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)

I love deadlines. I like the whooshing sound they make as they fly by.

— Douglas Adams

6 Greedy Algorithms

6.1 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 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 $L[b]F[a] - L[a]F[a]$. But this change is an improvement, since

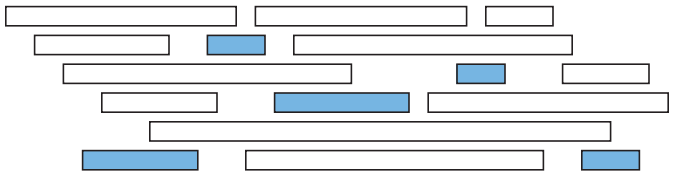
$$\frac{L[a]}{F[a]} > \frac{L[b]}{F[b]} \implies L[b]F[a] - L[a]F[a] < 0.$$

Thus, if the files are out of order, we can improve the total cost by swapping some mis-ordered adjacent pair. □

6.2 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 has all of its classes on the same day every week, referred to as "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 7:06pm, and so on. In the interests 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.



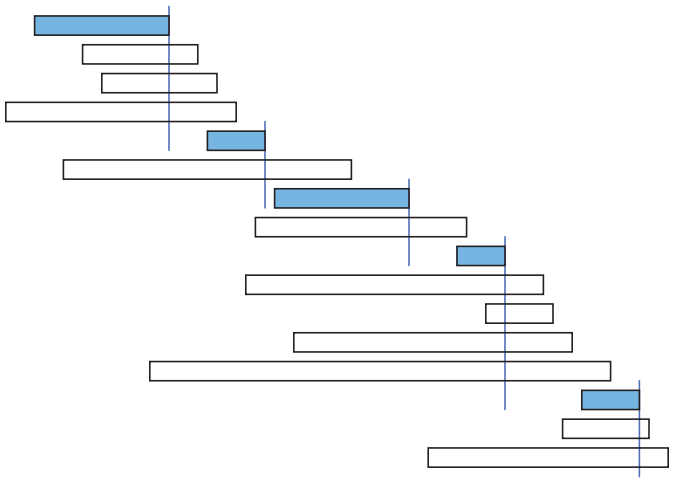
A maximal conflict-free schedule for a set of classes.

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 be the set of classes that end *before* class 1 starts, and let L_8 be 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.[1]

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.)

```
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]
```

[1] But you should still work out the details yourself. The dynamic programming algorithm can be used to find the "best" schedule for any definition of "best", but the greedy algorithm I'm about to describe only works that "best" means "biggest". Also, you need the practice.

This algorithm clearly runs in $O(n \log n)$ time.

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. □

6.3 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.

6.4 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 ($\cdot\cdot\cdot$) 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. (Note that the code tree is *not* a binary search tree. We don't care at all about the sorted order of symbols at the leaves. (In fact. the symbols may not have a well-defined order!)

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:[2]

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

In 1952, David Huffman developed the following greedy algorithm to produce such an optimal code:

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:[3]

> 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 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

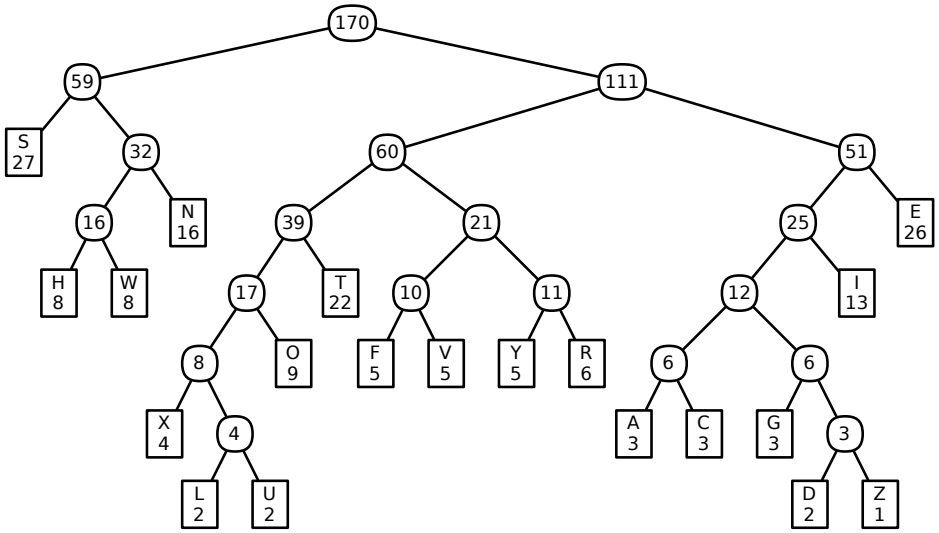
Huffman's algorithm picks out the two least frequent letters, breaking ties arbitrarily—in this case, say, Z and D—and merges them together into a single new character DZ with frequency 3. This new character becomes an internal node in the code tree we are constructing, with Z and D as its children; 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

[2]This looks almost exactly like the cost of a binary search tree, but the optimization problem is very different: code trees are **not** search trees!

[3]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.



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

For example, the code for A is 110000, and the code for S is 00. 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, 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 as 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
```

6.5 Matroids

Many problems that can be correctly solved by greedy algorithms can be described in terms of abstract combinatorial objects called *matroids*. Matroids were first described in 1935 by the mathematician Hassler Whitney as a combinatorial generalization of linear independence of vectors.

A matroid $\mathcal{M}$ is a finite collection of finite sets that satisfies three axioms:

- **Non-emptiness:** The empty set is in $\mathcal{M}$. (Thus, $\mathcal{M}$ is not itself empty.)
- **Heredity:** If a set X is an element of $\mathcal{M}$, then any subset of X is also in $\mathcal{M}$.
- **Exchange:** If X and Y are two sets in $\mathcal{M}$ and $|X| > |Y|$, then there is an element $x \in X \setminus Y$ such that $Y \cup \{x\}$ is in $\mathcal{M}$.

The sets in $\mathcal{M}$ are typically called *independent sets*; for example, we would say that any subset of an independent set is independent. The union of all sets in $\mathcal{M}$ is called the *ground set*. An independent set is called a *basis* if it is not a proper subset of another independent set. The exchange property implies that every basis of a matroid has the same cardinality. The *rank* of a subset X of the ground set is the size of the largest independent subset of X. A subset of the ground set that is not in $\mathcal{M}$ is called *dependent* (surprise, surprise). Finally, a dependent set is called a *circuit* if every proper subset is independent.

Most of this terminology is justified by Whitney's original example:

> **Linear matroid:** Let A be any $n \times m$ matrix. A subset $I \subseteq \{1, 2, \ldots, n\}$ is independent if and only if the corresponding subset of columns of A is linearly independent.

The heredity property follows directly from the definition of linear independence; the exchange property is implied by an easy dimensionality argument. A basis in any linear matroid is also a basis (in the linear-algebra sense) of the vector space spanned by the columns of A. Similarly, the rank of a set of indices is precisely the rank (in the linear-algebra sense) of the corresponding set of column vectors.

Here are several other examples of matroids; some of these we will see again later. I will leave the proofs that these are actually matroids as exercises for the reader.

- **Uniform matroid $U_{k,n}$:** A subset $X \subseteq \{1, 2, \ldots, n\}$ is independent if and only if $|X| \leq k$. Any subset of $\{1, 2, \ldots, n\}$ of size k is a basis; any subset of size $k+1$ is a circuit.
- **Graphic/cycle matroid $\mathcal{M}(G)$:** Let $G = (V, E)$ be an arbitrary undirected graph. A subset of E is independent if it defines an *acyclic* subgraph of G. A basis in the graphic matroid is a spanning tree of G; a circuit in this matroid is a cycle in G.
- **Cographic/cocycle matroid $\mathcal{M}^*(G)$:** Let $G = (V, E)$ be an arbitrary undirected graph. A subset $I \subseteq E$ is independent if the complementary subgraph $(V, E \setminus I)$ of G is *connected*. A basis in this matroid is the complement of a spanning tree; a circuit in this matroid is a *cocycle*—a minimal set of edges that disconnects the graph.
- **Matching matroid:** Let $G = (V, E)$ be an arbitrary undirected graph. A subset $I \subseteq V$ is independent if there is a matching in G that covers I.
- **Disjoint path matroid:** Let $G = (V, E)$ be an arbitrary *directed* graph, and let s be a fixed vertex of G. A subset $I \subseteq V$ is independent if and only if there are edge-disjoint paths from s to each vertex in I.

Now suppose each element of the ground set of a matroid $\mathcal{M}$ is given an arbitrary non-negative weight. The ***matroid optimization problem*** is to compute a basis with maximum total weight. For example, if $\mathcal{M}$ is the cycle matroid for a graph G, the matroid optimization problem asks us to find the maximum spanning tree of G. Similarly, if $\mathcal{M}$ is the cocycle matroid for G, the matroid optimization problem seeks (the complement of) the *minimum* spanning tree.

The following natural greedy strategy computes a basis for any weighted matroid:

```
GreedyBasis(M, w):
  X[1..n] ← ⋃M (the ground set)
  sort X in decreasing order of weight w
  G ← ∅
  for i ← 1 to n
      if G ∪ {X[i]} ∈ M
          add X[i] to G
  return G
```

Suppose we can test in $F(n)$ whether a given subset of the ground set is independent. Then this algorithm runs in $O(n \log n + n \cdot F(n))$ time.

Theorem 7. *For any matroid $\mathcal{M}$ and any weight function w,* GreedyBasis$(\mathcal{M}, w)$ *returns a maximum-weight basis of $\mathcal{M}$.*

Proof: Let $G = \{g_1, g_2, \ldots, g_k\}$ be the independent set returned by GREEDYBASIS$(\mathcal{M}, w)$. If any other element could be added to G to get a larger independent set, the greedy algorithm would have added it. Thus, G is a basis.

For purposes of deriving a contradiction, suppose there is an independent set $H = \{h_1, h_2, \ldots, h_\ell\}$ such that

$$\sum_{i=1}^{k} w(g_i) < \sum_{j=1}^{\ell} w(h_i).$$

Without loss of generality, we assume that H is a basis. The exchange property now implies that $k = \ell$.

Now suppose the elements of G and H are indexed in order of decreasing weight. Let i be the smallest index such that $w(g_i) < w(h_i)$, and consider the independent sets

$$G_{i-1} = \{g_1, g_2, \ldots, g_{i-1}\} \quad \text{and} \quad H_i = \{h_1, h_2, \ldots, h_{i-1}, h_i\}.$$

By the exchange property, there is some element $h_j \in H_i$ such that $G_{i-1} \cup \{h_j\}$ is an independent set. We have $w(h_j) \geq w(h_i) > w(g_i)$. Thus, the greedy algorithm considers *and rejects* the heavier element h_j before it considers the lighter element g_i. But this is impossible—the greedy algorithm accepts elements in decreasing order of weight. □

We now immediately have a correct greedy optimization algorithm for *any* matroid. Returning to our examples:

- Linear matroid: Given a matrix A, compute a subset of vectors of maximum total weight that span the column space of A.
- Uniform matroid: Given a set of weighted objects, compute its k largest elements.
- Cycle matroid: Given a graph with weighted edges, compute its maximum spanning tree. In this setting, the greedy algorithm is better known as *Kruskal's algorithm*.
- Cocycle matroid: Given a graph with weighted edges, compute its minimum spanning tree.
- Matching matroid: Given a graph, determine whether it has a perfect matching.
- Disjoint path matroid: Given a directed graph with a special vertex s, find the largest set of edge-disjoint paths from s to other vertices.

The exchange condition for matroids turns out to be crucial for the success of this algorithm. A *subset system* is a finite collection $\mathcal{S}$ of finite sets that satisfies the heredity condition—If $X \in \mathcal{S}$ and $Y \subseteq X$, then $Y \in \mathcal{S}$—but not necessarily the exchange condition.

Theorem 8. *For any subset system $\mathcal{S}$ that is not a matroid, there is a weight function w such that* GREEDYBASIS$(\mathcal{S}, w)$ *does* ***not*** *return a maximum-weight set in $\mathcal{S}$.*

Proof: Let X and Y be two sets in $\mathcal{S}$ that violate the exchange property—$|X| > |Y|$, but for any element $x \in X \setminus Y$, the set $Y \cup \{x\}$ is not in $\mathcal{S}$. Let $m = |Y|$. We define a weight function as follows:

- Every element of Y has weight $m + 2$.
- Every element of $X \setminus Y$ has weight $m + 1$.
- Every other element of the ground set has weight zero.

With these weights, the greedy algorithm will consider and accept every element of Y, then consider and reject every element of X, and finally consider all the other elements. The algorithm returns a set with total weight $m(m+2) = m^2 + 2m$. But the total weight of X is at least $(m+1)^2 = m^2 + 2m + 1$. Thus, the output of the greedy algorithm is not the maximum-weight set in $\mathcal{S}$. □

Recall the Applied Chaos scheduling problem considered earlier in this lecture. There is a natural subset system associated with this problem: A set of classes is independent if and only if not two classes overlap. (This is just the graph-theory notion of 'independent set'!) This subset system is *not* a matroid, because there can be maximal independent sets of different sizes, which violates the exchange property. If we consider a *weighted* version of the class scheduling problem, say where each class is worth a different number of hours, Theorem 8 implies that the greedy algorithm will *not* always find the optimal schedule. (In fact, there's an easy counterexample with only two classes!) However, Theorem 8 does *not* contradict the correctness of the greedy algorithm for the original *unweighted* problem, however; that problem uses a particularly lucky choice of weights (all equal).

6.6 Scheduling with Deadlines

Suppose you have n tasks to complete in n days; each task requires your attention for a full day. Each task comes with a *deadline*, the last day by which the job should be completed, and a *penalty* that you must pay if you do not complete each task by its assigned deadline. What order should you perform your tasks in to minimize the total penalty you must pay?

More formally, you are given an array $D[1..n]$ of deadlines an array $P[1..n]$ of penalties. Each deadline $D[i]$ is an integer between 1 and n, and each penalty $P[i]$ is a non-negative real number. A *schedule* is a permutation of the integers $\{1, 2, \ldots, n\}$. The scheduling problem asks you to find a schedule π that minimizes the following cost:

$$cost(\pi) := \sum_{i=1}^{n} P[i] \cdot [\pi(i) > D[i]].$$

This doesn't look anything like a matroid optimization problem. For one thing, matroid optimization problems ask us to find an optimal *set*; this problem asks us to find an optimal *permutation*. Surprisingly, however, this scheduling problem is actually a matroid optimization in disguise! For any schedule π, call tasks i such that $\pi(i) > D[i]$ *late*, and all other tasks *on time*. The following trivial observation is the key to revealing the underlying matroid structure.

The cost of a schedule is determined by the subset of tasks that are on time.

Call a subset X of the tasks *realistic* if there is a schedule π in which every task in X is on time. We can precisely characterize the realistic subsets as follows. Let $X(t)$ denote the subset of tasks in X whose deadline is on or before t:

$$X(t) := \{i \in X \mid D[i] \leq t\}.$$

In particular, $X(0) = \emptyset$ and $X(n) = X$.

Lemma 9. *Let $X \subseteq \{1, 2, \ldots, n\}$ be an arbitrary subset of the n tasks. X is realistic if and only if $|X(t)| \leq t$ for every integer t.*

Proof: Let π be a schedule in which every task in X is on time. Let i_t be the tth task in X to be completed. On the one hand, we have $\pi(i_t) \geq t$, since otherwise, we could not have completed $t-1$

other jobs in X before i_t. On the other hand, $\pi(i_t) \le D[i]$, because i_t is on time. We conclude that $D[i_t] \ge t$, which immediately implies that $|X(t)| \le t$.

Now suppose $|X(t)| \le t$ for every integer t. If we perform the tasks in X in increasing order of deadline, then we complete all tasks in X with deadlines t or less by day t. In particular, for any $i \in X$, we perform task i on or before its deadline $D[i]$. Thus, X is realistic. □

We can define a *canonical schedule* for any set X as follows: execute the tasks in X in increasing deadline order, and then execute the remaining tasks in any order. The previous proof implies that a set X is realistic if and only if every task in X is on time in the canonical schedule for X. Thus, our scheduling problem can be rephrased as follows:

> Find a realistic subset X such that $\sum_{i \in X} P[i]$ is maximized.

So we're looking for optimal subsets after all.

Lemma 10. *The collection of realistic sets of jobs forms a matroid.*

Proof: The empty set is vacuously realistic, and any subset of a realistic set is clearly realistic. Thus, to prove the lemma, it suffices to show that the exchange property holds. Let X and Y be realistic sets of jobs with $|X| > |Y|$.

Let t^* be the largest integer such that $|X(t^*)| \le |Y(t^*)|$. This integer must exist, because $|X(0)| = 0 \le 0 = |Y(0)|$ and $|X(n)| = |X| > |Y| = |Y(n)|$. By definition of t^*, there are more tasks with deadline $t^* + 1$ in X than in Y. Thus, we can choose a task j in $X \setminus Y$ with deadline $t^* + 1$; let $Z = Y \cup \{j\}$.

Let t be an arbitrary integer. If $t \le t^*$, then $|Z(t)| = |Y(t)| \le t$, because Y is realistic. On the other hand, if $t > t^*$, then $|Z(t)| = |Y(t)| + 1 \le |X(t)| < t$ by definition of t^* and because X is realistic. The previous lemma now implies that Z is realistic. This completes the proof of the exchange property. □

This lemma implies that our scheduling problem is a matroid optimization problem, so the greedy algorithm finds the optimal schedule.

```
GreedySchedule(D[1..n], P[1..n]):
  Sort P in increasing order, and permute D to match
  j ← 0
  for i ← 1 to n
      X[j+1] ← i
      if X[1..j+1] is realistic
          j ← j+1
  return the canonical schedule for X[1..j]
```

To turn this outline into a real algorithm, we need a procedure to test whether a given subset of jobs is realistic. Lemma 9 immediately suggests the following strategy to answer this question in $O(n)$ time.

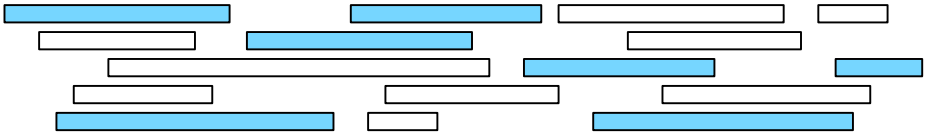
```
Realistic?(X[1..m], D[1..n]):
  ⟨⟨X is sorted by increasing deadline: i ≤ j ⟹ D[X[i]] ≤ D[X[j]]⟩⟩
  N ← 0
  j ← 0
  for t ← 1 to n
      if D[X[j]] = t
          N ← N+1;  j ← j+1
      ⟨⟨Now N = |X(t)|⟩⟩
      if N > t
          return False
  return True
```

If we use this subroutine, GREEDYSCHEDULE runs in $O(n^2)$ time. By using some appropriate data structures, the running time can be reduced to $O(n \log n)$; details are left as an exercise for the reader.

Exercises

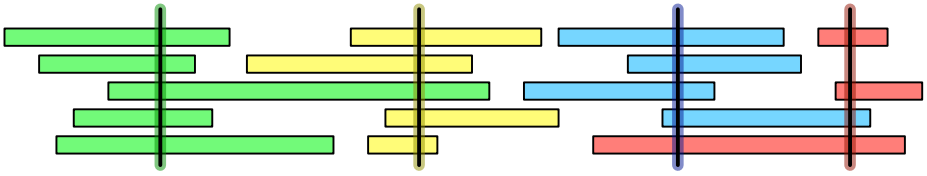
1. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a *tiling path* if the intervals in Y cover the intervals in X, that is, any real value that is contained in some interval in X is also contained in some interval in Y. The *size* of a tiling cover is just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



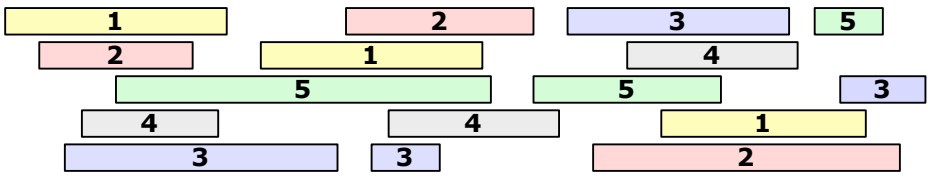
A set of intervals. The seven shaded intervals form a tiling path.

2. 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)

3. 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.



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

4. Prove that for any graph G, the 'graphic matroid' $\mathcal{M}(G)$ is in fact a matroid.

5. Prove that for any graph G, the 'cographic matroid' $\mathcal{M}^*(G)$ is in fact a matroid.

6. Prove that for any graph G, the 'matching matroid' of G is in fact a matroid. *[Hint: What is the symmetric difference of two matchings?]*

7. Prove that for any directed graph G and any vertex s of G, the resulting 'disjoint path matroid' of G is in fact a matroid. *[Hint: This question is **much** easier if you're already familiar with maximum flows.]*

8. Let G be an undirected graph. A set of cycles $\{c_1, c_2, \ldots, c_k\}$ in G is called *redundant* if every edge in G appears in an even number of c_i's. A set of cycles is *independent* if it contains no redundant subset. A maximal independent set of cycles is called a *cycle basis* for G.

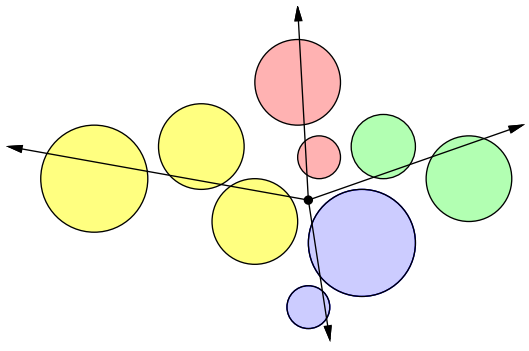
   (a) Let C be any cycle basis for G. Prove that for any cycle γ in G, there is a subset $A \subseteq C$ such that $A \cap \{\gamma\}$ is redundant. In other words, γ is the 'exclusive or' of the cycles in A.

   (b) Prove that the set of independent cycle sets form a matroid.

   *(c) Now suppose each edge of G has a weight. Define the weight of a cycle to be the total weight of its edges, and the weight of a *set* of cycles to be the total weight of all cycles in the set. (Thus, each edge is counted once for every cycle in which it appears.) Describe and analyze an efficient algorithm to compute the minimum-weight cycle basis in G.

9. Describe a modification of GREEDYSCHEDULE that runs in $O(n \log n)$ time. *[Hint: Store X in an appropriate data structure that supports the operations "Is $X \cup \{i\}$ realistic?" and "Add i to X" in $O(\log n)$ time each.]*

10. 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 a ray r intersects a circle!c in $O(1)$ time. It's not that hard to write this subroutine, but it's not the interesting part of the problem.