COMP 251 Study guide

Francis Piche

February 25, 2018

Contents

1 Disclaimer

The material of this document was transcribed from Prof. Adrian Vetta's lecture recordings for COMP251 in the winter of 2018. Extra notes, clarifications or interpretations added by me may not be correct. All images are taken directly from Prof. Adrian Vetta's lecture slides. I claim no ownership over these images or the content taken from the slides.

2 Preliminaries

In this course an algorithm is considered **good** if it:

- Works
- Runs in polynomial time. Meaning it runs, in $O(n^k)$ time. Where n is (always) the size of the problem. (Number of elements in a list to be sorted etc.)
- Scales multiplicatively with computational power. (If your computer is twice as fast, the problem is solved at least twice as fast)

A bad algorithm is one that:

- Doesn't always work
- Runs in exponential time or greater. Meaning: $O(k^n)$ time.
- Does not scale well with computational power. (Your computer is twice as fast, but barely any performance boost).

Part I
Recursive Algorithms

I won't be going into detail on the specifics of things like how recursion works, MergeSort, BinarySearch, solving recurrences, Big O, etc. as it's considered prerequisite material. If you need some review, my COMP250 study guide is still publicly available.

3 Divide + Conquer Algorithms

Examples:

- MergeSort
- BinarySearch

3.1 MergeSort

The MergeSort algorithm involves splitting a list of n elements in half, sorting each half recursively, and merging the sorted lists back into one. It takes time $T(\frac{n}{2})$ to sort the list of half size, and time $O(n)$ to merge the list back together. So the recurrence relation for MergeSort is given by:

$$T(n) = 2T(\frac{n}{2}) + cn$$

where c is some constant.

Theorem 1. MergeSort runs in time $O(nlog(n))$.

Proof. Add **dummy numbers** (extra "padding" to the list), until n is a power of two. $n = 2^k$. We can do this because $O()$ gives an **upper bound**, and adding numbers will make our solution take longer than the real one. Doing this will make solving the recurrence easier.

Unwinding the formula:

$$\begin{aligned} T(n) &= 2(2(T(\frac{n}{4}) + c\frac{n}{2}) + cn \\ &= 2^2(T(\frac{n}{4}) + 2cn) \\ &= 2^3(T(\frac{n}{8}) + 3cn) \\ &= 2^4(T(\frac{n}{16}) + 4cn) \end{aligned}$$

Notice we have a pattern emerging.

$$= 2^k(T(1)) + kcn$$

Recall $2^k = n$, so $k = log_2(n)$ and $T(1) = 1$ so:

$$= n + nlog_2(n)$$

Which is $O(nlogn)$. □

3.2 Binary Search

Binary search involves splitting your sorted list into two, and searching that half. So our recurrence is given by:

$$T(n) = T(\frac{n}{2}) + c$$

where c represents the constant work (comparisons, setting new bounds etc.)

Theorem 2. Binary Search is $O(log_2(n))$.

Proof. Again we add dummy numbers so that n is a power of two. $n = 2^k$

We begin with our recurrence:

$$T(n) = T(\frac{n}{2}) + c$$

$$= T(\frac{n}{4}) + c + c$$

$$= T(\frac{n}{8}) + c + c + c$$

$$= T(\frac{n}{2^k}) + kc$$

$$= T(1) + log_2(n)$$

since $k = log_2(n)$ which is $O(log_2(n))$. □

3.3 Run Time of Divide + Conquer in General

Divide and Conquer is a technique of solving problems that involves taking one large problem of size n , and breaking it down into a smaller problems of size $\frac{n}{b}$, and solving those problems recursively. They are then combined to produce a solution in time poly-time: $O(n^d)$.

So the run-time of a divide and conquer algorithm is:

$$T(n) = aT(\frac{n}{b}) + O(n^d)$$

In the case of MergeSort, $a = 2$, $b = 2$, $d = 1$.

In the case of BinarySearch, $a = 1$, $b = 2$, $d = 0$.

3.4 Aside on Recurrences: Domain Transformation

Note that the recurrence for MergeSort is really:

$$T'(n) \leq T'(\lfloor n/2 \rfloor) + T'(\lceil n/2 \rceil) + cn$$

Which we simplified by adding dummy entries. However, we can also say this:

$$T'(n) \leq 2T'(\frac{n}{2} + 1) + cn$$

But the +1 doesn't fit with our previous method.

We'll use **domain transformation** to solve this, starting with:

$$T(n) = T'(n + 2)$$

$$\leq T'(\frac{n+2}{2} + 1) + c(n + 2)$$

plugging in our expression from above

$$\leq T'(\frac{n+2}{2} + 1) + c'(n)$$

absorbing the +2 into c.

$$= T'(\frac{n}{2} + 2) + c'(n)$$

simplifying the fraction.

$$= T(\frac{n}{2}) + c'n$$

from our domain transformation at the beginning. Solving this the usual way, we get:

$$T(n) = O(nlog(n))$$

But again from our domain transformation:

$$T(n) = T'(n + 2)$$

, so

$$T'(n) = T(n - 2) = O(nlog(n))$$

So we've shown that $T'(n)$ has the same upper bound as $T(n)$.

4 Master Theorem

Theorem 3. If $T(n) = aT(n/b) + O(n^d)$ for constants $a > 0$, $b > 1$, $d \geq 0$, then:

$$\begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d log(n)) & \text{if } a = b^d \\ O(n^{log_b(a)}) & \text{if } a > b^d \end{cases}$$

These cases are just a few that occur often in practice when dealing with divide + conquer algorithms.

Proof. First we'll need two things. One is the geometric series, and the other is a law of logarithms. Professor Vetta proved them in class, and honestly I doubt you'd be asked to prove them on an exam, but it's good proof practice to go through them so I'll do it here.

$$\sum_{k=0}^{l} x^k = \frac{1 - x^{l+1}}{1 - x}$$

Proof:

Starting with:

$$(1 - x) \sum_{k=0}^{l} x^k$$

We can expand it out:

$$= \sum_{k=0}^{l} x^k - \sum_{k=0}^{l} x^{k+1}$$

Simplifying the sigma notation:

$$= \sum_{k=0}^{l} x^k - \sum_{k=1}^{l+1} x^k$$

All terms will cancel except:

$$= x^0 - x^{l+1} = 1 - x^{l+1}$$

Divide through by $1 - x$

$$= \frac{1 - x^{l+1}}{1 - x}$$

Our second fact to derive is this law of logs:

$$x^{log_b(y)} = y^{log_b(x)}$$

Using the power rule of logarithms:

$$log_b(x)log_b(y) = log_b(y^{log_b(x)})$$

similarly,

$$log_b(x)log_b(y) = log_b(x^{log_b(y)})$$

so,

$$log_b(x^{log_b(y)}) = log_b(y^{log_b(x)})$$

Now we're ready for the proof.

Assume n is a power of b, and split up the problem into all it's chunks.

$$T(n) = n^d + a(\frac{n}{b})^d + a^2(\frac{n}{b^2})^d + ... + a^l(\frac{n}{b^l})^d$$

(this is just if you'd "unwound" the whole recursion down to it's simplest form like we did in the MergeSort/Binary Search proofs.)

Each term is the amount of work it will take at each level of the recursion.

Notice you can factor out:

$$= n^d(1 + a(\frac{1}{b})^d + a^2(\frac{1}{b^2})^d + ... + a^l(\frac{1}{b^l})^d$$

$$= n^d(1 + (\frac{a}{b})^d + (\frac{a}{b^d})^2 + ... + (\frac{a}{b^d})^l)$$

That looks like a geometric series! So let's look at the cases:

Case 1: $a < b^d$

Applying the geometric series formula:

$$= n^d \sum_{k=0}^{l} (\frac{a}{b^d})$$

$$= n^d \frac{1 - (\frac{a}{b^d})^{l+1}}{1 - \frac{a}{b^d}}$$

we can remove the $\frac{a}{b^d}^{l+1}$ term with this inequality (since the term doesn't depend on n):

$$\leq n^d \frac{1}{1 - \frac{a}{b^d}}$$

which is $O(n^d)$.

Case 2: $a = b^d$

Since $\frac{a}{b^d} = 1$:

$$= n^d(1 + 1 + 1 + ... + 1)$$

There are $l + 1$ terms, but we said n was a power of b, ($n = b^l$) so, $l = log_b(n)$,thus:

$$= n^d(log_b(n) + 1)$$

which is $O(n^d log_b(n))$

Case 3: $a > b^d$

Again from geometric series, and multiplying through by -1:

$$n^d \frac{(\frac{a}{b^d})^{l+1} - 1}{\frac{a}{b^d} - 1}$$

Again this inequality holds:

$$\leq n^d \frac{(\frac{a}{b^d})^{l+1}}{\frac{a}{b^d} - 1}$$

Which is $O(n^d(\frac{a}{b^d})^l)$ which we can simplify:

$$(\frac{n}{b^l})^d a^l$$

but $n = b^l$, so:

$$= (1)a^l$$
$$= a^{log_b(n)}$$

now by our second fact:

$$= n^{log_b(a)}$$

which is $O(n^{log_b(a)})$

□

It's **much** more important to understand the proof than it is to memorize the theorem.

4.1 Tree Method to Prove Master Theorem

A more intuitive way to think of the proof is with a *Recursion Tree*.

The root node of the tree has label n, and each node has a children (except the leaves). a is called the *branching factor*. Each child is labelled $\frac{n}{b^d}$ where d is the depth. The labels represent the size of the sub problems.

The number of nodes at each level is a^d.

Case 1 is when the root level "dominates" all other levels, so the running time is just $O(f(n))$ where $f(n)$ is the amount of work at the root level.

Case 2 is when all levels are roughly the same weight. So the total running time is just $O(f(n)l)$ where l is the number of levels.

Case 3 is when the leaves dominate, so the running time is $O(a^l)$ since the leaves each take time $O(1)$, and there are a^l of them.

5 Multiplication

5.1 Grade School Multiplication

This takes n^2 multiplications when you multiply two n-digit numbers. so the runtime is $\Omega(n^2)$

5.2 Russian Peasant Multiplication

Super weird looking algorithm but it works!

```
Mult(x,y){
   if x = 1 then output y
   if x is odd then output y + Mult(floor(x/2),2y)
   if x is even then output Mult(x/2, 2y)
}
```

This actually comes from if you take the binary representation of x: say $x = 46_{10}$ then $x = 101110_2$. The bits that are 1's will have the y added step, and the zero bits will just have the doubling step. Weird right?

Notice that this means the number of steps is just the number of bits in x. The number of digits in the result will be at most $2n$, so if we need to then add these, we add at most n numbers of $2n$ digits so takes time $O(n^2)$

5.3 Divide + Conquer Multiplication

Notice that a number x can be written as:

$$x = x_n x_{n-1} ... x_{\frac{n}{2}+1} x \frac{n}{2} ... x_2 x_1$$

where the x_i are the digits.

Then we have:

$$x = 10^{\frac{n}{2}} x_L + x_R$$

where n is the number of digits, x_L is the first $\frac{n}{2}$ digits, and x_R is the last $\frac{n}{2}$

So by expanding:

$$xy = (10^n x_L y_R + 10^{\frac{n}{2}}(x_L y_R + x_R y_L) + x_R y_R$$

Notice that this now involves four products of $\frac{n}{2}$ digit numbers. So the recursion is:

$$T(n) = 4T(\frac{n}{2}) + O(n)$$

We have $a = 4$,$b = 2$,$d = 1$, which is case 3 of the master theorem.

Which means the running time is:

$$O(n^{log_2(4)})$$

which simplifies to:

$$O(n^2)$$

Thanks to Gauss, we can actually use this fact:

$$x_L y_R + x_R y_L = x_R y_R + x_L y_L - (x_R - x_L)(y_R - y_L)$$

which is actually only 3 unique products. (adding is cheap)

So our new running time is:

$$T(n) = 3T(\frac{n}{2}) + O(n)$$

which is case 3 of the master theorem, so

$$O(n^l og_2(3))$$

$$= O(n^{1.59})$$

5.4 Fast Fourier Transforms

These are $O(nlog(n))$ for multiplying n-bit numbers. They'll be studied more in-depth at the end of the course (time-permitting).

5.5 Multiplying Matrices

There are n multiplications to calculate each entry of the result matrix, and there are n^2 entries, so $O(n^3)$

Using divide + conquer, divide into 4 sub-matrices:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ x_{d1} & x_{d2} & x_{d3} & \dots & x_{dn} \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

So if we let:

$$x = \begin{bmatrix} A & B \\ C & D \end{bmatrix} y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

then:

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

So multiplying involves eight products with $\frac{n}{2}$x$\frac{n}{2}$ and the recurrence is:

$$T(n) = 8T(\frac{n}{2}) + O(n^2)$$

which is Case 3 of the master theorem, so runtime is $O(n^{log_2 8})$ which is $O(n^3)$, no improvement.

There actually is a trick to do better.

Claim:

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

is the same as:

$$\begin{bmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{bmatrix}$$

where:

$$S_1 = (B - D)(G + H)$$
$$S_2 = (A + D)(E + H)$$

$$S_3 = (A - C)(E + F)$$
$$S_4 = (A + B)H$$
$$S_5 = A(F - H)$$
$$S_6 = D(G - E)$$
$$S_7 = (C + D)E$$

which is only 7 products! (The additions are negligible)

So we have:

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

Which is Case 3 of the master theorem, so $O(n^log_2(7))$which is $O(n^{2.81})$

5.6 Fast Exponentiation

Method of taking exponents in a fast way, since doing:

$$x * x * x * x ... * x$$

is super slow.

```
FastExt(x,n){
if n=1 output x
else
   if n is even output FastExp(x, floor(n/2))^2)
   if n is odd output FastExp(x, floor(n/2))^2)*x
}
```

So our recurrence looks like:

$$T(n) = T(floor(\frac{n}{2}) + O(1)$$

(since we're halving the problem, and doing some constant work at each step)

This is Case 2 of the Master Theorem, so the runtime is $O(log_2 n)$

6 The Median Problem

6.1 The Selection Problem

Want to find the kth smallest number in a set S.

Select(S,k)
If $\mid S \mid = 1$ then output x_1.

Else:
Set S_L = all numbers less than x_1
Set S_R = all numbers greater than x_1
If $\mid S_L \mid = k-1$ then output x_1 (since if you have k-1 things smaller than x_1, that can only mean x_1 is the kth smallest element)
If $\mid S_L \mid > k-1$ then output Select(S_L, k) (since that means the kth smallest element must be within that set)
If $\mid S_L \mid < k-1$ then output Select(S_R, k-1-$\mid S_L \mid$)(-1 since you know its not x_1, and - $\mid S_L \mid$ since you know its not in any of those, so you want the k-1-$\mid S_L \mid$-th element of S_R.)

The runtime of this algorithm is almost entirely dependent on the choices of pivots, since if you get a "bad" pivot every time, then you would recurse on a set of size n-1.

$$T(n) = (n-1) + T(n-1)$$

.

.

.

$$= \frac{1}{2}(n(n+1))$$

which is $O(n^2)$.
We could instead choose our pivot randomly.

The pivot would separate the list into sizes from $\frac{n}{4}$ to $\frac{3n}{4}$ with probability $\frac{1}{2}$, and so the pivot would be good half the time. So the expected running time is:

$$T(n) \leq \frac{1}{2}T(\frac{3n}{4}) + \frac{1}{2}T(n) + O(n)$$

$$\frac{1}{2}T(n) \leq \frac{1}{2}T(\frac{3n}{4} + O(n)$$

$$T(n) \leq T(\frac{3n}{4}) + O(n)$$

which satisfies Case 1 of the master theorem which is $O(n)$.

But what if we want to be **certain** that the worst case will never happen?

6.2 Median of Medians

Divide the set S into groups of size 5. Sort each group and find the median of each group. If you were to find the median of these medians,there would always be less than $\frac{7}{10}n$ elements in your two groups, which is pretty good. The reason this comes up is:

There's $\frac{n}{5}$ groups overall. Imagine the everything was sorted. Each group of 5 is sorted, and the groups are sorted by their medians. So there's $\leq \frac{n}{5} * \frac{1}{2} = \frac{n}{10}$ groups to the left of the median of medians. There's 3 elements above than the median in its own group, so there's $\leq \frac{3n}{10}$ elements smaller (to the top left) than the median, which means there's $\leq \frac{7n}{10}$ elements larger (to the bottom right) than the median.

So the max size of the sets is $\frac{7n}{10}$

Finding the median of the medians is done recursively, by partitioning into 5 groups, and putting a recursive call on finding the pivot.

So the recursive formula is:

$$T(n) \leq T(\frac{7n}{10}) + T(\frac{n}{5}) + O(n)$$

Notice the Master Theorem doesn't apply here, instead we need to use the recursion tree method.

First our problem of size n is broken into two problems, one of size $\frac{7n}{10}$ and the other size $\frac{2n}{10}$. Continuing down recursively, we actually get one side of

the tree ending before the other. Namely, the $\frac{7n}{10}$ side will reach the leaves later than the $\frac{3n}{10}$ side.

However, up until the point that this end is reached, we're doing $(\frac{9}{10})^l n$ work at each level. Beyond this, the work needed at each level only decreases, so it's $\leq (\frac{9}{10})^l n$ These terms are geometrically decreasing, so the first term dominates, and we get $O(n)$.

7 Finding the Closest Pair of Points in the Plane

How fast can we solve this?

7.1 Exhaustive Search

Calculate the distance between every pair of points, choose the shortest pairwise distance. $O(n^2)$. Is there a faster algorithm?

In one-Dimension, notice that the closest pair of points needs to be next to each other on the line. So we only need to find how far each **pair** is. $(n-1$ distances to calculate).

7.2 2-D case

Simply taking the closest in their x-coordinate (or y coordinate) doesn't work since they could be close in x but very far in y.

A divide + conquer approach is to separate the points into two groups of size $\frac{n}{2}$, so we want our dividing line to pass through the median x-coordinate.

We can now recursively search for the closest pairs in each group.

But what if the closest pair is **between** the two groups?

So we have to check to see if there's a better solution with an endpoint in each group. How can we do this efficiently? (This is the bottleneck step).

7.3 Widening the Bottleneck

Notice that by solving the subproblems recursively we can find the smallest distance between two points in both the left and right subproblems call this δ. So we know that if a better solution exists, it will be within δ from the dividing line.

This seems much better! But what if all the points are within δ of the dividing line? Well then this doesn't help much.

There's actually a trick we can do.

We can break up the area into squares of size $\frac{\delta}{2}$, and no two points will lie in the same square. This is because if two points are in the same square, then there are on the same side of the dividing line. These points are within $\delta\frac{\sqrt{2}}{2}$ (by construction of the boxes)from eachother, but this is $< \delta$, so this contradicts the minimality of δ.

We can now use this fact to derive another fact:

Suppose there's a point on either side of the dividing line with distance less than δ. We can prove that there will be at most 10 points between them in the y-ordering. (Within the area filled with boxes).

Proof

Since the squares are of size $\frac{\delta}{2}$, then the two points are either on the same row, or one is within two rows above the other. (or else it would be further than δ) Now, since there can only be one point per-box, there's at most 10 points between them. (count the boxes for yourself!)

Now recall the 1-D case, we can now just look at every pairwise distance on a group where the points are at most 11 apart (rather than the ones that are next to each other as before). So you need to find the distance between a given point, and the next 11 distances.

So at most $11n$ distances to calculate.

7.4 The Finished Algorithm

- Find the point with the median x-coordinate
- Partition using this point
- Recursively find the closest pair of points in each half
- Find the closest pair within the small range given by δ, by checking the nearest 11 points (in the y-ordering) for each point.
- Among the three pairs found, (left, right, crossing) output the closest pair.

7.5 The Runtime (Enhanced)

Two subproblems of size $\frac{n}{2}$, and the work at each level is: finding the median $O(n)$, partitioning $O(n)$, making the smaller group (within δ of dividing line) $O(n)$, applying the 1-D algorithm $O(n)$. So our recurrence looks like:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

which is case 2 of the master theorem, so $nlog(n)$.

Part II
Graph Algorithms

For a review of basic graph terminology, see my COMP250 study guide. And I'm going to assume you took MATH240 and know the basics of graph theory from there. Lecture 7 had a review of these basics, but I wont include them here. I will only go over the theorems that were proved.

8 Theorems About Undirected Graphs

8.1 Handshaking Lemma

Theorem 4. In an undirected graph, there are an even number of vertices with odd degree.

Proof. We start with:

$$2 \mid E \mid = \sum_{v \in V} deg(v)$$

Since each edge is double counted when summing the degrees. (Each edge (u, v) contributes 1 degree to u and 1 degree to v)

This is the same as:

$$= \sum_{v \in Odd} deg(v) + \sum_{v \in Even} deg(v)$$

(the sum of the vertices of even degree plus the odd degree ones)

Rearranging we get:

$$\sum_{v \in Odd} = 2 \mid E \mid - \sum_{v \in Even} deg(v)$$

Now we know at the sum of the even degrees is even, and we know that $2x$ is even for any x. So the right hand side is always even. Therefore the left hand side must be even. But for the sum of odd numbers to be even, there must be an even number of odd terms.

□

8.2 Leaf Existence

Theorem 5. Lemma: A tree T with $n \geq 2$ vertices has at least one leaf.

Proof. A tree is connected, which means there's no vertices with degree 0. A leaf is a vertex with degree 1, so to get a contradiction, assume every vertex has degree ≥ 2.

Take the longest path $P = v_1, v_2, v_3 ... v_{l-1}, v_l$

But every vertex has degree greater than 2, so v_l has a neighbour $x \neq v_{l-1}$ so v_l forms an edge with something in the path which would create a cycle and thus be a contradiction (since all trees have no cycles). If the neighbour was not on the path, then P was not the longest path. □

8.3 Number of edges in a Tree

Theorem 6. A tree with n vertices has $n-1$ edges

Proof. By induction:

Base Case:

A tree on one vertex has zero edges
linebreak Induction Step:

Assume that any tree on $n-1$ vertices has $n-2$ edges. Take a tree with $n \geq 2$ vertices. By the previous lemma, there exists a leaf vertex v. Let $T' = T - v$. Then T' is a tree on $n-1$ vertices, which has $n-2$ edges by the induction hypothesis. Adding back v, we get that T is a tree on $n-1$ vertices with $n-2$ vertices. □

8.4 Halls Theorem

Theorem 7. A bipartite graph, with $\mid X \mid = \mid Y \mid$ contains a perfect matching $\Leftrightarrow \forall S \subseteq X, \mid \Gamma(S) \mid \geq \mid S \mid$

Proof. ($\Rightarrow$)
If there is a set $S \subseteq$ with $\mid \Gamma(S) \mid < \mid S \mid$, then the graph cannot have a perfect matching, since there would not be enough things in the neighbourhood for the things in S to match to.
($\Leftarrow$)
Take a maximum cardinality matching M in the graph. If M is perfect we're done, if not, then there exists a vertex x_0 who is not matched in X. If Halls condition holds, then x_0 has a neighbour y_0. Suppose y_0 is matched to x_1. Again if Halls condition holds, then x_0, x_1 have another neighbour say y_1.

We now repeat this process until eventually it terminates (it will since we

have a finite number of vertices). It will terminate when we reach x_k who is unmatched.

We now create an m-alternating path from y_k to x_0. This path is m-augmenting, so we augment, and receive a larger matching. Contradiction, M was not maximal.

□

9 Breadth First Search

9.1 Generic Search Algorithm

```
Put root into bag
   while bag not-empty
      remove v from the bag
         if v is unmarked
            mark v
            for each arc(v,w)
               put w into the bag
```

A vertex is discovered when it is marked. Notice that there can be multiple copies of a vertex in the bag, and that this actually wont affect our performance.

9.1.1 Revised Generic Search Algorithm

Instead of adding vertices, we'll add arcs.

```
Put (*,r) into a bag
   while bag not-empty
      remove (u,v) from the bag
         if v is unmarked
            mark v
            set p(v) to u //keep track of "predecessor" of v
            for each arc(v,w)
               put (v,w) into the bag
```

Keeping track of the predecessor will be useful later.

9.1.2 The Running Time

We look at each arc out of v only once, when v is first marked. The arc is then added to the bag once, and removed once. So, we get a runtime proportional to the number of arcs.

$$\Rightarrow O(m)$$

9.1.3 Validity

Theorem 8. Let G be a connected, undirected graph. Then the search algorithm finds every vertex in G.

Proof. We need to show that every vertex v is marked by the algorithm. We will use induction on the length of the smallest path from the vertex to the root.

Base Case:
$k = 0$ then v is the root, and only the root exists, so trivially true.
Induction Step:
Assume true for a path of length $k - 1$ from the root. Now assume there is a path P with k edges from v to r. So let

$$P = \{v = v_k, v_{k-1}, ..., v_1, v_0 = r\}$$

Then there is a path:

$$Q = \{u = v_{k-1}, ..., v_1, v_0 = r\}$$

So by the induction hypothesis, u is marked. Then after we mark u, all edges incident to u would have been added, so (u, v) would have been added. And so later, (u, v) would be removed and v would be marked. □

We can prove that for directed graphs, every vertex that has a directed path from r is marked in the same way.

9.2 Search Trees

Theorem 9. The predecessor edges made by the search algorithm on a connected, undirected graph G is a tree rooted at r.

Proof. By induction on the number of marked vertices, k.
Base Case: $k = 1$
Induction Step:
Assume true for the first $k-1$ vertices. Let v be the kth vertex to be marked. Assume v was marked when we removed the edge (u, v). This means that u is the predecessor of v. But (u, v) was added to the bag when we marked u, so u must be in the set of the first $k-1$ vertices to be marked. Thus, by the induction hypothesis, when we add the edge $(p(v), v) = (u, v)$, we are adding a leaf, so the new graph formed is still a tree. □

9.3 Choices of Bags

We can use a Queue to get BFS, if we use a Stack we get DFS, if we use a Priority Queue, we get minimum spanning tree.

9.4 BFS Trees

The edges are added to the queue in order of their distance from r. The vertices are marked in order of their distance from r.

Theorem 10. For any vertex v, the path from v to r given by the search tree T of predecessor edges is a shortest path.

Proof. Left as exercise. □

9.4.1 Structure

The structure of these trees can be broken down into "layers", where each layer is the set of vertices at a given distance from the root.
Any vertex $v \in S_l$ is at distance l from r in T, and the same is true in the whole graph G.
This implies that for every edge in the graph that is not in the tree (u, v), u and v are either in the same layer or in adjacent layers. If this was not the case, say u was in S_3 and v was in S_6, then we could get from the root to v in less than 6 steps.

9.4.2 BFS on Bipartite Graphs

Theorem 11. A graph G is bipartite $\Leftrightarrow$ it contains no odd length cycles.

Proof. $\Rightarrow$
Assume G contains an odd length cycle C.

$$C = \{v_0, v_1 ... v_2 k\}$$

Without loss of generality we can assume $v_0 \in Y$, therefore $v_1 \in X$, and so on. We eventually get down to $v_2 k \in X$ but it is a cycle so $v_2 k$ forms an edge with v_0, and we said $v_0 \in Y$, which means $v_2 k \in Y$, it can't be both in X and in Y, contradiction.
$\Leftarrow$
Assume G has no odd length cycles. Choose a root vertex r, and run BFS. Let X be the set of all odd layers of the BFS tree. Let Y be the set of all even layers of the BFS tree. Since every edge in the graph goes to either adjacent layers or the same layer, we know that if we have no edges in the same layer, then we'll have that every edge goes from X to Y.

Assume there's a non-tree edge (u, v) with u and v in the same layer. Let z be the closest common ancestor of u and v in the search tree. Let P be the path from u to z in the tree, and let Q be the path from v to z in the tree. The length of P is the same as Q since u and v are in the same layer. But then the cycle

$$C = P \cup Q \cup (u, v)$$

has an odd number of edges. So (u, v) cannot exist. □

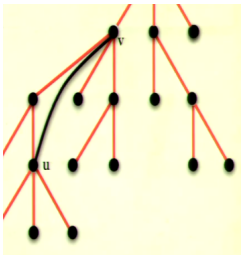
10 Depth First Search

We use the generic search algorithm using a stack.

10.1 DFS Trees

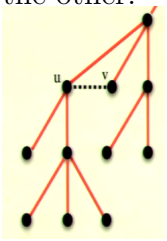
The DFS tree is much different than the BFS tree. DFS partitions the edges of an undirected graph into two types:
Tree Edges: Predecessor edges in the DFS tree at T
Back Edges: Edges where one endpoint is the ancestor of the other endpoint

in T.



Here (u,v) is a back edge.

We cannot have Cross Edges: Where neither endpoint is an ancestor of the other.



10.2 Recursive DFS

We can also do DFS recursively:

```
RecursiveDFS(r)
   mark r
   for each edge (r,v)
      if v is unmarked
         set p(v) = r
         RecursiveDFS(v)
```

10.3 Ancestral Edges

Theorem 12. Let T be a DFS tree in an undirected graph G. Then for every edge (u, v) either u is an ancestor of v in T or v is an ancestor of u.

Proof. Wlog assume u is marked before v. At the time u is marked, the algorithm will recurse on each arc incident to u.

Case 1: v is unmarked when the RecursiveDFS(u) examines (u, v).
Then the parent of v is then u, and so (u, v) is an ancestral tree edge.
Case 2: v is already marked. But v was marked after u, so it was marked during RecursiveDFS(u). So we have a series of vertices

$$\{u = w_0, w_1 ... w_{l-1}, w_l = v\}$$

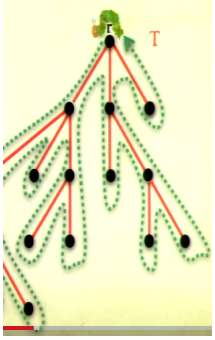
where $p(w_k) = w_{k-1}$ (the parent of each vertex is the previous vertex). This means that u is an ancestor of v, so (u, v) is a back edge.

□

Corollary Every non-tree edge is a back edge.

10.4 Previsit and Postvisit

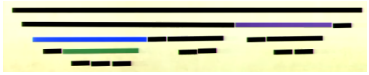
The way DFS explores the vertices of a graph is given by this picture:



We can add a "clock" that will keep track of the order in which the vertices were visited.

$Pre(v)$ is the time at which we arrive at a subtree rooted at v. $Post(v)$ is the time at which we leave a subtree rooted at v.

So we can represent each vertex by an interval of time. If we take the interval for every vertex, we get what's called a **Laminar Family**. Meaning, every interval is either completely disjoint, or completely overlapping.



If we draw an edge between each interval and the smallest interval that contains it, we actually build up the DFS tree again!

10.5 Directed Graph BFS Tree Structure

Now we can have four types of edges:

Tree arcs: Same as before
Forward Arcs: Arcs (u, v) where u is an ancestor of v
Backward Arcs: Arcs (u, v) where v is an ancestor of u
Cross Arcs: Non-Ancestral arcs (u, v) where u is marked after v.

Note that for cross arcs, the other way around is not possible because after visiting v, we must visit all descendants of it, before moving back up the tree and going down the other branch containing u.

We still have intervals. In a tree arc (u, v), the interval of v is contained in the interval of u. In a forward arc, the same is true. In a backward arc however, the interval of u is contained in the interval of v. In cross arcs, the intervals of u and v are disjoint.

So we have this list of properties:
For tree arcs:

$$post(v) < post(u)$$

For forward arcs:

$$post(v) < post(u)$$

For backward arcs:

$$post(u) < post(v)$$

For cross arcs:

$$post(u) < post(v)$$

So the only different one is for backward arcs.

10.6 Example: Directed Acyclic Graphs

How can we determine if a graph is acyclic?

Theorem 13. A directed graph G is acyclic $\Leftrightarrow$ DFS produces no backward arcs.

Proof. $\Rightarrow$
Suppose DFS gives a backward arc (u, v). By definition, then u is a descendant of v in the DFS tree T. Then there exists a path:

$$P = \{v = v_0, v_1, ..., v_k = u\}$$

which means $P \cup (u, v)$ is a directed cycle in G.
$\Leftarrow$
Assume DFS gives no backward arcs. Suppose there's a directed cycle:

$$C = \{v_0, v_1, ..., v_k, v_0\}$$

Since there's no backward arcs we have that:

$$post(v_0) > post(v_1) > ... > post(v_k) > post(v_0)$$

But $post(v_0)$ can't be greater than itself.

□

Corollary There is a linear time algorithm to test whether or not a graph is acyclic. Just run DFS and check if any arc is a backward arc.

10.7 Example: Topological Ordering

A topological ordering is when the vertices of a graph can be horizontally ordered such that every arc is from right to left.

Theorem 14. A directed graph G has a topological ordering $\Leftrightarrow$ DFS produces no backward arcs.

Proof. $\Rightarrow$ If DFS produces a backward arc then G contains a cycle C. Let the cycle:

$$C = \{v_0, v_1, ..., v_k, v_0\}$$

where, wlog, v_0 is the leftmost vertex of the cycle in the order. But then v_0, v_1 goes from left to right, which is not allowed.
$\Leftarrow$
Assume DFS gives no backward arcs. Then for every arc (u, v) we have:
$post(u) > post(v)$
so simply order the vertices by their post numbers. □

Part III
Greedy Algorithms

11 Scheduling

11.1 Task Scheduling

A firm can process 1 task a time. The job of customer i takes t_i time. We want to minimize the sum of the waiting times. Any job cannot be started until the previous one is finished.

First, we sort the jobs by length, shortest to longest. Simply schedule them in that order.

We must now prove this works.

Theorem 15. The greedy algorithm outputs an optimal schedule.

Proof. We will use an exchange argument.
Let the greedy algorithm schedule in the order: $\{1, 2, ..., n\}$
Assume there's a better schedule S. Then there must be a pair of jobs i and j such that:

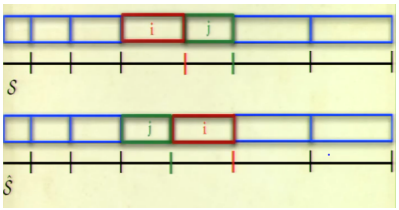
Job i is scheduled immediately before job j by schedule S

Job i is longer than job j

If we don't have this property, then it's sorted. The waiting time of job i is currently when job i finishes, and the waiting time of job j is when job j finishes.

Swap jobs i and j. Everything else stays the same. Specifically, the waiting time of every unchanged job stays the same.

Now the new waiting time of job j is better than both of the old ones, and the waiting time of job i is the same as the waiting time of the old job j.



So this configuration is better, and that contradicts the assumption that S was an optimal schedule. □

11.1.1 Running Time

All we did was sort, so it's $O(nlog(n))$

11.2 Class Scheduling

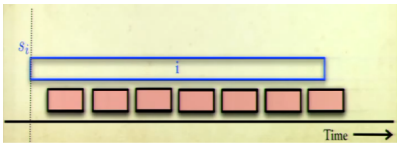
There is one classroom. There's a set $I = \{1, 2, .., n\}$ of classes that want to use the room class i has a start time s_i and a finish time f_i . The goal is to book as many classes as possible.

This is also known as the interval selection problem.

11.2.1 First Start

Select the class that starts earliest, iterate on the remaining classes that do not conflict with this one.

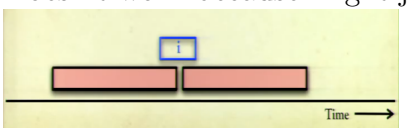
This doesn't work because the first class might also be the longest.



11.2.2 Shortest-Duration

Select the shortest class first, then iterate.

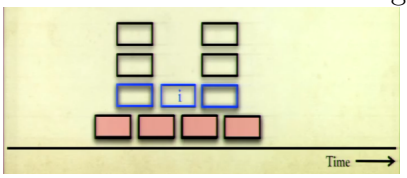
Doesn't work because might just be in an unlucky position.



11.2.3 Minimum Conflict

Select the class that conflicts with the fewest number of classes first. Then iterate.

Doesn't work because we might get a configuration like this:



Here the optimal solution has 4 classes, but we chose the configuration that has only 3, we chose i since i conflicts with only 2, whereas the red ones all conflict with 3 or 4. Next time we iterate, we just have two stacks of 3, so we can choose any from the left or right, and we're stukc with having at most 3 classes in our solution.

11.2.4 Last Start

Select the class that starts last, and iterate on the classes that do not conflict with this selection.

This one works!

It is symmetric to selecting the class that finishes first, then iterating on the classes that don't conflict with this selection.

Here's the pseudocode:

```
FirstFinish(I)
Let class1 be the class with the earliest finish time
Let X be the set of classes that clash with class1
output {1} U FirstFinish(I\X)
```

Lemma There is some optimal solution that selects Class 1.

Proof

Recall the classes are indexed such that $f_1 \leq f_2 \leq ... \leq f_n$.

Take an optimal schedule S and assume Class 1 is not in it. Let i be the lowest index class in S.

We claim we can replace i with 1. We know that f_i is before s_j for any $j \in S$. But f_1 is before s_j, so we know Class 1 doesn't conflict with any class in $S - \{j\}$.

So this ordering is at least as good as before, and contains class 1.

Theorem 16. The first-finish algorithm outputs an optimal schedule.

Proof. By induction on the cardinality of the optimal solution $\mid opt(I) \mid$

Base Case:
Let the solution have size 1, then this is trivially the optimal solution.

Induction Step:
Assume true for size k. Let the optimal solution have size $k+1$. First finish outputs $\{1\} \cup FirstFinish(I - X)$. But by the lemma, 1 is part of some optimal solution, S^*.

This means $S^* - \{1\}$ is an optimal solution with size k. So by the induction hypothesis, we have an optimal solution. □

There are at most n iterations of the algorithm, and it takes n time to find the class that finishes earliest in each iteration. So $O(n^2)$, but you could get it down to $O(nlogn)$ if you're careful about implementation.

12 The Shortest Path Problem

If every arc a has a length associated to it (a weight), l_a, then the length of a path P is:

$$l(P) = \sum_{a \in P} l_a$$

How do we then find the shortest path from s to every other vertex in the graph.

It turns out, we can find all the shortest paths in one go!

12.1 Dijkstra's Shortest Path Algorithm

We'll need to keep track of a few things. We need to keep track of the "tentative" distance from the root to our current vertex. We also need to keep track of which nodes we're done with.

1.) Initially, assign the first vertex's tentative distance to 0, set everything else to ∞.

2.) For each unmarked arc coming out of the current vertex, calculate the distance from the current vertex to the next, and add it to the distance from the root to the current. (To get the total). If this value is less than the current tentative distance of that vertex, replace it.

3.) Once you've calculated all neighbors of the current vertex, mark it as visited.

4.)Set the current vertex to be the one with the lowest tentative distance.

12.1.1 Special Case, All Arcs Have Distance 1

In this case, we actually get exactly Breadth First Search! Try it out ;).

12.1.2 Shortest Path Graph

Let S^k be the set of vertices in S at the end of the kth iteration, where S is the set of vertices we're "done with".

Let T^k be the set of arcs in T at the end of the kth iteration, where T is the set of arcs we fix to be in our final result.

Notice that all arcs in T^k are between vertices in S^k because when we add a vertex to S, we add the arc between it and it's predecessor (another vertex in S), to T. This means that $G^k = (S^k, T^k)$ is a directed graph.

Finally, G^n is the final output of the algorithm.

12.1.3 Shortest Path Tree

Theorem 17. The Graph G^k is a directed tree rooted as s.

Proof. Base Case: k=1
S^1 only contains s, and T^1 is empty. One vertex is a trivial tree.

Induction Step:
Assume true for G^{k-1}. Let v_k be the vertex added to S at the kth iteration. So $S^k = S^{k-1} \cup \{v_k\}$. This means that $T^k = T^{k-1} \cup (pred(v_k), v_k)$. So v_k has in-degree 1, and out-degree 0, which means v_k is a leaf. So G^k is still a directed tree rooted at s. □

Theorem 18. G^k gives the true shortest path distances from s to every vertex in S^k.

Proof. Base Case: $k = 1$
Trivially true. The label, $d(s)$ is 0, which is the shortest path distance, $d^*(s)$

Induction Step:
Assume true for G^{k-1}. That is, $d^{k-1}(v) = d^*(v)\ \forall v \in S^{k-1}$

Let v_k be the vertex added to S in the kth iteration. Take the shortest path P from s to v_k that uses as many arcs in common to G^k as possible. Now assume for a contradiction that $d^k(v_k) < d^*(v_k)$. (basically this path P follows the tree, then eventually jumps out to get to v_k and we're assuming

this is faster than just following the tree).

Let x be the last vertex of G^{k-1} in P. Let $y \notin S^k$ be the vertex after x in P. If this y doesn't exist, (there's no vertex after x) then we're done, since that means $P \subseteq G^k$. Assume it does exist.

Since y is on the shortest path from s to v_k and the arc-lengths are non-negative, each sub-path is also a shortest path, then the path from s to y is shorter than the path from s to v_k.

Since we're assuming P is the optimal path:

$$d^*(y) \leq d^*(v_k) < d^k(v_k)$$

But since $x \in S^{k-1}$ we have:

$$d^k(y) \leq d^{k-1}(x) + l(x, y)$$

(basically meaning that the distance from s to y is at most the distance from x to y.) And by our induction hypothesis:

$$= d^*(x) + l(x, y)$$

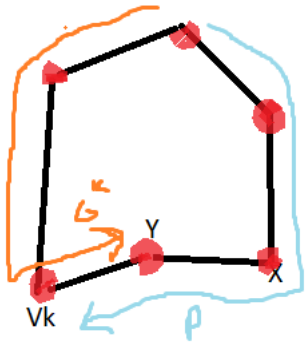
and by our assumption:

$$= d^*(y)$$

So we've now proved:

$$d^k(y) \leq d^*(y) < d^k(v_k)$$

A picture to see whats going on:



This is a contradiction because we've shown that the path down the left to y is shorter than the path to v_k, but we assumed the optimal path was from the right, and that meant that the distance to v_k had to be larger than the distance to y. □

12.1.4 The Running Time

There are n iterations, there are at most n distance updates at each iteration. So at most $O(n^2)$ but again we can improve it to $O(mlogn)$ using a heap.

13 Huffman Codes

13.1 Data Encoding

Suppose we want to encode the alphabet in binary. How many bits do we need to encode every letter?

Five bits since $2^5 \geq 26$

How do we measure the quality of an encoding? A natural measure would be the length of the encoding. But what if some letters are used very often? We would want these to have a smaller size.

Let f_i be the frequency at which a letter i appears in the alphabet. Then:

$$cost = \sum_{i \in A} l_i f_i$$

13.1.1 Morse Code

Morse code follows this idea. It uses less bits for the frequently used letters, and less bits for the less common ones. But there's problem with it. It cannot be binary because it's ambiguous whether 101 means 101 or 1, 0, 1. So Morse code is actually ternary. It uses pauses to signify the end of a letter.

How can we get around this?

13.2 Prefix Codes

A coding system is prefix-free if no codeword is a prefix of another codeword. Morse is not prefix free, since in 1101, 1 means t, 11 means m , 110 means g and 1101 means q. So t is a prefix of m is a prefix of g is a prefix of q.

13.3 Binary Tree Representations

We can use a binary tree T to represent a prefix-free binary code. Each left edge has label 0 and each right edge as label 1.

The leaf vertices are the letters of the alphabet. The codeword for a letter are the labels on the path from root to leaf.

Theorem 19. A binary coding system is prefix-free $\Leftrightarrow$ it has a binary tree representation.

Proof. ($\Leftarrow$)
In a binary tree representation the letters are at the leaves. This means that the path P_x from the root to leaf x and the path P_y from the root to a leaf y must diverge at some point.

So the codeword for x cannot be a prefix of the codeword for y.

($\Rightarrow$)

Given a binary coding system, we can define a binary tree recursively. A letter whose code word started with a 0 is placed in the left subtree. Otherwise it is placed in the right subtree. Then just recurse on the next letter. □

Observe that the cost of the tree is:

$$cost(T) = \sum_{i \in A} f_i d_i(T)$$

where d_i is the depth of the node i.

Proof

We have the definition of cost:

$$cost(T) = \sum_{i \in A} f_i l_i(T)$$

The length of the word is just the sum of the edges in the word

$$= \sum_{i \in A} \sum_{e \in P_i} 1$$

Which is exactly the same as the depth in the tree.

$$= \sum_{i \in A} f_i d_i(T)$$

13.3.1 Letter to Leaf Assignment

How should we assign letters to leaves? The least frequent letters should be at the deepest leaf. So, we can just sort all the frequencies, and start adding each least frequent letter to the deepest leaf.

13.3.2 Tree Shape

But what should the shape of the tree be?

Let $n_e = \sum_{i \in A : e \in P_i} f_i$ be the number of letters (weighted by frequency) whose root-to-leaf paths use edge e in T. (How many letters use this edge)

$$cost(T) = \sum_{e \in T} n_e$$

Proof
We start with our first observation:

$$cost(T) = \sum_{i \in A} f_i d_i(T)$$

The depth is just the sum of the edges in the path from root to the vertex.

$$= \sum_{i \in A} f_i \sum_{e : e \in P_i} 1$$

Changing the order of summation:

$$= \sum_{e \in T} \sum_{i \in A : e \in P_i} f_i$$

Which is exactly our definition.

$$= \sum_{e \in T} n_e$$

13.4 The Key Formula

The key to designing a good coding system is the following formula:

Theorem 20. Let $\hat{T}$ be the tree formed from T by removing a pair of sibling-leaves a and b and labelling it's parent by z where $f_z = f_a + f_b$ then:

$$cost(T) = cost(\hat{T}) + f_a + f_b$$

Observation 1 is telling us that the least frequent letters should be siblings, and observation 2 tells us how to find the optimal shape of the tree.

13.5 The Algorithm

```
Huffman(A,f)
   if A has two letters then
      encode one letter with 0 and the other with 1
   else
      let a and b be the most infrequent letters
      merge a and b into a new node z with frequency z = a + b
      recurse on the new set
      create the tree by adding a and b as children of z in the
          completed tree
```

13.5.1 Proof Of Correctness

Theorem 21. The Huffman Coding Algorithm gives the minimum cost encoding.

Proof. By Induction on the size of A.

Base Case: $\mid A \mid = 2$
Each letter has codeword length 1.

Induction Step:
Assume works for $\mid A \mid = k$. Take $\mid A \mid = k+1$.

First, the algorithm merges the two smallest leaves into one vertex, so there are now $k-1$ elements. We know that an optimal solution for $\hat{A}$ where $|A| = k-1$ exists, by our induction hypothesis. Then, the algorithm can extend this to a solution for A by attaching leaves labeled a and b as children of the vertex in $\hat{A}$, z, that we merged earlier. □

13.5.2 Running Time

There are n-2 iterations, each iteration takes $O(n)$ to find the two least frequent letters and update the alphabet.So, $O(n^2)$. Again, with heaps we can get it to $O(nlog(n))$.

14 Minimum Spanning Tree Problem

Given a graph, each edge e has a cost c_e, where all edge costs are distinct.

So the cost of a tree T is:

$$c(T) = \sum_{e \in T} c_e$$

14.1 Kruskal's Algorithm

Sort the edges $\{e_1, e_2, ..., e_m\}$ by cost, least to greatest.
Set $T = \phi$
For each $i = \{1, 2, ..., m\}$
Let $e_i = (u, v)$

if u and v are in different components of the tree, then add this edge to T.

14.2 Prim's Algorithm

Set $T = \{a\}$
If $V(T) \neq V(G)$ then
Let e be the minimum cost edge in $\delta(T)$ (edges leaving a vertex in T)
At this edge to T. (and it's vertices)

14.3 Boruvka's Algorithm

Set $T = \phi$
If T has more than one component $\{S_1, S_2, ..., S_l\}$ then
For $i = \{1, 2, ..., l\}$ let e_i be the minimum cost edge in $\delta(S_i)$
Add all of these edges to the tree.

14.4 Running Times

14.4.1 Kruskal's Running Time

It takes $O(mlogm)$ to sort the edges, and there's m iterations of the loop. Within the loop we have to search the tree to see if u and v are in different components. This takes time $O(n)$.

So we have:

$$O(mlogm + mn) = O(mn)$$

14.4.2 Prim's Running Time

We have n iterations of the loop. Within the loop, we have to exhaustively search for the minimum edge in time $O(m)$.

So:

$$O(mn)$$

14.4.3 Boruvska's Running Time

We have at most n components, finding the minimum edge takes $O(m)$, and there are $\leq logn$ iterations.

So:

$$O(mnlogn)$$

14.5 Proof That They All Work

First, notice that for a chicken to cross a road and get to a chicken coop, it must cross the road an odd number of times.

We'll also need this fact:

Theorem 22. The Cut Property of a minimum spanning tree is this: Assume the edge costs are distinct. If e is the cheapest edge in some cut $\delta(S)$ then e is in the minimum spanning tree.

Proof. Let $e = (u, v)$ be the cheapest edge in a cut $\delta(S)$ recall that $\delta(S)$ is the edges leaving a subset of vertices S.

Let T^* be a minimum spanning tree, and to get a contradiction, assume $e \notin T^*$.

Since T^* is a spanning tree there is a unique path P in T^* from u to v.

Observation. If $\hat{e} \in P$ then $(T^* \setminus \hat{e} \cup e$ is a spanning tree. (Basically we can replace an edge from the path joining u, v with u, v itself).

If a chicken is walking along P must cross the cut $\delta(S)$.

Observation. There is at least one edge $\hat{e} \in P \cap \delta(S)$.

We know from the beginning that e is the lowest cost edge, so

$$c_{\hat{e}} > c_e$$

$$\Rightarrow (T * -\hat{e}) \cup e$$

is a cheaper spanning tree than T^*. This contradicts the assumption that T^* was the minimum cost tree.

Essentially what we did was, replace $\hat{e}$, by e since we know that doing so gives us a cheaper spanning tree.

□

This proves all our algorithms simultaneously.

In the case of Prim's algorithm, we literally added edges based on if they were the minimum edge in the cut $\delta(T)$, so it directly uses this theorem.

In Baruvka's algorithm,we add edges from the cut $\delta(S)$ for each component, so again using the theorem.

In Kruskal's algorithm, we add edges if u, v are in different components. Let S be one of the two. Then e_i is the cheapest edge in $\delta(S)$ since we look at edges by order of cost. So this one also works.

14.6 The Cycle Property

Theorem 23. Assume distinct edge costs. If e is the most expensive edge in some cycle C, the e is not in the MST.

Proof. Let $e = (u, v)$ be the most expensive edge in the cycle C.

So $P = C - e$ is a path from u to v.

Assume for a contradiction that e is in the MST T^*.

Let $(S, V - S)$ be the cut introduced by $T^* - e$.

Observation 1: If $\hat{e} \in \delta(S)$ then $(T^* - e) \cup \hat{e}$ is a spanning tree.(basically we can replace e with $\hat{e}$ and get a spanning tree, since $\hat{e}$ also joins the two sets.)

Observation 2: There is at least one edge $\hat{e} \in P \cap \delta(S)$ (there's an edge that crosses the cut that is also part of the path.)

But $c_{\hat{e}} < c_e$ so we can replace e by $\hat{e}$ which contradicts the fact that T^* was the MST. □

14.7 The Reverse Delete Algorithm

Sort the edges by cost, most expensive to cheapest.

For each edge:
If G $\setminus\{e_i\}$ is connected then set G = G $\setminus\{e_i\}$

So basically take the most expensive edge, and if the graph is still connected without it, then throw it away.

14.7.1 Runtime of Reverse Delete

There are m iterations, and at each one need to check that graph is still connected (using BFS or DFS) in $O(m)$. So the running time is $O(m^2)$

14.7.2 Proof of Reverse Delete

First notice that G $\setminus\{e_i\}$ being connected means there was a cycle including e_i, and since we've sorted in reverse order, e_i is the most expensive edge, so by the cycle property, this algorithm works.

15 The Clustering Problem

Given a collection of objects, O we want to partition the objects into a set of clusters $\{S_1, ..., S_k\}$. A "good" clustering has similar objects in the same clusters.

We represent the problem by a weighted graph G.

There is a vertex for each object O, and an edge between each pair of objects.

The weight $d_{ij} \geq 0$ of an edge represents the disimilarity of object i and object j.

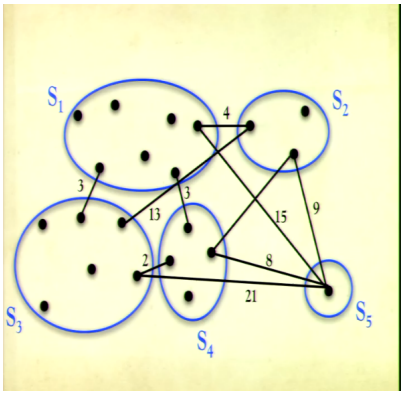
The quality of a clustering has no optimal definition. (Depends on application)

15.1 Maximum Spacing Clustering

Maximize the distances between the clusters. In other words, partition the vertices into k clusters so that the minimum distance between two vertices in different clusters is maximized.

Given a clustering $\{S_1, S_2, ..., S_k\}$ we define the distance between two clusters as:

$$d(S_l, S_m) = min_{i \in S_l, j \in S_m} d_{ij}$$



So here we just want to maximize the minimum black line (here it's 2) so the quality is (2).

15.2 Reverse-Delete Clustering Algorithm

Sort the edges by cost, highest to lowest.

For each edge:
If G $\setminus\{e_i\}$ has k components or less, then set G = G $\setminus\{e_i\}$

Notice, this is exactly the MST problem, where in MST, $k = 1$

15.2.1 Proof Of Reverse-Delete Clustering

First, observe this:

Theorem 24. A connected graph contains a spanning tree as a subgraph

Proof. Simply grow a BFS tree from any root vertex. □

Next, observe this fact:

Theorem 25. We can remove an edge, and the number of components increases by at most 1.

Proof. Originally, u, v are in the same component S_1. S_1 contains a spanning tree T.

Case 1: e is not in T. Then S_1 remains a component after deletion of e

Case 2: e is in T for every spanning tree in S_1. Then S_1 is broken into two components on the deletion of e. □

Now our algorithm:

Proof. Let e_l be the edge whose deletion causes the number of components to increase from $k-1$ to k.

This means that the algorithm deleted all the edges up to e_l.

When we delete e_l we have the clustering $S = \{S_1, ..., S_k\}$

But this means that only the edges up to e_l can cross between the clusters. Since we organized these to be largest to smallest, that means e_l is the shortest edge between two clusters. So the quality is determined by e_l.

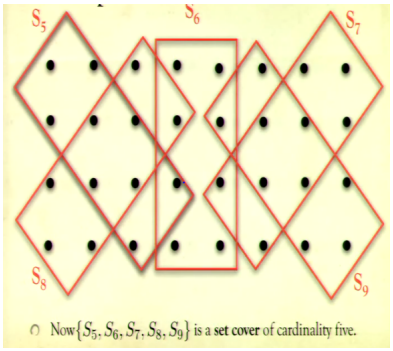
We now need to show that this is the optimal solution.

Any other clustering $S^* = \{S_1^*, ..., S_k^*\}$ with k components must separate the endpoints of at least one edge with an endpoint in the edges up to e_l from below. (edges smaller than e_l).

But then we'll have separated two clusters by an edge shorter than e_l which is worse than S. □

16 The Set Cover Problem

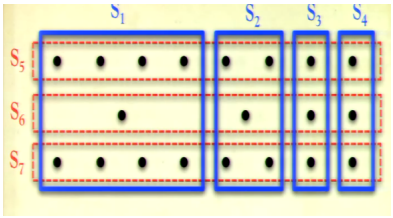
Given a collection of n items, I, and another collection of sets $S = \{S_1, .., S_m\}$ we want to find the smallest collection of sets in S that contain all of the elements of I.



16.1 The Greedy Set Cover Algorithm

Repeat until $I = \phi$
Let $\hat{S} = argmax \mid S \cap I \mid$ (pick the set that covers the most items in I)
Set $S = S - \{\hat{S}\}$ and $I = I - \hat{S}$

This doesn't work!



But we'll see that it's pretty close.

16.2 Approximation Algorithms

An algorithm A is an a-approximation algorithm if for any instance I:

It runs in time poly($|I|$)

It always outputs a feasible solution S.

It always guarantees: $cost(S) \leq \alpha * OPT$ where OPT is the optimal solution, and α is the desired approximation. (here we're looking at a minimization problem)

The greedy set cover algorithm is an approximation algorithm. So we want to find α for it, so we know how good it actually is.

Observation
If the optimal set cover has cardinality k then for any $X \subseteq I$, then there is some set S that covers at least $\frac{1}{k}|X|$ items of X.

Proof Let the optimal solution be $\{S_1^*, ..., S_k^*\}$
Let the sets $\{S_1^*, ..., S_k^*\}$ cover every item in I.

So they cover every item of any subset X of I. So since X is covered by k sets, there must be some set that covers greater than one kth fraction of X

16.3 Proof that Greedy Set Cover Almost Works

Theorem 26. If the optimal set cover has cardinality k then the greedy algorithm finds a solution of cardinality at most $kln(n)$.

Proof. wlog let the greedy algorithm output $\{S_1, ..., S_T\}$ and let the optimal solution be $\{S_1^*, ..., S_k^*\}$.

We want to show that $T \leq kln(n)$

Let I_t be the uncovered items in the start of step t For example, I_1 is just I.

Since $I_t \subseteq I$, by the observation before, there's a set that covers at least $\frac{1}{k}|I|$ items in I_t

This means in step t it must pick a set the covers at least $\frac{1}{k}|I|$ items in I_t.

$$\Rightarrow |I_{t+1}| \leq |I_t| - \frac{1}{k}|I_t|$$
$$= (1 - \frac{1}{k})|I_t|$$
$$.$$
$$.$$
$$.$$
$$\leq (1 - \frac{1}{k})(1 - \frac{1}{k})...(1 - \frac{1}{k})|I_t|$$

We iterate t times, so

$$= (1 - \frac{1}{k})^t |I_1|$$

And since I_1 is just n:

$$= (1 - \frac{1}{k})^t n$$

Key fact: $1 - x < e^{-x} \forall x \neq 0$

So if we let $\frac{1}{k} = x$, then we have:

$$|I_{t+1}| < (e^{-\frac{1}{k}})^t n$$
$$= e^{-\frac{t}{k}} n$$

Now setting $t = kln(n)$, then:

$$|I_{t+1}| < (e^{-\frac{kln(n)}{k}})^t n$$
$$= e^{-ln(n)} n$$
$$= 1$$
$$\Rightarrow |I_t| = |I_{kln(n)+1}| < 1$$

which means that it's empty, which means we're done at step t. So there was $t = kln(n)$ steps, we finished, meaning there's at most $kln(n)$ sets that the algorithm picked. □

So this is a $log(n) - approximation$ algorithm for this problem. This is actually a bad approximation, but it's the best we've come up with unless we solve P=NP.

16.4 Running Time

There are n iterations, and there's at most n distance updates at each iteration, so $O(n^2)$

16.5 The Hitting Set Problem

Given a collection S of m elements, and a collection I of $\{I_1, .., I_n\}$ sets that are subsets of S.

We want to select as few elements as possible such that there is at least one element selected in every set.

In other words, we want to find the smallest set X of elements such that every set I is "hit".

It was left as an exercise to show that this is exactly the same as the set cover problem.

17 Matriods

Given a set E of elements where each element has a weight $w_e \geq 0$. There is a collection $\mathcal{F}$ of feasible subsets of E. Feasible meaning a valid solution to the problem.

Each set $F \in \mathcal{F}$ is a valid solution with weight:

$$w(F) = \sum_{e \in F} w_e$$

So the weight of a solution is the sum of the element weights.

The problem is then to find a feasible set in $\mathcal{F}$ with the maximum weight.

17.1 The Hereditary Property

$\mathcal{F}$ satisfies the hereditary property if:

$$F \in \mathcal{F} \Rightarrow \hat{F} \in \mathcal{F}, \forall \hat{F} \subseteq F$$

Basically subsets of a feasible solution are also a feasible solution.

This arises in the interval selection problem. E is the set of intervals, $F \in \mathcal{F}$ if F is a disjoint collection of intervals.

The Maximum Weight Spanning tree problem also has this property. Where E is the set of edges, and $F \in \mathcal{F}$ if F is a forest. (If you solve each component of the forest, you can put them together into the larger solution). (Subsets of forests are also forests)

17.2 The Greediest Algorithm

A generic algorithm for a hereditary set system:

Sort the elements by weight $\{w_1 \geq w_2 \geq ... \geq w_m\}$.
Set $T = \phi$
For $i = \{1, 2, ..., m\}$
If $T \cup e_i \in \mathcal{F}$ then set $T \leftarrow T \cup e_i$.

Basically, if making my solution bigger by adding the current element is still in the valid solution, then do so.

17.2.1 The Running Time

Sorting time is $O(mlogm)$, and there are m iterations. Each test for feasibility takes time T. So $O(mlogm + mT)$. So as long as test for feasibility is fast, then our algorithm is fast.

17.2.2 Does It Work?

This is essentially Kruskal's algorithm, which works in that scenario.

This does **NOT** work for the interval selection problem.

When does it work then?

17.3 The Augmentation Property

$\mathcal{F}$ satisfies the augmentation property if:

$$F, \hat{F} \in F \text{ and } |F| > |\hat{F}|$$

$$\Rightarrow \exists e \in \hat{F} \text{ such that } \hat{F} \cup e \in \mathcal{F}$$

Basically, if there are two feasible sets where one is strictly larger than the other, there is an element in the bigger set that could be added to the smaller set, and still overall be a feasible solution.

17.4 What is a Matroid?

A matroid is a non-empty set system $M = (E, \mathcal{F})$ that satisfies both the Hereditary and Augmentation Properties.

Notice that that hereditarity and non-emptyness $\Rightarrow \phi \in \mathcal{F}$

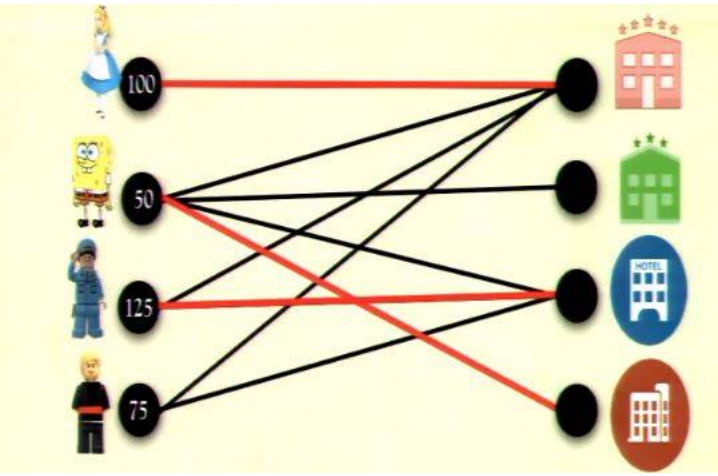
17.4.1 Examples of Matroids

E is the set of edges in a graph. $F \in \mathcal{F}$ if F is a forest. Basically, subsets of forests are still forests, and you can augment forests by adding an edge in a bigger forest.

E is a finite set of vectors in a vector space. $F \in \mathcal{F}$ if F is a collection of linearly independent vectors.

E is the set of left vertices in a bipartite graph. $F \in \mathcal{F}$ if there is a matching in the graph that matches each vertex in F to a distinct vertex on the right. (Halls theorem!)

The online auction problem:



Here you want to maximize profits.

The Job Scheduling Problem with Deadlines:

One job can be processed per day, each job has a deadline, and a late cost. Minimize the losses and complete the most jobs before the deadlines.

As an exercise, prove that these are Matroids.

17.5 Characterization of Matroids

The greediest algorithm works when matroids are the structure we're dealing with!

Theorem 27. A hereditary, non-empty set system M is a matrid $\Leftrightarrow$ the greediest algorithm outputs the optimal solution in M for any set of weights w.

Proof. ($\Rightarrow$)
First, the greediest algorithm works on matroids:

Let the algorithm output $\{e_1, ..., e_l\}$ where:

$$w(e_1) \leq w(e_2) \leq ... \leq w(e_l)$$

Let the optimal solution be $\{e_1^*, ..., e_l^*\}$ where:

$$w(e_1^*) \leq w(e_2^*) \leq ... \leq w(e_k^*)$$

Notice that since we have a matroid, the augmentation property holds. So we have $l \geq k$, otherwise, the algorithm would have selected another element. (The greediest algorithm needs to select at least as many elements as the true solution).

Now we want to show that $w(e_i) \geq w(e_i^*)$ (the greedy solution is at least as good as the optimal one).

Suppose not. Let j be the smallest index with $w(e_j) < w(e_j^*)$. Now consider:

$$\hat{F} = \{e_1, ..., e_{j-1}\} \text{ and } F = \{e_1^*, ..., e_j^*\}$$

So by the augmentation property, $\exists e_i^* \in F$ such that $\hat{F} \cup e_i \in \mathcal{F}$

Or in English, $\hat{F}$ is a subset of a feasible solution, so it's a feasible solution itself, so we can add e_i from the optimal solution and still be feasible.

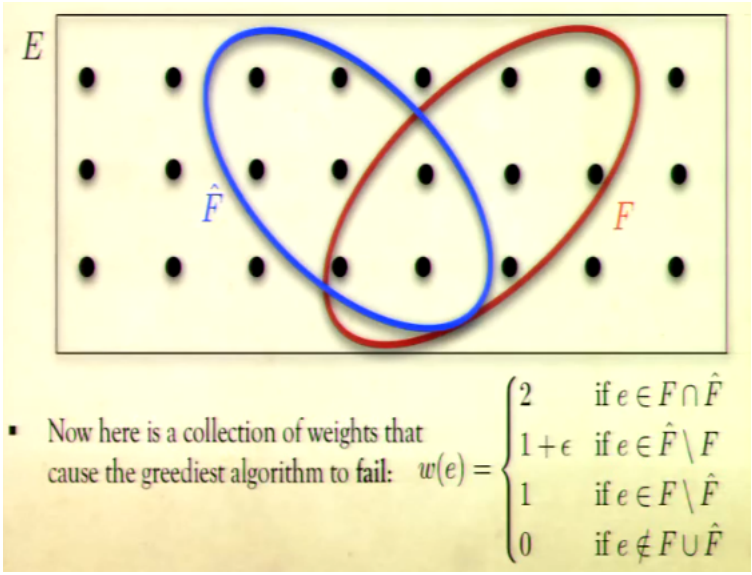
But since we ordered everything, $w(e_i^*) \geq w(e_j^*) > w(e_j)$. This is a contradiction because then the greediest algorithm should have chosen e_i^* instead of e_j, since choosing e_j produces a lesser result.

($\Leftarrow$) Take a hereditary set system M that is not a matroid. Then:

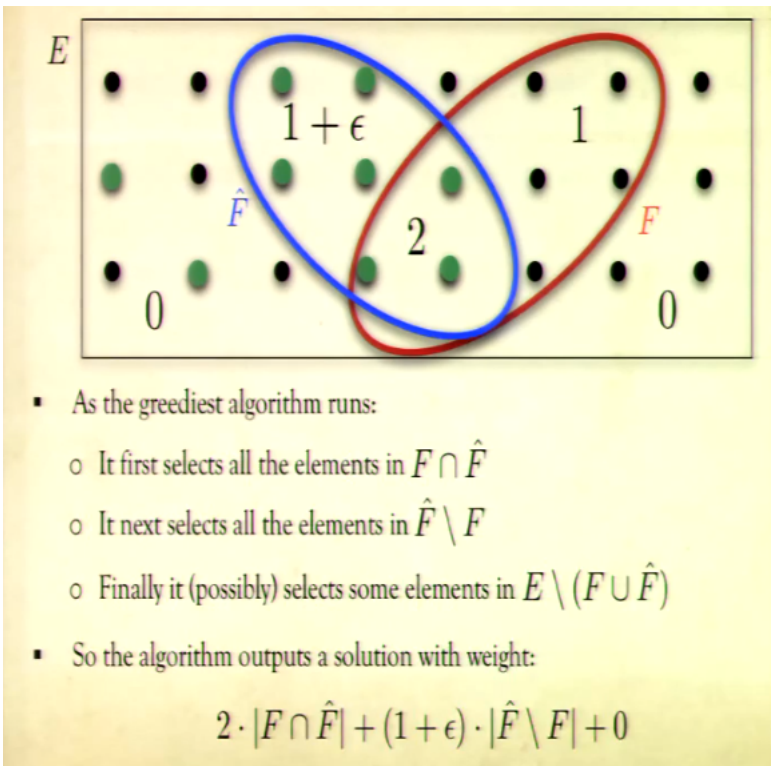
$$\exists F, \hat{F} \in \mathcal{F} \text{ with } |F| > |\hat{F}| \text{ but } \nexists e \in F - \hat{F} \text{ s.t } \hat{F} \cup e \in \mathcal{F}$$

Or in English, you have two feasible solutions with one larger than the other, but you cannot augment the smaller one with an element of the larger and still have a feasible solution.

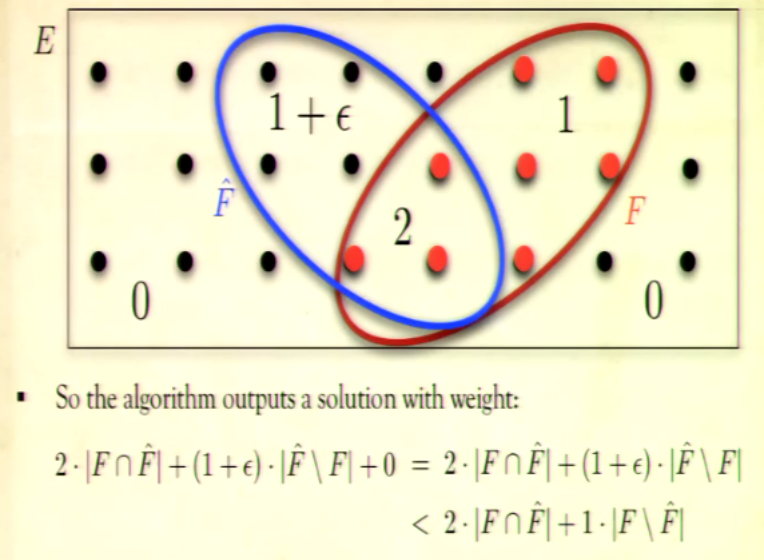
Here's a counter example:



The greedy algorithm produces this solution:



But this would have been better:



□

Part IV
Dynamic Programming

18 Fibonacci Numbers

Base Cases:

$$F(0) = 1, F(1) = 1$$

Recurrnce:

$$F(n) = F(n-1) + F(n-2) \forall n \geq 2$$

Models the growth of asexual populations.

18.1 Closed Form

The above recurrence can be solved and this form is obtained:

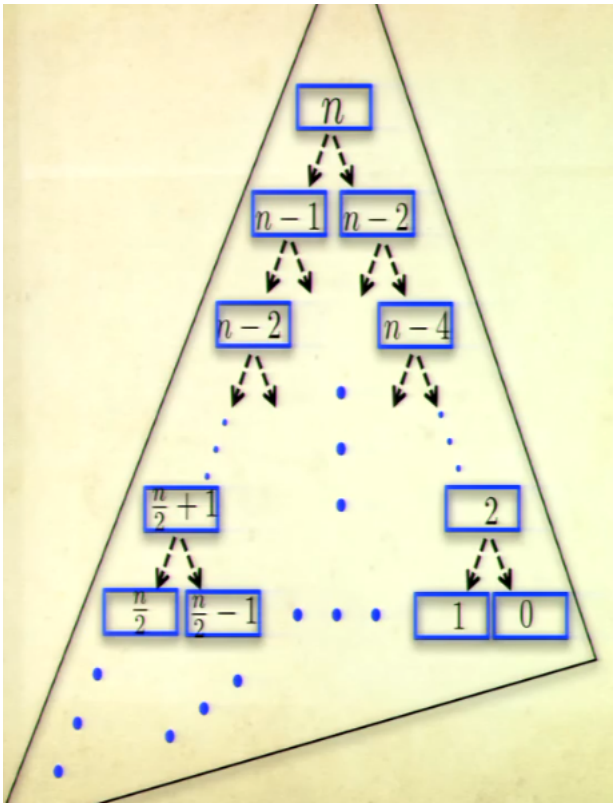
$$F(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$$

Which turns out to be $\Theta(1.618^n)$.

But how long would it take to solve the recurrence by unwinding the recursive formula? (Not by using the techniques in MATH240).

18.2 Recursive Tree of Fibonacci Formula

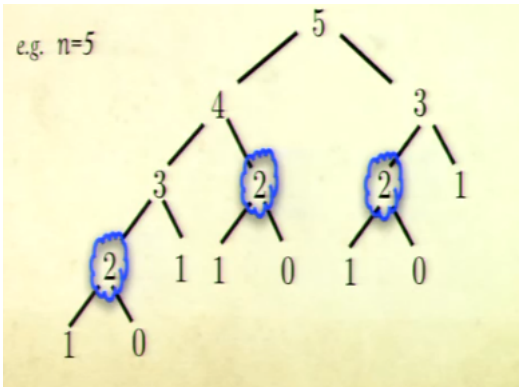
Well, at each level we're doing a constant amount of work. So we'll just add a +1 every time. We split into two sub-problems, so the work increases as 2^l at each level. But how many levels?



Well, we can see that the right side of the tree is decreasing by 2 each time, so there are $\frac{n}{2}$ levels on that side. So the running time is at least $\Omega(2^{\frac{n}{2}}) = \Omega(\sqrt{2}^n)$ Thats so slow!

18.2.1 Why is it so slow?

The problem is that the recursive formula solves the same subproblem many times.



Here it solves for $n = 2$, 3 times. The solution is storage.

19 Dynamic Strategies

19.1 Top-Down (Memoization)

Our Fibonacci example:

If $n < 1$ then return 1.

If $F(n)$ is undefined
then set $F(n) \leftarrow F(n-1) + F(n-2)$

So really you only solve problems once.

There are $n - 1$ additions, which may be n digits long, so $O(n^2)$ (adding n-digit numbers is $O(n)$).

19.2 Bottom-Up

Just start at the base cases and accumulate up to n. Essentially with iteration.

19.3 Top-down vs Bottom-Up

Sometimes you don't need to solve all of the subproblems, so top-down solves only the ones that absolutely need to be solved. So is faster in those cases.

19.4 Difference Between Divide + Conquer and DP

Much like divide and conquer, we're dividing up our problems into sub-problems, solving them and putting them together into a solution. However in DP, these sub-problems may overlap, until the Divide and Conquer problems.

19.5 Formula for Running Time of Dynamic Program

Let k be the number of sub-problems in the recursive formula, and let l be the number of sub-problems overall, then the running time is:

$$O(kl)$$

20 Interval Scheduling Revisited

When we used greedy algorithms, we said it would be useful to have a more general version.

20.1 Weighted Interval Selection

There is a set $I = \{1, 2, .., n\}$ of intervals.Each interval has a start time s_i, finish time f_i, and now a value v_i. The goal is to maximize the value of our set S, where S is the set of disjoint intervals with the maximum total value $\sum_{i \in S} v_i$.

Recall the first finish algorithm worked for non-weighted algorithms. Does it still work for the wighted interval selection? NO!

20.2 Dynamic Approach

Recall that for a dynamic program we want the solution $\mathcal{P}$ to be computable using a set of sub-problems, each with a natural ordering from smallest to largest. We also need there to be a polynomial number of sub-problems.

So how do we define the sub-problems?

In this situation, we have n intervals, $\{1, 2, ..n\}$. We'll simply define a sub-problem $\{1, 2, ..., l\}$, and find the maximum value for this set of intervals.

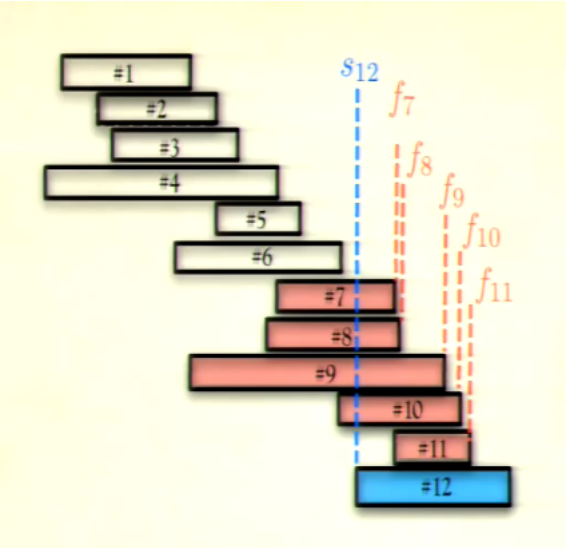
But we can change the labels $\{1, 2...n\}$ to whatever we want, so whats the best labeling of the intervals?

Again, the best way to do it is by finish times.

20.2.1 Why This Ordering?

First, notice that any interval finishing after the start interval n, clashes with it.

Next, notice that intervals that clash with n are consecutive.



Proof. Let interval l be the lowest index interval finishing after n starts. Then, by our first observation, l clashes with n. But for any interval $i > l$, we have that $f_i \geq f_l$, since we ordered by finish times. This means that $f_i > s_n$, which means that i also clashes with n. So the intervals $\{l, l+1, ..., n-1\}$ all clash with n. Conversely, no interval i with index $i < l$ since, $s_i < f_i \leq s_n < t_n$. Which concludes our proof.

□

Let $h(n)$ be the highest index interval that is disjoint from n. If we select interval n, then $\{h(n)+1, h(n)+2, ..., n-1\}$ are conflicting.

So if we select interval n, then we can only pick the next to be from 1 to $h(n)$. (This is our subproblem). What if we don't select interval n? Then we try to find an optimal solution from groups 1 to $n-1$.

Note that this same property holds for any $h(j) < j$

So the recursive formula for our algorithm is:

$$optimal(j) = max\{optimal(j-1), v_j + optimal(h(j))\} \ \forall j \geq 1$$

Basically, take either the optimal solution for not picking j, or take j, (and thus it's value, v_j) and find the optimal solution for the things up to $h(j)$, and see which one is better.

However, as we saw with Fibonacci numbers, solving this with the formula above will be too slow. So we'll need a dynamic program.

20.3 Weighted Interval Selection Algorithm

Sort the intervals by finish times.
Set $opt(0) = 0$
For $j = \{1, 2, .., n\}$
Let $h(j) < j$ be the highest index interval that is disjoint from j.

Set $opt(j) = max\{opt(j-1), v_j + opt(h(j))\}$ If $opt(j) + opt(j-1) then parent(j) = j-1, else parent(j) = h(j)$

This requires storing a pointer to the parent of each interval, so that we can know which intervals to select in the end by following the pointers back.

20.3.1 The Running Time

Recall, the running time of a DP in general is the number of sub-problems multiplied by the time required to use the recursive formula at each step.

We have n sub-problems, and to solve a sub-problem we examine the solutions to 2 smaller subproblems (constant time), so this is $O(n)$

20.4 Key Points

The four steps we did in this problem, can be used in almost any dynamic program. First, understand the underlying structure of the program. Second, set up a recursive formula. Third, solve the recursion by a bottom-up dynamic program. Last, to recover the solution, follow the pointers backwards.

Note that most of the work is in step 1.

21 Classifying Dynamic Problems

With the 4 steps in mind, we need to define the sub-problems clearly. We need them to be computable using these sub-problem, we want an ordering of the sub-problems, a polynomial number of sub-problems, and a recurrence giving the solution in terms of the subproblems.

21.1 Structure of Sub-problems

There are 4 main classes of dynamic sub-problem structures. 1.) One-sided interval 2.) Box Structure 3.) Two-sided interval 4.) Tree Structure.

21.1.1 One-Sided Interval

The input is $\{x_1, x_2, ..., x_n\}$ and the sub-problems are of the form $[j] = \{x_1, .., x_j\}$ Basically the subproblems are just pieces of the input list. For example, the weighted interval selection problem.

21.1.2 Box-Structure

The input is $\{x_1, x_2, ..., x_n\}$, $\{y_1, y_2, ..., y_n\}$. And the sub-problems are of the form: $[i, j] = \{x_1, x_2, ..., x_i, y_1 y_2, .., y_j\}$. Essentially the one-sided interval, but on two dimensions. Example: Knapsack problem, humpty dumpty.

21.1.3 Two-Sided Interval

The input is $\{x_1, x_2, ..., x_n\}$ and the subproblems look like: $\{x_i, x_{i+1}, ..., x_j\}$ basically an interval that doesn't start at 1. Example: RNA secondary structure problem.

21.1.4 Tree Structure

The input is a rooted tree with a vertex set $\{x_1, x_2, ..., x_n\}$ and the sub-problems are sub-trees rooted at x_j.

22 Knapsack Problem

Say you have a bag with capacity W, there are n items to put in it. Item i has value v_i and weight w_i. We want to get the maximum total value into the knapsack.

$$max \sum_{i=1}^{n} v_i x_i$$

such that we're less than W, and x_i is 0 or 1.

The dimensions of the box-structure are the items, and the capacity. What happens if we select item n? We lose capacity, so $W < -W - w_n$, and the

set of items loses n. If we don't pick n, we cast it aside, so the capacity is unchanged, and the set of items loses n.

22.1 Recursive Formula

$$opt(j, w) = max\{opt(j-1, w), v_j + opt(j-1, w-w_j)\}$$

22.2 Dynamic Program

Set opt(0,w) = 0 $\forall w \leq W$
Set opt(j,w)=0 $\forall j, w$
For $j = \{1, 2, ..n\}$
For $w = \{1, 2, ..., W\}$ Set opt(j,w) = $max\{opt(j-1, w), v_j + opt(j-1, w-w_j)\}$
Output opt(n,W)

We also need to add pointers if we want to output the optimal solution.

22.3 Running Time

There are nW sub-problems, and we need to examine the solutions to two sub-problems for each one, so $O(nW)$. This is a $pseudo-polynomial$, algorithm, meaning its polynomial in the number of weights. If W is small, it's all good.

23 The Humpty-Dumpty Problem

There is a skyscraper with T storeys. You have k eggs. Whats the maximum story from which an egg does not break? Solve with minimum number of egg-drop tests.

You could try from floor 0 and work your way up, but that might take T tests.

You could try binary search, but then $k \leq log(T)$

23.1 Dynamic Approach

The two dimensions are: the number of eggs remaining, the number of storeys remaining. Let $d(T, k)$ be the number of drops needed to guarantee we can solve the problem.

If the egg breaks, then we only have $k-1$ eggs left, and $t-1$ storeys left. So $d(T, v) \leq 1 + d(t-1, k-1)$

Set $d(\tau, 1) = \tau$
Set d(0,k) = 0
For $\tau = \{1, 2, ..\tau\}$
For $k = \{1, 2, .., k\}$
Set $d(\tau, k) = 1 + min\{max\{d(t-1, k-1), d(T-t, k)\}\}$

23.2 Running Time

There are Tk sub-problems, and to solve a sub-problem we need to examine the solutions to 2T smaller problems, so $O(T^2k)$.