

(a) Describe and analyze a greedy algorithm to find the minimum number of refueling stops needed to complete your trip. Don't forget to prove that your algorithm is correct.

(b) But what you really want to minimize is the total *cost* of travel. Show that your greedy algorithm in part (a) does *not* produce an optimal solution when extended to this setting.

(c) Describe an efficient algorithm to compute the locations of the fuel stations you should stop at to minimize the total cost of travel.

21. You've been hired to store a sequence of n books on shelves in a library. The order of the books is fixed by the cataloging system and cannot be changed; each shelf must store a contiguous interval of the given sequence of books. You are given two arrays $H[1..n]$ and $T[1..n]$, where $H[i]$ and $T[i]$ are respectively the height and thickness of the ith book in the sequence. All shelves in this library have the same length L; the total thickness of all books on any single shelf cannot exceed L.

(a) Suppose all the books have the same height h and the shelves have height larger than h, so every book fits on every shelf. Describe and analyze a greedy algorithm to store the books in as few shelves as possible. *[Hint: The algorithm is obvious, but why is it correct?]*

(b) That was a nice warmup, but now here's the real problem. In fact the books have different heights, but you can adjust the height of each shelf to match the tallest book on that shelf. (In particular, you can change the height of any empty shelf to zero.) Now your task is to store the books so that the sum of the heights of the shelves is as small as possible. Show that your greedy algorithm from part (a) does *not* always give the best solution to this problem.

(c) Describe and analyze an algorithm to find the best matching between books and shelves as described in part (b).

22. A string w of parentheses (and) is ***balanced*** if it satisfies one of the following conditions:

- w is the empty string.
- $w = (x)$ for some balanced string x
- $w = xy$ for some balanced strings x and y

For example, the string

$$w = ((())()())(()())()$$

is balanced, because $w = xy$, where

$$x = ((())()()) \qquad \text{and} \qquad y = (()())().$$

(a) Describe and analyze an algorithm to determine whether a given string of parentheses is balanced.

(b) Describe and analyze a greedy algorithm to compute the length of a longest balanced subsequence of a given string of parentheses. As usual, don't forget to prove your algorithm is correct.

For both problems, your input is an array $w[1..n]$, where for each i, either $w[i] = ($ or $w[i] =)$. Both of your algorithms should run in $O(n)$ time.

23. One day Alex got tired of climbing in a gym and decided to take a large group of climber friends outside to climb. They went to a climbing area with a huge wide boulder, not very tall, with several marked hand and foot holds. Alex quickly determined an "allowed" set of moves that her group of friends can perform to get from one hold to another.

The overall system of holds can be described by a rooted tree T with n vertices, where each vertex corresponds to a hold and each edge corresponds to an allowed move between holds. The climbing paths converge as they go up the boulder, leading to a unique hold at the summit, represented by the root of T.

Alex and her friends (who are all excellent climbers) decided to play a game, where as many climbers as possible are simultaneously on the boulder and each climber needs to perform a sequence of *exactly* k moves. Each climber can choose an arbitrary hold to start from, and all moves must move away from the ground. Thus, each climber traces out a path of k edges in the tree T, all directed toward the root. However, no two climbers are allowed to touch the same hold; the paths followed by different climbers cannot intersect at all.

(a) Describe and analyze a greedy algorithm to compute the maximum number of climbers that can play this game. Your algorithm is given a rooted tree T and an integer k as input, and it should compute the largest possible number of disjoint paths in T, where each path has length k. Do ***not*** assume that T is a binary tree. For example, given the tree below as input, your algorithm should return the integer 8.

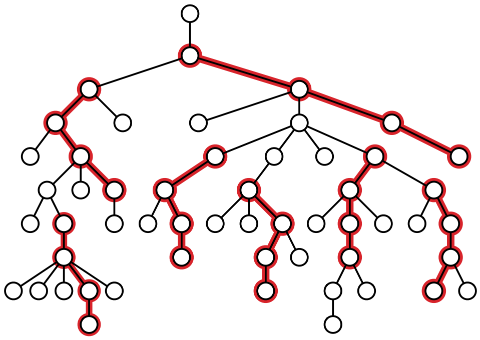


Figure 4.4. Seven disjoint paths of length $k = 3$. This is *not* the largest such set of paths in this tree.

(b) Now suppose each vertex in T has an associated *reward*, and your goal is to maximize the total reward of the vertices in your paths, instead of the total number of paths. Show that your greedy algorithm does *not* always return the optimal reward.

(c) Describe an efficient algorithm to compute the maximum possible reward, as described in part (b).

24. Congratulations! You have successfully conquered Camelot, transforming the former battle-scarred hereditary monarchy into an anarcho-syndicalist commune, where citizens take turns to act as a sort of executive-officer-for-the-week, but with all the decisions of that officer ratified at a special bi-weekly meeting, by a simple majority in the case of purely internal affairs, but by a two-thirds majority in the case of more major. . . .

As a final symbolic act, you order the Round Table (surprisingly, an actual circular table) to be split into pizza-like wedges and distributed to the citizens of Camelot as trophies. Each citizen has submitted a request for an angular wedge of the table, specified by two angles—for example: Sir Robin the Brave might request the wedge from $17.23°$ to $42°$, and Sir Lancelot the Pure might request the $2°$ wedge from $359°$ to $1°$. Each citizen will be happy if and only if they receive *precisely* the wedge that they requested. Unfortunately, some of these ranges overlap, so satisfying *all* the citizens' requests is simply impossible. Welcome to politics.

Describe and analyze an algorithm to find the maximum number of requests that can be satisfied. *[Hint: The output of your algorithm should not change if you rotate the table. Do not assume that angles are integers.]*

25. Suppose you are standing in a field surrounded by several large balloons. You want to use your brand new Acme Brand Zap-O-Matic$^{\text{TM}}$ to pop all the balloons, without moving from your current location. The Zap-O-Matic$^{\text{TM}}$ shoots a high-powered laser beam, which pops all the balloons it hits. Since each shot requires enough energy to power a small country for a year, you want to fire as few shots as possible.

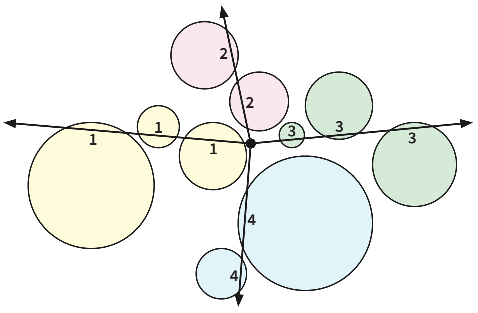


Figure 4.5. Nine balloons popped by four shots of the Zap-O-Matic$^{\text{TM}}$

The *minimum zap* problem can be stated more formally as follows. Given a set C of n circles in the plane, each specified by its radius and the (x, y) coordinates of its center, compute the minimum number of rays from the origin that intersect every circle in C. Your goal is to find an efficient algorithm for this problem.

(a) Suppose it is possible to shoot a ray that does not intersect any balloons. Describe and analyze a greedy algorithm that solves the minimum zap problem in this special case. *[Hint: See Exercise 2.]*

(b) Describe and analyze a greedy algorithm whose output is within 1 of optimal. That is, if m is the minimum number of rays required to hit every balloon, then your greedy algorithm must output either m or $m+1$. (Of course, you must prove this fact.)

(c) Describe an algorithm that solves the minimum zap problem in $O(n^2)$ time.

♥(d) Describe an algorithm that solves the minimum zap problem in $O(n \log n)$ time.

Assume you have a subroutine that tells you the range of angles of rays that intersects an arbitrary circle c in $O(1)$ time. This subroutine is not difficult to write, but it's not the interesting part of the problem.