COMP 251: Algorithms & Data Structures

Owen Lewis

Winter 2018

Contents

On Copyright and Stuff

These are just my notes from class. I just wrote down whatever the the teacher put up on the board, and then I typed the stuff up later because it helps me better internalize it and also because I can't read my own handwriting after a week or so. I'm not claiming to own anything here (except for the pictures that I made, I guess.) Don't sell these notes. That's a real dick move.

1 Overview of Graph Theory

1.1 Definitions

A graph $G = (V, E)$ is a set V of vertices (a.k.a. nodes) and a set E of edges (denoting vertex pairs). We set $n = |V|$, and $m = |E|$. A graph is said to be *undirected* when for any edge $(u, v) \in E$ there exists an edge $(v, u) \in E$ for some nodes u, and v. A graph is said to be *directed* if it is not undirected. In other words, the edge set of a directed graph consists of ordered pairs where the edge set of an undirected graph consists of unordered pairs.

A *walk* is a set of vertices $\{v_0, v_1, \ldots, v_\ell\}$ such that $(v_i, v_{i+1}) \in E$, $\forall\ 0 \leq i \leq \ell$. A walk where $v_0 = v_\ell$ is said to be a *circuit* or a *closed walk*. A circuit where every edge in the graph is used exactly once is known as an *Eulerian circuit*. A *cycle* is a walk $\{v_0, v_1, \ldots, v_\ell\}$ such that every vertex is distinct except $v_0 = v_\ell$. A cycle where every vertex of the graph is used exactly once is known as a *Hamiltonian cycle*. A walk where every vertex is distinct is said to be a *path*.

A graph is said to be *connected* if for each $u, v \in V$ there exists a walk from u to v. A graph is said to be *disconnected* id it is not connected. Each connected subgraph of a graph is called a *component*. A connected graph therefore has exactly one component.

A connected component with no cycles is called a *tree*. A graph whose components are all trees is said to be a *forrest*. A tree is said to be *spanning* if it contains every vertex in the graph. A vertex in a tree with at most one neighbour is called a *leaf*.

A *matching* is a set of vertex-disjoint edges i.e. each edge is incident to at most one other edge in a matching. A matching is said to be *perfect* if every vertex is incident to exactly one edge in the matching.

A *clique* is a set of pairwise adjacent vertices. In *independent set* (a.k.a. a *stable set*) is set of pairwise non-adjacent vertices.

A *bipartite graph* is a graph such that the vertex set V can be partitioned as $V = X \cup Y$ where each edge has one node in X and the other node in Y. Note that X and Y are necessarily independent sets.

1.2 Some Theorems for Undirected Graphs

Theorem: (Handshaking Lemma) Let $G = (V, E)$ be an undirected graph, let $\Gamma(v) := \{u : (u, v) \in E\}$ be the set of neighbours of a node v, and let the *degree* $\deg(v)$ of a vertex v equal the cardinality of $\Gamma(v)$. Then there are an even number of vertices with odd degree.
Proof: First note that since we're double-counting the number of pairs where

(v, e) is an edge incident to v

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

Since the degree of a vertex is either even or odd, we can partition V into a set of odd-degree vertices $\mathcal{O}$, and a set of even-degree vertices $\mathcal{E}$. This gives us

$$\sum_{v \in V} \deg(v) = \sum_{v \in \mathcal{O}} \deg(v) + \sum_{v \in \mathcal{E}} \deg(v)$$

which implies

$$\sum_{v \in \mathcal{O}} \deg(v) = 2 \cdot |E| - \sum_{v \in \mathcal{E}} \deg(v)$$

since both the $2 \cdot |E|$ term is even (obvious) and the $\sum_{v \in \mathcal{E}} \deg(v)$ term is even (sum of even numbers) then the $\sum_{v \in \mathcal{O}} \deg(v)$ term must also be even. □

Theorem: (Euler's Theorem) If G is an undirected graph then G contains an Eulerian circuit if and only if every vertex has even degree.
Proof: Easy proof by induction

Lemma: A tree T with $n \geq 2$ vertices has at least one leaf vertex.
Proof: Trees are connected so there exists no vertices with degree 0 when $n \geq 2$. Suppose each vertex has degree of at least 2. Then consider the longest path $P \subseteq T$, $P = \{v_1, v_2, \ldots, v_{\ell-1}, v_\ell\}$. Since $\deg(v_\ell) \geq 2$, $\exists$ a neighbour (of v_ℓ) $x \in P$ with $x \neq v_{\ell-1}$. If $x = v_{\ell+1}$ then P is not the longest path, a contradiction. Therefore, for P to be the longest path, x must be somewhere else in P, but this creates a cycle, another contradiction. Thus there must exist at least one node v such that $0 < \deg(v) < 2$ – a leaf. □

Theorem: A tree with n vertices has exactly $n - 1$ edges.
Proof: Simple proof by induction.
Base case: A tree with one vertex trivially has 0 edges.
Induction Hypothesis: Assume any tree with $n - 1$ vertices has $n - 2$ edges.
Inductive Step: Take a tree with $n \geq 2$ vertices. By the previous lemma this tree contains a leaf vertex v. This implies that $T \setminus \{v\}$ is a tree with $n - 1$ vertices and by the induction hypothesis $T \setminus \{v\}$ is a tree with $n - 2$ edges, which imples that T is a tree with $n - 1$ edges. □

Theorem: (Hall's Theorem) Let $G = (X \cup Y, E)$ with $|X| = |Y|$ be a bipartite graph. G contains a perfect matching if and only if $\forall\ B \subseteq X$, $|\Gamma(B)| \geq |B|$ (Hall's condition).
Proof: Firstly, the ($\Rightarrow$) direction is fairly obvious. If $B \subseteq X$ with $\Gamma(B) < |B|$ then the graph can't have a perfect matching. The ($\Leftarrow$) direction is a bit trickier. Suppose Hall's condition is satisfied. Then, take the maximum cardinality

matching M is the graph. If M is perfect then we are done. Otherwise there must exist an unmached vertex b_0.

- Since Hall's condition holds, we have $|\Gamma(\{b_0\})| \geq |\{b_0\}| = 1$ so b_0 must have at least one neighbour s_0.
- Suppose s_0 is matched in M to b_1.
- Since Hall's condition holds, we have $|\Gamma(\{b_0, b_1\})| \geq |\{b_0, b_1\}| = 2$ so $\{b_0, b_1\}$ must have at least one neighbour $s_1 \neq s_0$.
- Suppose s_1 is matched in M to b_2.
- Since Hall's condition holds, we have $|\Gamma(\{b_0, b_1, b_2\})| \geq |\{b_0, b_1, b_2\}| = 3$ so $\{b_0, b_1, b_2\}$ must have at least one neighbour $s_2 \notin \{s_0, s_1\}$.
- ...

we repeat this argument as long as we can. Since the graph contains a finite number of vertices this process must terminate, but it can only terminate when we reach an unmatched node s_k. Using the edges we've formed in M we can create a path P from b_0 to s_k that alternates between using non-matching edges and using matching edges. Swapping the matching edges with the non-matching edges gives us one more matching edge (as we have an odd number of edges.) This is still a valid matching as the internal nodes of P are still incident to exactly one matching edge. Also, the end nodes, b_0 and s_k were previously unmatched but are now incident to exactly one edge in the new matching. Thus M isn't the maximum capacity matching – a contradiction. □

1.3 """Data Structures""" for Representing Graphs

1.3.1 Adjacency Matrices

For a graph, an *adjacency matrix* M is a matrix suxh that

1. There is a row for each vertex
2. There is a column for each vertex
3. The $ij - th$ entry is defined as $M_{ij} = \begin{cases} 1, (i,j) \in E \\ 0, (i,j) \notin E \end{cases}$

Note that in an undirected graph the matric is symmetric around the diagonal because $(i, j) \equiv (j, i)$. Of course this is not necessarily true of directed graphs.

1.3.2 Adjacency Lists

An *adjacency list* of an undirected graph is such that for each vertex v of V we store a list of its neighbours. For a directed graph we have two lists: one in which we store the in-neighbours of v and one in which we store the out-neighbours of v.

1.3.3 Adjacency Matrices vs. Adjacency Lists

The main difference between the two is the amount of storage required to implement them.

- An adjacency matrix requires we store $\Theta(n^2)$ numbers
- An adjacency list requires we store $\Theta(m)$ numbers

In any graph $m = O(n^2)$. This means that for a sparse graph adjacency lists are highly favourable in terms of space complexity.

Verifying whether an edge exists, however, is much faster in an adjacency matrix – when using the array representation of a matrix it takes $O(1)$ time, where verifying the existance of an edge takes $O(\log n)$ time for an ordered adjacency list (using binary search), and $O(n)$ time if the adjacency list is not ordered (using sequential search).

2 Divide & Conquer

A *divide and conquer* algorithm ideally breaks up a problem of size n into smaller sub-problems such that:

- There are exactly a sub-problems
- Each sub-problem has a size of at most $\frac{1}{b} \cdot n$
- Once solved, the solutions to the sub-problems must be combined in $O(n^d)$ time to produce a solution to the original problem

Therefore the time-complexity of a divide and conquer algorithm satisfies a recurrence relation given by

$$T(n) = a \cdot T\Big(\frac{n}{b}\Big) + O(n^d)$$

2.1 The Master Theorem

Lemma 1: $\sum_{k=0}^{\ell} \tau^k = \frac{1-\tau^{\ell+1}}{1-\tau}$, for any $\tau \neq 1$.

Proof:

$$\begin{aligned}(1-\tau)\sum_{k=0}^{\ell}\tau^k &= \sum_{k=0}^{\ell}\tau^k - \sum_{k=1}^{\ell+1}\tau^k \\ &= \tau^0 - \tau^{\ell+1} \\ &= 1 - \tau^{\ell+1} \\ \sum_{k=0}^{\ell}\tau^k &= \frac{1-\tau^{\ell+1}}{1-\tau}\end{aligned}$$

□

Lemma 2: $x^{\log_b y} = y^{\log_b x}$ for any base $b \in \mathbb{R}$.
Proof: From the power rule of logarithms we have

$$\log_b x \cdot \log_b y = \log_b \left(y^{\log_b x}\right)$$

similarily, we have

$$\log_b x \cdot \log_b y = \log_b \left(x^{\log_b y}\right)$$

therefore

$$\log_b \left(x^{\log_b y}\right) = \log_b \left(y^{\log_b x}\right)$$

thus

$$x^{\log_b y} = y^{\log_b x}$$

□

Theorem: (The Master Theorem) If a recurrence relation is of the form $T(n) = a \cdot T\Big(\frac{n}{b}\Big) + O(n^d)$, for constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{, if } a < b^d \text{ [Case I]} \\ O(n^d \cdot \log n) & \text{, if } a = b^d \text{ [Case II]} \\ O(n^{log_b a}) & \text{, if } a > b^d \text{ [Case III]} \end{cases}$$

Proof: By adding dummy numbers, we may assume that n is a power of b, i.e. $n = b^\ell$, for some $\ell \in \mathbb{N}_0$. Then

$$T(n) = n^d + a\Big(\frac{n}{b}\Big)^d + a^2\Big(\frac{n}{b^2}\Big)^d + \cdots + a^\ell\Big(\frac{n}{b^\ell}\Big)^d$$

simlifying, we get

$$T(n) = n^d\left(1 + \frac{a}{b^d} + \Big(\frac{a}{b^d}\Big)^2 + \cdots + \Big(\frac{a}{b^d}\Big)^\ell\right)$$

we now have three cases:

Case I: $\dfrac{a}{b^d} < 1$
Set $\tau := \dfrac{a}{b^d}$ then we have

$$T(n) = n^d \sum_{k=0}^{\ell} \tau^k$$

applying lemma 1 we get

$$T(n) = n^d\left(\frac{1 - \tau^{\ell+1}}{1 - \tau}\right) \leq n^d\Big(\frac{1}{1 - \tau}\Big)$$

but $\dfrac{1}{1 - \tau}$ is a constant so $T(n) \leq n^d$ and thus

$$T(n) = O(n^d)$$

Case II: $\dfrac{a}{b^d} = 1$
Then we have that

$$T(n) = n^d\big(1 + 1 + 1^2 + \cdots + 1^\ell\big) = n^d\big(\ell + 1\big)$$

but $n = b^\ell \Rightarrow \ell = \log_b n$, thus

$$T(n) = O(n^d \cdot \log n)$$

Case III: $\dfrac{a}{b^d} > 1$

Set $\tau := \dfrac{a}{b^d}$ then we have

$$T(n) = n^d \sum_{k=0}^{\ell} \tau^k$$

applying lemma 1 we get

$$T(n) = n^d \left(\frac{\tau^{\ell+1} - 1}{\tau - 1} \right) \leq n^d \left(\frac{\tau^{\ell+1}}{\tau - 1} \right)$$

but since $\tau - 1$ is a constant we get

$$\begin{aligned} T(n) &= n^d O(\tau^{\ell+1}) \\ &= O(n^d \tau^{\ell+1}) \\ &= O(n^d \tau^{\ell}) \\ &= O\left(\left(\frac{a}{b^d} \right)^{\ell} n^d \right) \\ &= O\left(\left(\frac{n}{b^{\ell}} \right)^{d} a^{\ell} \right) \end{aligned}$$

but $n = b^{\ell}$ so

$$T(n) = O(a^{\ell})$$

and $\ell = \log_b a$ so

$$T(n) = O(a^{\log_b n})$$

and applying lemma 2 gives

$$T(n) = O(n^{\log_b a})$$

This completes the proof. □

2.2 MergeSort

MergeSort is an algorithm to sort n numbers into non-decreasing order.

MergeSort($x_1, x_2, \ldots, x_n$)
    if $n = 1$
        return x_1
    else
        return Merge(MergeSort($x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor}$), MergeSort($x_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, x_n$))

Where the Merge function is a linear time algorithm combining two sorted lists (by comparing the first element in each list and moving the smaller element of

the two into our new list.)

MergeSort is correct!

- It calls itself on smaller instances until the division process terminates when it reaches a base case where each list has size 1
- MergeSort works trivially on the base cases
- This sort of strong induction-type proof works for just about all divide and conquer algorithms

MergeSort is efficient!

- The recurrence relation that we can construct to model the running time of MergeSort is given by

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

  as we break the problem into two sub-problems of size $\frac{n}{2}$, plus a linear time sub-routine to combine the solutions (Merge.)
- This recurrence relation is in the proper form to use the Master Theorem and indeed we're in its Case II.
- By the master theorem the running time of MergeSort is $O(n \cdot \log n)$.
- The running time can also be proved by unwinding the recurrence.

2.3 Binary Search

BInary Search is an algorithm to search for whether a key k exists in a given sorted list.

BinarySearch$(a_1, a_2, a_n; k)$
    while $n > 0$
        if $a_{\lfloor \frac{n}{2} \rfloor} = k$
            return true
        if $a_{\lfloor \frac{n}{2} \rfloor} > k$
            return BinarySearch$(a_1, \ldots, a_{\lceil \frac{n}{2} \rceil - 1}; k)$
        if $a_{\lfloor \frac{n}{2} \rfloor} < k$
            return BinarySearch$(a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, a_n; k)$
    return false

Binary Search works!

- It works for the same strong induction argument as MergeSort

Binary Search is efficient!

- The recurrence relation that we can construct to model the running time of Binary Search is given by

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

  as we break the problem into one sub-problem of size $\frac{n}{2}$.

- This recurrence relation is case II of the master theorem.
- By the master theorem the running time of MergeSort is $O(\log n)$.
- Again, the running time can also be proved by unwinding the recurrence.

2.4 Fast Multiplication

In COMP 250 we found an $O(n^2)$ algorithm to multiply 2 n-digit numbers (in any base). Let's try to do better using divide and conquer. Consider the example:

Let $\mathbf{x} := x_n x_{n-1} \dots x_{\frac{n}{2}+1} x_{\frac{n}{2}} \dots x_2 x_1$
Let $\mathbf{y} := y_n y_{n-1} \dots y_{\frac{n}{2}+1} y_{\frac{n}{2}} \dots y_2 y_1$

Then $\mathbf{x} = 10^{\frac{n}{2}}\mathbf{x}_L + \mathbf{x}_R$, where $\mathbf{x}_L = x_n \dots x_{\frac{n}{2}+1}$, and $\mathbf{x}_R = x_{\frac{n}{2}} \dots x_1$
Similarily $\mathbf{y} = 10^{\frac{n}{2}}\mathbf{y}_L + \mathbf{y}_R$, where $\mathbf{y}_L = y_n \dots y_{\frac{n}{2}+1}$, and $\mathbf{y}_R = y_{\frac{n}{2}} \dots y_1$

Thus

$$\begin{aligned}\mathbf{x} \cdot \mathbf{y} &= (10^{\frac{n}{2}}\mathbf{x}_L + \mathbf{x}_R)(10^{\frac{n}{2}}\mathbf{y}_L + \mathbf{y}_R) \\ &= 10^n \cdot \mathbf{x}_L\mathbf{y}_L + 10^{\frac{n}{2}}(\mathbf{x}_L\mathbf{y}_R + \mathbf{x}_R\mathbf{y}_L) + \mathbf{x}_R\mathbf{y}_R\end{aligned}$$

So we've found a divide and conquer algorithm to multiply two n-digit numbers. Here we're breaking the problem down into four sub-problems of size $\frac{n}{2}$, with a few additions thrown in there (can be done in linear time). Therefore, the recurrence for this algorithm is given by

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

then, by the master theorem, the running time of this algorithm is $O(n^2)$. Hmmm. This isn't better. Well thanks to our good friend Carl "G-Money" Gauss, all is not lost. Gauss was studying the product of complex numbers

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

which seemingly involved 4 multiplications, when he noticed that actually we can do it with only 3. Note that

$$(bc + ad) = (a + b)(c + d)$$

Now we can use this exact same trick but we'll replace the i with $10^{\frac{n}{2}}$. This gives

$$(\mathbf{x}_L\mathbf{y}_R + \mathbf{x}_R\mathbf{y}_L) = (\mathbf{x}_R + \mathbf{x}_L)(\mathbf{y}_R + \mathbf{y}_L) - \mathbf{x}_R\mathbf{y}_R - \mathbf{x}_L\mathbf{y}_L$$

So now we can break our problem into only 3 sub-problems! Our new recurrence is

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

so finally, by the master theorem, $T(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$

2.5 Fast Matrix Multiplication

Let's try to efficiently multiply two $n \times n$ matrices. Let

$$X := \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \text{ and } Y := \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

where X, and Y are two $n \times n$ matrices and $A, B, \ldots H$ are eight $\frac{n}{2} \times \frac{n}{2}$ matrices. Then their product Z is given by

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

Therefore multiplying two $n \times n$ matrices involves eight products of $\frac{n}{2} \times \frac{n}{2}$ matrices, and n^2 additions. Therefore the recurrence relation is given by

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

By the master theorem, this then takes $O(n^3)$ time. That's ok but again, we can do better. To do better however, we need a little algebraic trick. To do this, we need to define 7 matrices, $S_1, \ldots, S_7$ given by

$$\begin{array}{ll} S_1 = (B - D) \cdot (G + H) & S_2 = (A + D) \cdot (E + H) \\ S_3 = (A - C) \cdot (E + F) & S_4 = (A + B) \cdot H \\ S_5 = A \cdot (F - H) & S_6 = D \cdot (G - E) \\ S_7 = (C + D) \cdot E & \end{array}$$

then the product of X, and Y is given by

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

This means that we only need to do 7 multiplications! Therefore our new recurrence is

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

and thus by the master theorem $T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$

2.6 Fast Exponentiation

Consider the following algorithm to quickly calculate x^n:

FastExp(x, n)
    if $n = 1$
        return x_1
    else if n is even
        return FastExp$(x, \lfloor \frac{n}{2} \rfloor)^2$
    else if n is odd
        return $x \cdot$FastExp$(x, \lfloor \frac{n}{2} \rfloor)^2$

Assuming that n is a power of two, we get that recurrence is given by

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

and thus by the master theorem this algorithm has time complexity of $O(\log n)$.

2.7 The Selection Problem

Suppose we're given a list of number and we want to find the k^{th} smallest number in that list. We can use the following divide and conquer algorithm:

Select$(\mathcal{S}, k)$
    if $|\mathcal{S}| = 1$
        return x_1
    else
        set $\mathcal{S}_L := \{x_i \in \mathcal{S} : x_i < x_1\}$
        set $\mathcal{S}_R := \{x_i \in \mathcal{S} : x_i \geq x_1\}$

        if $|\mathcal{S}_L| = k - 1$
            return x_1
        else if $|\mathcal{S}_L| > k - 1$
            return Select$(\mathcal{S}_L, k)$
        else if $|\mathcal{S}_L| < k - 1$
            return Select$(\mathcal{S}_R, k - 1 - |\mathcal{S}_L|)$

Well.. this actually sucks because it runs in $\Omega(n^2)$ time. Note that it would actually be faster to just sort the list and then pull the k^{th} element (this can of course be done in $O(n \cdot \log n)$ time.) One idea that we can exploit to improve this is to use a random pivot rather than deterministically selecting x_1 as a pivot.

We will define a *good pivot* to be such neither $\mathcal{S}_L$, nor $\mathcal{S}_R$ contain more than $\frac{3}{4}$ of $\mathcal{S}$, and a *bad pivot* to be otherwise. Note that by this definition the probability that a pivot will be either good or bad (respectively) is 0.5.

For randomized algorithms we're always interested in the *expected runtime* of the algorithm rather than the worst case runtime. The recurrence relation, $\bar{T}(n)$, for the expected runtime for the randomized version of the selection is then given by

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

where the $\frac{1}{2} \cdot \bar{T}\Big(\frac{3n}{4}\Big)$ term represents the recursive call given a good pivot, and the $\frac{1}{2} \cdot \bar{T}(n)$ term represents the recursive call on a bad pivot. It's less than or equal to because we will only ever use one of those terms at once but not both. We can clean this recurrence up a bit to find the expected runtime.

$$\begin{aligned}
\bar{T}(n) &\leq \frac{1}{2} \cdot \bar{T}\Big(\frac{3n}{4}\Big) + \frac{1}{2} \cdot \bar{T}(n) + O(n) \\
\frac{1}{2} \cdot \bar{T}(n) &\leq \frac{1}{2} \cdot \bar{T}\Big(\frac{3n}{4}\Big) + O(n) \\
\bar{T}(n) &\leq \bar{T}\Big(\frac{3n}{4}\Big) + O(n)
\end{aligned}$$

Applying the master theorem to that recurrence gives $\bar{T}(n) = O(n)$

This is good but let's try to do better, i.e. let's see if we can find a deterministic linear time algorithm. It turns out we can but it requires a little trick called the *median of medians*.

It's best to illustrate this concept with an example. Suppose we have a list, $\mathcal{S} = \{x_1, x_2, \ldots, x_n\}$, from which we want to find the median of medians. To do this we partition the list into $\left\lceil \frac{n}{5} \right\rceil$ groups of cardinality 5. Denote each such group by $G_1, \ldots, G_{\lceil \frac{n}{5} \rceil}$ Then we'll sort each group and let z_i be the median of each group G_i. Let m be the median of the set $\mathcal{Z} = \{z_1, \ldots, z_{\lceil \frac{n}{5} \rceil}\}$. Predictably, we call m the *median of medians*. This is significant because m will always be a good pivot. This is because there will always be at least $\frac{3n}{10} - 1$ numbers in $\mathcal{S}$ less than m, and $\frac{3n}{10} - 1$ numbers in $\mathcal{S}$ at least as large as m. Therefore $|\mathcal{S}_R| \leq \frac{7n}{10}$, and $|\mathcal{S}_L| \leq \frac{7n}{10}$.

Now that we have a deterministic way to guarntee we have a good pivot, all that's left is to bake it into a selection algorithm. Luckily, our algorithm won't vary too much since all we have to do is add a few steps at the beginning.

```
DetSelect(S, k)
    if |S| = 1
        return x_1
    else
        partition S into ⌈n/5⌉ groups of 5
        for j in range (1, ⌈n/5⌉)
            let z_j be the median of the group G_j.
        let Z = {z_1, ..., z_⌈n/5⌉}
        set m := DetSelect(Z, ⌈n/10⌉)

        set S_L := {x_i ∈ S : x_i < m}
        set S_R := {x_i ∈ S : x_i ≥ m}

        if |S_L| = k − 1
            return m
        if |S_L| > k − 1
            return DetSelect(S_L, k)
        if |S_L| < k − 1
            return DetSelect(S_R, k − 1 − |S_L|)
```

The recursive formula for the running time of this algorithm is

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

where the $T\left(\frac{n}{5}\right)$ term is the time to find the median of medians, the $T\left(\frac{7n}{10}\right)$ term is pivoting based on the median of medians. Unfortunately, this doesn't work with the master theorem and we can't simplify it down to a form that does (unlike with the randomized algorithm, we need both of the $T(\dots)$ terms.) Since it doesn't work with the master theorem, we need to use the recusion tree method. From the recursion tree method, $T(n) = O(n)$. So there it is.. a linear time deterministic algorithm to select the k^{th} smallest element from a list!

2.8 The Closest Pair of Points in a Plane

Suppose we're given a set $\mathcal{P}$ of n points in a 2D plane, $\mathcal{P} = \{p_1, p_2, \dots, p_n\}$ and want to find the closest 2. For each $i \leq i \leq n$, let $p_i = (x_i, y_i)$.

The easiest (correct) solution to think of is an exhaustive search where we calculate every distance between any two points and return the smallest pair. There are $\binom{n}{2}$ possible pairs of points, and calculating the distance between any two can be done in constant time, so this means the runtime is $O(n^2)$. This is

okay, it's certainly polynomial-time efficient, but let's do it faster.

First, perhaps it will be handy to consider the 1D case (the closest two pairs of points on a line.) Note that in this case, the closest pair of point must be adjacent in their x-ordering. Therefore, assuming the list is already sorted according to their x-ordering, all we need to do is calculate $n-1$ pairwise distances, which can be done in $O(n)$ time. Of course if the list is not sorted we have to do that first, so the closest pair can be found in $O(n \cdot \log n)$ time. If we try to mimic that idea in the 2D case we come up with the following algorithm:

Pair2D($\mathcal{P}$)

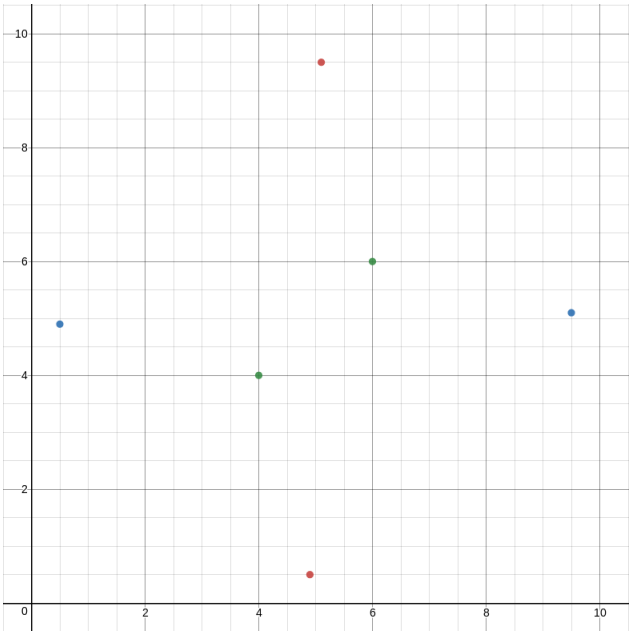
sort the pairs by their x-coordinate
sort the pairs by their y-coordinate

find the closest pair of points with respect to their x-coordinates
find the closest pair of points with respect to their y-coordinates

return the pair from the above 2 with the smallest distance between them

---

Sorting the lists takes $O(n \cdot \log n)$ time, and finding the closest points takes $O(n)$ time, so this algorithm take $O(n \cdot \log n)$ time. Does it work though? Nope. Consider this example:



It's clear here that the two green points are the closest together, but the algorithm would have returned either the pair of red points or the pair of blue points.

Let's try to come up with a divide and conquer algorithm to solve this problem. We can start by partitioning $\mathcal{P}$ into two sets, $\mathcal{P}_L$, and $\mathcal{P}_R$, of cardinality $\frac{n}{2}$. This gives us three possible outcomes:

1. The closest pair of points is in $\mathcal{P}_L$
2. The closest pair of points is in $\mathcal{P}_R$
3. The closest pair of points has one point in $\mathcal{P}_L$, and the other point in $\mathcal{P}_R$

To summarize this idea so far, we have the algorithm

DCPair2D($\mathcal{P}$)
    find the point q with the median x-coordinate
    partition $\mathcal{P}$ into $\mathcal{P}_L$, and $\mathcal{P}_R$

    find the closest pair of points in $\mathcal{P}_L$
    find the closest pair of points in $\mathcal{P}_R$
    find the closest pair of points with one point in $\mathcal{P}_L$ and the other in $\mathcal{P}_R$

    return the pair from the above 3 with the smallest distance between them

---

The recursive formula for this algorithm is given by

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

The $O(n^2)$ term comes from the work needed to compare all $\frac{n}{2}$ points in $\mathcal{P}_L$ with all $\frac{n}{2}$ points in $\mathcal{P}_R$. $\big(\frac{n}{2} \times \frac{n}{2} = \frac{n^2}{4}$ comparisons.$\big)$ Therefore, by the master theorem, the running time is $O(n^2)$.

This isn't any better than exhaustive search because there's a bottleneck step where we compare the points in $\mathcal{P}_L$ with those in $\mathcal{P}_R$. Indeed we're pretty much just doing exhaustive search but on only $\frac{n}{2}$ elements. With this in mind, let's try to slim this step down. Note that since we've solved both the left and right sub-problems, there's a minimum pairwise distance δ, given by

$$\delta = \min(\delta_L, \delta_R)$$

for which the distance between the closest two points will not exceed, even if the two points are not in the same partition. Therefore, when looking for pairs of points with one point in each partition, we only need to look for points within δ of the dividing line! From now on we will denote the set of points within δ of

the dividing line by $\mathcal{P}_M$. This trick does work, but not trivially, so it has to be proved.

Lemma: Let $p_i \in \mathcal{P}_L$, and let $p_j \in \mathcal{P}_R$. If $d(p_i, p_j) \leq \delta$ then $\{p_i, p_j\} \subseteq \mathcal{P}_M$.

Proof: WLOG assume that $p_i \notin \mathcal{P}_M$. Then $d(p_i, p_j) > \delta$. Similarily, if $p_j \notin \mathcal{R}_M$. Then $d(p_i, p_j) > \delta$. Thus if the pair of points is not within one of the two original sub-problems, it must be contained within $\mathcal{P}_M$. □

Let's implement this trick in the last algorithm we wrote up.

DCPair2D($\mathcal{P}$)

find the point q with the median x-coordinate
partition $\mathcal{P}$ into $\mathcal{P}_L$, and $\mathcal{P}_R$

find the closest pair of points in $\mathcal{P}_L$
find the closest pair of points in $\mathcal{P}_R$
find the closest pair of points in $\mathcal{P}_M$

return the pair from the above 3 with the smallest distance between them

---

There's still a problem. $\mathcal{P}_M$ can possibly contain all (or most) of the points! This means that our algorithm can *still* take $\Omega(n^2)$ time to measure the distances between the points in $\mathcal{P}_L \cap \mathcal{P}_M$, and the points in $\mathcal{P}_R \cap \mathcal{P}_M$. Once again, however, this was fortunately not all for nothing. Since $\mathcal{P}_M$ covers only a narrow band of the x-axis, we'll consider the area of the plane within δ of the dividing line (i.e. the area pf $\mathcal{P}_M$.) Suppose we divide this area into small squares of size $\frac{\delta}{2}$ then we have the following lemma:

Lemma: No two points from $\mathcal{P}$ will lie within the same square of width $\frac{\delta}{2}$.

Proof: WLOG take two points in a square in $\mathcal{P}_L$. Then their pairwise distance d is

$$d \leq \sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \frac{\delta}{\sqrt{2}} < \delta$$

which is a contradiction as δ is the smalles distance between two points. □

Theorem: Let $p_i \in \mathcal{P}_L$, and let $p_j \in \mathcal{P}_R$. If $d(p_i, p_j) \leq \delta$ then p_i, and p_j have at most 10 points between them in $\mathcal{P}_M$ with respect to their y-ordering.

Proof: WLOG suppose that p_i is below p_j in the y-order. Then p_j is in the same row of squares as p_i, or it is somewhere in the next two higher rows, or else they won't be within δ of eachother (a contradiction). Therefore we are working

in a rectangle that is at most three rows of squares high and four columns of squares wide. By the previous lemma we can have at most one point in any square and therefore in our 4×3 rectangle there can be at most 12 points. Thus, by choosing the the points on either extreme of the y-ordering of our rectangle, there can be at most 10 points between. □

This result implies that the idea underlying the algorithm from the 1D case can be applied after all! We just have to tweak it slightly so as to look for points that are 11 apart rather than just adjacent, i.e. we'll have to calculate at most $11n$ pairwise distances rather than at most $n - 1$. Thus we either find a pair of points in $\mathcal{P}_M$ closer than δ or we don't, but if we don't then we can conclude that no such pair of points exists, and we can do so in $O(n)$ time!

Let's update our algorithm to reflect this.

DCPair2D($\mathcal{P}$)
    find the point q with the median x-coordinate
    partition $\mathcal{P}$ into $\mathcal{P}_L$, and $\mathcal{P}_R$

    find the closest pair of points in $\mathcal{P}_L$
    find the closest pair of points in $\mathcal{P}_R$
    find the closest pair of points in $\mathcal{P}_M$ using the (updated) 1D algorithm

    return the pair from the above 3 with the smallest distance between them

---

Now that we've gotten the bottleneck step down to $O(n)$, the new recurrence relation for this algorithm is

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

where the $O(n)$ term comes from finding the median with respect to the x-coordinates (using algorithm from previous lecture), partitioning $\mathcal{P}$, finding $\mathcal{P}_M$ and applying the 1D algorithm. Thus, by the master theorem, the time complexity of this algorithm is $O(n \cdot \log n)$. (If you think about it, the new most expensive step is just sorting the points in $\mathcal{P}_M$ with respect to the y-coordinates! If you don't think that's hella cool then you have no soul.)

Like most divide and conquer algorithms, this one is correct and the proof follows immediately from strong induction, where the proof that the base cases are correct follows from the proofs of the above various lemmas and theorems.

3 Graph Algorithms

3.1 The Generic Search Algorithm

Suppose we want to search a graph to determine whether there exists a path between a root node r and some other node in the graph. Then we get the following algorithm:

```
Search(r)
    put r into a bag
        while the bag is not empty
            remove a node v from the bag
            if v is unmarked
                mark v
                for each arc (v, w)
                    put w into the bag
```

Where we say a new vertex (and therefore a path to that new vertex) is discovered when it's marked. We can write this algorithm in a bit more 'formal' pseudo-code as such:

```
Search(r)
    set B := {r}
        while B ≠ ∅
            let v ∈ B
            set B ← B \ {v}
            if v is unmarked
                mark v
                for each arc (v, w)
                    set B ← B ∪ {w}
```

Note that this algorithm allows multiple copies of the same node to be put into the bag. This *could* pose a problem as it would require our algorithm excecute superfluous steps to terminate. We can get around this by putting edges into the bag instead of nodes. Let's update our pseudo-code to reflect this:

```
Search(r)
    set B := {(r, *)}
        while B ≠ ∅
            let (u, v) ∈ B
            set B ← B \ {(u, v)}
            if v is unmarked
                mark v
                set p(v) ← u
                for each arc (v, w)
                    set B ← B ∪ {(v, w)}
```

Where $(r, *)$ is some pseudo-edge and $p(v)$ is the parent of v.

3.1.1 Time Complexity

We search every edge out of v only once (when u is first marked). The edge is added to the bag once and removed from the bag once. Thus the time complexity is $O(m)$ (where m is the cardinality of the edge set.)

3.1.2 Validity

Theorem: Let G be a connected, undirected graph. The generic search algorithn finds every vertex in G.

Proof: We must show that every vertex is marked by the algorithm. We will do this by induction on the smallest number k of edges in a path from the vertex to the root.
Base Case: $k = 0$. Then v is the root node r, but r is marked by the algorithm.
Induction Hypothesis: Assume any vertex v that has a path of $k - 1$ or fewer edges to the root r is marked.
Inductive Step: Assume there exists a path P with k edges from v to r. i.e.

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

Then there is a path Q with $k - 1$ edges from $u = v_{k-1}$ to r. i.e.

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

By our induction hypothesis u is marked. After we mark u we pace the edges incident to it in the bag. Therefore the edge (u, v) is added to the bag. Thus when (u, v) is removed from the bag we will mark v if it is not marked already. □

3.1.3 Search Trees

Note that after running this algorithm, each non-root vertex has exactly one predecessor.

Theorem: Let G be a connected, undirected graph. Then the predecessor edges form a tree rooted at r.

Proof: We will prove this by induction on the number k of marked vertices.
Base Case: $k = 1$. The root vertex r is the first one marked. Trivially, this forms a tree.
Induction Hypothesis: Assume the predecessor edges for the first $k-1$ marked vertices form a tree rooted at r.
Inductive Step: Let v be the k^{th} vertex to be marked. Assume that v was marked when the edge (u, v) was removed. Then $u = p(v)$. But (u, v) was added to the bag when we marked vertex u. Therefore u is in the set S of the first $k-1$ vertices to be marked. By the induction hypothesis, the predecessor edges for S form a tree T rooted at r. Thus $T \cup (p(v), v)$ is a tree rooted at r on the first k marked vertices. □

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

3.1.4 The Bag

The bag is just shorthand for "some data structure". In this algorithm, the bag can be any data structure we want but our choice has some important consequences. Specifically, for the graph search algorithm we really want to use one of three data structures, each of which will give a distinct algorithm:

1. Queue: The generic search algorithm becomes Breadth-First Search
2. Stack: The generic searcg algorithm becomes Depth-First Search
3. Priority Queue: The generic search algorithm becomes a Minimum Spanning Tree algorithm

3.2 Breadth-First Search

Breadth-First search is an implementation of the generic search algorithm using a queue as the data structure (the bag.) Because we use a queue, edges are added in the order of their distance from r. The vertices are then marked in order of their distance from r.

3.2.1 Breadth-First Search & Bipartite Graphs

Theorem: Let G be a graph. Then G is bipartite if and only if it contains no odd-length cycles.

Proof: ($\Rightarrow$) Assume G contains as a subgraph an odd-length cycle C. Let

$$C := \{v_0, v_1, \ldots, v_{2k}\}$$

WLOG assume $v_0 \in Y$. Then

$$v_0 \in Y \Rightarrow v_1 \in X \Rightarrow v_2 \in Y \Rightarrow v_3 \in X \ldots \Rightarrow v_{2k} \in Y$$

but $(v_0, v_{2k}) \in E$. Thus $v_0 \in Y \Rightarrow v_{2k} \in X$, a contradiction. Thus G is not bipartite.
($\Leftarrow$) Assume G contains no odd-length cycles. Then select an arbitrary root vertex r and run the BFS algorithm. We want to show that if G contains no odd-length cycles we can find two sets X, and Y such that each edge goes only between X and Y. Recall that for every edge (u, v) we have that u, and v are either in the same layer or are in adjacent layers. Now set

$$X := \bigcup_{\ell \text{ odd}} S_\ell, \text{ and, } Y := \bigcup_{\ell \text{ odd}} S_\ell$$

where S is a set of nodes in the layer of a BFS-generated search tree, and ℓ is the level of the node form the root. We claim that this choice of X, and Y gives a bipartite graph. Suppose that for each non-tree edge (u, v), that u, and v are in adjacent layers. Assume that there exists 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 T from u to v; let Q be the part in T from v to z. Since u, and v are in the same layer $|P| = |Q|$. But then the cycle $C = P \cup Q \cup (u, v)$ has an odd number of edges, so (u, v) can't exist, a contradiction. □

3.3 Depth-First Search

Running the generic search algorithm but using a stack as our bag gives depth-first search. An interesting property of depth-first search is that it partitions the edges of an undirected graph into two types:

1. Tree edges: Predecessor edges in the DFS tree T
2. Back edges: Edges where one endpoint is an ancestor of the other endpoint in T

we cannot have edges of the following type:

3. Cross edges: Edges where neither endpoint is an ancestor of the other in T

We can also define depth-first search recursively as such:

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

3.3.1 Ancestral Edges Theorem

Theorem: Let T be a depth-first search-generated tree in an undirected graph G. Then for each edge (u, v), either u is an ancestor of v in T, or v is an ancestor of u.

Proof: WLOG assume that u is marked before v. Consider the time u is marked during RecursiveDFS(u). In RecursiveDFS(u), the algorithm examines every arc incident to u. This leads to two cases:
Case 1: v is unmarked when RecursiveDFS(u) examines (u, v). Then RecursiveDFS(u) sets $p(v) \leftarrow u$. Therefore (u, v) is a tree edge.
Case 2: v is marked when the algorithm examines (u, v). But v must have been marked after u, so it was marked during RecursiveDFS(u). Thus we have a series of vertices $\{u = w_0, w_1, \ldots, w_{\ell-1}, w_\ell = v\}$ where $p(w_k) = w_{k-1}$. Therefore u is an ancestor of v in T and is thus a back edge. □

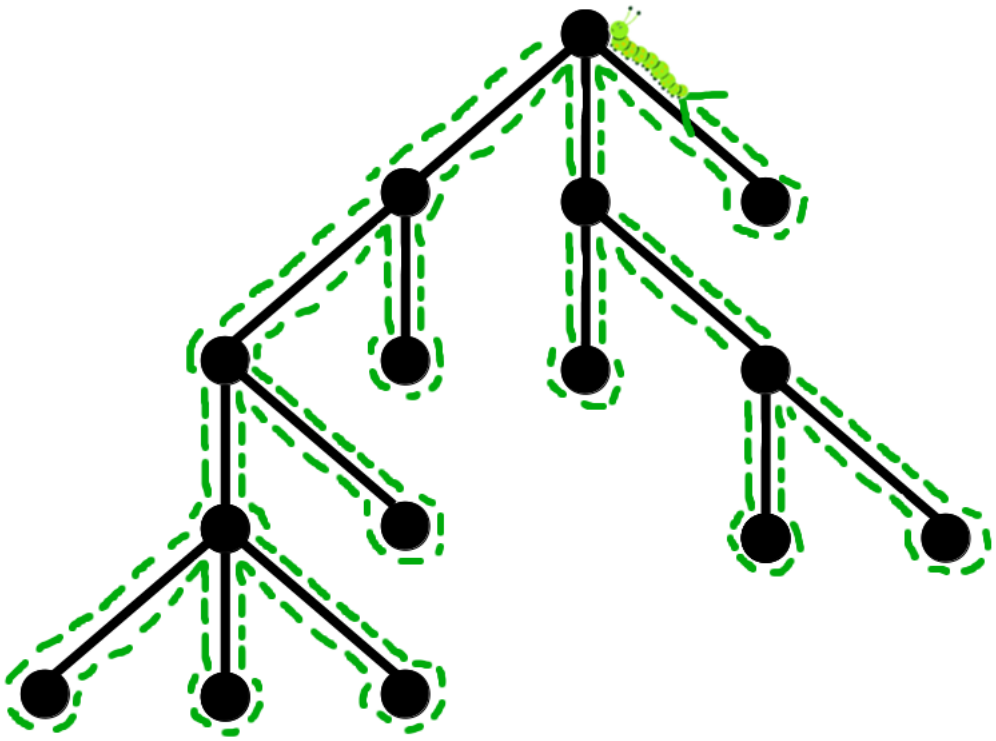
Corollary: Let T be a DFS tree in an undirected graph G. Then every non-tree edge is a back edge.

3.3.2 Previsit & Postvisit

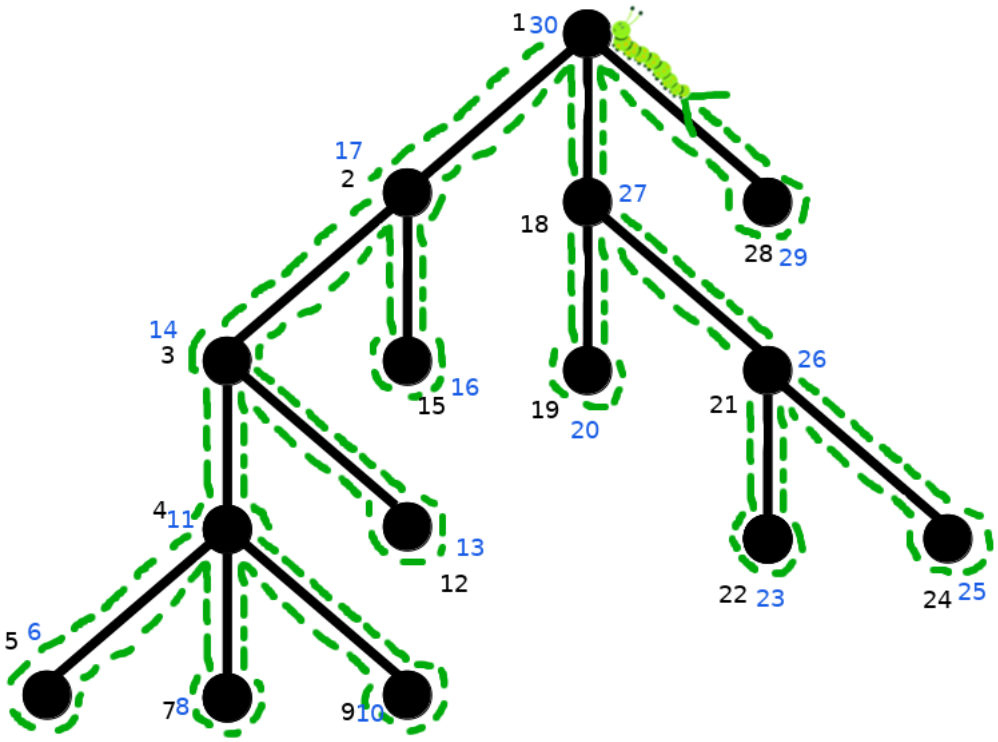
We can add a clock to the RecursiveDFS(r) algorithm to record when we visit a vertex for the first time (previsit) and when we visit a vertex for the last time (postvisit). Let's update our recursive algorithm to include this.

```
clock ← 0
RecursiveDFS(r)
    mark r
    pre(r) ← clock
    clock++
    for each edge (r, v)
        if v is unmarked
            set p(v) ← r
            RecursiveDFS(v)
    post(r) ← clock
    clock++
```

To visualize this imagine that a caterpillar is crawling along the graph like so:



The clock records the first time the caterpillar visits a node and the last time the caterpillar visits a node. Then, let the black number be the clock on the previsit, and let the blue number be the clock on the postvisit. We get this



Note that pre(v) is the time that the caterpillar arrives at the subtree rooted at v, and post(v) is the time when the caterpillar leaves that subtree.

Depth-first search expores one subtree of the tree at-a-time. This means that we can represent each subtree as an interval over $\mathbb{R}$. More strongly, the set of intervals is laminar (for every pair of intervals, I_1, I_2, either I_1, and I_2 are disjoint, or one is a subset of the other.)

3.3.3 Postvisit Times

For an arc (u, v) the following always hold:

1. Tree arcs: post(v) < post(u)
2. Forward arcs: post(v) < post(u)
3. Backward arcs: post(u) < post(v)
4. Cross arcs: post(v) < post(u)

Note that only for backward arcs is post(u) < post(v). This is a useful result.

3.3.4 Directed Acyclic Graphs

Theorem: A directed graph G is acyclic if and only if depth-first search produces no backward arcs.

Proof: ($\Rightarrow$) Let DFS give a backward arc (u, v). By definition u is a descendant of v in the DFS tree T. Therefore there exists a path P, given by $P = \{v = v_0, v_1, \ldots, v_k = u\} \in T$. But then $P \cup (u, v)$ is a directed cycle in T, a contradiction.
($\Leftarrow$) Assume DFS gives no backward arcs. Suppose there is a directed cycle $C = \{v_0, v_1, \ldots, v_k, v_0\} \in T$. Then, as there are no backward arcs we have that $\text{post}(v_0) > \text{post}(v_1) > \text{post}(v_2) > \ldots > \text{post}(v_k) > \text{post}(v_0)$, a contradiction. □

Corollary: We have an $O(m)$ algorithm to test whether the graph is acyclic.

Proof: Run DFS and check for backward arcs.

3.3.5 Topological Orderings

A directed graph has a "Topological Ordering" if the vertices can be ordered such that every arc is from right to left.

Theorem: A directed graph G has a topological ordering if and only if DFS produces no backward arcs.

Proof: ($\Rightarrow$) If DFS produces a backward arc then G contains a cycle C. Let the cycle $C = \{v_0, v_1, \ldots, v_k, v_0\}$, where WLOG v_0 is the leftmost vertex of the cycle in the order. But then the arc (v_0, v_1) goes from left to right, a contradiction.
($\Leftarrow$) Assume DFS gives no backward arcs. Then for every arc (u, v), we have that $\text{post}(u) > \text{post}(v)$. Thus, if we place each vertex v at coordinate $\text{post}(v)$ on the x-axis, then every arc goes from right to left and hence G has a topological ordering. □

Corollary: We have an $O(m)$ algorithm to test whether or not we have a topological ordering.

Corollary: A directed graph is acyclic if and only if it has a topological ordering.

3.3.6 Disconnected Graphs & Strongly Connected Components

For disconnected undirected graphs, a depth-first search from a vertex r will find the entire component containing the vertex r. Therefore, if we want to discover the entire graph, not just the component containing r, we must run DFS from another vertex (in a different component).

In a directed graph, a subgraph S is said to be strongly connected if for every vertex pair $u, v \in S$, there exists a directed path from v to u and vice versa. DFS can be used to decompose a directed graph into its strongly connected components very quickly.

Theorem: There is an $O(m)$ algorithm to find stromngly connected components of directed graphs.

4 Greedy Algorithms

Greedy algorithms are a class of algorithms characterized by how they make locally optimal choices at each step. They have a few properties:

- They're fast!
- They're usually very simple!
- They're easy to code!
- They rarely work!

Thus for this topic in the course the goal is to understand some basic (working) greedy algorithms along with some basic techniques and when such techniques can succesfully be applied.

4.1 The Task Scheduling Problem

Suppose a firm receives a set J of n job orders from some customers, but the firm can only process one order at once. Suppose the job of customer i will take t_i units of time to complete. Assume each customer wants their job done as quickly as possible (and will pay accordingly.)

Assume the firm processes the jobs in order $\{1, 2, \ldots, n\}$. Then the wating time of customer ℓ will be

$$w_\ell = \sum_{i=1}^{\ell} t_i$$

Under the assumption that the customers will pay according to time, the firm wants to minimize the sum of the waiting times. That is, they want to minimize

$$\sum_{\ell=1}^{n} w_\ell = \sum_{\ell=1}^{n} \sum_{i=1}^{\ell} t_i$$

We'll use a greedy approach to solve this problem:

Schedule(J)
    sort the list of orders in non-decreasing order of their completion time
    schedule the jobs in this order

Unfortunately, unlike divide and conquer algorithms that generally work, and where most of the proofs follow directly from strong induction (requiring only the base cases be verified), greedy algorithms don't work in general and thus have to be proved in more creative ways. Let's now prove the task scheduling algorithm works.

Proof: Let the algorithm schedule the tasks in the order $\{1, 2, \ldots, n\}$. Assume there is a better schedule $\mathcal{S}$. Then there must be a pair of jobs i, j such that

1. Job i is (WLOG) scheduled immediately before job j by $\mathcal{S}$.
2. Job i is longer than job j. In other words, $t_i > t_j$.

This implies that exchanging jobs i, and j leads to a more optimal schedule $\hat{\mathcal{S}}$. Observe that after exchanging the two jobs the waiting time for all the other jobs remains the same (draw a picture if you need to convince yourself.) That is,

$$\hat{w}_k = w_k, \forall k \neq i, j$$

but obviously

$$\hat{w}_i + \hat{w}_j < w_i + w_j$$

a contradiction. Thus $\mathcal{S}$ was the optimal solution. □

The runtime of this algorithm is $O(n \log n)$ as the only real step is sorting the list of jobs by length.

4.2 The Interval Selection Problem (Class Scheduling)

Suppose there is one classroom and as set $I = \{1, 2, \ldots, n\}$ of n classes that want to use the room. Suppose each class $i \in I$ has a start time s_i, and a finish time f_i. Our goal is to book as many classes into that room as possible (in a certain period of time) but no more than one class can use the room at once. Let's try using a greedy algorithm to solve this problem. There's a couple different possibilities we can try:

1. First start: Choose the class that starts first and iterate over the remaining non-conflicting classes
2. Shortest duration: Select the class that is the shortest and then iterate over the remaining non-conflicting classes
3. Minimum Conflict: Find the course that conflicts with the fewest other courses and iterate over the remaining non-conflicting classes
4. First finish: Choose the class that finishes first and iterate over the remaining non-conflicting courses.

But do any of these actually work? Let's see.

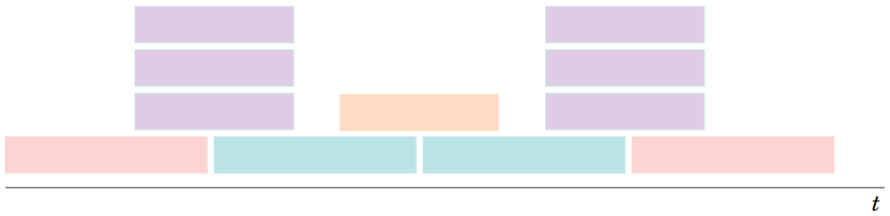
Does first start work? Nope. Consider the example:

Note that that algorithm would select the blue interval where the eight orange intervals would be the optimal solution.

How about shortest duration? Nope. Consider the following:

This algorithm would select the blue interval where the two orange intervals would be the optimal solution.

What about minimum conflict? Nope.



This algorithm would select the orange interval because it only conflicts with the two blue intervals, and then two of the purple intervals, but the blue intervals plus the red intervals along the bottom is the optimal solution even though the red intervals each conflict with the three purple intervals, and the blue intervals each conflict with four intervals.

First finish? Yep this one works. First we'll write down the pseudo-code algorithm and then we'll prove that it works.

```
FinishFirst(I)
    sort the classes by their finish time in non-decreasing order
    initialize the set S := ∅ to be the set of scheduled classes
    while I ≠ ∅
        let i be the lowest index class in I
        set S ← S ∪ {i} and set I ← I \ {i}
        for each class j ∈ I
            if s_i < f_i
                set I ← I \ {i}
    return S
```

Before we prove the algorithm we need some help from a lemma

Lemma: Let c be the class with the earliest finish time. Then there is an optimal solution that contains c.

Proof: Recall the classes are indexed such that $f_1 \leq f_2 \leq \cdots \leq f_n$. Then take an optimal solution $\mathcal{S}$ and assume that $c \in \mathcal{S}$. Let i be the lowest index in $\mathcal{S}$. We claim that $\mathcal{S} \setminus \{i\} \cup \{c\}$ is a feasible allocation of maximum size. This follows as $f_i \leq s_j$ for any $j \in \mathcal{S} \setminus \{i\}$. But $f_1 \leq f_i \leq s_j$ so c doesn't conflict with any class in $\mathcal{S} \setminus \{i\}$. Therefore $\mathcal{S} \setminus \{i\} \cup \{c\}$ is feasible with cardinality equal to $|\mathcal{S}|$. □

We can now prove that the algorithm works.

Proof: We'll proceed by induction on the cardinality of the optimal solution.
Base Case: Let $|\text{opt}(I)| = 1$. Then the algorithm outputs c and this is trivially an optimal solution.
Induction Hypothesis: If $|\text{opt}(I)| = k$ then the algorithm outputs an optimal solution.
Inductive Step: Let $|\text{opt}(I)| = k + 1$. Then the algorithm outputs $\{c\} \cup$ FirstFinish$(I \setminus k)$. But by the previous lemma there exists an optimal solution $\hat{\mathcal{S}}$ containing c. Thus $\hat{\mathcal{S}} \setminus \{I\}$ is an optimal solution for the subproblem $I \setminus X \Rightarrow |\text{opt}(I)| = k$. But then by our induction hypothesis, the algorithm gives an optimal solution to the sub-problem $I \setminus X \Rightarrow |\text{FirstFinish}(I \setminus X)| = k \Rightarrow |\{c\} \cup \text{FirstFinish}(I \setminus X)| = k + 1$. Thus the algorithm outputs an optimal solution. □

There are at most n iterations, and it takes $O(n)$ time to find a class that finishes first. Therefore the runtime is $O(n^2)$. With better implementation, we can get the runtime down to $O(n \log n)$.

4.3 Dijkstra's Shortest Path Alorithm

Given a directed graph $G = (V, A)$ with a source vertex $s \in V$, let each arc arc $a \in A$ have a non-negative length ℓ_a. Then the length of a path $P \subseteq G$ is

$$\ell(P) = \sum_{a \in P} \ell_a$$

Our goal is to find the shortest path from s to every other vertex $v \in V$. We can do this using *Dijkstra's Shortest Path Algorithm.*

Dijkstra(s, G)
    set $\mathcal{S} := \emptyset$
    set $d(s) := 0$, and set $d(v) := \infty$, $\forall v \in V \setminus \{s\}$
    while $\mathcal{S} \neq V$
        if $v = \text{argmin}_{u \in V \setminus \mathcal{S}}$
            set $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
        for each arc (v, w)
            if $d(w) > d(v) + \ell_{vw}$ and $w \in V \setminus S$
                set $d(w) = d(v) + \ell_{vw}$

This is indeed a greedy algorithm because it myopically selects the vertex with the smallest label. Note this algorithm only calculates the shortest path distances, but it's easy to tweak it to keep track of the actual paths themselves:

Dijkstra(s, G)
    set $\mathcal{S} := \emptyset$
    set $\mathcal{T} := \emptyset$
    set pred$(s) \leftarrow$ NULL, pred$(v) \leftarrow$ --
    set $d(s) := 0$, and set $d(v) := \infty$, $\forall v \in V \setminus \{s\}$
    while $\mathcal{S} \neq V$
        if $v = \text{argmin}_{u \in V \setminus \mathcal{S}}$
            set $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
            set $\mathcal{T} \leftarrow \mathcal{T} \cup (\text{pred}(u, v))$
        for each arc (v, w)
            if $d(w) > d(v) + \ell_{vw}$ and $w \in V \setminus S$
                set $d(w) = d(v) + \ell_{vw}$
                set pred$(w) \leftarrow v$

Note that this is just a generalization of Breadth-First search!

To show that the algorithm works first observe that $\mathcal{S}$ is a set of vertices and $\mathcal{T}$ is a set of arcs. Then let $\mathcal{S}^k$ be the set of vertices in $\mathcal{S}$ at the end of the

k^{th} iteration, and let $\mathcal{T}^k$ be the set of arcs in $\mathcal{T}$ at the end of the k^{th} iteration. Furthermore observe that the arcs in $\mathcal{T}^k$ are between the nodes in $\mathcal{S}^k$. Therefore $\mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$ is just a directed graph. Note that $\mathcal{G} = \mathcal{G}^n$ is the final outcome of the algorithm. Next consider the following lemma.

Lemma: The graph $\mathcal{G}^k$ is a directed tree rooted at s.

Proof: We will proceed by induction on k.
Base Case: $k = 1$. Then $\mathcal{S}^1$ contains only s, and $\mathcal{T}^1 = \emptyset$ as pred(s) = NULL. Therefore it's trivially an arborescence.
Induction hypothesis: $\mathcal{G}^{k-1} = (\mathcal{S}^{k-1}, \mathcal{T}^{k-1})$ is an arborescence rooted at s.
Inductive Step: Consider $\mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$. Let v_k be the vertex added to $\mathcal{S}$ in the k^{th} iteration. Then $\mathcal{S}^k = \mathcal{S}^{k-1} \cup \{v_k\} \Rightarrow \mathcal{T} = \mathcal{T}^{k-1} \cup (\text{pred}^{k-1}(v_k), v_k)$. Therefore v_k has in-degree exactly 1 in G^k, and out-degree exactly 0. Then by our induction hypothesis $\mathcal{G}^{k-1}$ is an arborescence rooted at s. Furthermore by definition, $\text{pred}^{k-1}(v_k) \in \mathcal{S}^{k-1} \Rightarrow \mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$ is an arborescence rooted at s with v_k as a leaf. □

Now we just need to show that the paths in the outputted arborescence are the shortest paths in the graph and we'll have proved the algorithm.

Theorem: The graph $\mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$ gives the shortest distances from s to every vertex in $\mathcal{S}^k$.

Proof: Again, we'll proceed by induction on k.
Base Case: $k = 1$. Then this is trivially true as $d^1(s) = 0 = d^*(s)$.
Induction Hypothesis: Assume $\mathcal{G}^{k-1} = (\mathcal{S}^{k-1}, \mathcal{T}^{k-1})$ gives the shortest distance from s to every vertex in $\mathcal{S}^{k-1}$. i.e. $d^{k-1}(v) = d^*(v)$, $\forall v \in \mathcal{S}^{k-1}$.
Inductive Step: Consider the graph $\mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$. Let v_k be the vertex added to $\mathcal{S}$ in the k^{th} iteration. Take the sortest path P from s to v_k with as many arcs in common with $\mathcal{G}^k$ as possible. Let x be the last vertex of $\mathcal{G}^{k-1}$ in P, and let $y \notin \mathcal{S}^{k-1}$ be the vertex after x in P. Such a y exists or we are done. Since y is on the shortest path from s to v_k and the arc lengths are non-negative, we have that $d^*(y) \leq d^*(v_k) < d^k(v_k)$. We have a strict inequality or we'd be done. But since $x \in \mathcal{S}^{k-1}$ we have that $d^k(y) \leq d^{k-1}(x) + \ell(x, y) = d^*(x) + \ell(x, y) = d^*(y)$. And therefore $d^k(y) \leq d^*(y) < d^k(v_k)$, which contradicts our choice of v_k as $d^k(y) < d^k(v_k)$. □

There are n iterations and there are at most n distance updates in each iteration. Therefore the runtime is $O(n^2)$. We can implement this algorithm faster, in $O(m \log n)$ time, using a heap.

4.4 The Binary Encoding Problem

Suppose we want to encode an alphabet $\mathcal{A}$ in binary. Let's suppose for the sake of simplicity that we're using the english alphabet. That alphabet has 26 letters so we need at most 5 bits ($2^5 = 32 > 26$) to encde the alphabet as $a = 00000, b = 00001, \ldots, z = 11010$. To measure whether this is a good encoding, we need to come up with some definition of the cost of an encoding. The most intuitive way to do this is to sum up how long the final encoding is (in bits). Note that some letters are more frequent than others. Let f_i be the frequency with which each letter i appears in the alphabet. Then the cost of the encoding is given by

$$\text{cost} = \sum_{i \in \mathcal{A}} \ell_i f_i$$

where ℓ_i is the length (number of bits of a letter's encoding). Our goal then is to minimize this cost. Note that we can reduce the cost by giving shorter encodings to higher-frequency letters such as e, or t, and longer encodings to lower-frequency letters, such as z, or q. Morse code is an encoding that does exactly this! Consider the table

a	01	b	1000	c	1010	d	100
e	0	f	0010	g	110	h	0000
i	00	j	0111	k	101	l	0100
m	11	n	10	o	111	p	0110
q	1101	r	010	s	000	t	1
u	001	v	0001	w	011	x	1001
y	1011	z	1100				

Note that the two most frequent letters, e, and t, have only one bit, where unfrequent letters such as j, and z all have 4. Note also that since we're giving some numbers fewer bits, we can get away with using only 4 bits for our least frequent letters instead of 5.

The problem with this, however, is that Morse code isn't actually a binary code. Consider the following

$$\begin{array}{ll} 1101 \equiv \text{q} & 1101 \equiv \text{ttet} \\ 1101 \equiv \text{ma} & 1101 \equiv \text{gt} \\ 1101 \equiv \text{tk} & 1101 \equiv \text{met} \end{array}$$

Indeed in real life Morse code is a ternary code where the third character is a rest after each letter.

4.4.1 Prefix Codes

A coding is said to be "prefix-free" if no codeword is a prefix of another. Note that due to the above example, Morse code is not prefix-free.

4.4.2 Binary Tree Representation

We can use binary trees to get prefix-free codes. Suppose we have a binary tree T. Let each left edge have the label 0, and each right edge have the label 1. Suppose each leaf node is associated with a letter. Then the codeword for a letter are the lables on the edges in the path from the root to the leaf associated with the letter.

Theorem: A binary coding is prefix-free if and only if it has a binary tree representation.

Proof: ($\Leftarrow$) In a binary tree representation the letters are leaves. This means that there exists a path P_x from the root to the leaf x, and a path P_y from the root to the leaf y for distinct letters x, and y. SInce $x \neq y$, the paths P_x, and P_y must diverge at some point. Thus the codeword for x can't be a prefix of the codeword for y.
($\Rightarrow$) Given a binary coding system we can recursively define a binary tree representation where a letter whose codeword starts with 0 is placed on the left subtree, and where a letter whose codeword is 1 is placed on the right subtree. □

Now given a tree T, what is the cost of that tree? Observe that the cost of any given letter is just the length of the path from the root to the letter in the tree because it corresponds directly with the length of the encoding. That is,

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

where d_i is the depth in the tree of the i^{th} letter. We can prove this a little more formally.

Proof: We start with the definition of the cost.

$$\text{cost}(T) = \sum_{i \in \mathcal{A}} f_i \ell_i(T)$$

But the length of an encoding for a letter is just the number of edges in the path from the root to the leaf associated with that letter, so

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

but that second sum is just the depth

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

□

From this observation, we see that given a tree T with a specified shape, it's easy to determine the best way to assign the letters to the leaves; all we do is sort them by frequency and put the lowest-frequency letters near the bottom depth. Now all we have to do is find the best possible shape for a tree and we'll be done. This requires a bit of a trick. Let

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

be the number of letters (weighted by frequency) whose root-leaf paths use edge e in T. Then

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

This needs to be be proved.

Proof: From the previous observation we have

$$\begin{aligned}
\text{cost}(T) &= \sum_{i \in \mathcal{A}} f_i d_i(T) \\
&= \sum_{i \in \mathcal{A}} f_i \cdot \sum_{e: e \in P_i} 1 \\
&= \sum_{e \in T} 1 \cdot \sum_{i \in \mathcal{A}: e \in P_i} f_i \\
&= \sum_{e \in T} \sum_{i \in \mathcal{A}} f_i \\
&= \sum_{e \in T} n_e
\end{aligned}$$

□

4.4.3 The Key Formula

Let $\hat{T}$ be the tree formed by removing a pair of sibling-leaves a, and b, and by labeling their parent by z where $f_z = f_a + f_b$. Then

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

So now we knoww how to recursivelly find the optimal shape of the tree. Putting this all together gives us the Huffman Encoding Algorithm.

4.4.4 The Huffman Coding Algorithm

Suppose we're given an alphabet $\mathcal{A}$ and a frequency list $\mathbf{f}$. Then we have the following algorithm to give an efficient binary encoding.

```
Huffman(𝒜, f)
    if |𝒜| = 2
        encode one letter with 0 and the other with 1
    else
        let a, b ∈ 𝒜 be the most infrequent letters
        set Â ← (𝒜 \ {a, b}) ∪ {z}
        set f_z = f_a + f_b
        let T̂ = Huffman(Â, f)
        create T by adding a, b as children of the leaf z in T̂
```

Now we must prove that this algorithm gives a minimum cost encoding.

Proof: Proceed by induction on $|\mathcal{A}|$
Base Case: $|\mathcal{A}| = 2$. Both letters are given codewords of length 1 so this can't be improved.
Induction Hypothesis: Assume Huffman coding works if $|\mathcal{A}| = k$.
Inductive Step: Assume $|\mathcal{A}| = k + 1$. Let a, and b be the least frequent letters. Then a, and b are siblings for any $\hat{T}$, and $\text{cost}(T) = \text{cost}(\hat{T}) + f_a + f_b$. The best choice for $\hat{T}$ is the optimal solution for $\hat{\mathcal{A}}$. By by the induction hypothesis, this is exactly the choice made recursively by the algorithm. □

There are $n - 2$ iterations of the algorithm. Each iteration takes $O(n)$ time to find the two least-frequent letters and update the alphabet. Therefore the running time is $O(n^2)$. This can be improved to $O(n \log n)$ by implementing it with a heap.

4.5 Minimum Spanning Tree Problem

Suppose we're given a graph $G = (V, E)$ where each edge $e \in E$ has an associated non-negative cost c_e. For convenience assume that all edges are distinct. Then the cost of a tree $T \subseteq E(G)$ is

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

Our goal is to find a spanning tree T that minimizes the cost. We'll see three algorithms to do this. All three of the following algorithms work, but we'll prove them later.

4.5.1 Kruskal's Algorithm

Kruskal(G)
    sort the edges $\{e_1, e_2, \ldots, e_m\}$ by cost in increasing order
    set $T := \emptyset$
    for each $i \in \{1, 2, \ldots, m\}$
        let $e_i = (u, v)$
        if u, v are in different components
            set $T \leftarrow T \cup \{e_i\}$

Sorting the edges takes time $O(m \log m)$. We then iterate m times where each iteration takes $O(n)$ time. Therefore the runtime is $O(mn + m \log m) = O(mn)$. This runtime can be improved to $O(m \log n + m \log^* n)$ by using the data structure for disjoint sets, where $\log^* n$ is the number of nested logs that are required to get a number less than or equal to 1.

4.5.2 Prim's Algorithm

Prim(G)
    let $a \in V$ be some vertex
    set $T := \{a\}$
    while $V(T) \neq V$
        let e be the minimum cost edge in $\delta(T)$
        set $T \leftarrow T \cup \{e\}$

Prim's algorithm takes n iterations. Finding the minimum cost edge during each iteration takes $O(m)$. Therefore the runtime is $O(mn)$. This can be improved to $O(m + n \log n)$ by using Fibonacci heaps.

4.5.3 Borůvka's Algorithm

Borůvka(G)
    set $T := \emptyset$
    while T has more than one component $\{S_1, S_2, \ldots, S_\ell\}$
        for each $i \in \{1, 2, \ldots, \ell\}$
            let ℓ_i be the minimum cost edge in $\delta(T)$
            set $T \leftarrow T \cup e_1 \cup e_2 \cup \cdots \cup e_\ell$

We have at most n components. Finding the minimum cut $\delta(S_i)$ takes at most $O(m)$. We have $\log n$ iterations. Therefore the runtime is $O(mn \log n)$. This can be improved to $O(m \log^* n)$.

4.5.4 The Cut Properties of Minimum Spanning Trees

Theorem: (Cut Property) Assume the edge costs are distinct. If e is the cheapest edge in some cut $\delta(S)$ then e is in the MST.

Proof: Let $e = (u, v)$ be the cheapest edge in a cut $\delta(S)$. Let T^* be a minimum spanning tree and assume $e \notin T^*$. Since T^* is a spanning tree, there exists a unique path $P \subseteq T$ from u to v. Note that if $\hat{e} \in P$ then $(T^* \setminus \hat{e}) \cup e$ is still a spanning tree. Also note that in any walk from u to v we must cross the cut, so there exists at least one edge $\hat{e} \in P \cap \delta(S)$. But $c_{\hat{e}} > c_e \Rightarrow (T^* \setminus \hat{e}) \cup e$ is a cheaper spanning tree than T^*, a contradiction. □

Note that this property trivially proves that the above three algorithms are correct as they all look for the minimum cost edge in a cut, and the theorem implies that that minimum cost edge must be in the MST.

4.5.5 The Cycle Property of Minimum Spanning Trees

Theorem: (Cycle Property) Assume all edge costs of a graph are distinct. If e is the most expensive edge in some cycle C then e is not in the minimum spanning tree.

Proof: Let $e = (u, v)$ be the most expensive edge in the cycle C. Since C is a cycle, $P = C \setminus \{e\}$ is a path from u to v. Assume for a contradiction that e is in the minimum spanning tree T^*. Let $\delta(S) = (S, V \setminus S)$ be the cut induced by $T^* \setminus \{e\}$. Note that if $\hat{e} \in \delta(S)$ then $(T^* \setminus \{e\}) \cup \{\hat{e}\}$ is a spanning tree. Also note that there exists at least one edge $\hat{e} \in P \cap \delta(S)$. Therefore, as e is the most expensive edge in C we have $c_{\hat{e}} < c_e$, however this implies that $(T^* \setminus \{e\}) \cup \{\hat{e}\}$ is a cheaper spanning tree than T^*, a contradiction. □

4.5.6 The Reverse Delete Algorithm

RevDel(G)
    sort the edges $\{e_1, e_2, \ldots, e_m\}$ in increasing order by cost
    for each $i \in \{m, m-1, \ldots, 2, 1\}$
        if $G \setminus \{e_i\}$ is connected
            set $G \leftarrow G \setminus \{e_i\}$

This algorithm has m iterations. Checking whether the graph is connected can be done in $O(m)$ time using any graph search algorithm. Thus the running time is $O(m^2)$.

The fact that this algorithm works follows directly from the cycle property.

4.6 Clustering

Given a collection $\mathcal{O}$ of objects, we want to partition it into a set of clusters $\{S_1, S_2, \ldots, S_k\}$. The basic objective of a good clustering is that similar objects are placed into the same cluster, and dissimilar objects are placed into different clusters. We can represent the clustering problem by a weighted graph $G = (V, E)$ such that there is a vertex for each object in $\mathcal{O}$, there is an edge between every pair of objects, and where the weight (or length) associated with the edge represents the dissimilarity between the two objects.

4.6.1 The Quality of a Clustering

Unfortunately, we can only give an informal definition of the quality of a clustering as it varies greatly depending on the application. In general though, with a clustering there will be an objective function that determines the quality.

4.6.2 The Maximum Space Clustering Problem

In the maximum space clustering problem we want to parition the set of objects into k clusters (subsets), $\{S_1, S_2, \ldots, S_k\}$, so that the minimum distance between any two of the k subsets is maximized. We define the distance between two clusters as

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

Thus the objective function for the maximum space clustering problem is given by

$$\max_{\mathcal{S}} \min_{\ell,m} d(S_\ell, S_m) = \max_{\mathcal{S}} \min_{\ell,m} \min_{i \in S_\ell, j \in S_m}$$

It turns out that if we tweak the reverse-delete minimum spanning tree algorithm, we get an efficient algorithm to solve this clustering problem. The idea is that instead of disallowing an edge to be removed when the graph becomes disconnected, we remove edges until there are at most k components. The pseudo-code for this algorithm then looks like this:

RevDelCluster(G)
    sort the edges $\{e_1, e_2, \ldots, e_m\}$ in increasing order by cost
    for each $i \in \{m, m-1, \ldots, 2, 1\}$
        if $G \setminus \{e_i\}$ has k components or less
            set $G \leftarrow G \setminus \{e_i\}$
    return the k components of G

Just like the reverse-delete algorithm for the minimum spanning tree problem, the time complexity for this algorithm is $O(m^2)$

To prove that this algorithm is correct, we need some help from a couple of

lemmas.

Lemma: A connected graph contains a spanning tree as a subgraph.

Proof: This is a direct consequence of running breadth-first search. If a graph is connected then BFS discovers all vertices. □

Lemma: If we remove an edge $e = (u, v)$ from a graph then the number of components increases by at most one.

Proof: As there exists an edge $e = (u, v)$, the vertices u, and v are in the same component, say S_1. Since S_1 is a component, and therefore connected by definition, then by the previous lemma it contains a spanning tree T as a subgraph. This leads to two cases:
Case 1: $e \notin T$. Then after removing e, S_1 remains a component as it still contains the spanning tree T.
Case 2: $e \in T$ for every spanning tree in S_1. Then since T can't contain a cycle, removing e partitions T into exactly two components. Thus removing e creates either zero additional components or one additional component. □

Now that we have all the tools we need to prove the correctness of the algorithm, let's do it.

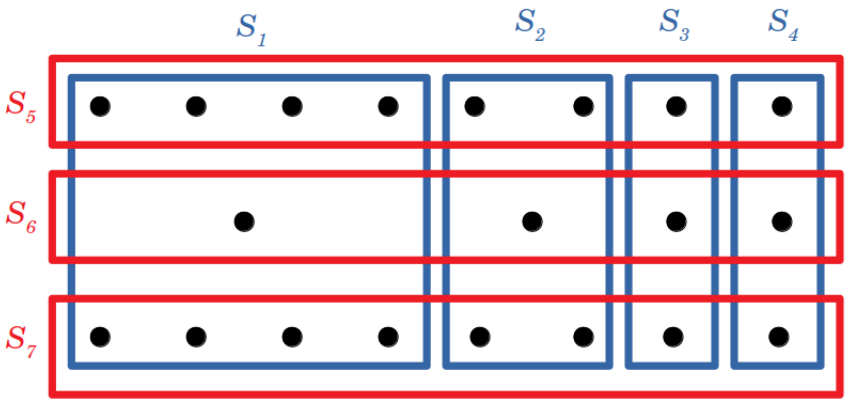
Proof: Let e_ℓ be the edge whose deletion causes the number of components to increase from $k - 1$ to k. When we delete e_ℓ we obtain the clustering $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$. But this means that only the edges $\{e_m, e_{m-1}, \ldots, e_\ell\}$ can cross between two different clusters in $\mathcal{S}$. This implies that e_ℓ is the shortest edge between two clusters in $\mathcal{S}$. Now we need to show that this is the best solution. Any cluster $\mathcal{S}^* = \{S_1^*, S_2^*, \ldots, S_k^*\}$ must separate the endpoints of at least one edge in $\{e_1, e_2, \ldots, e_\ell\}$. But then the quality of $\mathcal{S}^*$ is at most d_{e_ℓ} as all the other edges are cheaper. Thus the reverse delete clustering algorithm gives an optimal solution to the maximum space clustering problem. □

4.7 Set Cover Problem

Suppose we're given a groundset $I = \{i_1, i_2, \ldots, i_n\}$ of items, and a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ such that $S_i \subseteq I$, $\forall i \leq n$. We then want to select as few sets from $\mathcal{S}$ as possible so that every element in I is contained in at least one set. In other words we want to find the smallest cardinality subset of $\mathcal{S}$ that covers the groundset. Here's a simple algorithm to do this:

SetCover($\mathcal{S}$, I)
while $I \neq \emptyset$
let $\hat{S} = \underset{S \in \mathcal{S}}{\operatorname{argmin}}(|S \cap I|)$
set $S \leftarrow S \setminus \{\hat{\mathcal{S}}\}$ set $I \leftarrow I \setminus \hat{\mathcal{S}}$

Does this algorithm work? Not really. Consider the example:



The algorithm will select the blue sets $\{S_1, S_2, S_3, S_4\}$, even though the red sets $\{S_5, S_6, S_7\}$ are the optimal solution.

4.7.1 Approximation Algorithms

An algorithm $\mathcal{A}$ is an *approximation algorithm* if for any instance I, the following conditions hold.

- The algorithm runs in polynomial time
- The algorithm returns a feasible solution $\mathcal{S}$
- It gives the approximation guarantee:

$$\text{cost}(\mathcal{S}) \leq \alpha \cdot \text{opt}$$

i.e. the cost of the solution is within a factor α of the optimal solution.

The above set cover problem is an approximation algorithm . We want to now figure out what it's value of α is.

Lemma: If the optimal set cover has cardinality k, then for any $X \subseteq I$ there exists an $\mathcal{S}$ that covers at least $\frac{1}{k}|X|$ items of X.

Proof: Let the optimal solution be $\{S_1^*, S_2^*, \ldots, S_K^*\}$. Then that set covers every item in $I \Rightarrow$ it covers every item in X. As X is covered by k sets, there must be some set that covers at least $\frac{1}{k}$ of X. $\square$

Note that this lemma proves that the above algorithm is approximately correct.

Theorem: If the optimal set cover has cardinality k, then the greedy algorithm finds a solution of size at most $k \cdot \ln n$.

Proof: WLOG let the algorithm output $\{S_1, S_2 \ldots, S_T\}$ and let the optimal solution be $\{S_1^*, S_2^*, \ldots, S_k^*\}$. We want to show that $T \leq k \cdot \ln n$. Let I_t be the set of uncovered items at the start of step T. Then $I = I_1$. Since $I_t \subseteq I$, by the above lemma there is a set that covers at least $\frac{1}{k}|I_t|$ items in I_t. Therefore we have

$$\begin{aligned}
|I_{t+1}| &\leq |I_t| - \frac{1}{k}|I_t| \\
&= \Big(1 - \frac{1}{k}\Big)|I_t| \\
&= \Big(1 - \frac{1}{k}\Big)\Big(1 - \frac{1}{k}\Big)|I_{t-1}|
\end{aligned}$$

doing this t times, we get

$$\begin{aligned}
&= \Big(1 - \frac{1}{k}\Big)^t |I_t| \\
&= \Big(1 - \frac{1}{k}\Big)^t n
\end{aligned}$$

and thus we have an upper bound on $|I_{t+1}|$. But note that as $1-x < e^{-x}, \forall x \neq 0$,

$$< (e^{-1/k})^t n$$

setting $t = k \cdot \ln n$ gives us

$$\begin{aligned}
&= e^{\frac{-k \cdot \ln n}{k}} n \\
&= e^{-\ln n} \\
&= 1
\end{aligned}$$

and thus $|I_{k \cdot \ln n + 1}| < 1$. But note that since we're dealing with integers, then if the cardinality of $I_{k \cdot \ln n + 1}$ is less than one then it's zero, which is when the algorithm terminates. Thus the algorithm will terminate on or before step $k \cdot \ln n$. $\square$

Corollary: The above greedy algorithm is a $\log n$ approximation for the set

cover problem.

This doesn't really seem to be a great approximation guarantee as $\log n$ is unbounded, however the following theorem say that it's probably they best we'll get.

Theorem: There is no approximation algorithm for the set cover problem with guarantee $(1 - o(1)) \cdot \log n$, unless P = NP.

The proof for this theorem is way beyond the scope of this course.

This algorithm has at most n iterations with at most n coverage updates at each step. Thus the running time is $O(n^2)$.

4.8 Matroids

4.8.1 The Generic Greedy Problem

Suppose we're given a set E of elements each with a non-negative weight w_e, and a collection $\mathcal{F}$ of feasible sets where each $F \in \mathcal{F}$ is a valid solution to some problem. The weight of a set F is defined by

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

Our problem is to find a feasible set with the maximum possible weight.

4.8.2 The Hereditary Property

A collection of feasible sets is said to satisfy the hereditary property if

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

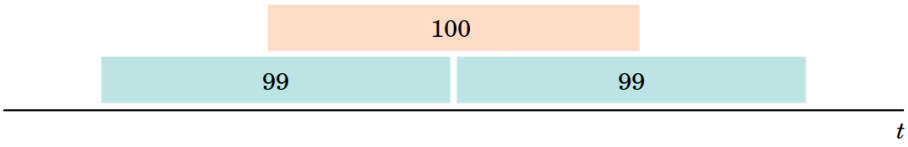
4.8.3 The Greediest Algorithm

A generic algorithm to solve the generic greedy problem for sine hereditary system is:

Greediest(E)
    sort the elements by weight
    set $T := \emptyset$
    for each $i \in \{1, 2, \ldots, m\}$
        if $T \cup e_i \in \mathcal{F}$
            set $T \leftarrow T \cup e_i$

This involves sorting elements which takes $O(m \log m)$ time. We have m iterations where we can test for feasibility in time t. Therefore, the running time for the greediest algorithm is $O(m \log m + mt)$. Thus this algorithm is efficient as long as t is polynomial.

So does this algorithm work? Well it works for the minimum spanning tree problem as it's exactly Kruskal's algorithm. Note however that it doesn't work with the weighted interval selection problem. Consider the example:



The greediest algorithm would choose the one interval with weight 100 when the two intervals with weight 99 are the optimal solution.

A natural question then is when does this algorithm work? What is it about the structure of the MST problem but not the weighted interval selection problem that makes the greediest algorithm work? Spoilers it's a matroid. First though we need to look at something called the augmentation property.

4.8.4 The Augmentation Property

$\mathcal{F}$ is said to satisfy the augmentation property if when

$$F, \hat{F} \in \mathcal{F}, \text{ with } |F| > |\hat{F}| \text{ then } \exists e \in F \setminus \hat{F}, \text{ such that } \hat{F} \cup \{e\} \in \mathcal{F}$$

in other words given two feasible sets with one being strictly larger than the other in cardinality, then we can remove one element from the larger set and add it to the smaller set while maintaining feasiblilty of the smaller set.

4.8.5 Matroids

A matroid $M = (E, \mathcal{F})$ is a non-empty set system that satisfies both the herediraty property and the augmentation property. Note that having the hereditary property and non-emptiness implies $\emptyset \in \mathcal{F}$.

Some matroids include

- Graphical Matroids:
  - E is the edges in a set
  - $F \in \mathcal{F}$ if F is a forrest
- Linear Matroids:
  - E is a finite set of vectors

- F is a collection of linearly independent vectors

- Transversal Matroids:
  - E is the set of leaf vertices in a bipartite graph
  - $F \in \mathcal{F}$ if there is a matching in the graph that matches each vertex in F to another distinct vertex on the right

Theorem: A hereditary non-empty set system is a matroid if and only if the greediest algorithm outputs the optimal solution in M for any set of weights W.

Proof: ($\Rightarrow$) First let's show that the greediest algorithm always works on matroids. Suppose the algorithm outputs $\{e_1, e_2, \ldots, e_\ell\}$ where $w(e_1) \geq w(e_2) \geq \cdots \geq w(e_\ell)$. Let the optimal solution be $\{e_1^*, e_2^*, \ldots, e_k^*\}$ where $w(e_1^*) \geq w(e_2^*) \geq \cdots \geq w(e_k^*)$. By the augmentation property, we have that $\ell \geq k$. We claim that $w(e_i) \geq w(e_i^*)$ for each $1 \leq i \leq \ell$. Assume the contrary. Let j be the smallest index with $w(e_j) < w(e_j^*)$. Now consider the sets $F = \{e_1, e_2, \ldots, e_{j-1}\}$, and $\hat{F} = \{e_1^*, e_2^*, \ldots, e_j^*\}$. By the augmentation property $\exists e_i^* \in F$ such that $\hat{F} \cup e_i \in \mathcal{F}$. But $w(e_i^*) \geq w(e_j^*) > w(e_j)$, a contradiction as the greediest algorithm should have chosen e_i^* at step j instead of e_j.
($\Leftarrow$) Now take a hereditary set system M that's not a matroid. Then $\exists F, \hat{F} \in \mathcal{F}$ with $|F| > |\hat{F}|$, but $\nexists e \in F \setminus \hat{F}$ such that $\hat{F} \cup \{e\} \in \mathcal{F}$. Let $\varepsilon > 0$ then consider the following collection of weights that we claim will cause the algorithm to fail.

$$w(e) = \begin{cases} 2 & \text{if } e \in F \cap \hat{F} \\ 1+\varepsilon & \text{if } e \in \hat{F} \setminus F \\ 1 & \text{if } e \in F \setminus \hat{F} \\ 0 & \text{if } e \notin F \cup \hat{F} \end{cases}$$

As the algorithm runs it first selects the elements in $F \cap \hat{F}$. Then it selects all the elements in $\hat{F} \setminus F$. Finally it selects some elements in $E \setminus (F \cup \hat{F})$. Therefore the algorithm outputs a solution with weight

$$\begin{aligned} 2 \cdot |F \cap \hat{F}| + (1+\varepsilon) \cdot |\hat{F} \setminus F| + 0 &= 2 \cdot |F \cap \hat{F}| + (1+\varepsilon) \cdot |\hat{F} \setminus F| \\ &< 2 \cdot |F \cap \hat{F}| + (1) \cdot |\hat{F} \setminus F| \\ &= \text{opt} \end{aligned}$$

Thus the algorithm output a lower-weight solution than the optimal and hence failed with these weights. $\square$

5 Dynamic Programming

"I spent the Fall quarter [of 1950] at RAND. My first task was to find a name for multistage decision processes. An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term 'research' in his presence. You can imagine how he felt, then, about the term 'mathematical'. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying — I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities"

– Richard Bellman, 1984, on the origins of the term 'dynamic programming'

5.1 Fibonacci Numbers

Recall that the Fibonacci Numbers are defined by

$$f(0) = 1,\ f(1) = 1,\ f(n) = f(n-1) + f(n-2),\ \forall n \geq 2$$

We can solve recurrence relation to obtain a closed form equation given by

$$f(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^2\right) = \Theta(1.618^n)$$

Using the recursion tree method we get that $f(n) = \Omega\big(\sqrt{2}^n\big)$. Well that's atrocious.

One way to improve on this is to just store data. Indeed storing data is the underlying idea behind dynamic programming. There are two basic approaches for implementing dynamic programming.

1. Top-down (memoization)
2. Bottom-up

5.1.1 Top-Down

The best way to explain both this approach and bottom-up is by illustrating them through an example – the Fibonacci numbers. Consider the algorithm:

FibMemo($f(n)$)
    if $n \leq 1$
        return 1
    else
        if $f(n)$ is undefined
            set $f(n) \leftarrow$ FibMemo($f(n-1)$) + FibMemo($f(n-2)$)

Note that this is similar to DFS insofar as we dive down to solve the base cases and then we work our way back up the tree.

This takes $n-1$ additions, where in each iteration we add two numbers. Adding two n-bit numbers takes $O(n)$ time. Thus the running time for this algorithm is $O(n^2)$. That's waaay better than that monster we had originally.

5.1.2 Bottom-Up

In fact there is an even simpler way to solve this problem using the bottom-up approach. Consider the following algorithm:

FibB-up($f(n)$)
    set $f(0) \leftarrow 1$
    set $f(1) \leftarrow 1$
    for $i \in \{2, 3, \ldots, n\}$
        set $f(n) \leftarrow f(n-1) + f(n-2)$
    return $f(n)$

5.1.3 Top-Down versus Bottom-Up

Top-down is generally faster because it solves only the sub-problems that need to be solved. Bottom-up, however, is generally cleaner and conceptually simpler.

5.1.4 Dynamic Programming vs. Divide and Conquer

Dynamic programming is very similar to divide and conquer as in both methods, we're dividing a problem$\mathcal{P}$ into k sub-problems $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_k$, and combining them for a final answer. The difference, however, is that in divide and conquer, the sub-problems must be completely disjoint, where in dynamic programming,

they're allowed to overlap. Specifically, $\mathcal{Q}_1$, and $\mathcal{Q}_2$ may share some common sub-sub-problem $\hat{\mathcal{Q}}$.

5.1.5 What's Needed for Dynamic Programming?

To solve a dynamic program we need the following properties to hold:

- The solution $\mathcal{P}$ must be computable using the set of sub-problems $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_k$
- We should have a natural ordering of the sub-problems from smallest to largest
- We should have a polynomial number of sub-problems
- We want a recurrence giving the solution to $\mathcal{P}$ in terms of the solution to $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_k$

If the above conditions hold, then the running time of a dynamic program is equal to the number of sub-problems multiplied by the time required to use the recursive formula at each step.

5.2 The (Weighted) Interval Selection Problem

Suppose we have a set I of n intervals, each with a start time s_i, and with a finish time f_i. Now suppose that each interval has an associated weight v_i. Our goal is to find a set S of disjoint intervals with the maximum total weight

$$w(S) = \max \sum_{i \in S} v_i$$

To try a dynamic programming approach, first we need to define the sub-problems. The key issue to define them with a natural ordering. We have n objects $[n] = \{1, 2, \ldots, n\}$. Our trick will be to define a subproblem $[\ell]$ as $[\ell] = \{1, 2, \ldots, \ell\}$. The (sub)-problem $[\ell]$ is then to find the maximum value disjoint set of intervals when we can only use the intervals in $\{1, 2, \ldots, \ell\}$. Thus we have the natural ordering

$$[1] \prec [2] \prec \cdots \prec [n]$$

Note that we still have the flexibility to change the labels of the intervals. Like in the greedy interval selection problem, let's order the intervals by finish time so that

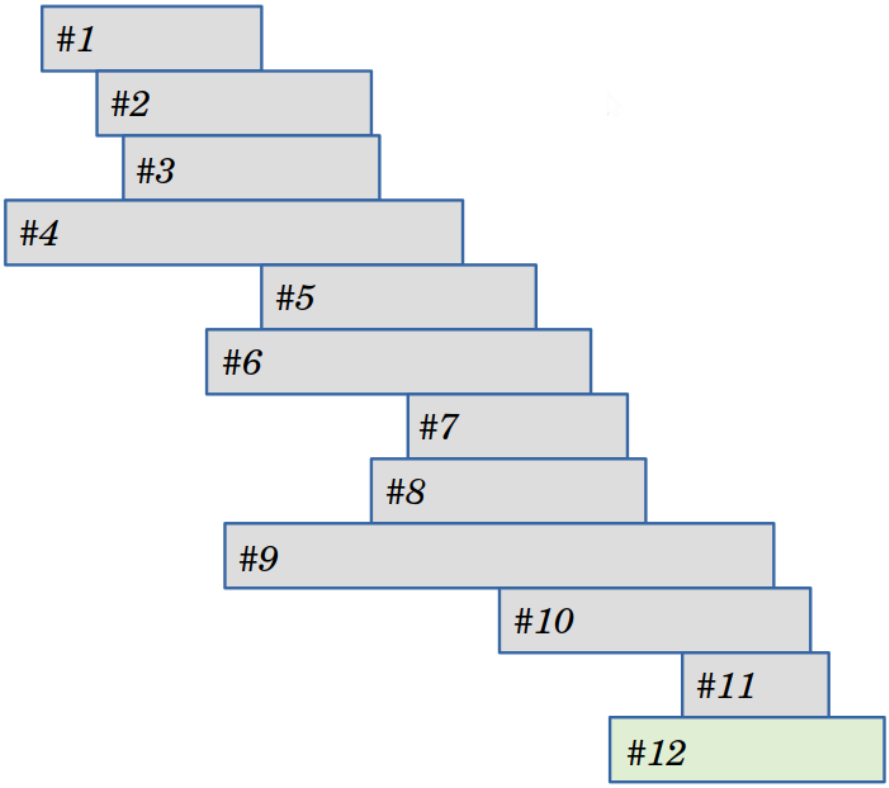
$$f_1 \leq f_2 \leq \cdots \leq f_n$$

This is a good choice of an ordering because it works. The key to figuring out why, however, is to look at the structure of the problem. To do this, we consider two questions (that we'll ask ourselves in every dynamic programming from now on:)

1. What happens if the algorithm selects n?

2. What happens if the algorithm doesn't select n?

First consider the picture to give some motivation for the following lemmas:



Lemma: Take any interval i that finishes after the start of an interval n. Then if we select interval n we cannot select interval i.

Proof: Interval n is the last interval $f_j \leq f_n$. But by assumption, $s_n < f_j \Rightarrow s_n < f_j \leq f_n \Rightarrow$ interval j, and interval n clash at time $f_j - \varepsilon$ for some $\varepsilon > 0$. Since they clash, we can't choose both. □

Lemma: The intervals that clash with interval n are consecutive.

Proof: Let ℓ be the lowest index interval with $f_\ell > s_n$. Then by the first lemma, interval ℓ clashes with interval n. But for any interval $i > \ell$, we have that $f_i \geq f_\ell$ by the ordering criteria $\Rightarrow f_i > s_n \Rightarrow$ interval i also clashes with interval n. Thus all intervals $\{\ell, \ell+1, \ldots, n-1\}$ clash with n. Conversely, no interval i with index $i < \ell$ clashes with interval n, as $s_i < f_i \leq s_n < t_n \Rightarrow$ interval i is disjoint from interval n. Thus the interval n clashes exactly with the set $\{\ell, \ell+1, \ldots, n-1\}$. □

From these two lemmas, it's natural define some notation for the highest index interval that's disjoint from an interval n. Let $h(n)$ be such an interval. Thus by the above lemmas if we select an interval n, the set of intervals $\{h(n)+1, h(n)+2, \ldots, n-1\}$ becomes infeasible.

Now we have what we need to proceed to answer the two questions from earlier. If the algorithm selects the interval n, it should also select the optimal solution for the subproblem consisting of intervals from the set $\{1, 2, \ldots, h(n)\}$. If the algorithm doesn't select n, then it should simply select an optimal solution for the subproblem consisting of the intervals from the set $\{1, 2, \ldots, n-1\}$. Now that we've got this structure, we can recurse to find the optimal solution, by using the recursive formula given by

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

with the base case given by $\text{opt}(0) = 0$.

Recall that in the same way that solving the Fibonacci numbers recursively, it's really bad (exponential time) to solve this recursively, unless we store some data, i.e. unless we use a dynamic program. Consider the algorithm:

WIntSelect(I)
    sort the intervals by finish time
    set opt(0) := 0
    for each $j \in \{1, 2, \ldots, n\}$
        let $h(j) < j$ be the highest index interval disjoint from j
        set $\text{opt}(j) = \max\{\text{opt}(j-1), v_j + \text{opt}(h(j))\}$
    return opt(n)

But note that this dynamic program only gives the value of the weight of the optimal solution, not the actual set of intervals. We can fix this by adding some pointers to the algorithm. Namely, we'll change this algorithm to be

```
WIntSelect(I)
    sort the intervals by finish time
    set opt(0) := 0
    for each j ∈ {1, 2, ..., n}
        let h(j) < j be the highest index interval disjoint from j
        set opt(j) = max{opt(j − 1), v_j + opt(h(j))}
        if opt(j) = opt(j − 1)
            setp(j) ← j − 1
        else
            set p(j) ← h(j)
    return opt(n)
```

Once we have the optimal solution, we can just follow the pointers back to recover the optimal solution set.

This algorithm breaks the problem into two subproblems, each taking $O(n)$ time to complete. Thus the total running time of the algorithm is $O(n)$.

5.3 More on the Structure of a Dynamic Program

The four steps to a dynamic program are:

1. **Structure:** Understand the structure of the underlying problem
2. **Recursion:** Use the structure of the problem to formulate a recurrence
3. **Optimal Value:** Solve the recursion through a (bottom-up) dynamic program
4. **Solution Recovery:** Follow the pointers backwards to recover the optimal solution

Understanding the structure is both the most important step, and the hardest step to solve. Luckily for us, the structure of a dynamic program falls into one of four cases:

1. One-Sided Interval
2. Box Structure
3. Two-Sided Interval
4. Tree Structure

5.3.1 One-Sided Interval

The input is $\{x_1, x_2, \ldots, x_n\}$, and the sub-problems are of the form $[j] = \{x_1, x_2, \ldots, x_j\}$. The subproblems are ordered according to the rule $[i] \preceq [j]$ if and only if $i \leq j$. An example is the weighted interval selection problem that we've already seen.

5.3.2 Box Structure

The input is $\{x_1, x_2, \ldots, x_n\}$, and $\{y_1, y_2, \ldots, y_m\}$. The sub-problems are of the form $[i, j] = \{x_1, x_2, \ldots, x_i, y_1, y_2, \ldots, y_j\}$. The subproblems are ordered according to the rule $[i, j] \preceq [k, \ell]$ if and only if $i \leq k$, and $j \leq \ell$. Some examples of dynamic programs with the box structure are the knapsack problem, and the Humpty-Dumpty problem.

5.3.3 Two-Sided Interval

The input is $\{x_1, x_2, \ldots, x_n\}$, and the sub-problems are of the form $[i, j] = \{x_i, x_{1+1}, \ldots, x_j\}$. The subproblems are ordered according to the rule $[i, j] \preceq [k, \ell]$ if and only if $k \leq i$, and $j \leq \ell$. An example is the weighted interval selection problem is the RNA secondary structure problem.

5.3.4 Tree Structure

The input is a rooted tree with vertex set $\{x_1, x_2, \ldots, x_n\}$ and the sub-problem $[j]$ is given by the subtree rooted at x_j. The subproblems are ordered by $[i] \preceq [j]$ if and only if x_i is a descendant of x_j. The independent set on a tree problem is an example of a problem employing this structure.

5.4 Knapsack Problem

Suppose a theif has a bag with capacity W, and suppose there are n elements to steal. Item i has value v_i, and weight/size w_i. The theif wants to steal the subset of items with the maximum total value that fits into the knapsack. This problem has the box structure, where the first dimension represents the items, and where the second dimension represents the weight. To understand this structure in more detail, we must ask ourselves the same two questions as from the weighted interval selection problem:

1. What happens if the algorithm selects item n?
   - The remaining capacity of the bag is $W - w_i$
   - The set of remaining items is $\{1, 2, \ldots, n-1\}$
2. What happens if the algorithm doesn't select item n?
   - The remaining capacity of the bag is W
   - The set of remaining items is $\{1, 2, \ldots, n-1\}$

From these, we can deduce that the recurrence relation for this problem is given by

$$\text{opt}(j, w) = \max\{\text{opt}(j-1, w),\ v_j + \text{opt}(j-1, W-w_i)\}\ \forall j \geq 1,\ \forall w \leq W$$

with base cases $\text{opt}(j, w) = 0$ if $j = 0$ or $w \leq 0$.

Much like the weighted interval selection problem, simply solving this recurrence is far too slow. Let's try solving it using a dynamic program.

```
Knapsack(I, W)
    set opt(0, w) := 0, ∀w ≤ W
    set opt(j, w) := 0, ∀j, and ∀w ≤ 0
    for each j ∈ {1, 2, ..., n}
        for each w ∈ {1, 2, ..., W}
            set opt(j, w) = max{opt(j − 1, w), v_j + opt(j − 1, W − w_i)}
    return opt(n, W)
```

There are $2n \cdot W$ subproblems, thus the running time is $O(nW)$. Note that this is only a pseudo-polynomial time algorithm. It is not truly polynomial as if the integers have b bits, then W could be 2^b. Indeed, for a polynomial time algorithm we really want it to be either:

- Wealky polynomial: The algorithm runs in time $\text{poly}(n, b)$
- Strongly polynomial: The algorithm runs in time $\text{poly}(n)$

If W is small, then this won't cause huge problems for us, however.

Unfortunately, we don't think that there actually is a polynomial time algorithm to solve this problem, unless of course P=NP.

5.5 Humpty Dumpty Problem

Suppose we have a T-storey skyscraper and k identical eggs. We want to find the highest storey from which we can drop an egg without it breaking, whilst using the fewest number of eggs as possible.

Our first intuition may be to just start on the lowest floor and keep moving up if the egg doesn't break. The requires $O(T)$ tests. Let's try to do better.

Next we may try using binary search. This is a little better as it requires only $O(\log n)$ tests, but this still may fail as it's possible that $k \ll \log T$.

Let's try a dynamic programming approach. We have two parameters: the

number of eggs, and the number of (remaining) stories. Let $d(T, k)$ be the number of egg drops required to guarantee that we can successfully drop an egg safely. Like always, we first have to look at the structure. If the egg breaks, then the answer is in the range $\{0, 1, \ldots, t-1\}$, but then we only have $k-1$ eggs remaining. Therefore, the number of drops is less than or equal to $1 + d(t-1, k-1)$. Similarily, if the egg doesn't break, then the answer is in the range $\{t, \ldots, T-1, T\}$, but then we still have k eggs. Thus the number of drops is at most $1 + d(T-t, k)$. From this we can derive our recursive formula as

$$d(T, k) = 1 + \min_{t:1 \leq t \leq T} \{\max\{d(t-1, k-1),\ d(T-t, k)\}\}$$

with base cases given by $d(\tau, 1) = \tau$, $\forall \tau \geq 1$, and $d(0, \kappa) = 0$, $\forall \kappa$. Filling this into a dynamic program gives us the following algorithm.

HumptyDumpty(T, k)
    set $d(\tau, 1) = \tau$, $\forall \tau \geq 1$
    set $d(0, \kappa) = 0$, $\forall \kappa$
    for each $\tau \in \{1, 2, \ldots, T\}$
        for each $\kappa \in \{1, 2, \ldots, k\}$
            set $d(\tau, \kappa) = 1 + \min_{t:1 \leq t \leq T} \{\max\{d(t-1, k-1),\ d(T-t, k)\}\}$
            if $d(\tau, \kappa) = 1 + d(t-1, k-1)$
                set $p([\tau, \kappa]) \leq [t-1, \kappa-1]$
            else
                set $p([\tau, \kappa]) \leq [\tau-1, \kappa]$
    return $d(\tau, \kappa)$

There are Tk subproblems. To solve each subproblem, we need to look at $2T$ smaller problems. Thus the time complxity is $O(T^2 k)$, which is polynomial time.

5.6 The Bellman-Ford Algorithm

Recall the shortest path problem from when we looked at Dijkstra's algorithm. Recall that Dijkstra's fails on negative edge lengths. Sometimes, however, we come across negative edges in applications (for example in some finatial network we can have negative costs versus positive profits.) Therefore we need to come up with an algorithm that won't break when it encounters negative edges.

One's first intuition may be to just change the graph into an instance of the graph without negative edges, perhaps by setting the negative edges all to 0. We note right away that that intuition is garbage because the shortest paths can change. Let's think about this a bit more. If we have negative arc lengths, then it's certainly possible that we also have negative length cycles. Thus, if we have no negative cycles, the shortest walk in the graph has the same length as the

sortest path, however if we do have negative cycles, then the shortest walk can be shorter than the shortest path. For now, let's assume no negative cycles. Then, as it turns out, we can find the shortest path using dynamic programming. Indeed, for every vertex v, we can find the shortest path from s to v, using at most k arcs, where we have to solve this $n-1$ times, as each vertex has $k = n-1$ arcs.

This problem has the box structure, where the first parameter is the vertices, and the second parameter is the number of arcs in the path.

1. What happens if the shortest path to v has fewer than k arcs?
   - The shortest path to v with fewer than k arcs has at most $k-1$ arcs.
2. What happens if the shortest path to v has exactly k arcs?
   - The shortest path to v with at most k arcs consists of a shortest path to an in-neghbour u of v with at most $k-1$ arcs plus the arc (u, v).

From the above thought experiment, we can derive our recursive formula

$$\text{opt}(k, v) = \min\left\{\text{opt}(k-1, v),\ \min_{u \in \Gamma^-(v)} \text{opt}(k-1, u) + \ell_{uv}\right\}$$

with base cases given by $\text{opt}(0, s) = 0$, and $\text{opt}(0, v) = \infty\ \forall v \neq s$.

We can now use this recursive formula to write our dynamic program.

BellmanFord(G, s)
    set $\text{opt}(0, s) := 0$
    set $\text{opt}(0, v) := \infty$
    for each $k \in \{1, 2, \ldots, n-1\}$
        for each $v \in V$
            set $\text{opt}(k, v) = \min\left\{\text{opt}(k-1, v),\ \min_{u \in \Gamma^-(v)} \text{opt}(k-1, u) + \ell_{uv}\right\}$
    for each $v \in V$
        return $\text{opt}(n-1, v)$

There are n^2 subproblems in which we solve n smaller problems. Thus with this crude analysis the running time is $O(n^3)$. Using a more nuanced analysis we have that the running time is $O(mn)$.

5.6.1 Negative Cycle Detection

Recall that in the above, we assumed no negative cycles. But what if the graph contais negative cycles? Well it turns out that the Bellman-Ford algorithm doubles as a negative cycle-detector. Thus to find negative cycles, we just have to run the algorithm again. To see why this works, we need some help from some

lemmas.

Lemma: If the graph contains no negative cycles, then

$$\text{opt}(n, v) = \text{opt}(n-1, v),\ \forall v \in V$$

Proof: Any path from s to v uses at most $n-1$ arcs $\Rightarrow$ any walk W from s to v that uses n arcs contains a cycle C that has non-negative length, and therefore $\ell(W) \leq \ell(W \setminus C)$. Thus there is also a shortest path P from s to v that is also a shortest walk, hence using n or more arcs won't help us. Thus $\text{opt}(n, v) = \text{opt}(n-1, v),\ \forall v \in V$. □

Lemma: If $\text{opt}(n, v) = \text{opt}(n-1, v),\ \forall v \in V$, then the graph contains no cycles.

Proof: We have that $\text{opt}(n, v) = \text{opt}(n-1, v),\ \forall v \in V$. This implies that row n of the dynamic programming table is exactly the same as row $n-1$. Since nothing has changed, if we run the algorithm further, we get $\text{opt}(n, v) = \text{opt}(n+1, v) = \text{opt}(n+2, v), \ldots,\ \forall v \in V$. But then no vertex x can lie on a negative cycle or otherwise the length of the best walk found to x must decrease. Thus, the graph contains no negative cycles. □

Note that in proving these lemmas, we have directly proved the following theorem.

Theorem: A graph contains no negative cycles iff $\text{opt}(n, v) = \text{opt}(n-1, v),\ \forall v \in V$. Furthermore, the graph contains a negative cycle if there exists a vertex $x \in V$ such that $\text{opt}(n, x) < \text{opt}(n-1, x)$. Moreover, we can find the negative cycle by following the predecessor arcs back from x.

Therefore, the Bellman-Ford algorithm outputs either the shortest path, or a negative cycle.

5.6.2 Distributed Algorithms

Unlike Dijkstra's algorithm, the Bellman-Ford algorithm is a *distributed algorithm*. This means that it can be implemented by many different processors across different machines.

5.6.3 Testing for Negative Cycles

Given a set $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ of shortest paths in a graph, we have an efficient algorithm to test whether there is a negative cycle. Namely, since we're given the graph $G = (V, A)$, and we know the arc lengths, then for each vertex $v \in V$ we can calculate $d(v) = P_v$, and then for each arc $(u, v) \in A$, we can test whether $d(v) \leq d(u) + \ell_{uv}$. If the answer is yes, then there's no negative cycle, and otherwise there is one. More formally, consider the following theorem:

Theorem: Given a set of paths $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ with $d(v) = \ell(P_v)$, these paths are the shortest if and only if $d(s) = 0$, and $d(v) \leq d(u) + \ell_{uv}$, $\forall v \in V$.

Proof: ($\Leftarrow$) Suppose there is a path $\mathcal{Q}_i$ from s to v_i such that $\ell(\mathcal{Q}_i) < \ell(P_i)$. WLOG, assume $\mathcal{Q}_i = \{s = v_1, v_2, \ldots, v_i\}$. Thus

$$\begin{aligned}
\ell(P_i) &= d(v_i) \\
&\leq d(v_{i-1}) + \ell_{v_i v_{i-1}} \\
&\leq d(v_{i-2}) + \ell_{v_{i-2} v_{i-1}} \\
&\leq \cdots \\
&\leq d(v_1) + \ell_{v_1 v_2} + \cdots + \ell_{v_{i-1} v_i} \\
&= 0 + \ell_{v_1 v_2} + \cdots + \ell_{v_{i-1} v_i} \\
&= \ell(\mathcal{Q}_i)
\end{aligned}$$

($\Rightarrow$) If $\mathcal{P}$ contains the shortest paths, then $d(s) = 0$. Moreover, we must have $d(v) \leq d(u) + \ell_{uv}$, or else we can find shorter paths from u to v. □

5.7 All-Pairs Shortest Paths

Given a directed graph $G = (V, A)$ with arc lengths ℓ_a, we want to find the shortest path between every pair of vertices. We will assume that there are no negative cycles in G (if there were then we could just run Bellman-Ford n times starting from a different vertex each time. This would take $O(n^4)$, or with a more nuanced analysis $O(n^2 m)$, which kinda sucks but it does the trick.) Indeed, the algorithm that we'll derive to do this is called the Floyd-Warsahll algorithm.

First, we order the vertices in some arbitrary order $\{1, 2, \ldots, n\}$. Then we create a sub-problem for each tripple of vertices $\{u, v, k\}$. Here, $\text{opt}(u, v, k)$ is the shortest path from u to v using only the vertices $\{1, 2, \ldots, k\}$ internally on the path.

Then, we'll take the optimal path $\mathcal{P}$ for $\text{opt}(u, v, k)$.

1. What happens if $k \in \mathcal{P}$?
   - Then $\text{opt}(u, v, k) = \text{opt}(u, v, k-1)$.
2. What happens if $k \notin \mathcal{P}$?
   - Then $\text{opt}(u, v, k) = \text{opt}(u, k, k-1) + \text{opt}(k, v, k-1)$.

This has npt just a box structure, but a 3D box structure. We can then solve this by a recurrence. Our recursive formula is

$$\text{opt}(u, v, k) = \min\{\text{opt}(u, v, k-1),\ \text{opt}(u, k, k-1),\ \text{opt}(k, v, k-1)\}$$

$\forall u, \forall v, \forall k \geq 1$, and with base cases given by

$$\text{opt}(u, v, 0) = \begin{cases} 0 & \text{if } u = v \\ \ell_{uv} & \text{if } (u, v) \in A \\ \infty & \text{if } (u, v) \notin A \end{cases}$$

Let's now write a dynamic program to solve this problem.

FloydWarshall(G)
  set $\text{opt}(u, v, 0) = 0$, $\forall v \in V$
  set $\text{opt}(u, v, 0) = \ell_{uv}$, $\forall (u, v) \in A$
  set $\text{opt}(u, v, 0) = \infty$, $\forall (u, v) \notin A$
  for each $k \in \{1, 2, \ldots, n\}$
    for each $u \in \{1, 2, \ldots, n\}$
      for each $v \in \{1, 2, \ldots n\}$
        set $\text{opt}(u, v, k) = \min\{\text{opt}(u, v, k-1),\ \text{opt}(u, k, k-1),\ \text{opt}(k, v, k-1)\}$

There are n^3 subproblems that each contain 3 smaller problems. Thus the running time is $O(n^3)$.

5.8 Independent Set on a Tree

6 Network Flows

6.1 Ford-Fulkerson Algorithm

6.2 Max Flow - Min Cut Theorem

6.3 Model Extensions

6.4 Bipartite Matching

6.5 Applications of Network Flows

6.6 Maximum Capacity Augmenting Path Algorithm

7 Data Structures

7.1 Priority Queues & Heaps

7.2 Hashing

7.3 String Matching

7.4 Binary Search Trees

7.5 Data Structure for Disjoint Sets

7.6 Segment Trees