Contents

1 Asymptotic Notation

Question What does it mean for an algorithms to be efficient?

Answer It's polynomial in the input size

> **Note**.
>
> We say that $g(n)$ is **in** $\Omega(n), \mathcal{O}(n)$, etc... Not that $g(n)$ **equals** $\Omega(n), \mathcal{O}(n)$, etc...

1.1 Big $\mathcal{O}$ Notation

Asymptotic upper bound.

We say that

$$f(n) \in \mathcal{O}(g(n))$$

If $\exists C > 0, \exists n_0 \geq 1$ such that $\forall n \geq n_0$

$$f(n) \leq Cg(n)$$

Eventually, f is below a constant multiple of g. Big $\mathcal{O}(g(n))$ represents an asymptotic *upper bound* of $f(n)$.

1.2 Little o Notation

$$f(n) \in o(g(n))$$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Meaning that f grows **slower** than g.

> **Example**.
>
> If $f(n) = 10n^2$, then $f(n) \in o(n^3)$ because $\lim_{n\to\infty} \frac{10n^2}{n^3} = 0$.
>
> Also, note that $f(n) \in \mathcal{O}(n^2)$, if we let $C = 11$, then asymptotically, $10n^2 < 11n^2$. **However**, $f(n) \notin o(n^2)$, since $\lim_{n\to\infty} \frac{10n^2}{n^2} \neq 0$. Moreover, $f(n) \notin o(Cn^2)$ for any $C \in R$ since, for a fixed C, $\lim_{n\to\infty} \frac{10n^2}{Cn^2} = \frac{10}{C} \neq 0$.

> **Note**.
>
> If $f(n) \in o(n)$, then $f(n) \in O(n)$

1.3 Big Ω Notation

Asymptotic lower bound.

$$f(n) \in \Omega(g(n))$$

If $\exists B > 0, \exists n_0 \geq 1$ such that $\forall n \geq n_0$

$$f(n) \geq Bg(n)$$

Eventually, f is above a constant multiple of g. This represents an asymptotic *lower bound*.

Example.

For example, we have $g(n) = 10n^2 \in \Omega(n^2)$.

1.4 Little ω Notation

$$f(n) \in \omega(g(n))$$

If $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$. Meaning that f grows **faster** than g.

Note.

If $f(n) \in \omega(n)$, then $f(n) \in \Omega(n)$

1.5 Big Θ Notation

Asymptotic upper and lower bound.

$$f(n) \in \Theta(g(n))$$

Meaning that $\exists B > 0, C > 0, n_0 \geq 1$ such that $\forall n \geq n_0$

$$Bg(n) \leq f(n) \leq Cg(n)$$

Eventually, f is between two constant multiples of g. Here, we bound f by two constant multiples of g.

Note.

If $f(n) \in \Theta(n)$, then $f(n) \in \mathcal{O}(n)$ and $f(n) \in \Omega(n)$

Note.

- $o <$
- $\mathcal{O} \leq$
- $\Theta =$
- $\Omega \geq$
- $\omega >$

1.6 Running Time Analysis

Note.

We will be evaluating some recurrence equations in this class.

We need to count elementary steps when evaluating our algorithms. For instance the number of times we go through a loop. Inside this loop, we should also consider how expensive one such pass is, as it might not be constant.

Using the right data structures can impact this a lot.

1.6.1 Running Time Classes

- $\mathcal{O}(1)$: If we can get it
- $\mathcal{O}(\log^k(n))$: This is a good target for a data structure. Shows up in binary search.
- $\mathcal{O}(n^{1/k})$ for $k \in \{1, 2, \dots\}$
- $\mathcal{O}(n^k)$: Not great but not bad. Occurs in lists a lot
- $\mathcal{O}(n \log(n))$: A good sorting algorithm. (Merge, Quick, ...) It's a lower bound on *comparison based* sorting algorithm.
- $\mathcal{O}(n^2)$: For some things, this isn't bad but it's not great either.
- $\mathcal{O}(n^k)$ for $k \geq 2$: Slow. Fine for small data. This can happen in searches of smaller subsets.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \in \Theta(n^k)$$

- $\mathcal{O}(2^n)$: Terrible, but sometimes it's all you get. Determining if a boolean expression is satisfiable. Checking all subsets.
- $\mathcal{O}(n!)$: Bad.

2 Graphs

Definition. **Undirected Graph**

An **Undirected Graph** $G = (V, E)$ is a finite set of vertices V and edges E, which are **unordered** pairs of vertices, where for some edge $e \in E$, $e = \{v_1, v_2\}$ is an edge connecting vertex v_1 and vertex v_2.

■

Note.

Vertices are sometimes called *Nodes*.

Definition. **Directed Graph**

A **Directed Graph**, *also known as a Digraph*, $G = (V, E)$ is a finite set of vertices V and edges E, which are **ordered** pairs of vertices, where for some edge $e \in E$, $e = (v_1, v_2)$ is an edge pointing from vertex v_1 to vertex v_2.

■

Note.

An edge $e = \{u, v\} \in E$ is *uniquely* defined by vertices $u, v \in V$. You **cannot** have two edges going from the same starting vertex to the same destination vertex in a Graph.

However in a Digraph, you **can** have both $e = (u, v)$ and $e' = (v, u)$, where e goes from u to v, and e' goes from v to u.

Definition. **Adjacence and Incidence**

For some Graph G with some edge $e = \{v_1, v_2\}$, we say that v_1 and v_2 are **adjacent**, and e is **incident** on the vertices.

■

Definition. **Subgraph**

A **Subgraph** of $G = (V, E)$ is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ where, for any edge $e = \{v_1, v_2\} \in E'$, $v_1, v_2 \in V'$.

Author Note.

G' is a subgraph of G if all of its edges and vertices are in G. It can be connected to the rest of G, or not.

■

Definition. **Head and Tail**

For some Digraph G with some edge $e = (v_1, v_2)$, we say that e has **tail** v_1 and **head** v_2.

■

Definition. **Degree**

The **degree** of a vertex $\deg(v)$ is its number of *incident* edges.

The **in degree** of a vertex $\text{indeg}(v)$ is its number of edges with v as the head.

Author Note.

This is the number of edges pointing *into* v.

The **out degree** of a vertex $\text{outdeg}(v)$ is its number of edges with v as the tail.

Author Note.

This is the number of edges pointing *out of* v.

■

For some undirected graph $G = (V, E)$, we commonly define $n = |V|$, and $m = |E|$. Then we have the following identities

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

Author Note.

For all vertices, the sum of all degrees is twice the number of edges. This should make sense, as edges are attached to two vertices.

For a Digraph, we have that

$$\sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v) = m$$

Author Note.

Since the graph is directed, each edge has a start and end, so it should make sense that the sum of all in degrees should match the sum of all out degrees, which is the number of edges.

2.1 Representations

2.1.1 Adjacency Matrix

Represented by a $|V| \times |V|$ matrix indexed by vertices, with

$$a_{i,j} = \begin{cases} 1 \text{ if (i, j)} \in E \\ 0 \text{ otherwise} \end{cases}$$

Note.

This has size $\Theta(n^2)$

2.1.2 Adjacency List

Represented by a list of vertices v where, for each $v \in V$, we are given a list of all its neighbors. Usually, this list is composed only of *outgoing* neighbors from v.

> **Note**.
>
> This has size $\Theta(n+m)$

2.2 Paths & Cycles

Definition. **Path**

A **Path** is a sequence of vertices $(v_0, v_1, ..., v_k)$ such that $(v_i, v_{i+1}) \in E$ for all $i \in \{0, ..., k-1\}$.

The length of a path is its number of edges, k

■

Definition. **Simple Path**

A path is **Simple** if all its vertices and edges are distinct

■

Definition. **Cycle**

A **Cycle** is a path with $v_0 = v_k$ and all other vertices distinct, with $k \geq 3$.

Do note that, for a *Digraph*, two vertices connected by edges *can* form a cycle. In which case, $k \geq 2$. Additionally, it's important that the edge is in the right direction for it to count as a cycle.

■

Definition. **Distance**

The **Distance** between two vertices u and v is the length of *a shortest path* between them.

■

2.3 Connectivity

Can you get from one vertex to another at all?

Definition. **Component**

A **Component** is a set of vertices such that for all pairs of vertices in the set, there is a path between them. No superset of vertices has this property.

> **Note**.
>
> A single vertex is still a component.

A component is really represented only by its vertices.

■

Definition. **Connected Graph**

We say that an undirected graph is **Connected** if, for all $u, v \in V$, there exists a path from u to v.

> **Note.**
>
> The path does not need to be direct.

■

Definition. **Strongly Connected**

A Digraph is **Strongly Connected** if you can get from any vertex to any other vertex. For all $u, v \in V$ there exists a path from u to v and from v to u.

> **Note.**
>
> The path does not need to be direct.

■

Definition. **Strongly Connected Component**

A **Strongly Connected Component** of a Digraph is a maximal set of vertices for which the induced subgraph is strongly connected.

■

> **Note.**
>
> Components partition a graph, and the union of the partitions always produce the whole graph.

2.4 Edge Bounds

Let's look at various types of graphs and how many edges they can have.

2.4.1 Undirected Graphs

With self loops

$$0 \leq m \leq n + \binom{n}{2}$$

Without self loops

$$0 \leq m \leq \binom{n}{2}$$

2.4.2 Directed Graphs

Self loops

$$0 \leq m \leq n + 2\binom{n}{2}$$

Without self loops

$$0 \leq m \leq 2\binom{n}{2}$$

2.5 Breadth First Search

Breadth First Search splits the graph into layers L_j. L_0 is the starting vertex. L_{j+1} is the set of vertices in L_j which are not already neighbors of $L_0 \cup \cdots \cup L_{j-1}$.

Lemma.

A vertex in L_j has distance j from s, the starting vertex.

Proof.

By induction on j. Base $j = 0$, $L_0 := \{s\}$ is the set of vertices with distance 0. If $L_0, ..., L_j$ satisfy the lemma, then the vertices in L_{j+1} must have distance at least $j+1$ from s (otherwise they would be in $L_0, ..., L_j$ instead), but there exists a path of length $j+1$ to all vertices in L_{j+1} since they are adjacent to vertices in L_j. Vertices in L_{j+1} therefore satistfy the claim so the claim follows for all j by induction.

■

2.5.1 Pseudocode

For a graph G, starting vertex s, and a function f, we have

```
BreadthFirstSearch(G, s, f):
  # initialize a stack with the starting vertex
  queue = [s]
  visited = ∅

  while stack is not empty
    v = pop(stack)

    # perform action on v
    f(v)

    for all neighbors u of v:
      if u ∉ visited:
        push u to queue
        add u to visited
      end
    end
  end
end
```

2.5.2 Time Complexity

Loop over all vertices $v \in V$, for each, do $\mathcal{O}(\deg(u) + 1)$ operations. Total asymptotic running time is

$$\mathcal{O}(\sum_{v \in V}(\deg(n) + 1)) = \mathcal{O}(2\deg(n) + n) = \mathcal{O}(2m + n) \in \mathcal{O}(m + n)$$

for $n = |V|, m = |E|$.

2.5.3 Algorithmic Connectivity

To understand connectivity in a graph, we construct a **spanning tree**.

> *Definition.* **Spanning Tree**
> A **Spanning Tree** is a subgraph that includes all the vertices of the graph, and is a tree.
>
> ■

We specify such tree by giving the parent $\text{pr}[v]$ of every vertex v with $\text{pr}[\text{root}] = \emptyset$.

Generic Algorithm for constructing a spanning tree for the component rooted at S.

```
1 Let T be the graph with one vertex, s, with pr[s] = ∅.
2
3 While there is an edge {u, v} joining a vertex u of T with a vertex v
4 not in T.
5
6   Add vertex v and edge {u, v} to T
7   Set pr[v] = u
```

Claim: The resulting graph is always a tree.

Lemma.

This tree spans the component containing s.

> **Proof.**
> Any vertex in this tree has a path to s: repeatedly take parents until they lead to s. There is a path between any two vertices u and v in the tree: consider paths from u to s and v to s: join them to get a path from u to v. On the other hand, if u is not in T, then there is no path from u to s. If there were, would could follow the path and find an edge from a vertex not in T to a vertex in T. But no such edge can remain when the algorithm terminates.
>
> ■

2.5.4 Bipartiteness

This is an algorithmic application of BFS.

Question. In the search tree of the graph, which edges don't show up in the original graph?

> **Note**.
>
> The only non-tree edges that we could possibly have are between vertices of the same layer, or adjacent layers. After all if it were not, the vertex would be higher in the search tree.

Lemma.

Let T be a BFS tree. and let u and v be vertices of T with u in layer L_i and v in layer L_j. Suppose that $\{u, v\}$ is an edge of graph G. Then $|i - j| \leq 1$.

Proof.

Suppose WLOG that u is added before v. Consider the two following cases,

1. v is already in the tree when u becomes active (i.e. you are looking at its neighbors).

   Then $\{u, v\}$ is a non-tree edge, and $pr[v]$ (the parent of v) must have joined the tree before u, s $j = \text{layer}(pr[v]) + 1 \leq i + 1$.

2. v is not in the tree when u becomes active.

   Then $\{u, v\}$ is a tree edge and $j = i + 1$.

This is a nice property of BFS, and it's useful to tell if a graph is **bipartite**.

■

Definition. **Bipartiteness**

We say that a Graph $G = (V, E)$ is **Bipartite** if there is a partition (A, B) of $V = A \cup B$ with $A \cap B = \emptyset$, such that for all edges $\{u, v\} \in E$, either $u \in A$ and $v \in B$ or $v \in B$ and $v \in A$.

Author Note.

One way to think of bipartiteness is to color each vertex. Start with any vertex of the graph and paint it blue, paint all of its neighbors red, then paint each neighbor of *those* vertices blue again, etc...

Do this until the whole graph is painted. If at any point you encounter a colored vertex which you must change the color of, that means that the graph has an odd cycle, and so it isn't bipartite.

■

Theorem.

A connected graph with BFS tree T is bipartite if and only if there is no non-tree edge joining vertices in the same layer of T.

Proof.

($\Leftarrow$)

Consider the bipartition $A = L_0 \cup L_2 \cup \cdots$, and $B = L_1 \cup L_3 \cup \cdots$.

This shows that the graph is bipartite since all non-tree edges join vertices whose layers differ by one.

Also, the tree edges always join vertices in adjacent layers.

($\Rightarrow$)

Suppose that $\{u, v\}$ is a non-tree edge between vertices u and v in the same layer of the BFS tree. Suppose their *nearest common ancestor* is m layers higher. Then there is a path from u to v in the BFS search tree, and the length of that path is $2m$ (m on the way up, m on the way down). Adding edge $\{u, v\}$ gives a cycle of length $2m + 1$.

Claim. A cycle of odd length cannot be bipartite.

Proof of Claim. Indexing the adjacent vertices by 1, 2, ..., $2n + 1$, to be bipartite, the odd vertices will be

on one side, and the even vertices will be on the other side. But 1 and $2m+1$ are both odd, and connected, so the edge between them joins vertices on the same side.

■

2.5.5 Connectivity and Directed Graphs

Question: Given a directed graph G, how could we tell if it is connected?

One approach is to run BFS on every vertex, but the runtime of that would be $\mathcal{O}(n(m+n))$, which (as we'll see) is wasteful.

Lemma.

If vertices u and v are *mutually reachable*, and v and w are mutually reachable, then u, w are mutually reachable.

Author Note.

In other words, reachability is transitive.

Proof.
Go from u to v, and then v to w. Similarly, go from w to v, and v to u.

$$u \to \cdots \to v \to \cdots \to w$$

■

To tell if a graph is **strongly connected**, fix a vertex s and construct

- BFS starting from s
- BFS starting from s, reversing the direction of all edges.

The graph is strongly connected iff both searches reach the *entire graph*.

> **Note**.
>
> A Strongly Connected graph doesn't need to be complete, you just need to be able to get from any vertex to any other, but maybe not directly.

This is surprising because to tell if a graph is strongly connected, you only need to run BFS twice!

2.6 Depth First Search

Main Idea: Explore the graph in order of increasing distance from s.

Say a vertex is exhausted if it is not adjacent to any vertex outside the tree.

2.6.1 Pseudocode

For a graph G, starting vertex s, and a function f, we have

```
1  DepthFirstSearch(G, s, f):
2    # initialize a stack with the starting vertex
3    stack = [s]
4    visited = ∅
5
6    while stack is not empty
7      v = pop(stack)
8
9      # perform action on v
10     f(v)
11
12     for all neighbors u of v:
13       if u ∉ visited:
14         push u to stack
15         add u to visited
16       end
17     end
18   end
19 end
```

Note.

The pseudocode for `DepthFirstSearch` is almost identical to `BreadthFirstSearch`, the only difference is the use of a `stack` data structure where a `queue` data structure was used for `BreadthFirstSearch`.

3 Trees

For this section we talk about undirected graphs.

Definition. **Forest**

A **Forest** is a graph with no cycles.

■

Definition. **Tree**

A **Tree** is a connected forest.

■

Note.

These definitions stem from the fact that graphs can be disconnected.

Note.

All trees are forests.

An n vertex tree has $m = n - 1$, where $n = |V|$, and $m = |E|$.

Definition. **Parent & Children Vertices**

If we designate one vertex of a tree as the root, then the unique neighbor of any vertex that is closer to the root is called the **parent**, and the other neighbors are called its **children**.

■

Definition. **Leaf**

A **Leaf** of a tree is a degree-1 vertex.

For a tree, a leaf is at the bottom and has no children.

■

Definition. **Directed Acyclic Graph**

A Digraph with no *directed cycles* is called a **Directed Acyclic Graph**, or **DAG**.

■

Note.

Any vertex in a tree can be the root!

4 Topological Sorting

Definition. **Topological Ordering**

A **Topological Ordering** is an order in which we can perform the operations sequentially so that all required inputs are available when an operation is performed.

■

A typical example of this is **boolean circuits**. You can evaluate any of the first layer, in any order. But you can't start from the end, because the later operations depend on the previous ones.

Lemma.

If G has a topological ordering, then G is a DAG.

Proof.

By contradiction, suppose that G has a topological ordering $v_1, v_2, ..., v_n$, and has a directed cycle C.

Let v_i be the lowest index vertex on C. Let v_j be the vertex just before v_i on C. Then (v_j, v_i) is an edge with $1 \leq i < j \leq n$, but we must have $1 \leq j < i \leq n$ in a topological ordering since (v_j, v_i) is an edge.

Author Note.

The idea here is that, we need input v_j to perform input v_i (since $j < i$ in the ordering) but we also need v_i to perform v_j because the vertices are in a cycle. Therefore nothing can ever be done, and so this is not a topological ordering.

■

Now we'll show that every DAG has a topological ordering, and how to find it.

Lemma.

Every DAG has a vertex with indegree 0.

Proof.

By contrapositive, If every vertex has positive in degree, then there is an directed cycle.

Start from any vertex. Walk along some edge in the backward direction (possible since the in degree is positive). After at most n steps, we must have visited some vertex twice, so we must have visited some directed cycle.

■

Lemma.

Every DAG has a topological ordering

Proof.

By induction.

Base Case. A 1-vertex graph is a DAG.
Inductive Hypothesis.

Assume the claim holds for any DAG with at most n vertices.
Inductive Step.
Given an $(n+1)$ vertex DAG, find an in degree 0 vertex, and delete it (and associated edges).

The resulting graph is an n vertex DAG, since deletion can't create a cycle. So the claim follows by induction.

■

Note.

The order in which we delete produces a topological ordering.

Note.

Topological orderings are **not** unique.

4.1 Pseudocode

This code checks if a graph is a DAG. This is done by adding every vertex with out degree 0 to a set S. Then for each member v of S, it checks all neighboring vertices u while removing edge $\{v, u\}$. If u then has no edges pointing to it, it is added to S as well.

This process is repeated until termination, when S becomes empty. Finally, G is checked and if vertices remain, then we know that G is not a DAG.

```
1  TopoSort(G):
2    # this set contains all vertices with indegree 0
3    let S be an empty set
4
5    for all vertices v:
6      # suppose easy access to this information,
7      # can be done through pre-processing
8      if v has indegree 0:
9        add v to S
10
11     # we keep track of the indegree
12     # of v through count[v]
13     set count[v] = indeg(v)
14
15   while S is not empty:
16     remove a vertex v from S
17     output v
18
19     for each vertex u that v points to:
20       remove edge {v,u} from G
21
22       # since we removed an edge going into u
23       decrement count[u]
24
25       # if nothing now points to u
```

```
26        if count[u] = 0:
27          add u to S
28
29    if G is nonempty:
30      # the graph is not a DAG
31      return error
```

4.1.1 Running Time

It takes $\mathcal{O}(n)$ to initialize the graph, and get information about the in degree of each vertex v of G.

Each operation in the `for` loop then takes time $\mathcal{O}((\text{outdeg}(v)) + 1)$ so we get the total runtime

$$\mathcal{O}(n + \sum_{v \in V}(\text{outdeg}(v) + 1)) = O(n + m + 1) \in \mathcal{O}(n + m)$$

5 Greedy Algorithms

Definition. **Greedy Algorithm**

A **Greedy Algorithm** is an algorithm that makes a sequence of small choices, each of which is locally as good as possible.

■

Greedy algorithms aren't always the best. There might be overall solutions that are better than what a greedy algorithm can give. However for certain well behaved problems, they can be very good.

Sometimes, depending on what you're greedy about, they can be really good, or not that great.

5.1 Making change

Suppose that you have the following coin denominations.

$$c_1, c_2, \ldots, c_n$$

Given a value v find a way to express v using as few coins as possible. You can use multiple coins of the same denomination.

5.1.1 Pseudocode

```
Change(v):
  if v = 0:
    return

  let c be the largest coin of denomination less than v

  add c to the list of coins

  Change(v - c)
```

Note.

This algorithm doesn't always produce the best solution.

For instance, consider the coins $1, 3, 4$, and you want to make change for 6. Then the algorithm would return $4, 1, 1$, even though $3, 3$ would be a better solution.

Sometimes, the algorithm *is* optimal, for instance consider the same example as above, but make change for 5 instead.

Consider the example with coins

$$1, 5, 10, 25, 100$$

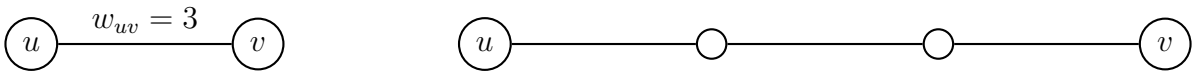
Then, it is always possible to find the optimal solution.

5.2 Shortest Path

Recall that BFS finds the shortest paths in unweighted graphs. What if there is a weight w_{uv} for each edge $(u, v) \in E$?

The length of a path is the sum of the edge weights.

> **Note**.
>
> Suppose that our Graph lives in a **metric space**. Informally, this is to say that you can't get from vertex u to v faster than directly. It's a discrete version of the triangle inequality.



Let's look at an alternative...

5.2.1 Dijkstra's Algorithm

Let G be our graph, and s be our starting vertex. Dijkstra's algorithm finds the minimum distance of every node to s.

```
Dijkstra(G, s):
  # we break our graph G up into its
  # edge set E and vertex set V
  let E, V = G

  # d[v] represents the distance of vertex v
  # from s, so the distance from s to s is 0
  let d[s] = 0

  # this is the parent of s
  let pr[s] = ∅

  # we initialize all vertices that aren't s as
  # being infinitely far, since we don't know how
  # far they are
  for all v ∈ V \ {s}:
    d[v] = ∞

  mark all vertices as unexplored

  while ∃ unexplored vertices:
    let u be the unexplored vertex with smallest d[u]

    for each neighbor v of u:
      let w_uv ∈ E be the weight of the edge connecting u to v

      if d[u] + w_uv < d[v]:
        # this is a better upper bound
        d[v] = d[u] + w_uv
        pr[v] = u
```

```
      end if
    end for

    marked u as explored
  end while
```

5.2.2 Proof of Dijkstra's algorithm

Note.

Dijkstra's algorithm requires all edge weights to be **positive**.

Lemma.

When a vertex v is explored, $d[v]$ is the distance from s to v.

Proof.

By induction on the number of explored vertices.

By induction on the number of explored vertices. Clearly, it holds when only s is explored.

Suppose that the lemma holds when there are k explored vertices. Let v be the $(k+1)^{\text{st}}$ explored vertex, and let $\text{pr}[v] = u$.

Suppose for the sake of contradiction, that the shortest path from s to v does not go through u. Instead, let $x \neq u$ be the last explored vertex on the shortest path, and let y be its unexplored neighbor on this path.

Note that $y \neq v$, otherwise the algorithm would choose $u = x$.

$$\begin{aligned}
\text{dist}(s,v) &< d[v] \\
&\leq d[y] && \text{since we chose to add } v \text{ before } y \\
&= \text{length of a path from } s \text{ to } y \text{ found} \\
&\leq \text{dist}(s,v) && \text{since, by contradiction, } y \text{ is on the shorted path from } s \text{ to } v
\end{aligned}$$

But this is a contradiction, so the shorted path from s to v must go through u, and it must be the case that

$$\text{dist}(s,v) = d[u] + w_{uv} = d[v]$$

■

5.2.3 Running Time

The running time of Dijkstra's algorithm is very similar to the other algorithms we've seen so far. It can be shown that the running time is $\mathcal{O}(m+n)$.

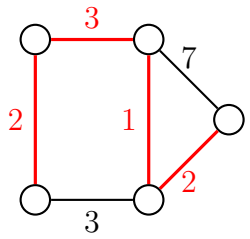
5.3 Minimum Spanning Trees

We're interested in finding a spanning tree for a given graph. But now we don't want to find the distance between any two vertices, but we want to *minimize the total minimum weight of the edges in the tree.*

These weights just have to be real numbers, they don't have to be positive.

If we want to have a communications networks between nodes, this might be applicable. The edge weight might represent the cost of building that cable, or the distance, or anything else that you want.

Example



In this graph, the red edges form a minimum spanning tree. Note that there can be multiple minimum spanning trees from a single graph.

Question

How do we know which edges to add to our tree?

Definition. **Cut**

A **Cut** is a *bipartition* (a partition into two sets) of the vertices, $V = A \cup B$ with $A \cap B = \emptyset$.

A cut is **non-trivial** if both A and B are non-empty.

An edge **crosses** the cut if it has one end in A and one end in B.

■

Lemma.

Assume that all edge weights are distinct. Let (A, B) be a non-trivial cut, and let e be the minimum weight edge crossing this cut. Then, any minimum spanning tree must contain e.

We will prove this using an *exchange argument* (we will use this again a lot for greedy algorithms as well as scheduling algorithms.)

Proof.

Let T' be a spanning tree that does *not* contain $e = \{u, v\}$. Since T' is spanning, there is *a unique, simple* path from u to v.

Since e crosses the cut, there must be *at least* one edge along the path from u to v that crosses the cut (if there is more than one, that's fine), call this edge e'.

At this point, our graph is partitioned into two parts, A, and B. e is not in the MST, but e′ is. Both e and e′ cross from A to B. What we're gonna do here is swap edge e and e′

Construct a new spanning tree T from T' by deleting e' and adding e.

This is still a tree since the number of vertices is n, the number of edges is $n-1$, and the graph is fully connected. Therefore it cannot contain any cycles.

Since the weight of e is less than the weight of e', this change lowers the total edge weight of the spanning tree.

■

Using this lemma, let's now look at various algorithms for finding MSTs.

5.3.1 Kruskal's Algorithm

Main Idea: *Add the lowest cost edge that doesn't create a cycle.*

Process

1. Sort all edges in G by weight, lowest to highest.
2. Iterate through sorted edges, add it to the Minimum Spanning Tree **if** it doesn't create a cycle.

Correctness

Theorem.

Kruskal's Algorithm always outputs a Minimum Spanning Tree.

Proof.
Suppose that the algorithm adds the edge $e = \{u, v\}$ to the forest F. Consider the cut induced by the component of u in F.

One side of this cut is everything connected to u, and the other side is everything connected to v.

Clearly, e crosses this cut and, by definition, it is the minimum weight edge with that property. So by the Lemma, any MST must include e.

Note that this works at any stage along the algorithm. So the only thing that we need now is to show that the output is a spanning tree, and if it is, it will automatically be minimal.

Note also that it clearly does not create a cycle. If the graph were not connected, then it could add some edge without creating a cycle, so the output must be a tree.

■

Let's now look at another way to produce Minimum Spanning Trees: Prim's algorithm.

Running Time

For Kruskal's algorithm, a *union-find* data structure is helpful. If done properly, the running time can be $\mathcal{O}(m \log n)$.

5.3.2 Prim's Algorithm

Main Idea: *Chose a root to grow a Spanning Tree from.*

Process

1. Chose a starting vertex. This is the root of the Minimum Spanning Tree.
2. Look at the set of all edges connected to the nodes in our Minimum Spanning Tree.
3. From these edges, only consider the ones that point to vertices that are **not** already in our MST. As you go, mark vertices as explored.
4. Of these edges, add the one with lowest weight.

Repeat this process until all vertices are explored.

Correctness

Theorem.

Prim's algorithm outputs a MST.

Proof.
Suppose the algorithm adds edge $e = \{u, v\}$ to the forest F. Consider the cut induced by component of the root (from which the algorithm builds the tree).

Clearly e crosses this cut and, by definition, it is the minimum weight edge with that property.

So by the Lemma, every MST contains e and so clearly, Prim's algorithm outputs a Spanning Tree.

■

Note.

Both of these algorithms pertain to undirected graphs. The graphs *could* have cycles.

Both of these algorithms are also *greedy*.

Implementing these algorithms efficiently require choosing good data structures.

Running Time

For Prim's algorithm, you often want a *priority queue* to store edges of the graph. If done properly, a running time of $\mathcal{O}(m \log n)$ is possible.

5.4 Scheduling

You have to be very careful in how you're greedy to get an algorithm that works.

This is another set of greedy algorithms.

Note.

Midterm

Divide and conquer might be the last topic before the midterm.

We'll look at 3 different examples of scheduling algorithms.

5.4.1 Interval Scheduling

Which classes should be in which rooms.

For now, suppose that we have one room, and a bunch of requests for it. How many classes can you schedule.

Suppose that you are given n request, which come with starting times s_i and finishing times f_i for an $1 \leq i \leq n$. Note that $f_i \leq s_i$.

We cannot hold two classes at the same time. A subset of requests is **compatible** if no two of the intervals overlap in time.

Goal

Our goal is to hold as many classes as possible, and the duration of a class does not matter to us (a 10 minute class is just as valuable as a 90 minute class).

Let's look at an example

> **Note**.
>
> In general, *solutions may not be unique, but we only need to find one.*

How do we write such an algorithm? We want to find the *best* solution. What's a natural way to be greedy here? In what order should we look at the intervals? This is going to matter a lot.

Idea 1.

What if we try to think about the shortest activities first? This is actually sub-optimal, If you have a really long task and another really long task, with a really short task right between the two long tasks, then you're not doing as best as you could.

Idea 2.

What if we try to find the activity with the minimum number of conflicts first? This is very natural, but not optimal. You can give a counter example to this, even if it's not easy.

Idea 3.

What if we sort activities by starting time? This is sub-optimal. What if the earliest starting interval finishes last? What if it knocks out 10 other intervals that could have been better?

Idea 4.

What if we consider each interval on its own, and then find the earliest finishing one? This is optimal! (among potentially many other solutions).

Find the earliest activity first,

Pseudocode

```
GreedyIntervalSchedule(s, f):
  sort tasks by increasing order of finishing times.
  # note that you will need to sort s in the same way as f

  let A = ∅
  let f_prev = −∞

  for i in 1 to n:
    # this interval starts after the previous interval finishes it doesn't
    # conflict, so add it
```

```
    if s_i > f_prev:
      add task i to A
      set f_prev = f_i
    end
  end
end
```

This is clearly $\mathcal{O}(n \log n)$ for sorting, since everything else is constant time.

Proof of Correctness

Why does this work?

1. There are no conflicts, since we only schedule a task that starts after the current one ends.
2. We'll show that the number of tasks is maximal by an **exchange argument**. We can also show that this algorithm stays ahead (see book.)

Theorem.

`GreedyIntervalSchedule` outputs an optimal schedule.

Proof.
Let $G = (g_1, g_2, ..., g_k)$ be the greedy schedule, with indices representing tasks in order of increasing finish time.

Let $B = (b_1, ..., b_l)$ be an optimal schedule, with $l \geq k$.

Let j be the first index where the schedules differ

$$g_1 = b_1, \ldots, g_j = b_j, g_{j+1} \neq b_{j+1} \ldots$$

Now switch $B' = (g_1, \ldots, g_{j-1}, g_j, b_{j+1}, \ldots, b_l)$. By the greedy choice, we know that the finishing time of $g_j \leq b_j$, so there are no conflicts, and it is just as long: l intervals.

Author Note.

This is the essence of the exchange argument. We started with a perfect schedule and demonstrated that we could exchange it inductively and retain an optimal schedule.

Repeating this process, we get a schedule

$$(g_1, \ldots, g_k, b_{k+1}, \ldots, b_l)$$

If $l > k$, then b_{k+1} is the index of some interval that the greedy algorithm could have scheduled, but it didn't which is a contradiction, so l must equal k.

Therefore the greedy algorithm is optimal.

■

Note.

Solutions to the greedy algorithm may not be unique! It's possible to use this exchange argument to produce

different optimal schedules.

5.4.2 Interval Partitioning

Suppose that we must schedule all intervals, given again by starting and finishing times, while minimizing the number of resources used (in this case, the number of classrooms, where each classroom is the same size).

What is the minimum number of classes necessary to accommodate all schedules?

Definition. **Interval Depth**
The **depth** of a set of intervals is the maximum

$$\max_t \Big\{ i \in \{1, \ldots, n\} : t \in [s_i, f_i] \Big\}$$

In other words, for every $1 \leq i \leq n$, this is the maximum number of overlapping intervals.

Which is the largest number of overlapping subsets.

■

Note.

The depth is *always* the optimal solution. It's possible to always reach it.

Greedy Approach

Scan through intervals in increasing order of start time. Assign each to any available resource from

$$\{1, \ldots, \text{depth}\}$$

Theorem.

This algorithm assigns a color from $\{1, \ldots, \text{depth}\}$ to every interval and no two overlapping intervals have the same color.

Proof.
When we assign a color, it is, by definition of the algorithm, different from the colors of all the overlapping intervals that are already scheduled. So because of that, we never assign the same color to overlapping intervals.

Suppose that, when we are considering the i^{th} interval, it overlaps t previous intervals. So, in the input, there exists $t + 1$ overlapping intervals. So the depth is at least $t + 1$, but conversely, that tells us that t is at most depth $- 1$. So there is always a color available.

■

The pseudocode for this algorithm was not shown in class, but it was mentioned that optimal runtime was $\mathcal{O}(n \log n)$.

5.5 Schedule to Minimize Lateness

We have a number of schedules, and only one room. We have to schedule all of them. Each schedule is defined by a duration and a deadline. There's some time by which we would like to finish the request. There is not specified starting time. Let's formalize this.

Setup: We are given tasks with duration t_i, and deadline d_i.

It might not be possible to always satisfy each deadline, but we want to minimize lateness.

If we schedule job i from the interval $[s_i, f_i]$ with $f_i = s_i + t_i$. We say that the lateness of job i, called l_i is defined as follows

$$l_i = \max(0, f_i - d_i)$$

In other words, it's either 0 if the job finishes before its deadline, or some positive real number otherwise.

Concretely, our goal is to schedule all jobs without overlaps, while minimizing the worst case lateness defined as $\max_i(l_i)$.

> **Note**.
>
> Here we are not trying to minimize *total* lateness, just the *worst* lateness.

Let's look at an example

Example

Let's look at the following jobs.

$J = \{(t_1 = 1, d_1 = 2), (t_2 = 3, d_2 = 1)\}$

> It's always in our best interest to have a job running at any time if we are free.
>
> From this job list, we can chose to run the jobs in the following order
>
> $1, 2$, for which the lateness would be $l_1 = 0, l_2 = 3$ or $2, 1$ for which the lateness would be $l_1 = 2, l_2 = 2$
>
> Since the worst case lateness in the second ordering is only 2, the second ordering is better.
>
> ■

We'll solve this problem using a greedy strategy. Let's first take a look at some natural approaches we could take for this problem.

Approaches

1. Go in order of increasing deadline. Start with the job with the earliest deadline d_i. This *will* actually prove to be optimal!
2. Go in order of shortest duration t_i first. If you have some jobs you could get done quickly, you might want to get those out of the way first. Unfortunately this is can be shown not to be optimal

   Consider the following jobs

   $J = \{(t_1 = 0, d_1 = 100), (t_2 = 10, d_1 = 10)\}$

Then we would do t_1 first, and then t_2, which would yield a max lateness of 10, whereas doing t_2 and then t_1 would yield 0 lateness.

3. Go in order of "slack": How much extra time remains after a job finishes? Jobs that give you the highest amount of slack sound really good to do first, surely. It can be shown that this is also not optimal.

What this illustrates is that the *way* in which we are greedy about a problem matters a lot. There are many things that sound sensible to do, but actually lead to sub-optimal solutions. It's worth taking the time to try to figure out what to be greedy about.

We'll now look into the first approach, and prove that it is optimal by an exchange argument.

To help understand why this is optimal, we'll first introduce the concept of an *inversion*.

Definition.

A schedule has an **Inversion** if job i comes before job j, but it is also the case that $d_i > d_j$.

Recall that our approach is to sort by increasing deadline. If in our ordering we have two jobs whose deadlines behave in the way defined above, it is what we call an inversion.

In our optimal schedule, we want no inversion.

Lemma.

All schedules with no inversions (there can be more than one!) and no idle time (this is assumed, just remove the idle time otherwise) have the same max lateness.

Proof.

Author Note.

How can this happen? Well let's first ask how we can have two different schedules with no inversions. This can only be the case if two jobs have the same deadline! After all if they did not, they would be in some order according to our greedy strategy.

The only flexibility in such a schedule is in the order of jobs with the same deadline. Moreover, any such jobs would be ordered consecutively. The finishing time of the last of these jobs doesn't depend on the ordering.

■

Lemma.

There is an optimal schedule with no inversions (and no idle time).

Proof.

We'll prove this using an exchange argument.

Goal. We'll suppose that we have an optimal schedule which *has* inversions, then we'll show that we can exchange out such inversions without losing quality.

If there is an inversion, then there must be jobs i, j which are adjacent (with i before j WLOG) with $d_i < d_j$. In other words, if we have an inversion (adjacent or not), then we *must* have an adjacent inversion. You can reason about this by contradiction, but the argument is a sort of intermediate value theorem.

Now we'll swap the two jobs and see what happens to the lateness. This decreases the total number of inversions. The claim now is that the max lateness is no worse.

Swapping i and j gives a schedule with one less inversion, and we claim that there is no worse lateness.

Why is the lateness no worse? Well suppose we have the following scenario. Job i comes before job j, and $d_j < d_i$ come before the start of job i. Then, it's clear that that the max lateness is driven by $f_j - d_j$.

Swapping i and j, the lateness is

$$\max(f_j - d_i, (s_i + f_j - s_j) - d_j) = \max(f_j - d_i, f_j - d_j - (s_j - s_i))$$

but $s_j - s_i \geq 0$. Clearly the lateness we get by doing this is going to be $l \leq f_j - d_j$ since $d_i > d_j$ and $s_j - s_i \geq 0$.

Repeating this, we eventually get a schedule with no inversions and no worse lateness.

■

Side Note.

> Exam one week from today. No calculator needed, can bring notes. For proofs, you don't need to write five pages. See solutions for examples.
>
> Format of the exam: similar to practice problems. Suggestion: make a question out of three short answer questions. Longer questions are problems are gonna be like homework questions. Proving mathematical fact, run algorithm that you know on some data, etc...
>
> Look at practice problems!
>
> Assignment 2 should be graded a bit before exam 2
>
> ■

6 Divide and Conquer

This is the last topic on the exam

Main Idea: Divide into sub problems, usually 2, but not necessarily. We' solve the sub problems recursively, and merge the solutions, maybe with some additional work, to solve the overall problem.

6.1 Merge Sort

To sort a list of length n

- Divide the list in half
- Sort each half, recursively, using this exact procedure.
- Merge the two sorted lists. You can do this in linear time! This is what makes Merge Sort great.

6.1.1 Running Time

Let $T(n)$ be the running time on instances of length n. Let's write a recurrence equation for Merge Sort.

$$T(n) = 2 \cdot T(n/2) + \mathcal{O}(n),\ T(1) = \mathcal{O}(1)$$

> **Note**.
>
> The first part of the recurrence represents the work necessary to complete Merge Sort on a list of length $n/2$, which there are 2 of. The second part represents the work necessary to merge the two lists together.

Solving this recurrence, we get that $T(n) = \mathcal{O}(n \log n)$.

Question: What if n isn't even?

It's important to note what we're doing here. The recurrence equation we described above represents an *upper bound* for the running time of Merge Sort. If the two lists aren't the same size, then the time needed to sort each will just be bounded above by the time necessary to sort twice the larger one, and merge twice the larger one. If the lists aren't the same size, the time can only be better.

Not *all* divide and conquer algorithms are $\mathcal{O}(n \log n)$. Recall Karatsuba multiplication, which is also divide and conquer. This algorithm was still $\mathcal{O}(n^2)$, but just had a better constant term.

6.1.2 Pseudocode

```
MergeSort(L):
  # get the size of the list
  let n = |L|

  # the list is length 1 or less
  if n ≤ 1:
    return L

  # get the lower midpoint of the list
  let m = ⌊n/2⌋

  # get the sorted left and right sublists
  let l = MergeSort(L[0 to m])
  let r = MergeSort(L[m to n])

  return Merge(l, r)
end
```

Where L is our list, and `Merge` is defined as

```
Merge(l, r):
  let S = []
  let i = j = 0

  while i < |l| and j < |r|:
    if l[i] ≤ r[j]:
      S.push(l[i])
      i++
    else:
      S.push(r[j])
      j++

  # in case the two lists aren't the same size
  # only one of these loops will be entered
  while i < |l|:
      S.push(l[i])
      i++
  while j < |r|:
      S.push(r[j])
      j++

  return S
end
```

6.2 Closest Points

Suppose we have n points in in the plane, and we want to find the two closest points.

6.2.1 One Dimensional Case

We explored a simpler version of this problem in one dimension. Suppose that instead the points are in a line, how do you find the closest two? If the points are given in arbitrary order you have to sort, and then go through them from left to right.

In one dimension, you sort by x coordinate but you can't just do that in 2 dimensions. Still, we can sort the points in *some* way that can help. Once the points are sorted, we can work with them in some other way.

Let's define the problem better.

6.2.2 Two Dimensional Case

Suppose that we are given n points in a plane, in no order whatsoever, and no two points overlap.

Main Idea: We could recursively subdivide the points into left and right based on the *median* x coordinate of the points, and divide left and right. Then we solve the closest pair problem on each half.

> **Note**.
>
> The closest pair of points *could* be across the line, our algorithm needs to consider this. In other words we want to check whether there is a closer pair with one on the left and one on the right.

As we recurse, we maintain lists of the points in our subsets sorted by x and y coordinate. We store this is in 2 separate lists by just sorting the indices on x and y.

Let's define some notation.

Suppose that the input points $P = \{p_1, \ldots, p_n\}$ where $p_i = (x_i, y_i)$.

- Let Q be the points in P with the first $\lceil n/2 \rceil$ x coordinates, and let R be the points in P with the last $\lfloor n/2 \rfloor$ x coordinates, in other words: the remaining points.
- We recursively find the closest points in Q, call them q_0^*, q_1^*, and R, call then r_0^*, r_1^*.
- Let δ be the minimum between $\text{dist}(q_0^*, q_1^*)$, and $\text{dist}(r_0^*, r_1^*)$.

  In other words, δ represents the distance between the two closest points either on the left side, or the right side of the line.
- Let x^* be the largest x coordinate of a point in Q.

  In other words, x^* is the point closest to the line on the side of Q. In fact, the line crosses over the point x^*.
- Let L be the line $x = x^*$.

Author Note.

So what we're doing here is that we look at every point in Q and R, and find the closest two and we define δ based on this distance.

But what if the closest two points are actually between Q and R? Let's explore this possibility in the next lemma.

Lemma. If there is a $q \in Q$ and $r \in R$ with $\text{dist}(q, r) < \delta$, then both q and r are within δ of L.

What this tells us is that we can restrict attention to a strip of length 2δ centered on L. This is nice, but keep in mind that all points could be in that strip.

Lemma.

If two points in S have distance less than δ, then they're within 15 positions of each other in the list of points sorted by y coordinate.

Why 15? Who knows! Its just some weird constant not explained in class. Just believe in the Lemma.

Proof.
Divide S into squares of side length $\frac{\delta}{2}$.

Claim. No two points can be in the same square. If they were, then δ would be smaller by definition of δ.

Suppose that two points in S are 16 or more positions apart in the list sorted by y coordinate. Then they have to differ by *at least* 3 rows of boxes, so they have distance at least

$$3 \cdot \frac{\delta}{2} > \delta$$

In other words, *because* their distance is greater than δ, then there *must* be *at least* 3 empty boxes between them.

■

6.2.3 Running Time

The cost of finding the two closest points is $T(n/2)$. Since there are two sides we have cost

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

Where $\mathcal{O}(n)$ is the cost of merging the two sides, and since the lists are sorted, the cost of this is linear.

6.3 Fast Fourier Transform

This is one of the best algorithms of all time

This is a great way to multiply polynomials, but it's also used in signal processing, and many other fields. For now, we'll restrict our problem to the following.

Suppose that we are given a polynomial $a(x), b(x)$ of degree $n - 1$

$$a(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

and

$$b(x) = b_0 + b_1 x + \cdots + b_{n-1} x^{n-1}$$

Suppose now that $c(x)$ is defined as $c(x) = a(x) \cdot b(x)$. We'll see how to multiply this next time.

Side Note.

> Everything about Divide and Conquer is on the exam. Dynamic Programming is not.
>
> ■

Side Note.

> The Fast Fourier transform uses some linear algebra. You aren't really responsible for this part on the exam, just know how exactly the task is divided.
>
> ■

We've been talking about Divide and Conquer algorithms, we looked at the closest points problem and we started talking about polynomial multiplication, which will motivate the Fast Fourier Transform.

Recall polynomial multiplication, we can suppose that the coefficients are in $\mathbb{R}$, or even $\mathbb{C}$ if we wanted to.

$$a(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$$

and

$$b(x) = b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}$$

Suppose now that $c(x)$ is defined as $c(x) = a(x)b(x)$.

What is $c(x)$?

$$\begin{aligned}
c(x) =& a(x) \cdot b(b) \\
=& \left(\sum_{j=0}^{n-1} a_j \cdot x^j\right)\left(\sum_{j=0}^{n-1} b_j \cdot x^j\right) \\
=& \sum_{j,k=0}^{n-1} a_j \cdot b_k \cdot x^{j+k} && \text{Can combine the sums} \\
=& \sum_{l=0}^{2n-2}\left(\sum_{j=0}^{l} a_j \cdot b_{l-j}\right)x^l && l = j+k, \text{ both go up to } n-1 \\
=& \sum_{l=0}^{2n-2} c_l x^l && \text{Define } c_l = \sum_{j=0}^{l} a_j \cdot b_{l-j}
\end{aligned}$$

Note.

If both $a(x)$ and $b(x)$ are both degree $n-1$, $c(x) = a(x)b(x)$ will be of degree $l = 2n-2$.

More generally, if $a(x)$ is degree n and $b(x)$ is degree m, then $c(x) = a(x)b(x)$ is degree nm.

What we can gather from this is that, if we were to compute this manually, it would be $\mathcal{O}(n^2)$. However using Divide and Conquer can really help us here.

Main Idea: Use polynomial Interpolation.

A degree d polynomial can be uniquely specified by its values at any $d+1$ points. Not necessarily its roots, but *any* points on that polynomial

Note.

If you plot 2 two points, there's a unique line that goes through them. If you plot 3 points, there is a unique parabola that goes through them. If you plot 4 points, there is a unique cubic, etc...

More generally, given n points, there is a unique polynomial of degree $n+1$ that goes through it.

This is a linear amount of work! If we can somehow divide and conquer on this, we have an algorithm. Let's devise a strategy.

6.3.1 Strategy

1. Evaluate the polynomials a and b on $2n-1$ points. Somehow, we have to do this in close to linear time.

   We'll have to pick these points carefully!

   Note.

   You need $2n-1$ points because $c(x)$ is degree $2n-2$, and you need one more point to identify c from that, from the previous note.

2. Evaluate $c(x)$ on those points.
3. Reconstruct the coefficients of $c(x)$ from the points.

Author Note.

We'll see that (1) and (3) are actually the same problems in reverse!

Now let's dive deeper into each step.

6.3.2 Process

1. *Evaluate a, b on $2n-1$ points*

   First we compute $\mathcal{O}(n)$ points on a and b, each of which takes $O(n)$ time to evaluate individually, since both polynomials have $\mathcal{O}(n)$ terms.

   Consider evaluating a polynomial of degree $d-1$ at d points $x_0, x_1, \ldots, x_{d-1}$

   $$a(x_j) = a_0 + a_1 x_j + \cdots + a_{d-1} x_j^{d-1}$$

   Assume d is even for simplicity. Define

$$\begin{aligned} a_{\text{even}}(x) =& a_0 + a_2 x + a_4 x^2 + \cdots + a_{d-2} x^{(d/2)-1} \\ a_{\text{odd}}(x) =& a_1 + a_3 x + a_5 x^2 + \cdots + a_{d-1} x^{(d/2)-1} \end{aligned}$$

Then we have

$$a(x) = a_{\text{even}}(x^2) + x \cdot a_{\text{odd}}(x^2)$$

Each of these polynomials are half the size.

Picking points carefully

Consider the points $1, -1, i, -i$. Squaring these points, we get $1, 1, -1, -1$. Squaring again, we get $1, 1, 1, 1$.

The natural choice for these points are the d^{th} **roots of unity**!

$$x_j = \omega_d^j$$

Where $\omega_d = e^{\frac{2\pi i}{d}}$.

Note.

If we square d^{th} roots of unity, we get half as many points.

Let's see this in action.

$x_j = \omega_d^j$. Squaring, we get

$$\begin{aligned} x_j^2 =& \omega_d^{2j} \\ =& e^{\frac{2\pi i}{d}(2j)} \\ =& e^{\frac{2\pi i}{d}(2j) \pmod d} \\ =& \omega_d^{2j \pmod d} \\ =& x_{2j \pmod d} \end{aligned}$$

If $T(d)$ is the cost of evaluating $a(x)$ at x_j for $j \in \{0, 1, \ldots, d-1\}$, then we have

$$\begin{aligned} T(d) =& 2T(d/2) + \mathcal{O}(d) \\ =& \mathcal{O}(d \log d) \end{aligned}$$

Is the cost of evaluating $a(x)$ on d points.

Note.

If our polynomial isn't perfectly even, we can pad it out to make it a power of 2. It doesn't really matter.

2. *Evaluate* $c(x)$
3. *Reconstruct* $c(x)$

   Consider how the coefficients of a polynomial relate to its evaluations at $\omega_d^0, \omega_d^1, \ldots, \omega_d^{d-1}$.

   So we have

$$\begin{aligned} a(x) =& a_0 + a_1 x a_2 x^2 + \cdots + a_{d-1} x^{d-1} \\ =& \begin{bmatrix} 1 & x & x^2 & \cdots & x^{d-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{d-1} \end{bmatrix} \end{aligned}$$

   So we have

$$\underbrace{\begin{bmatrix} a(\omega_d^0) \\ a(\omega_d^1) \\ a(\omega_d^2) \\ \vdots \\ a(\omega_d^{d-1}) \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} 1 & \omega_d^0 & (\omega_d^0)^2 & \cdots & (\omega_d^0)^{d-1} \\ 1 & \omega_d^1 & (\omega_d^1)^2 & \cdots & (\omega_d^1)^{d-1} \\ 1 & \omega_d^2 & (\omega_d^2)^2 & \cdots & (\omega_d^2)^{d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_d^{d-1} & \left(\omega_d^{d-1}\right)^2 & \cdots & \left(\omega_d^{d-1}\right)^{d-1} \end{bmatrix}}_{F} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{d-1} \end{bmatrix}}_{y}$$

   This is the discrete Fourier transform! As it turns out, F is a unitary matrix, meaning that its inverse is easy to compute.

   We have (x = a omega vector). We want (y = ai vector).

   We have that

$$x = \frac{1}{\sqrt{d}} F y$$

   So

$$y = \sqrt{y} F^{-1} x$$

   which reconstructs c.

   Since F is unitary, F^{-1} is just like F with ω replaced with $\frac{1}{\omega}$.

7 Dynamic Programming

This is another class of problems dividing into sub-problems, but more subtly. In general the sub-problems will be overlapping here, whereas in divide and conquer, they usually don't overlap. Essentially, by solving the smaller sub-problems first, and use the results of the smaller sub-problems to more easily solve the bigger sub-problem.

The best way to learn is by example.

7.1 Weighted Interval Scheduling

Recall the interval scheduling problem. We were given a bunch of intervals, specified by their starting times s_i and finishing times f_i. Our goal was to find a largest possible subset of non-overlapping intervals.

The new twist is that not every interval is worth the same, they come with their weights. Interval i has a weight $v_i \in \mathbb{R}$. Our goal is now to find a set of non-overlapping intervals S that maximizes the sum of the values of each of its intervals

$$\sum_{i \in S} v_i$$

Notice that the length of the interval doesn't really matter here (unless the weight is the length of course.)

7.1.1 Greedy Approach

Notice that our greedy algorithm no longer works for this. Suppose that we have two classes a and b in our schedule, where a starts and finishes inside of b, but suppose that b is worth more (has a higher v). Our greedy approach would pick a because it finishes first, but that would be sub-optimal.

We could try to schedule the highest weight interval first, but that also doesn't work. What if you had the most valuable interval a cover all others, but the sum of all others outweighs a, then this approach would pick a which is sub-optimal.

Dynamic Programming solves this problem very naturally, so we'll look at how to do this now.

7.1.2 Dynamic Programming Approach

First we sort the intervals so that the finishing times are non-decreasing. In other words, we want

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

which can be done in $\mathcal{O}(n \log n)$ time.

> **Note**.
>
> We write non-decreasing because two intervals can finish at the same time.

Ask: Is the last interval (interval with the last finishing time) part of the optimal solution?

Answer Possibly?

> **Note**.
>
> What's important here is that we ask about a feature of the input, and run over every possibility.

Let's start small with the question above. *Is the last interval part of the optimal solution?*

- If the answer is No, then the optimal solution is the same as if that last interval didn't exist at all.
- If the answer is Yes, then the optimal solution contains the interval, *plus* the optimal solution of the rest of the intervals *minus* the intervals that overlap with the last. In other words,

$$v_n + \text{opt}(1, \ldots, p_n)$$

  Where $p(j) = \max\{i < j : \text{intervals } i \text{ and } j \text{ are disjoint}\}$

Let $\text{opt}(j)$ be the optimal value for this scheduling problem on interval $\{1, \ldots, j\}$. Well now we can write a recurrence for $\text{opt}(j)$

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

$$\text{opt}(0) = 0$$

Where the first parameter, $\text{opt}(j-1)$ tells us that the j^{th} interval is not included, and the second tells us that it is.

> **Note**.
>
> In this formulation of the problem, we only return the optimal value, but not *how* to construct such a schedule.
>
> Changing the program to do this is actually surprisingly easy. At each recursive step, simply return the current optimal schedule as well as the current total sum.

7.1.3 Recursive Algorithm

Assume that the intervals are sorted by finishing time, and that we've pre-computed $p(j)$ for all intervals. These can be done in $\mathcal{O}(n \log n)$.

```
ComputeOpt(j):
  # base case
  if j = 0:
    return 0

  return max(
    ComputeOpt(j - 1),
    v_j + ComputeOpt(p(j))
  )
end
```

Claim: This algorithm is correct. *We've already shown this by the recurrence.*

Running Time

For this sort of program, it's helpful to write a recurrence that we can use to figure out the running time.

Let $T(j)$ denote the running time of `ComputeOpt(`j`)`, then

$$T(j) = T(j-1) + T(p(j)) + \underbrace{\mathcal{O}(1)}_{\text{Computing Max}}$$

$$T(0) = \mathcal{O}(1)$$

Firstly, we can note that $p(j)$ is at most $j-1$. It's certainly not greater than j, by definition. If this is the case, then we have

$$T(j) = 2 \cdot T(j-1) + \mathcal{O}(1)$$

In the worst case. This grows exponentially (like 2^j), oh no.

The reason that this algorithm is so bad is that it recomputes values over and over again. Dynamic Programming fixes this by *caching* values that we've already computed once.

Let's look at the Dynamic Programming approach now.

7.1.4 Recursive Dynamic Programming Algorithm

We collect all the solutions of the sub-problems, and if we've already computed sub-problems, we never have to compute it again.

> *Definition.* **Memoization**
>
> **Memoization** is the process of caching large computations, so that we don't have to compute them more than once.
>
> ■

We define some array $M[j]$ for $1 \leq j \leq n$, initially set to $M[j] = \emptyset$.

```
MemoComputeOpt(j):
  # base case
  if j = 0:
    return 0

  # if we've computed the value already
  if M[j] ≠ ∅:
    return M[j]

  # compute M[j]
  M[j] = max(
    ComputeOpt(j - 1),
    v_j + ComputeOpt(p(j))
  )

  return M[j]
end
```

We can then find the optimal solution by checking which term achieves the max.

Lemma.

The running time of `MemoComputeOpt(`n`)` is $\mathcal{O}(n)$.

Proof.

The running time of `MemoComputeOpt(`j`)` is $\mathcal{O}(1)$ plus the cost of its recursive calls. The running time `MemoComputeOpt(`n`)` is just $\mathcal{O}$(total number of recursive calls).

But notice, we make at most n recursive calls since there are only n values of $M[j]$, and once we've computed $M[j]$, we never call `MemoComputeOpt(`j`)` again.

Therefore the running time must be $\mathcal{O}(n)$.

■

There is actually another way to write this program that doesn't involve recursion. For that case, the running time analysis should be a lot clearer.

7.2 Tabular Dynamic Programming Algorithm

```
1 MemoComputeOpt(n):
2   let M[0] = 0
3
4   for j = 1 to n:
5     let M[j] = max(M[j - 1], v_j + M[p(j)])
6   end
7 end
```

Note.

The order of this `for` loop matters a lot! Since we're running through the values in order, we have that all values that we access are well defined. Take a moment to make sure that this makes sense.

The most involved part of Dynamic Programming is finding what the sub-problems should be. This mostly comes with practice.

7.3 Knapsack

Author Note.

In general, when doing dynamic programming, we want to express the solution in terms of a recurrence.

To be efficient, we need

- polynomially many sub-problems. Since we end up solving all sub-problems, we better not have too many.
- the solution of a problem should be efficiently computable from the solutions of its sub-problems.
- to be able to express the solution in terms of a recurrence involving only "smaller" sub-problems

Note.

The sub-problems should form a DAG. If it does, then there is a topological ordering in which the sub-problems can be computed so that we always have the information needed to do the next evaluation.

Constraints

We are given n items, where item i has a positive integer weight $w_i \in \mathbb{Z}^+$ for all $1 \leq i \leq n$, an upper bound $W \in \mathbb{Z}^+$ on the total weight.

Goal Find a subset $S \subseteq \{1, \ldots, n\}$ that maximizes the total weight $\sum_{i \in S} w_i \leq W$.

7.3.1 Approach

A natural idea in Dynamic Programming is to only consider the first j items. This will be our sub-problems.

Let $\text{opt}(j)$ be the optimal solution only items 1 through j.

Question: starting from the last item, is item n part of the optimal solution?

- **No**: Then the problem is identical with or without the item in the list, so we have

$$\text{opt}(n) = \text{opt}(n-1)$$

- **Yes**:

$$\text{opt}(n) = w_n + \text{opt}(n-1)$$

  But now we don't want to exceed weight $W - n$

We made a mistake! This is great because it shows that we did not come up with the right sub-problem. What we realized as we wrote this recurrence was that we needed to capture the weight of the knapsack in the recurrence.

Let's try that again.

Let $\text{opt}(j, w)$ be the optimal solution containing only items 1 through j, that adds to a weight less than or equal to w. Now we have the following recurrence.

Question: starting from the last item, is item n part of the optimal solution?

- **No**: Then, like before, the problem is the same as if that item weren't there at all, so we have

$$\text{opt}(n, w) = \text{opt}(n-1, w)$$

  And just like before, our backpack is no more full so the weight limit stays the same.

- **Yes**: Then we need to consider the weight of the backpack, and we have

$$\text{opt}(n, w) = w_n + \text{opt}(n-1, w - w_n)$$

  This time, we remove the last item from the list, *and* we make sure to adjust the weight capacity of the backpack.

Base Case:

If our weight limit is zero, then no matter what set of items we're given, we can't take any.

$$\text{opt}(k, 0) = 0 \text{ for any } 1 \leq k \leq n$$

> **Note**.
>
> We'll see that we don't actually need the base case above, due to the order in which we will fill our memo.

Now if $k = 0$, there's no items to take from, so similarly, no matter the weight limit

$$\text{opt}(0, w) = 0 \text{ for any weight } w$$

Recurrence

If the jth item is too heavy for our backpack, we don't consider it. Otherwise, we check for the max between taking item j, and not taking it. More formally

$$\text{opt}(j, w) = \begin{cases} \text{opt}(j-1, w) & \text{if } w_j > w \\ \max\left\{\text{opt}(j-1, w), w_j + \text{opt}(j-1, w-w_j)\right\} & \text{otherwise} \end{cases}$$

Notice that although this recurrence describes the algorithm, it does not describe the order in which to iterate the loop so that all current computed values pull from already defined results.

7.3.2 Pseudocode

```
Knapsack(n, W):
  # we add 1 because we run from 0 items and 0 weights
  let M be an (n + 1) × (W + 1) array of integers

  # set our base case
  let M[0][w] = 0 for all w from 0 to W

  for j = 1 to n:
    for w = 0 to W:
      if w_j > w:
        M[j][w] = M[j - 1][w]
      else:
        M[j][w] = max(M[j - 1][w], w_j + M[j - 1][w - w_j])
      end
    end
  end

  return M[n][W]
end
```

This algorithm is correct because it evaluates the recurrence above.

7.3.3 Running Time

The running time is $\mathcal{O}(n \cdot W)$ since the body of the loop takes constant time.

It's linear in n, and exponential in $\log(W)$. This is known as a pseudo-polynomial time algorithm.

7.4 Sequence Alignment

We have two strings, and we want to know how similar those two strings are.

We want to compare strings, line them up, possibly with gaps, to minimize the mismatches.

Example.

Suppose we have strings representing DNA sequences.

`GACGTTA` and `GAACGCTA`

We want to make them as similar as possible, we'll do this with alignment.

$$\begin{array}{cccccccc} G & A & \_ & C & G & T & T & A \\ G & A & A & C & G & C & T & A \end{array}$$

The third to last column is a *mismatch*.

Setup: To quantify the quality of an alignment, we define two types of penalties.

- δ: The *gap* penalty. This is the price we way of adding a gap in a string.
- α_{xy}: The *mismatch* penalty. This is the penalty of pairing up symbols x and y together. This value could depend on the symbols. For example in our DNA analogy, it might be better to pair `C`s with `G`s than `A`s.

  Note that if $x = y$, then $\alpha_{xy} = 0$.

Goal: Find the alignment of lowest total cost.

Example.

If we had instead done

$$\begin{array}{ccccccccc} G & A & \_ & C & G & \_ & T & T & A \\ G & A & A & C & G & C & T & \_ & A \end{array}$$

Then the cost would have been 3δ because there are 3 gaps, and there are no mismatch penalties.

Now we're going to formalize this idea of alignment.

Definition. **Matching & Alignment**

Given strings $x \in \Sigma^n, y \in \Sigma^m$ where Σ^i is our alphabet. The superscript is the length of the string.

A **matching** of sets I and J is a set of ordered pairs (i, j) with $i \in I$ and $j \in J$ such that each $i \in I$ and $j \in J$ appears at most once.

I, J are index sets of x, y respectively.

A matching M is $I = \{1, \ldots, n\}$ and $J = \{1, \ldots, m\}$ is an **alignment** if there are no "crossings". In other words, if $(i, j) \in M$ and $(i', j') \in M$ with $i < i'$, then $j < j'$.

Example.

$M = \{(1, 3), (2, 2)\}$ is a matching, but **not** an alignment, since $(1, 3)$ crosses over 2.

TODO What's a string that produces this M?

_AB

ACB

_BA

ABC

$M = \{(1, 1), (2, 3)\}$ is an alignment, but it has a gap.

A_B

ABC

■

Note.

In this problem, we are **not** allowed to mutate the strings. We can't change the letters, or their order. We can only add spaces between letters to align them as "best as possible", depending on the metric.

7.4.1 Algorithm

To find this, it's helpful to work backwards and ask what happens at the very end? Asking yes/no questions is helpful in this setting in order to best align the two strings.

Defining the sub-problem

Are the last two symbols matched? In other words, is $(n, m) \in M$?

Author Note.

This might not be the case, maybe the last characters are the same, and so it makes sense to do, maybe they're different, but the penalty is worth paying.

The point is that, if these characters aren't matched with each other, then there *must* be a gap. One of them has to be unpaired.

If not, then we know something more: either the last symbol or x or the last symbol of y must be unpaired.

Why?

If both x_n and y_m are matched, but not to each other, there must be a crossing.

Note.

Crossings are not allowed because including a crossing in our matching answer would mean swapping letters around in our original strings, which is not allowed.

TODO include picture from class (Mar 28, 14:47pm)

So we've identified the sub-problem! We want to focus on aligning $x_1, x_2, \ldots, x_i$ with $y_1, y_2, \ldots, y_j$. Let $\text{opt}(i, j)$ be the minimum cost of an alignment between these sub-strings.

We start from the end of the string and ask

Question: Is x_i aligned with y_j?

- **Yes**: Then

$$\text{opt}(i, j) = \alpha_{x_i, y_j} + \text{opt}(i - 1, j - 1)$$

- **No**: Is x_i unmatched?
  - **Yes**: Then

$$\text{opt}(i, j) = \delta + \text{opt}(i - 1, j)$$

  - **No**: Then y_j must be unmatched, so

$$\text{opt}(i, j) = \delta + \text{opt}(i, j - 1)$$

    Note that y must be unmatched here because if it were not, there would be a crossing with x and y.

> **Note**.
>
> It's important to note that, in general, we *don't know* if x_i is aligned with y_j, but *if* we knew the solutions to all the sub-problems, we *would* know.

In general, we have

$$\text{opt}(i, j) = \min\{\alpha_{x_i, y_j} + \text{opt}(i - 1, j - 1), \delta + \text{opt}(i - 1, j), \delta + \text{opt}(i, j - 1)\}$$

Which is just the minimum of the three scenarios above.

Initial Conditions: $\text{opt}(i, 0) = i \cdot \delta$, and $\text{opt}(0, j) = j \cdot \delta$.

If we're comparing index k of one string to index 0 of the other, the best possible scenario is to offset the string by k spaces, which costs $k \cdot \delta$.

7.4.2 Pseudocode

```
1 Align(x, y):
2   let A be an (n + 1) × (m + 1) array indexed by
3   i ∈ {0, 1, ..., n} and j ∈ {0, 1, ..., m}
4
5   A[i, 0] = i · δ for all i ∈ {1, ..., n}
6   A[0, j] = j · δ for all j ∈ {1, ..., m}
7
8   for i = 1 to n:
```

```
 9      for j = 1 to m:
10        A[i, j] = min{α_{x_i,y_j} + A[i − 1, j − 1], δ + A[i − 1, j], δ + A[i, j − 1]}
11      end
12    end
13
14    return A[n, m]
15  end
```

7.4.3 Running Time

The total running time is $O(nm)$. **TODO** talk about how

7.5 Shortest Paths in Graphs with Negative Weights

Side Note.

> Homework 3 is due April 5.
>
> ■

Author Note.

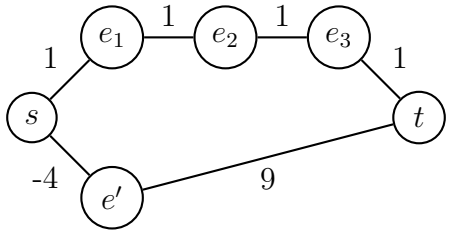
Note that just doing Dijkstra's algorithm wont work here, you also can't just offset the weights by the most negative weight.

In the unweighted setting, you could just use breadth first search. We talked about how, if you have positive weight edges, you could use Dijkstra's algorithm.

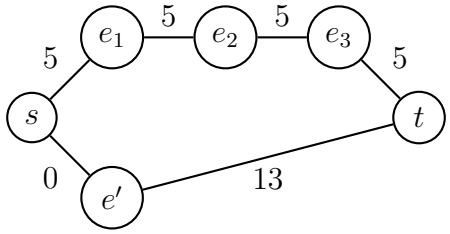
Today, we'll see what happens when the edge weights are necessarily positive.

7.5.1 What if I just shift all weights up by the lowest and do Dijkstra's?

This doesn't work. Consider the following graph where we try to go from s to t.



In this case, we can clearly see that the shortest path is the top one with a total weight of 4. Now let's see what happens when we add a weight of 4 to every edge.



Now the top path has a total weight of 20, and the bottom one has a weight of 13, so this is **not** the same graph as the one we started with.

What this shows us is that we can't simply solve this problem by offsetting the weights of such a graph.

7.5.2 The Problem

Given a digraph $G = (V, E)$ with edge weight c_{uv} for each edge $(u, v) \in E$. We're also given vertices $s, t \in V$. What is *a* minimum cost path from s to t? The cost of the path is the sum of the costs of the edges on that path.

Why should we care about negative edge weights? well it could be the case that in certain scenarios. For example, what if the edges represent financial transactions? In that case, we would want to make the total cost as low as possible.

7.5.3 Assumption

We're going to make an assumption to make the problem well defined.

G **does not contain negative cost cycles**. The reason we want to avoid this is because, if G did have negative cost cycles, you could just go around in circle in that cycle to make your weight arbitrarily low.

Note.

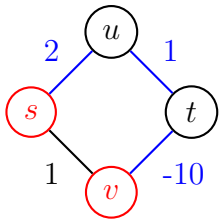
Our graph *may* have cycles! But no *negative cost* cycles.

7.5.4 Approaches

To solve the problem, we could think of modifying Dijkstra's algorithm in some way. Unfortunately we can see that this does not work. Having negative edge weights means you can't simply consider "one more edge". Let's look at an example.

Example.

In the following graph, the shortest path from s to v is not obvious. We would need to explore more of the graph to be able to find it.



7.6 Bellman-Ford Algorithm

Main Idea: Let $\text{opt}(i, v)$ be the minimum cost of a path from v to t using at most i edges.

By limiting the number of edges at each step, we have our sub-problem!

Why does this work? Why isn't the memo too big?

- There's only n vertices V
- The minimum cost path is simple, so only need to consider *paths of length* $\leq n - 1$

QUESTION Why do we care about this?

Question: Does the min-cost path for $\text{opt}(i, v)$ use all i edges?

- **No**: Then $\text{opt}(i, v) = \text{opt}(i-1, v)$, since we don't use all i edges, so the solution should be the same using one less edge.
- **Yes**: Then $\text{opt}(i, v) = \min_{u \in V} \big(\text{opt}(i-1, u) + c_{uv}\big)$.

  Here we consider the optimal way to get to a neighbor u of v, and then look for the best way to get from u to v.

Initial Conditions

- $\text{opt}(0, t) = 0$.

  t is our *target* vertex, it's also right where we happen to be, so the cost to get to it is 0.
- $\text{opt}(0, v) = \infty$ for all $v \neq t$.

  If you want to get to any vertex other than t in at most 0 edges, you can't.

Goal: Compute the shortest path from s to t using *at most* $n-1$ edges. In other words, $\text{opt}(n-1, s)$.

7.6.1 Pseudocode

```
1  BellmanFord(G, c, s, t):
2    let n be the number of vertices of G
3    let M be an n × n array
4
5    for all v ≠ t:
6      M[0, t] = 0
7      M[0, v] = ∞
8    end
9
10   for i = 1 to n − 1:
11     for v ∈ V:
12       M[i, v] = min{M[i − 1, v], min_{u∈V}(M[i − 1, u] + c_uv)}
13     end
14   end
15
16   return M[n − 1, s]
17 end
```

Where

- G is our graph.
- c is the list of edge weights of G.
- s is our start vertex.
- t is our target vertex.

7.6.2 Running Time

We definitely have a $\mathcal{O}(n^3)$ upper bound. We can do better than this though, by noticing that the cost of line 12 is $\mathcal{O}(\text{outdeg}(v) + 1)$.

So the total running time is

$$\mathcal{O}\Big(n \cdot \sum_{v \in V} (\text{outdeg}(v) + 1)\Big) = \mathcal{O}(n(m+n))$$

If the graph is sparse, this is quite good.

7.6.3 Negative Cycles

If we do have negative cycles, running this algorithm would continue to find lower and lower bounds for the path. At some point we can stop, but how do we know if there is one?

Bellman-Ford will find a negative cycle if one exists, provided that

1. there is a path from the cycle to t, where we're trying to go
2. it runs long enough to see the cycle.

> **Note**.
>
> This is not necessarily the case: what if the whole graph is one big cycle? Then you'll never figure this out because you'll get to your destination before noticing the cycle.

Lemma.

For any directed graph G, there exists a digraph G' with *one more* vertex than G such that G' has a path from a negative cycle to t, if and only if G has a negative cycle.

Proof.
This proof isn't formal, but let's describe the main idea.

You have some graph G. Create a new vertex t, and add an edge from all the vertices in G, to this new vertex t.

Now if there was a negative cycle in G, its *definitely* connected to t.

■

This solves the first of the two problems. Now we need to make sure that our algorithm can see the cycle, provided that there is one.

Remember when we considered that the whole graph was one big cycle? Well we only needed to run Bellman-Ford one more time (on all n edges instead of $n-1$) to notice that we're in a cycle. It turns out that this is completely generic, and will always work.

Lemma.

A digraph has no negative-weight cycles **if and only if** $\text{opt}(n-1, v) = \text{opt}(n, v)$ for all vertices v.

here, n is the number of vertices in G', not G.

In other words, we add *one more* row to the memo. If the last row is the same as the second to last row, then there are no negative cycles. If it *is* different, then there *is* a negative cycle.

Proof.

If there are no negative weight cycles, then this follows from the correctness of Bellman-Ford (the result of one more search should not change.)

Conversely, if it is the case that $\text{opt}(n-1, v) = \text{opt}(n, v)$ for all vertices v, then by induction, it *must* be the case that $\text{opt}(n, v) = \text{opt}(k, v)$ for all $k \geq n - 1$.

However, if there were a negative cycle, then $\text{opt}(k, v)$ could become arbitrarily small (because you could just go around the cycle as many times as you'd like.)

But you don't need to go around the cycle a million times to see if there is a cycle! You just need one more iteration of Bellman-Ford, and you check if things have changed.

■

Why did we add the vertex t?

Well remember that we always want to get to t. So we created a graph that you can reach from all vertices. By making sure we create such a graph, we make sure that Bellman-Ford will always explore the whole graph to check what the best way to get to t is.

8 Network Flow

Author Note.

We're still talking about graph algorithms, but ultimately, this is a new style of algorithm.

We're going to try to find augmenting paths in graphs. This is a really powerful algorithmic idea, and sometimes it's not obvious where it might show up.

Definition. **Flow Network**

We're going to have this notion of a **flow network**. This is a digraph $G = (V, E)$ with

- A designated **source** vertex $s \in V$. This is a vertex with no edges pointing into it.

$$\text{indeg}(s) = 0$$

- A designated **sink** vertex $t \in V$. This is a vertex with nothing coming out of it.

$$\text{outdeg}(t) = 0$$

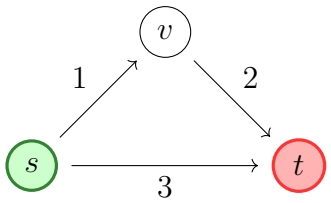
- A **capacity** $C_e \geq 0$ for all edges $e \in E$.

  These are essentially the weights of the edges. We also assume that they are **integers**.

For the problem that we're considering, there is only one sink and one source. Generally, there's no reason why we couldn't have multiple vertices with in-degree 0, or out-degree 0.

Let's look at an example flow network.

Example.



Definition. **Flow**

A **flow** is a function $f : E \to \mathbb{R}^+$ satisfying

- **Capacity** conditions $0 \leq f(c) \leq C_e$ for all $e \in E$

  An edge can't pump more liquid than its C_e value.

- **Conservation** conditions.

$$f^{\text{in}}(v) = f^{\text{out}}(v)$$

  For all vertices v, **except the source s and sink t.**

An edge transfers as much flow as is sent into it.

Here, $f^{\text{in}}(v)$ is the sum of all the flows going into v, and $f^{\text{out}}(v)$ is the sum of the flows going out of v.

$$f^{\text{in}}(v) = \sum_{e \text{ into } v} f(e)$$

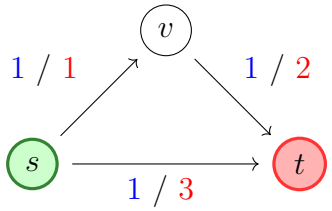
$$f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$$

8.1 Flow Examples

Let's look at an example of a flow.

Example.

This is a flow that satisfies the conditions above.



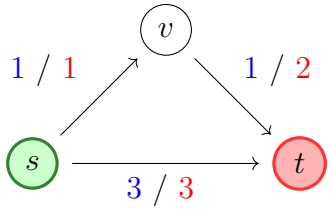
The edge weights are represented as

flow value / capacity

Note that this was not the best flow!

Example.

This is.



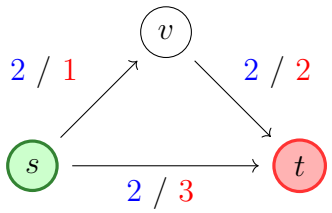
Notice that the edge from v to t is still at flow value 1, even if it can handle 2. That's because only 1 is coming into it.

We'll spend a lot of time trying to find the best possible flow of these graphs.

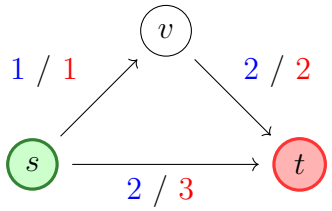
8.2 Invalid Flow Examples

Notice that there are also *invalid flows*.

Example.



The graph above is wrong because it doesn't respect the **capacity** constraint.



The graph above is wrong because it doesn't respect the **conservation** constraint.

Definition.
The **value** of a flow $v(f) = f^{\text{out}}(s)$ is the total flow that's leaving s.

■

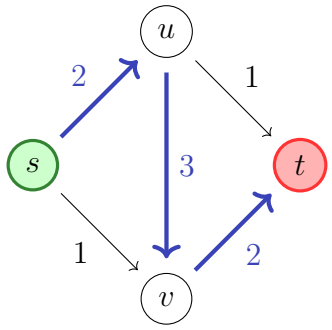
8.3 Finding a maximum flow

What happens if we try to be greedy? Spoiler alert: it won't work. Let's see why.

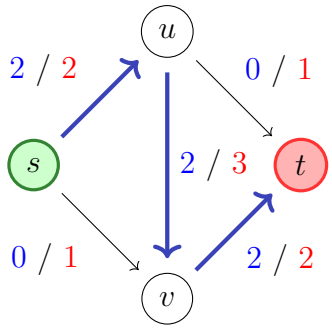
Here's the strategy:

1. Find an $s \to t$ path (using your favorite path finding algorithm)
2. Send as much flow along it as possible.

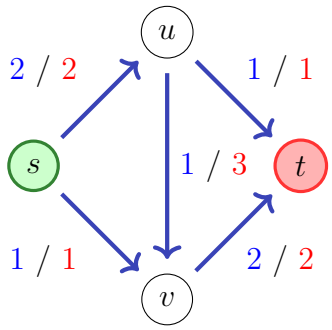
Consider the following flow generated by the greedy rule.



We can be more precise, here are the flow / capacities.



We get that the flow value at our sink t is 2. But notice that we're not using edges $s \to v$ or $u \to t$. Here's a better flow instead.



Doing this, we still respect the capacity and conservation conditions, and we get a flow value of 3 at our sink. This should convince you that the greedy approach is sub-optimal.

This is great and all but how did we come up with this better example? We want a general way to send as much flow as possible from s to t that will work in general. The general strategy that we'll use to do this stems from a simple idea. We'll allow ourselves to not only send flow forward along an edge, but also backwards. Sending flow backwards along an edge is analogous to decreasing the flow in the forward direction.

It should make sense that we should only allow ourselves to do this for edges that have flow in the forwards direction to begin with.

8.4 Residual Networks

To get around this issue, we're going to consider forward edges and backward edges in **residual networks**.

Definition. **Residual Network**

Given a flow network and a flow f, their **residual flow network** works as follows

- The vertices are those of the original flow network.
- **Forward Edges**: For each edge e with $f(e) < C_e$, include the edge e with capacity $C_e - f(e)$.

  In this case, we're not using e to its full capacity.
- **Backward Edges**: For each edge $e = (u, v)$ with positive flow value $f(e) > 0$, include the edge (v, u) with capacity $f(e)$.

  We should allow ourselves to decrease the flow along e by at most $f(e)$.

■

Note.

A Residual Flow Network is **only** defined for a *particular flow!*

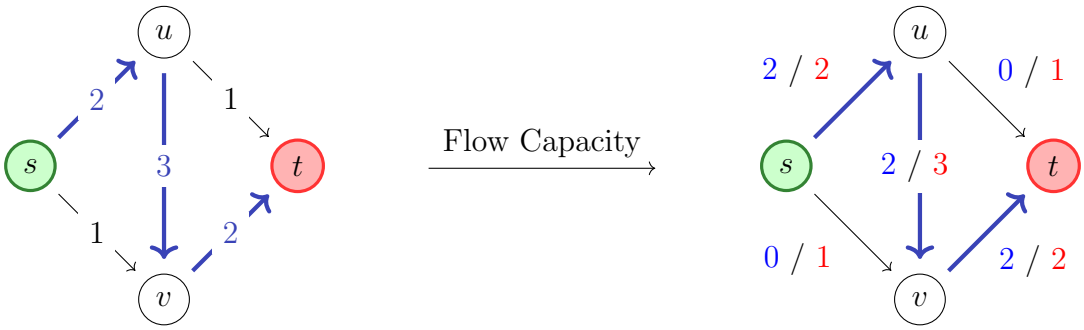
We want to send as much flow from the start to the destination.

The residual flow network "flows backwards through the graph".

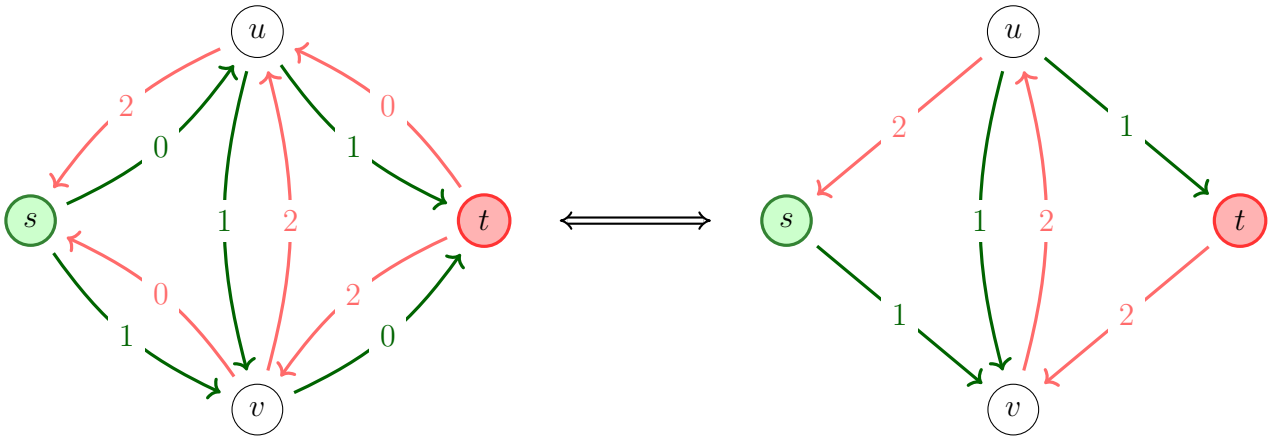
Let's look at an example

Example.

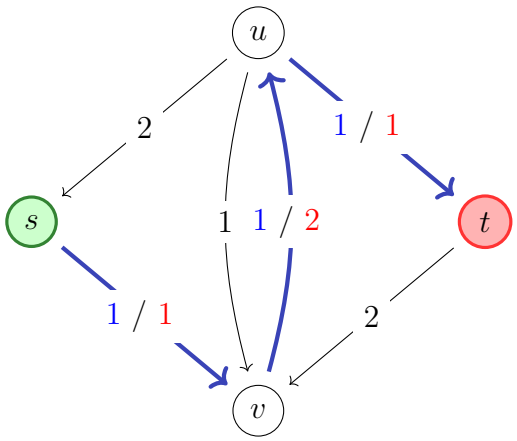
Consider the following flow network.



Let's construct the residual network of this particular flow using the rules from the last lecture.

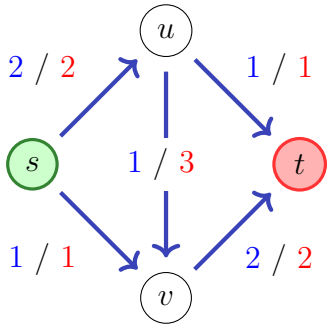


Now, we can use the residual network to send the following flow



What we've done here is that we've **augmented** the flow! We used the residual network of our original flow and

found a flow that we could send through it that we can now use to update our original flow.



This is the optimal flow that we found earlier!

This is exactly what our algorithm is going to do. We're going to start from zero flow, construct the residual network, send as much flow as possible along some path, augment the flow, construct the new residual network, etc... Until we can't send anymore flow from s to t.

Note.

It turns out that this approach is optimal. By repeating this process, we *will* find the optimal flow from s to t.

Definition. **Augmenting Path**
An **Augmenting Path** is a simple path from s to t in the residual network, with positive capacity.

■

8.5 The Ford-Fulkerson Algorithm

Here are the steps we need.

1. **Find an Augmenting Path.**

   We can do this using BFS or some other path-finding algorithm. It doesn't actually matter if we don't find the *best* path from s to t here, any path will work.

2. **Augment the Flow.**

   For a path P and a flow f, we let bottleneck(P, f) be the smallest capacity of any edge on P, **in the residual network corresponding to** f.

   Let's look at exactly how this works.

```
Augment(f, P):
  # done by walking along the path and
  # finding the minimum capacity
  let b = bottleneck(P, f)

  for e = (u, v) along P:
    if e is a forward edge:
      increase f(e) by b
    else:
      # e is a backwards edge
```

```
11
12        let e' = (v, u)
13        decrease f(e') by b
14    end
15  end
```

To figure out if an edge is forwards or backwards, you compare it to the edge from the original flow network. Remember, the graph we pass to `Augment` is the graph of the residual network, so its edges might differ from the original graph.

Lemma.

`Augment(`f`, `P`)` produces a valid flow.

> **Proof.**
>
> To prove this, we have to prove that `Augment(`f`, `P`)` respects the capacity, and conservation constraints on the original graph.
>
> **Capacity**
>
> Suppose e is a forward edge, we increase the flow by b, and every edge on P has residual capacity at least b (in other words, every edge along P can still take at least b more flow) so we still satisfy the capacity constraint.
>
> If e is a backward edge, we decrease the capacity by b. Since b is at most $f(e)$, we never decrease the flow by more than the maximum possible, so the resulting flow is non-negative. Again, we satisfy the capacity constraint.
>
> **Conservation**
>
> Whenever we add flow into an internal vertex (not the source or the sink), then we add the same flow going out.
>
> ■

Now we're ready to see the complete algorithm.

8.5.1 Algorithm

```
1  MaxFlow(G, C, s, t)
2    let f(e) = 0 for all edges e of G
3
4    while there is a simple path P from s to t
5    in the residual network of flow f:
6      update f to Augment(f, P)
7      update the residual network to use the new flow
8    end
9
10   return f
11 end
```

Where

- G is our graph
- C stores the capacity of each edge
- s is our source vertex
- t is our sink vertex

8.5.2 Proof of Termination

How do we know that this algorithm terminates? How do we know that it finds the max flow?

This is actually quite subtle, this algorithm actually could run indefinitely, *if we had not restricted ourselves to integer capacities*.

Lemma.

The value of the flow strictly increases, at every step of the algorithm.

> **Proof.**
> The first edge of augmenting path P starts from s. The flow along this edge is increased by the bottleneck capacity $b > 0$. Since P is simple, this is the only edge of P that includes s. So the value of the flow is increased by b.
>
> ■

Lemma.

At each step of the algorithm, the flow values and the residual capacities are all integers.

> **Proof.**
> This just simply follows from the basic fact that $\mathbb{Z}$ is closed addition.
>
> In other words, if you subtract an integer from an integer, you get an integer. If you add an integer to an integer, you get another integer.
>
> ■

From these two lemmas, we get that the flow increases by at least 1 at each step. Since the larger possible flow value is at most

$$C = \sum_{e} C_e$$

Where e is an edge leaving s.

This is finite, so we eventually must reach it. Thus the algorithm terminates.

> **Note**.
> Although C is the maximum flow, it might not be achievable.

8.5.3 Running Time

Additionally, C is an upper bounds for the number of iterations that the algorithm performs.

Since each pass through the loop can be implemented in time $\mathcal{O}(m+n)$, the overall running time is $\mathcal{O}(C(n+m))$.

8.6 Maximum Flows & Minimal Cuts

We saw that C was an upper bound for the maximum flow, but we can find other bounds for the flow by considering various types of *cuts*. Recall the definition of a cut: a bipartition of the vertices.

Definition.
We call an cut an s-t cut if $s \in A$ and $t \in B$.

■

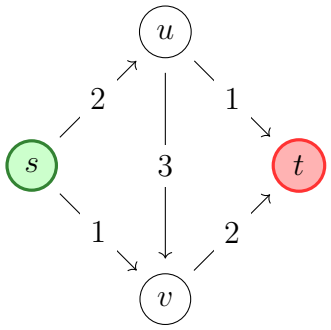
In a flow network. The capacity of an s-t cut is the total capacity of all edges leaving A.

$$C(A,B) = \sum_{e \text{ out of } A} C_e$$

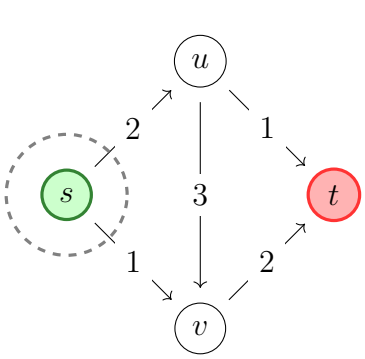
We only look at the edges that go from the A part of the cut, to the B part of the cut.

Example.

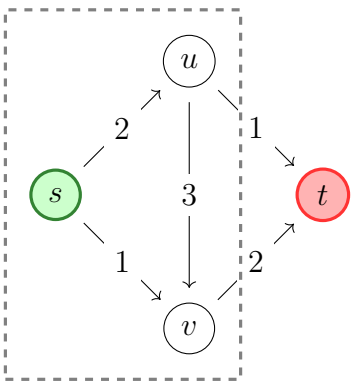
Let's look at our example from earlier.



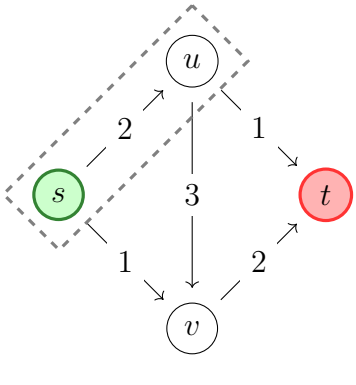
We can generate a cut by choosing any subset $A \subset V$ which includes s and doesn't include t.



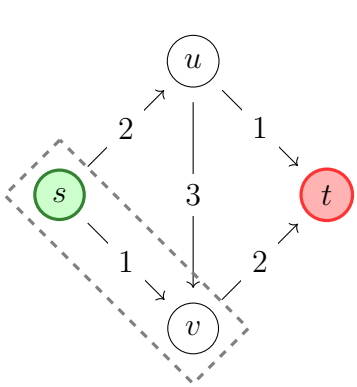
$A = \{s\}$
Capacity 3



$A = \{s, u, v\}$
Capacity 3



$A = \{s, u\}$
Capacity 5



$A = \{s, v\}$
Capacity 4

What we'll show next time is that the capacities of these cuts will give us upper bounds on the flow value.