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.

— Michael Douglas as Gordon Gekko, *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)

7 Greedy Algorithms

7.1 Storing Files on Tape

Suppose we have a set of n files that we want to store on a tape. 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

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

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 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 prove the greedy algorithm is actually correct, we simply prove that the output of any other algorithm can be improved by some sort of swap.

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?

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 1. But what if the sizes and the frequencies are both different? In this case, we should sort the files by the ratio L/f.

Lemma 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, since

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

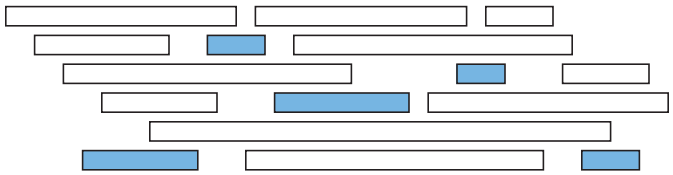
Thus, if two adjacent files are out of order, we can improve the total cost by swapping them. □

7.2 Scheduling Classes

The next example is slightly less trivial. Suppose you decide to drop out of computer science at the last minute 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.[1] 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 7:06pm, and so on. In the interest of graduating as quickly as possible, you want to register for as many classes as you can. (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. 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.

[1] but interestingly, *not* by the faculty



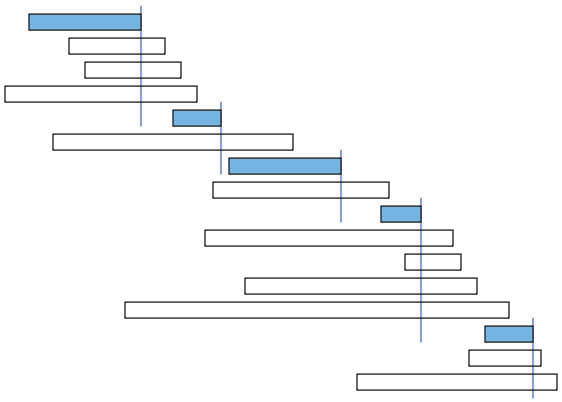
A maximal conflict-free schedule for a set of classes.

This problem has a fairly simple recursive solution, based on the observation that either you take class 1 or you don't. Let B_4 denote the set of classes that end *before* class 1 starts, and let L_8 denote the set of classes that start *later* than class 1 ends:

$$B_4 = \{i \mid 2 \le i \le n \text{ and } F[i] < S[1]\} \qquad L_8 = \{i \mid 2 \le i \le n \text{ and } S[i] > F[1]\}$$

If class 1 is in the optimal schedule, then so are the optimal schedules for B_4 and L_8, 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 $O(n^2)$ 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 most remaining classes. If this greedy strategy works, it suggests the following very simple 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!



The same classes sorted by finish times and the 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.

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

[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 need the practice.

To prove that this algorithm actually gives us a maximal 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. (See the figures on the previous page.) All we can claim is that at least one of the maximal schedules is the one that the greedy algorithm produces.

Lemma 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. *The greedy schedule is an optimal schedule.*

Proof: Let f be the class that finishes first, and let L be the subset of classes the 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 L. The greedy algorithm chooses f and then, by the inductive hypothesis, computes an optimal schedule of classes from L. □

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

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

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

where the classes c_i are different from the classes chosen by the greedy algorithm. By construction, the jth greedy choice g_j does not conflict with any earlier class $g_1, g_2, \ldots, g_{j-1}$, and since our schedule 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. This implies that g_j does not conflict with any of the later classes $c_{j+1}, \ldots, c_m$. Thus, the schedule

$$\langle g_1, g_2, \ldots, g_{j-1}, g_j, c_{j+1}, \ldots, c_m \rangle,$$

is conflict-free. (This is just a generalization of Lemma 3, which considers the case $j = 1$.) By induction, it now follows that there is an optimal schedule $\langle g_1, g_2, \ldots, g_k, c_{k+1}, \ldots, c_m \rangle$ that includes every class chosen by the greedy algorithm. But this is impossible unless $k = m$; if there were a class c_{k+1} that does not conflict with g_k, the greedy algorithm would choose more than k classes. □

7.3 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 degrading the solution.

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

7.4 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. 7-bit ASCII and Unicode's UTF-8 are both prefix-free binary codes. Morse code is a binary code, but it is not prefix-free; for example, the code for `S` ($\cdots$) includes the code for `E` ($\cdot$) as a prefix. 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. The length of a codeword for a symbol is the depth of the corresponding leaf.

Let me emphasize that 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 messages 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:[3]

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

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.

For 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 forget about the forty-four spaces, nineteen apostrophes, nineteen commas, three hyphens, and only one period, and just encode the letters. Here's the frequency table:

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 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

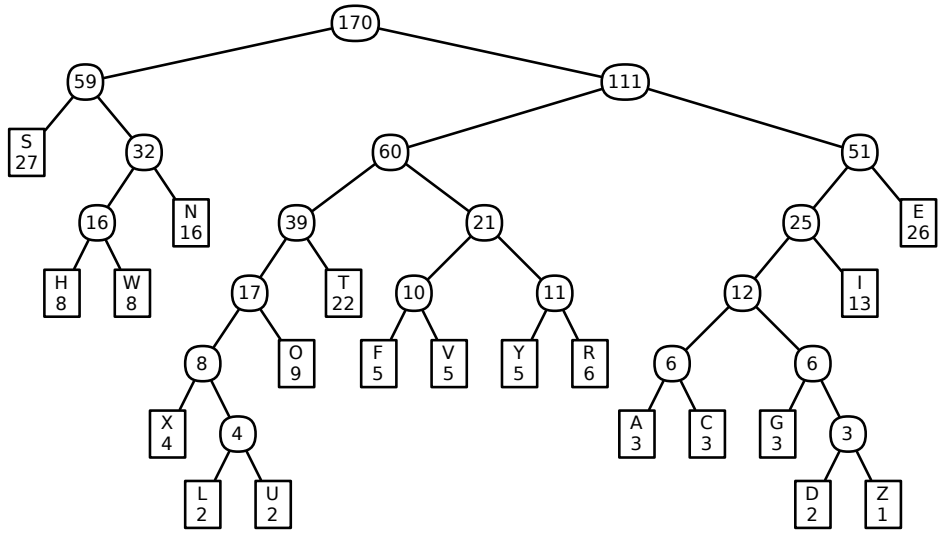
A	C	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	DZ
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3

[3] This looks almost exactly like the cost of a binary search tree, but the optimization problem is very different: code trees are not required to keep the keys in any particular order.

[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 (then open) problem of finding an optimal encoding to his class; Huffman solved the problem as a class project, in lieu of taking a final exam.

[5] A. K. Dewdney. Computer recreations. *Scientific American*, October 1984. Douglas Hofstadter published a few earlier examples of Lee Sallows' self-descriptive sentences in his *Scientific American* column in January 1982.

After 19 merges, all 20 characters 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 code is shown below. For example, the code for A is 110000, and the code for S is 00.



A Huffman code for Lee Sallows' self-descriptive sentence; the numbers are frequencies for merged characters

If we use this code, the encoded message starts like this:

1001	0100	1101	00	00	111	011	1001	111	011	110001	111	110001	10001	011	1001	110000	1101	...
T	H	I	S	S	E	N	T	E	N	C	E	C	O	N	T	A	I	

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

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 would assign different codes, possibly with different lengths, to various characters, but the overall length of the encoded message is the same for any 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. Encoding Lee 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 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. Since 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. The depth of x increases by some amount Δ, and the depth of a decreases by the same amount. Thus,

$$cost(T') = cost(T) - (f[a] - f[x])\Delta.$$

By assumption, x is one of the two least frequent characters, but a is not, which implies that $f[a] \geq f[x]$. 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 does not decrease the cost, either. Thus, T' is also an optimal code tree (and incidentally, $f[a]$ actually equals $f[x]$).

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! Essentially we're relying on the following recursive definition for a full binary tree: either a single node, or a full binary tree where some leaf has been replaced by an internal node with two leaf children.

Theorem 6. *Huffman codes are optimal prefix-free binary codes.*

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

Otherwise, let $f[1..n]$ be the original input frequencies, where without loss of generality, $f[1]$ and $f[2]$ are the two smallest. To keep things simple, let $f[n+1] = f[1] + f[2]$. By the previous lemma, we know that some optimal code for $f[1..n]$ has characters 1 and 2 as siblings.

Let T' be the Huffman code 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, $depth(i)$ denotes the depth of the leaf labelled i in either T or T'; if the leaf appears in both T and T', it has the same depth in both trees.)

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

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 actually implement Huffman codes efficiently, we keep the characters in a min-heap, where the priority of each character is its frequency. We can construct the code tree by keeping three arrays of indices, listing the left and right children and the parent of each node. The root of the tree is the node with index $2n-1$.

```
BUILDHUFFMAN(f[1..n]):
  for i ← 1 to n
      L[i] ← 0; R[i] ← 0
      INSERT(i, f[i])

  for i ← n to 2n − 1
      x ← EXTRACTMIN( )
      y ← EXTRACTMIN( )
      f[i] ← f[x] + f[y]
      L[i] ← x; R[i] ← y
      P[x] ← i; P[y] ← i
      INSERT(i, f[i])

  P[2n − 1] ← 0
```

The algorithm performs $O(n)$ min-heap operations. If we use a balanced binary tree as the heap, each operation requires $O(\log n)$ time, so the total running time of BUILDHUFFMAN is $O(n \log n)$.

Finally, here are simple algorithms to encode and decode messages:

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

HUFFMANENCODEONE(x):
  if x < 2n − 1
      HUFFMANENCODEONE(P[x])
      if x = L[P[x]]
          B[m] ← 0
      else
          B[m] ← 1
      m ← m + 1
```

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

Exercises

1. Consider the following alternative greedy algorithms for the class scheduling problem. For each algorithm, either prove or disprove that it always constructs an optimal schedule. Assume that all four algorithms break ties arbitrarily.

   (a) Choose the course that *ends last*, discard all conflicting classes, and recurse.

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

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

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

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

   (f) Discard the course with *longest duraction* and recurse, stopping when there are no more conflicts.

   (g) Discard a course that *conflicts with the most other courses* and recurse, stopping when there are no more conflicts.

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

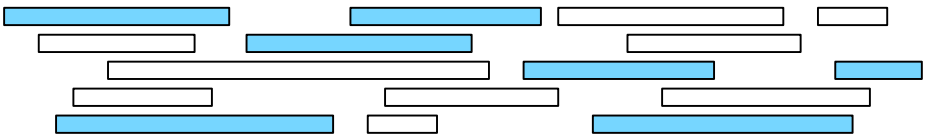
is now to choose a set of non-conflicting classes that give you the largest possible number of credit hours, given an array of start times, end times, and credit hours as input.

(a) Prove that the greedy algorithm described in the notes — Choose the class that ends first and recurse — does *not* always return an optimal schedule.

(b) Describe an algorithm to compute the optimal schedule in $O(n^2)$ time.

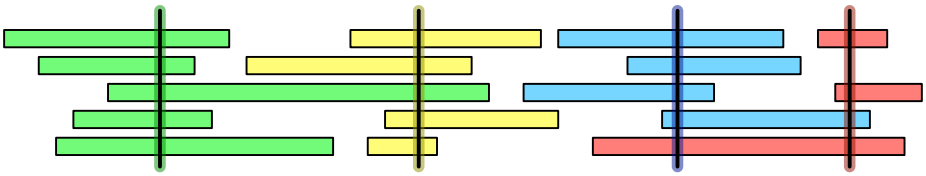
3. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a *tiling path* if the intervals in Y cover the intervals in X, that is, any real value that is contained in some interval in X is also contained in some interval in Y. The *size* of a tiling cover is just the number of intervals.

   Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, 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. The seven shaded intervals form a tiling path.

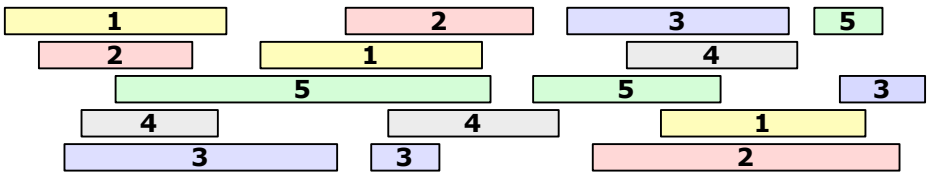
4. Let X be a set of n intervals on the real line. We say that a set P of points *stabs* X if every interval in X contains at least one point in P. Describe and analyze an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $X_L[1..n]$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

6. 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



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

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 A and a sorted array $c[1..n]$ of coin denominations, the smallest number of coins needed to make A cents in change. Assume that $c[1] = 1$, so that it is possible to make change for any amount A.

7. Suppose you have just purchased a new type of hybrid car that uses fuel extremely efficiently, but can only travel 100 miles on a single battery. The car's fuel is stored in a single-use battery, which must be replaced after at most 100 miles. The actual fuel is virtually free, but the batteries are expensive and can only be installed by licensed battery-replacement technicians. Thus, even if you decide to replace your battery early, you must still pay full price for the new battery to be installed. Moreover, because these batteries are in high demand, no one can afford to own more than one battery at a time.

   Suppose you are trying to get from San Francisco to New York City on the new Inter-Continental Super-Highway, which runs in a direct line between these two cities. There are several fueling stations along the way; 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 on the ICSH. 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 highway 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.

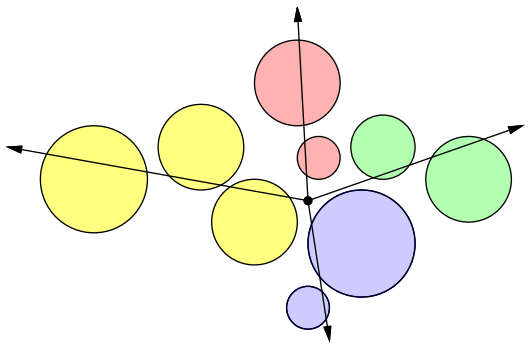
(c) Describe an efficient algorithm to compute the locations of the fuel stations you should stop at to minimize the total cost of travel.

8. Congratulations! You have successfully conquered Camelot, transforming the former battle-scarred kingdom with 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°. 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: Careful! The output of your algorithm must not change if you rotate the table. Do not assume that angles are integers.]*

9. Suppose you are standing in a field surrounded by several large balloons. You want to use your brand new Acme Brand Zap-O-Matic™ to pop all the balloons, without moving from your current location. The Zap-O-Matic™ 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.



Nine balloons popped by 4 shots of the Zap-O-Matic™

   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 INTERSECTS(r, c) that determines whether an arbitrary ray r 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.