CSE 101

Design and Analysis of Algorithms

Winter 2022

Taught by Professor Daniel Kane

Table of Contents

1 Introduction to Algorithms

In this class, we'll talk about algorithms. To start this class off, we'll consider the Fibonacci Numbers.

1.1 Problem: Fibonacci Numbers

Definition 1.1: Fibonacci Numbers

The **Fibonacci numbers** are the sequence defined by:

$$F_0 = F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2$$

1.1.1 Naive Algorithm

There is an easy recursive algorithm:

```
Fib(n)
    if n <= 1
        Return 1
    Else
        Return Fib(n - 1) + Fib(n - 2)
```

1.1.2 Runtime

Let $T(n)$ be the number of lines of code `Fib`(n) needs to execute.

- The `if` and `else` statements make up 2 lines of code.
- Otherwise, we need to run $1 + T(n-1) + T(n-2)$ lines.

So:

$$T(n) = \begin{cases} 2 & n \leq 1 \\ T(n-1) + T(n-2) + 3 & \text{Otherwise} \end{cases}$$

Here, if we want to compute `Fib`(100), the runtime would be:

$$T(100) \approx 2.87 \cdot 10^{21}$$

At a billion lines of code per second, this would take over 90,000 years to run.

1.1.3 Why So Slow?

This algorithm, in particular, has too many (redundant) recursive calls. For example:

$$f(5) = f(4) + f(3)$$

$$f(4) = f(3) + f(2)$$

$$f(3) = f(2) + f(1)$$

Already, there are some repeated calls. How can we make this more efficient?

1.1.4 Improving this Algorithm

First, we avoid recomputing things. When we do this by hand, we often already have the last two numbers. So, we can use an array to store the nth Fibonacci number.

1.1.5 Improved Algorithm

If we were to simulate finding the Fibonacci numbers by hand, we would have an algorithm similar to:

```
Fib2(n)
    Initialize A[0..n]
    A[0] = A[1] = 1
    For k = 2 to n
        A[k] = A[k - 1] + A[k - 2]
    Return A[n]
```

In the first two lines (initializing the array and setting initial values), there are 2 lines. In the `for` loop, we run $2(n-1)$ lines. Finally, we run 1 line of code. So, we have the final result of:

$$T(n) = 2n + 1$$

With this new algorithm, we have $T(100) = 201$. So, this is easily runnable on almost any computer.

Essentially, the power of algorithms is:

> Sometimes, the right algorithm is the difference between something working and not finishing in your lifetime.

1.2 Levels of Algorithm Design

In this class, there are several things we consider when designing algorithms.

- **Naive Algorithms:** Turn definition into algorithm. This is easy to wirte, good first pass, but often very slow. Good way to "test."
- **Toolkit:** Algorithms designed using standard tools the main focus of this course.
- **Optimized:** Use data structures or other ideas to make algorithm especially efficient.
- **Magic:** Sometimes, an algorithm requires a surprising new insight.

2 Undirected Graph Algorithms

Definition 2.1: Graph

A **graph** $G = (V, E)$ consists of two things:

- A collection V of vertices, or objects to be connected.
- A collection E of edges, each of which connects a pair of vertices.

For example, we can model the following as graphs:

- The internet, where V is the websites and E are links.
- The internet, where V are computers and E are physical connections.
- A highway system, where V are the intersections and E are the roads.

In computer science, there are a few examples of graphs:

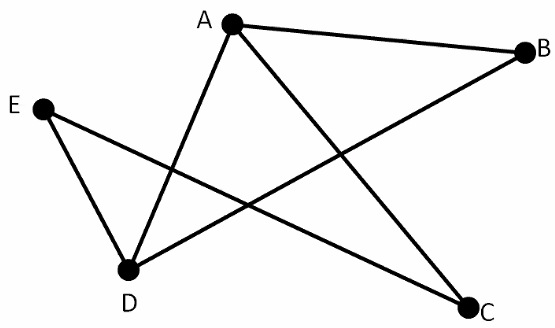
- The internet (webpages, physical connections, etc.)
- Social networks (especially with friendship, connections).
- Transitions between states of a program.
- Road maps.

Another question we might ask is, how do you store a graph in a computer?

- **Adjacency Matrix:** Store list of vertices and an array `A[i, j] = 1` if edge between v_i and v_j.
  - Slow space for dense graphs.
  - Slow for most operations.
- **Edge List:** List of all vertices with list of all edges.
  - More space-efficient for a sparse graph.
  - Hard to determine edges outof the single vertex.
- **Adjacency List:** For each vertex, store a list of neighbors.
  - Needed for DFS to be efficient.
  - We will usually assume this representation.

2.1 Drawing Graphs

When drawing graphs, we want to draw the vertices first. Then, for each edge, draw line segments or curves connecting those points.



This particular graph can be represented by $G = (V, E)$ where:

$$V = \{A, B, C, D, E\}$$

$$E = \{(A, B), (A, C), (A, D), (B, D), (C, E), (D, E)\}$$

Because this is an unordered graph, (A, B) and (B, A) mean the same thing.

2.2 Problem: Exploring Graphs

Suppose we are playing a video game and want to make sure that you've found all the areas in this level before moving on to the next one. How do we ensure that we found everything?

2.2.1 Basic Algorithm

Essentially, we want to:

```
Keep track of all areas discovered.
While there is an unexplored path:
    Follow path.
```

2.2.2 Systematize

Essentially, we need to keep track of:

- Which vertices we have discovered.
- Which edges have yet to be explored.

So, the explore algorithm will:

- Use a field `v.visted` to let us know which vertices we have seen.
- Store edges to be explored implicitly in the program stack.

```
explore(v):
    v.visited <- true
    for each edge (v, w):
        if not w.visited:
            explore(w)
            w.prev <- v     // If we want to keep track of path taken
```

2.2.3 Result

Theorem 2.1

If all vertices start unvisited, `explore(v)` marks as visited exactly the vertices reachable from `v`.

Proof. First, we note that we can only visit vertices that are reachable from v. If u is visited, then eventally we will visit every adjacent w. If there is a chain of vertices, then v will visit u_1 which will visit u_2 which will visit everything up to and including w. □

2.3 Problem: Depth First Search

`explore` only finds the part of the graph reachable from a single vertex. If you want to discover the entire graph, you may need to run it multiple times. This introduces an algorithm known as **depth first search**:

```
DepthFirstSearch(v):
    Mark all v in G as unvisited.
    For v in G:
        if not v.visited:
            explore(v)
```

Here, this runs in $O(|V|)$ time: we need to iterate over vertex.

2.3.1 Runtime of DFS

```
explore(v):                 // Run once per vertex O(|V|)
    v.visited <- true       // Run once per vertex
    for each edge (v, w):   // Run once per neighboring vertex O(|E|)
        if not w.visited:   // Run once per neighboring vertex
            explore(w)      // Run once per neighboring vertex
            w.prev <- v     // Run once per neighboring vertex
```

So, our final runtime is:

$$O(2|V| + |E|) = O(|V| + |E|)$$

Although $O(|V|+|E|)$ is linear time, in reality, graph algorithm runtimes depend on both $|V|$ and $|E|$. What algorithm is better may depend on the relative sizes of these parameters.

	Sparse Graphs	**Dense Graphs**
Runtime	$\|E\|$ small ($\approx V$)	$\|E\|$ large ($\approx V^2$)
Examples	Internet, Road Maps	Flight Maps, Wireless Networks

2.4 Problem: Connected Components

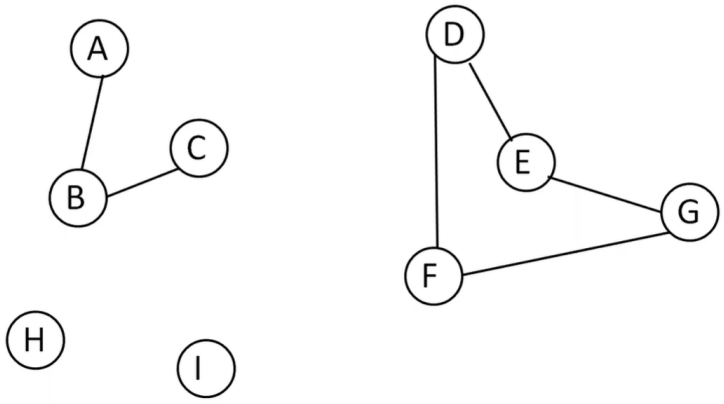
We want to understand which vertices are reachable from which others in a graph. We've already kind of done this using `explore(v)` to find which vertices are reachable from a given vertex.

For a more theoretical answer, consider the theorem:

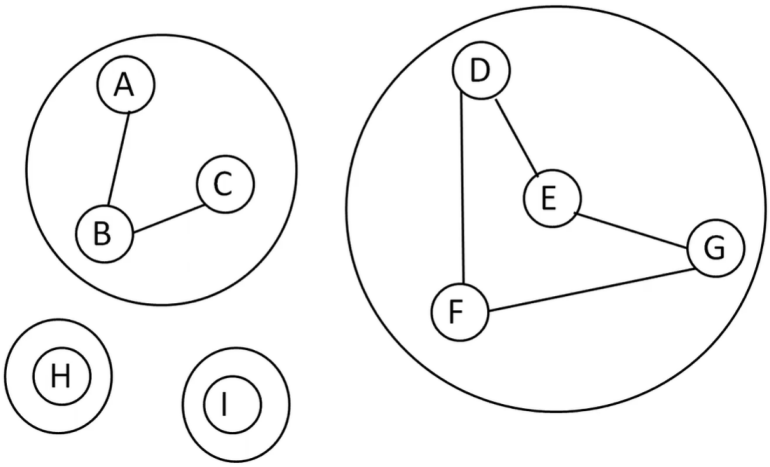
Theorem 2.2

The vertices of a graph G can be partitioned into connected components so that v is is reachable from w if and only if they are in the same connected component.

For instance, consider the following graph:



We can split this graph into four components:



Here, if there are two points in a component, then we can reach the first point from the second and vice versa. However, if there are two points in two different components, then this is not possible.

Problem Statement: Given a graph G, how do we compute its connected components?

- Easy Way: For each v, run `explore(v)` to find the vertices reachable from it. Group together into components. The runtime will be $O(|V|(|V| + |E|))$.
- Better Way: Run `explore(v)` to find the components of v. Then, repeat this on the unclassified vertices.

2.4.1 Applying Depth-First Search (DFS)

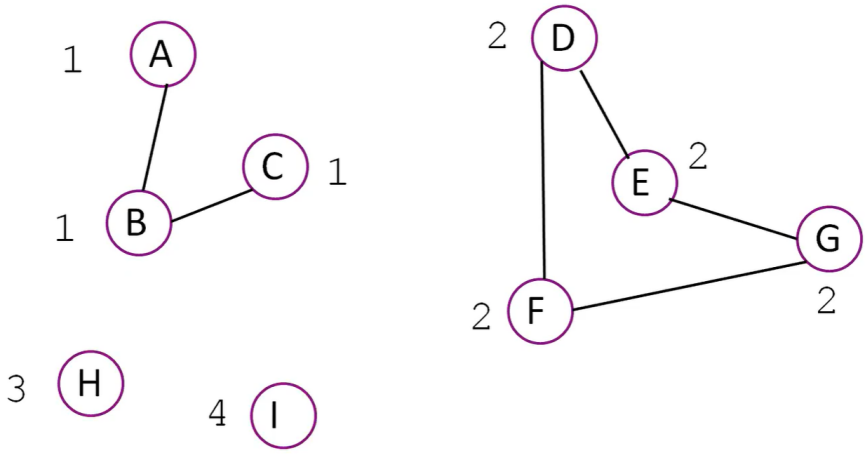
We can use depth-first search to simulate this. In fact, we can use the same algorithm above but with a bit of additional "information" to get our answer:

```
explore(v, CCNum):
    v.visited = true
    // CC is connected components
    v.CC = CCNum
    for each edge (v, w):
        if not w.visited:
            explore(w)
```

```
ConnectedComponents(G):
    CCNum = 0
    for each v in G:
        v.visited = false
    for each v in G:
        if not v.visited:
            CCNum++
            explore(v, CCNum)
```

The runtime is $O(|V| + |E|)$.

If we recall the graph above, running this algorithm would yield something like:



Where `CCNum` is `4`, or 4 connected components.

2.4.2 DFS: A Discussion

What does DFS actually do?

- No output.
- Marks all vertices as visited.
- Easier ways to do it.

However, DFS is a useful way to *explore the graph*. In particular, by augmenting the algorithm a bit, like we did with the connected components algorithm, we can learn useful things. In other words, DFS by itself is quite useless, but adding a bit of additional information will make it more useful.

2.4.3 Pre- and Post- Order

How can we augment this algorithm?

- Keep track of what algorithm does and in what order.
- Have a "clock" and note time whenever:
  - Algorithm visits a new vertex for the first time.
  - Algorithm finishes processing a vertex.
- Finally, we can record values as `v.pre` and `v.post`.

2.4.4 Computing Pre- and Post- Order

Consider the algorithm, which is the same thing as above but with some changes to make it useful:

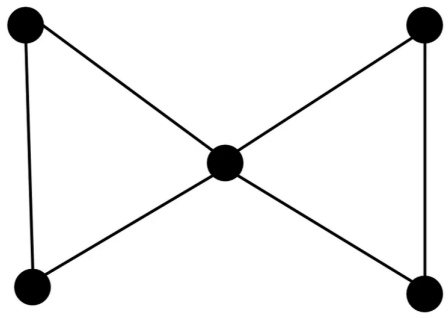
```
explore(v)
    v.visited = true
    v.pre = clock
    clock++
    For each edge (v, w)
        If not w.visited
            explore(w)
    v.post = clock
    clock++

DepthFirstSearch(G)
    clock = 1
    Mark all v in G as unvisited
    For v in G
        If not v.visited
            explore(v)
```

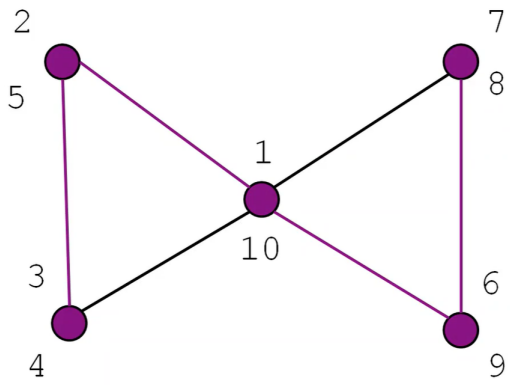
The runtime is still $O(|V| + |E|)$.

2.4.5 Example of Pre- and Post- Order

Consider the following graph:



After running the above algorithm, we get:



How did we get this?

- We assign `1` to the center vertex. `1` has an unexplored neighbor, which is the top-left vertex.

- We assign `2` to the top-left vertex. `2` has an unexplored neighbor, which is the bottom-left vertex.
- We assign `3` to the bottom-left vertex. `3` doesn't have any other unexplored neighbors.
- Since we can't go anywhere from `3`, we assign it a post-order value of `4`.
- Now that we're back at `2`, note that there aren't any other unexplored edges, so we give it a post-order value of `5`.
- We're now back at the center vertex. However, we still have more to explore.
- We assign `6` to the bottom-right vertex.
- We assign `7` to the top-right vertex.
- Since we can't go anywhere from `7`, we assign it a post-order value of `8`.
- Now that we're back at `6`, we assign it a post-order value of `9` since we can't explore anything else.
- Now that we're back at `1`, we assign it a post-order value of `10` since we can't explore anything else.

If the graph is connected, the first vertex will always be the last vertex. However, if the graph is disconnected, the first vertex may not necessarily be the last vertex.

2.4.6 Application of Orders

Proposition. *For vertices v, w, we can consider the intervals `[v.pre, v.post]` and `[w.pre, w.post]`. These intervals:*

1. *contain each other if v is an ancestor/descendant of w in the DFS tree.*
2. *are disjoint if v and w are cousins in the DFS tree.*
3. *never interleave (`v.pre < w.pre < v.post < w.post`).*

Proof. Assume that the algorithm finds v before w (`v.pre < w.pre`). If the algorithm discovers w after fully processing v, then:

- `v.post < w.pre`
- Intervals are disjoint
- v and w are cousins.

If the algorithm discovers w before fully processing v:

- The algorithm finishes processing w before it finishes v.
- `v.pre < w.pre < w.post < v.post`
- Represents nested intervals.
- v is an ancestor of w.

And so we are done. □

3 Directed Graph Algorithms

Often, an edge makes sense both ways. But, sometimes, streets are one directional.

Definition 3.1

A **directed graph** is a graph where each edge has a direction. We say that it goes from v to w.

Often, we draw arrows on the edges to denote direction.

A directed graph can be thought of as a graph of dependencies. For example, consider the following activities:

```
Breakfast                    Go to work

            Wake up

Shower                       Dress
```

Of course, you can't do all five of these activities at the same time; there are some dependencies. Some of them are:

- Wake Up → Breakfast → Go to Work.
- Wake Up → Go to Work.
- Wake Up → Shower → Dress → Go to Work.

In other words, you need to do A before you can do B, and so on. One of the things we might want to do is order this graph in a way that respects these dependencies.

3.1 Problem: DFS on Directed Graphs

We can use DFS on a directed graph. However, we only follow directed edges from v to w. The runtime is still $O(|V| + |E|)$, and `explore(v)` discovers all vertices reachable from v following only directed edges.

3.2 Problem: Topological Ordering

Essentially, a directed graph can be thought of as a graph of dependencies. An edge $v \mapsto w$ means that v should come before w. We can use something known as topological ordering to better understand this relationship.

Definition 3.2: Topological Ordering

A **topological ordering** of a directed graph is an ordering of the vertices so that for each edge (v, w), v comes before w in the ordering.

Remark: There can be multiple different topological orderings for the same graph.

A question that we might have is: does every directed graph have a topological ordering?

- No. Consider the classic counterexample:

$$\text{Chicken} \mapsto \text{Egg}$$

$$\text{Egg} \mapsto \text{Chicken}$$

  In other words, there is a *cycle* where the chicken goes to the egg and the egg goes back to the chicken.

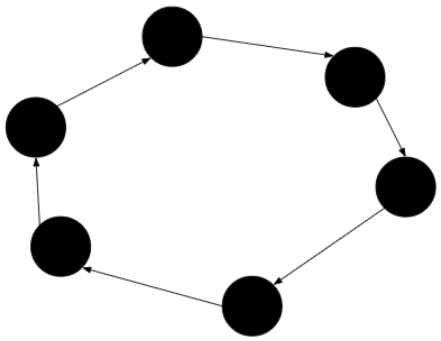
3.2.1 Cycles

Definition 3.3: Cycle

A cycle in a directed graph is a sequence of vertices $v_1, v_2, \ldots, v_n$ so that there are edges:

$$(v_1, v_2), (v_2, v_3), \ldots, (v_n, v_1)$$

For example, here is a cycle:



3.2.2 Obstacle

Proposition. *If G is a directed graph with a cycle, then G has no topological ordering.*

So, in other words, if G is a directed graph with at least one cycle *anywhere*, then it has no topological ordering.

Proof. Suppose we have a cycle $v_1, \ldots, v_n$. Assume for the sake of contradiction taht we have an ordering. Then, there are n vertices so one of them has to come first; say that v_i came first in the ordering. But, this is a cycle so there is an edge from v_{i-1} to v_i. However, because v_i was first so it must come first in the order, in contradiction to the order property. □

3.2.3 Directed Acyclic Graph (DAG)

Suppose we want to focus on directed graphs with no cycles. This brings us to the following definition.

Definition 3.4

A **directed acylic graph** (DAG) is a directed graph which contains no cycles.

The previous result said that only DAGs can be topologically ordered. However, is the reverse true? That is, is it the case that every DAG has a topological ordering? **Yes**.

3.2.4 Existence of Orderings

Theorem 3.1

Let G be a (finite) DAG. Then, G has a topological ordering.

Proof. We consider the last vertex in the ordering; in other words, whatever vertex comes last in this ordering. This vertex must be a sink, or a vertex with no outgoing edges. So, once we find the sink, we can put the graph at the end of the topological graph, and then order the remaining vertices. Now, using the lemma below, we want to show that every DAG has a topological ordering. We will use induction on $|G|$. Omitting the base case, we find a sink v. Then, create a graph $G' = G \setminus \{v\}$. We can inductively order G', which is still a DAG. Then, add v to the end of the ordering. □

3.2.5 Sinks

Lemma 3.1

Every finite DAG contains at least one sink.

Proof. Start at a vertex $v = v_1$. Then, we can "follow the trail," or in other words follow the edges (v_1, v_2), (v_2, v_3), ... Eventually, we will either find:

- Some vertices repeat (which creates a cycle). This can't happen if this is a DAG, though.
- Or, we get stuck (which means we found a sink).

So, we are done. □

3.2.6 Algorithm

Suppose we want to design an algorithm that, given a DAG G, computes a topological ordering on G. We can use the proof to create a naive algorithm.

- Find a sink v. This is done by following a chain of vertices until we are stuck.
- Compute the ordering on $G - \{v\}$.
- Place v at the end.

This algorithm can be written like so:

```
TopologicalOrdering(G)
    If |G| = 0
        Return {}
    Let v in G
    While there is an edge (v, w)
        v = w
    Return (Ordering(G - v), v)
```

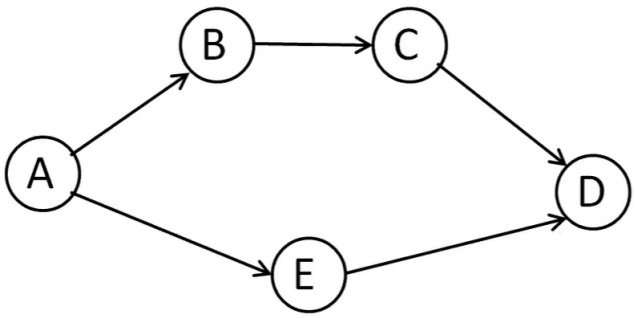
The runtime is $O(|V|^2)$. This is because we need $|V|$ time to find each sink and have $|V|$ sinks. This is suboptimal, however. Consider this optimal algorithm:

```
TopologicalOrdering(G)
    Run DFS(G) w/ Pre/Post Numbers
    Return Vertices in Reverse Postorder
```

This runs in $O(|V| + |E|)$.

3.2.7 Example: DAG

Consider the following DAG which we will call G:



We start at A since this is where the graph "begins."

- Using Algorithm 1:
  - First, we enumerate through the graph G until we find a sink. In this case:
    $$A \mapsto B \mapsto C \mapsto D$$
  - We *remove* D from the graph and add it to the ordering.
    ```
    D
    ```
  - Now, we enumerate through the graph $G - \{D\}$. In this case:
    $$A \mapsto B \mapsto C$$
  - We *remove* C from the graph and add it to the ordering.
    ```
    C -> D
    ```
  - Now, we enumerate through the graph $G - \{C, D\}$. In this case:
    $$A \mapsto B$$
  - We *remove* B from the graph and add it to the ordering.
    ```
    B -> C -> D
    ```
  - Now, we enumerate through the graph $G - \{B, C, D\}$. In this case:
    $$A \mapsto E$$
  - We *remove* E from the graph and add it to the ordering.
    ```
    E -> B -> C -> D
    ```
  - As we only have A left, we can add it to the ordering.
    ```
    A -> E -> B -> C -> D
    ```

    So, we are done.
- Using Algorithm 2:
  - First, we enumerate through the graph G until we find a sink. In this case:
    $$A \mapsto B \mapsto C \mapsto D$$

- We remove D from the graph and add it to the ordering.

```
D
```

- Now, we remember that we already made it to C and that C is a sink in $G - \{D\}$. So, we remove C and add it to the ordering:

```
C -> D
```

- We remember that we already made it to B and that B is a sink in $G - \{C, D\}$. So, we remove B and add it to the ordering:

```
B -> C -> D
```

- Now that we're back at A, we note that A has another edge. So, we go to that edge until we find a sink. In this case:

$$A \mapsto E$$

- We remove E from the graph and add it to the ordering.

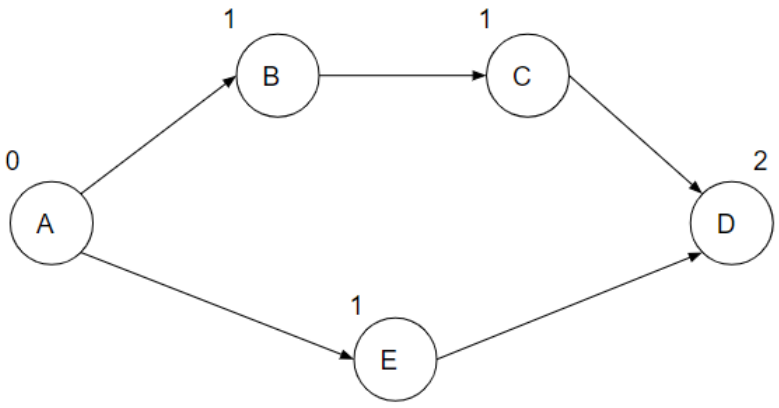
```
E -> B -> C -> D
```

- Now, A is the only vertex left in $G - \{B, C, D, E\}$. So, we add it to the ordering:

```
A -> E -> B -> C -> D
```

Here, we notice that algorithm 2 resembles depth-first seearch!

Another way to find a topological ordering for this graph is to consider the in-degrees of each vertex; i.e. the number of edges connecting **to** it. If we label each of G's vertices with its in-degrees, we will have:



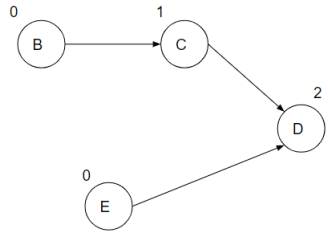
So, this makeshift algorithm is essentially:

```
TopologicalOrdering(G)
    ordering = []
    while G is not empty
        v = vertex in G with lowest in-degree (i.e. in-degree 0)
        ordering.append(v)
        Remove v from G
        Update in-degrees of existing vertices
    return ordering
```

- First, note that A has the lowest in-degree. So, remove A from the graph and add it to the ordering. The ordering now looks like:

```
[A]
```

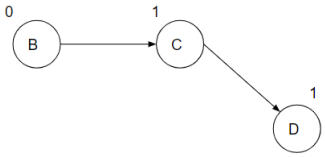
- The graph now looks like:



Here, B and E both have the lowest in-degree. So, we can remove one or the other. For the sake of consistency, remove E from the graph and add it to the ordering. The ordering now looks like:

```
[A, E]
```

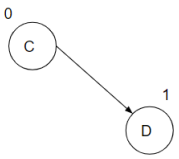
- The graph now looks like:



Here, note that B is the only node with the lowest in-degree. So, we remove it and add it to the ordering. The ordering now looks like:

```
[A, E, B]
```

- The graph now looks like:



Here, C has the lowest in-degree so we remove it and add it to the ordering. The ordering now looks like:

```
[A, E, B, C]
```

- By now, it's trivial to see that D has in-degree 0, so we add it.

```
[A, E, B, C, D]
```

  And we are done.

3.2.8 Topological Sort

This is a particularly useful algorithm.

- Many graph algorithms are relatively easy to find the answer for v if you've already found the answer for everything downstream of v.
  - We can topologically sort G.
  - Then, solve for v in reverse topological order.

3.3 Problem: Strongly Connected Components & Metagraphs

In undirected graphs, we had a very clean description of reachability: v was reachable from w if and only if they were in the same connected component. Well, this no longer works for digraphs. How do we achieve a clean description of reachability in a directed graph?

- Note that reachability is no longer symmetric. That is, we can reach w from v but not the other way around.
- Can we maybe allow reachability in either direction?
- Maybe we can allow the ability to follow edges in either direction? However, this treats a digraph as an undirected graph.

Definition 3.5: Strongly Connected Components

In a directed graph G, two vertices v and w are in the same **strongly connected component** if v is reachable from w and w is reachable from v.

3.3.1 Equivalence Relation

Let $v \sim w$ if v is reachable from w and vice versa.

Proposition. *This is an equivalence relation. Namely:*

- $v \sim v$ *(v is reachable from itself).*
- $v \sim w \implies w \sim v$ *(relation is symmetric).*
- $u \sim v$ *and* $v \sim w \implies u \sim w$.

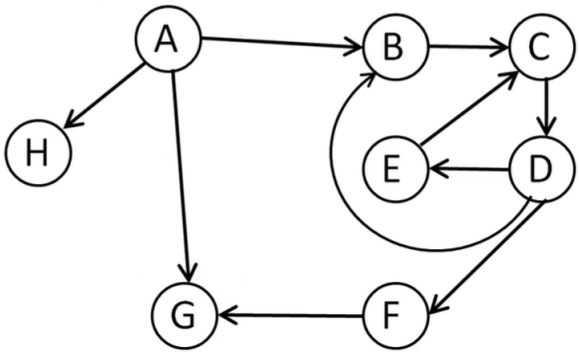
3.3.2 Relationship to Components

Essentially, when we have this equivalence relation, we can split a set into components (equivalence classes) so that $v \sim w$ if and only if v and w are in the same component.

Take any v, the set of all w so that $v \sim w$ is the component of v. Then, everything connects to everything else in this equivalence class and does not connect (both ways) to anything outside.

3.3.3 Example: SCCs

Consider the following graph:



How many strongly connected components does this graph have?

The answer is **5**. If we use the idea of equivalence classes above, then the equivalence classes are:

- B, C, D, E.
- A
- H
- G
- F

Here, we note that B, C, D, and E are all reachable from each other. However, these vertices and, for example, A are not reachable both ways.

3.3.4 Connectivity

Do strongly connected components completely describe connectivity in G?

- **No**. In directed cases, we can have an edge between strongly connected components. In the undirected case, connected components told you everything that there was to know. However, in the directed case, we can have these two strongly connected components consisting of just a single vertex v and w. However, just knowing these components doesn't tell you if there is an edge from v to w or vice versa. There could be an edge that could be extra information that we need to know.

```
[v] ---------> [w]
```

- We need extra information to describe how SCCs connect.

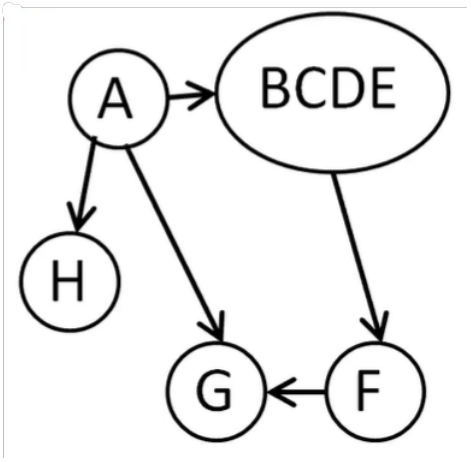
3.3.5 Metagraph

Definition 3.6: Metagraph

The **metagraph** of a directed graph G is a graph whose vertices are the strongly connected components of G, where there is an edge between C_1 and C_2 if and only if G has an edge between some vertex of C_1 and some vertex of C_2.

Remark: If you're given the strongly connected components and the metagraph of a graph G, then you can you can figure out connectivity within the full graph.

So, the corresponding metagraph of the above graph in the example would be:



3.3.6 Result

Theorem 3.2

The metagraph of any directed graph is a DAG.

Proof. Assume for the sake of contradiction that it is not a DAG. Then, the metagraph has a cycle. Now, let's suppose we have a bunch of strongly connected components; call these SCCs $C_1, C_2, \ldots, C_n$. Then, since the metagraph has a cycle, then the strongly connected components form a cycle. That is, we can go from C_1 to C_2 to C_n back to C_1. However, this implies that we have one giant strongly connected component since we can get from one C_i to another C_j $(i \neq j)$ and vice versa. □

3.3.7 Easy Algorithm to Compute SCC & Metagraph

Given a directed graph G, compute the SCCs of G and its metagraph.

- For each v, compute vertices reachable from v.
- Find pairs v, w so that v reachable from w and vice versa.
- For each v, the corresponding w's are in the SCC of v.

The runtime is $O(|V|(|V| + |E|))$.

3.3.8 Observation and Better Algorithm

Suppose that $SCC(v)$ is a sink in the metagraph.

- G has no edges from $SCC(v)$ to another SCC.
- We can run `explore(v)` to find all vertices reachable from v. This will contain all vertices in $SCC(v)$. But, it contains no other vertices.
- If v is in the SCC, then `explore(v)` finds exactly v's component.

With this observation, we consider the following strategy:

- Find v in a sink SCC of G.
- Run `explore(v)` to find the component C_1.

- Repeat process on $G \setminus C_1$.

The problem is, how do we find v that is in a sink?

Proposition. *Let C_1 and C_2 be SCCs of G with an edge from C_1 to C_2. If we run `DFS` on G, the largest postorder number of any vertex in C_1 will be larger than the largest postorder number in C_2.*

The reason why we care is because if v is the vertex with the largest postorder number, then:

- There is no edge to $SCC(V)$ from any other SCC.
- SCC is a source SCC.

However, we wanted a *sink* SCC. So, how do we relate these two?

- A sink is like a source, only with edges going in the opposite direction.

So, we define a reverse graph like so:

Definition 3.7: Reverse Graph

Given a directed graph G, the **reverse graph** of G (denoted G^R) is obtained by reversing the directions of all the edges of G.

Some properties of reverse graphs are:

- G and G^R have the same number of vertices and edges.
- $G = (G^R)^R$.
- G and G^R have the same SCCs.
- The sink SCCs of G are the source SCCs of G^R.
- The source SCCs of G are the sink SCCs of G^R.

The better algorithm we have is:

```
SCCs(G)
    Run DFS(G^R), record postorder
    Find v with largest v.post
    Set all vertices unvisited
    Run explore(v)
    Let C be the visited vertices
    Return SCCs(G - C) union {C}
```

The issue with our algorithm is that we recompute the postorder for every SCC we need to find. However, we don't need to do this; rather, after removing some strongly connected components to get G', the largest postorder number of vertices in G' is still in a sink component of G'.

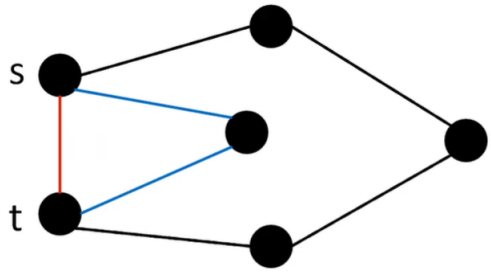
3.3.9 Better Algorithm

```
SCCs(G)
    Run DFS(G^R), record postorders
    Mark all vertices as unvisited
    For v in V in reverse postorder
        If v not in a component yet     // if v is not visited
            explore(v) on G-components found,
            marking new component
```

So, really, this is just 2 DFSs, so the runtime is $O(|V| + |E|)$.

4 Paths in Graphs

DFS and `explore` allow us to determine *if* it is possible to get from one vertex to another, and using the DFS tree, you can also find a path. However, this is often not an efficient path. This is because DFS picks one path and tries to go to the end of that path before trying a different path. If that path happens to be a not-so-ideal path, then we've taken a much longer path than necessary. For example, consider the following graph:



Suppose we wanted to get from s to t. Using DFS, we might take the black path; the one with 4 edges. How do we guarantee that we take the shortest path, i.e. the red path?

4.1 Problem: Finding Shortest Path in Unweighted Graph

Given a graph G with two vertices s and t in the same connected component, how do we find the *best* path from s to t? In fact, what do we mean by the best path?

- Least expensive.
- Best scenery.
- Shortest.

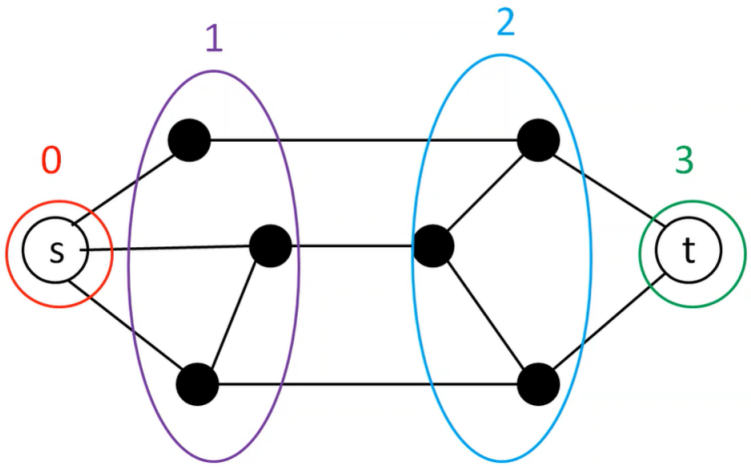
For now, we want the fewest edges.

4.1.1 Observation

Proposition. *If there is a path from vertices s to v with length at most d, then there is some w adjacent to v where there is a path a length at most $(d-1)$ from vertices s to w.*

Proof. If w is the next-to-last vertex on the path, then there are d edges to get you from s to v and that means the number of edges you have to take from s to w is only $d-1$. This means that if we know all of the vertices with distance at most $d-1$, then we can find all of the vertices with distance at most d. □

For example, consider the following (annotated) graph:



Here, suppose we start at s.

- The only place we can get with a path of length 0 is s, or the one vertex in the red circle labeled `0`.
- If we want to travel one edge, the only places we can get with a path of length one are the vertices in the purple circle labeled `1`.
- If we want to travel two edges, we can get to some vertex that is adjacent to one of the vertices in the purple circle. For each of the purple vertices, if we list all of the adjacent vertices, the only new vertices we can reach are the vertices in the cyan circle labeled `2`.
- Finally, if we want to travel three edges, we can get to something that is adjacent to the vertices at distance 2, which means we can get to t, or the vertex in the green circle labeled `3`.

By the way we constructed these sets (the circles), it's not possible to get to t at distance 2 because, for that to happen, it has to be adjacent to the vertices at distance 1, which it isn't.

4.1.2 Algorithm Idea

This observation gives us a pretty reasonable way to find an algorithm. In particular, if we want to find the best path length between s and t, we really need to find the best path length between s and every other vertex in the graph.

So, the idea is, for each d, create a list of all vertices at distance d from s.

- For $d = 0$, this is just $\{s\}$.
- For larger d, we want all new vertices adjacent to vertices at distance $d - 1$.

4.1.3 Algorithm

The algorithm is as follows:

```
1    ShortestPaths(G, s)
2        Initialize Array A
3        A[0] = {s}
4        dist(s) = 0
5        For d = 0 to n
6            For u in A[d]
7                For (u, v) in E
8                    if dist(v) undefined
9                        dist(v) = d + 1
10                       add v to A[d + 1]
```

4.1.4 Improving the Algorithm

How can we improve this?

- What if `dist(v)` undefined at end? We can set the distances of all vertices to infinity.[1]

```
1    ShortestPaths(G, s)
2+       For each v in V, dist(v) = infinity
3        Initialize Array A
4        A[0] = {s}
5        dist(s) = 0
6        For d = 0 to n
7            For u in A[d]
8                For (u, v) in E
9c                   if dist(v) = infinity
10                       dist(v) = d + 1
11                       add v to A[d + 1]
```

  Here, at the end of the algorithm, any vertices that could not be found will be assigned a distance of infinity to indicate that it could not be reached.

- The algorithm goes through `A[0]`, `A[1]`, and so on in order. We can just use a queue.

```
1    ShortestPaths(G, s)
2        For each v in V, dist(v) = infinity
3c       Initialize Queue Q
5        dist(s) = 0
6c       While Q not empty
7c           u = front(Q)
8            For (u, v) in E
9                if dist(v) = infinity
10c                  dist(v) = dist(u) + 1
11c                  Q.enqueue(v)
```

- What if we want to keep track of the paths?

```
1    ShortestPaths(G, s)
2        For each v in V, dist(v) = infinity
3        Initialize Queue Q
4        dist(s) = 0
5        While Q not empty
6            u = front(Q)
7            For (u, v) in E
8                if dist(v) = infinity
9                    dist(v) = dist(u) + 1
10                   Q.enqueue(v)
11+                  v.prev = u     // Keep track of path
```

  With this change, we can simply follow the chain of `previous` vertices, which is the path that was taken.

[1] Note that `+` next to a line number means added code and `c` next to a line number means changed line.

4.1.5 Breadth First Search

In our last change above, we note that we simply have BFS.

```
BFS(G, s)
    For v in V, dist(v) = infinity
    Initialize Queue Q
    Q.enqueue(s)
    dist(s) = 0
    While Q is not empty
        u = front(Q)
        For (u, v) in E
            If dist(v) = infinity
                dist(v) = dist(u) + 1
                Q.enqueue(v)
                v.prev = u
```

The total runtime is $O(|V| + |E|)$.

- In the first few lines, we have $O(|V|)$ time.
- In the while loop and the `front(Q)` lines, we have $O(|V|)$ iterations.
- In the edge iteration, we have $O(|E|)$ iterations.

4.1.6 DFS vs. BFS

- Similarities:
  - The way both algorithms process vertices is the same (`visited` for DFS vs. `dist < infinity` for BFS).
  - For each vertex, process all unprocessed neighbors.
- Differences:
  - DFS uses a stack to store vertices waiting to be processed.
  - BFS uses a queue.
- Big Effect:
  - DFS goes depth-first: very long path. Get a very "skinny" tree.
  - BFS is breadth first: visits all side paths. Get a very shallow tree since we process all of the neighbors.

4.1.7 Edge Length

The number of edges in a path is not always the right measure of distance. Sometimes, taking several shorter steps is preferably to taking a few longer ones.

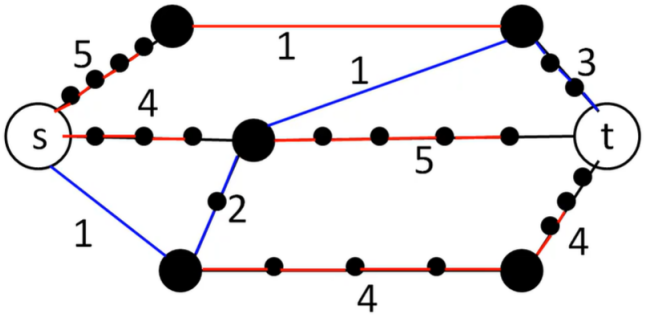
We can assign each edge (u, v) a non-negative length $\ell(u, v)$. The length of a path is the sum of the lengths of its edges.

4.2 Problem: Finding Shortest Path in Weighted Graph

Given a graph where each edge has some length ℓ, how do we find the shortest path between two vertices?

4.2.1 Trivial Way

The idea is the break edges into unit length edges. So, an edge of length 5 can actually be seen as 5 edges of length 1. With this conversion, we can run BFS.



Here, we see that the blue path has distance 7 (the lowest) while the red paths are the other paths that the BFS algorithm took.

Now, this *works*. However, there are some issues.

- BFS is linear time for the graph that we run it on. However, we're breaking this graph up so every edge can be represented as a unit edge (so an edge of weight n becomes n edges of weight 1). The runtime is $\mathcal{O}$(Sum of Edge Lengths).
- What if we have an edge of length one million?
- What if our edges have decimal weights (e.g. π, 15.12314352452, so on)?

4.2.2 Another Way

If we have very long edge lengths, most steps will just consist of advancing slightly (one unit) along a bunch of edges. The issue is, there may be a time period where all we're doing is advancing along one unit for each path that BFS is taking.

So, how do we "fast forward" through these boring steps? Well, occasionally, we have an interesting step where, for instance, the wavefront hits a new vertex. In other words, every time you hit a new vertex, you might need to update some things. But, ignoring that, all we're really doing is slowly advancing through each edge.

4.2.3 Algorithm

This algorithm is associated with the "ooze" that was mentioned in lecture.

```
Distances(G, s, l)
    dist(v) = 0
    While not all distances found
        Find minimum over (v, w) in E
            with v discovered w not
            of dist(v) + l(v, w)
        dist(w) = dist(v) + l(v, w)
        prev(w) = v
```

4.2.4 Why Does This Work?

Proposition. *Whenever the algorithm assigns a distance to a vertex v, that is the length of the shortest path from s to v.*

Proof. We use induction.

Base Case: We know that `dist(s) = 0`. The empty path has length 0.

Inductive Step: When assigning distances to w, suppose that all previously assigned distances are correct. We can imagine a bubble containing the vertices s and v, with edge denoted by dist(v). By the inductive hypothesis, we assume that all edge lengths inside this bubble is correct. Take a vertex w which is outside of the bubble so that the path length from v (inside the bubble) to w has length $\ell(v, w)$. This is the shortest path from s to any vertex outside the bubble.

□

4.2.5 Runtime of Initial Algorithm

This runs in $\mathcal{O}(|V| \cdot |E|)$ time. There are $\mathcal{O}(|V|)$ iterations and $\mathcal{O}(|E|)$ edges.

- This is too slow because every iteration we have to check every edge.
- The idea is that most of the comparison doesn't change much iteration to iteration. So, we can use this to save time.
- Specifically, record for each w the best value of dist(v) + $\ell(v, w)$.

4.2.6 Better Algorithm

Our better algorithm is based on the observations that we had above.

```
Distances(G, s, l)
    For v in V
        dist(v) = infinity
        done(v) = false
    dist(s) = 0
    done(s) = false

    while not all vertices done
        Find v not done with minimum dist(v)
        done(v) = true
        For (v, w) in E
            if dist(v) + l(v, w) < dist(w)
                dist(w) = dist(v) + l(v, w)
                prev(w) = v
```

The initialization is $O(|V|)$, and the while loop is $\mathcal{O}(|V|)$. The for loop is $\mathcal{O}(|E|)$ time. Thus, the runtime is:

$$\mathcal{O}(|V|^2 + |E|)$$

- This repeatedly asks for the smallest vertex. Even though not much is changing from round to round, the algorithm is computing the minimum from scratch every time.
- We can use a data structure to help answer a bunch of similar questions faster than answering each question individually (like the one above).

4.2.7 Priority Queue

A **priority queue** is a data structure that stores elements sorted by a **key** value. Its operations are:

- `insert`: Adds a new element to the priority queue.
- `decreaseKey`: Changes the key of an element of the priority queue to a specified smaller value.
- `deleteMin`: Deletes the element with the lowest key from the priority queue.

4.2.8 Even Better Priority Queue

```
Dijkstra(G, s, l)
    Initialize Priority Queue Q
    For v in V
        dist(v) = infinity
        Q.insert(v) // using dist(v) as key
    dist(s) = 0
    while Q not empty
        v = Q.deleteMin()
        For (v, w) in E
            if dist(v) + l(v, w) < dist(w)
                dist(w) = dist(v) + l(v, w)
                prev(w) = v
                // w has a faster path and needs to updated in
                // the priority queue
                Q.decreaseKey(w)
```

The runtime is as follows:

- We need to iterate $\mathcal{O}(|V|)$ times for the initial loop.
- In the `while` loop, we run $\mathcal{O}(|V|)$ times.
- We need to run through the edges $\mathcal{O}(|E|)$ times.

So, $\mathcal{O}(|V|)$ `insert`s + $\mathcal{O}(|V|)$ `deleteMin`s + $\mathcal{O}(|E|)$ `decreaseKey`. The runtime of Dijkstra's algorithm now depends on what priority queue we use.

4.2.9 Priority Queue Implementations

Here, we go over some implementations of priority queues.

- **Unsorted List:** Store n elements in an unsorted list.

Operation	Runtime	Explanation
`Insert`	$\mathcal{O}(1)$	Add it at the end of the list.
`DecreaseKey`	$\mathcal{O}(1)$	Assuming the operation comes with a pointer to where the element is stored in your data structure, take that element and decrease its key value.
`DecreaseMin`	$\mathcal{O}(n)$	We need to potentially scan the entire list.

  For Dijkstra, we would have $\mathcal{O}(|V|^2 + |E|)$.

- **Binary Heap:** Store elements in a balanced binary tree with each element having smaller key value than its children.

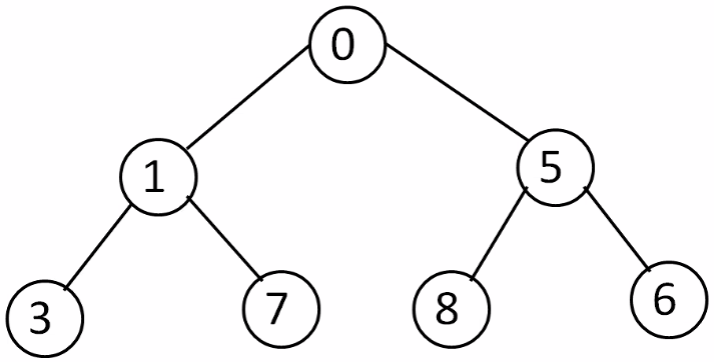


Figure: A binary heap.

The smallest key is at the top (`0`) and there are $\log n$ levels.

Operation	Runtime	Explanation
`Insert`	$\mathcal{O}(\log(n))$	Add the key at the bottom, then bubble the new key up until it's in the right place.
`DecreaseKey`	$\mathcal{O}(\log(n))$	We need to change the key. Then, we might need to bubble up the changed key until it's in the right place.
`DecreaseMin`	$\mathcal{O}(\log(n))$	We remove and then return the root node. Then, we move the bottom-most node to the root. After this, we might need to continuously bubble down the root node until it's in the right place.

For Dijkstra, we would have $\mathcal{O}(\log(|V|)(|V|+|E|))$.

- **d-ary Heap:** This is like a binary heap, but each node has d children.

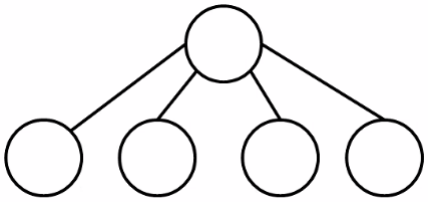


Figure: A 4-ary heap.

There are $\frac{\log(n)}{\log(d)}$ levels, so bubble up is faster. However, bubble down is slower since we need to compare more children.

Operation	Runtime	Explanation
`Insert`	$\mathcal{O}\left(\frac{\log(n)}{\log(d)}\right)$	Same thing as binary heap, essentially.
`DecreaseKey`	$\mathcal{O}\left(\frac{\log(n)}{\log(d)}\right)$	Same idea as binary heap, *but* the bubbling up is a lot faster.
`DecreaseMin`	$\mathcal{O}\left(\frac{d\log(n)}{\log(d)}\right)$	This is because, for bubble down, we need to compare more children; specifically, the d children.

For Dijkstra, we would have $\mathcal{O}\left(\frac{\log(|V|)(d|V|+|E|)}{\log(d)}\right)$. If the number of edges is substantially greater than the number of vertices, this can potentially be an improvement.

- **Fibonacci Heap:** This is an advanced data structure that uses amortization[2]. We are not concerned with its implementation.

Operation	Runtime
`Insert`	$\mathcal{O}(1)$
`DecreaseKey`	$\mathcal{O}(1)$
`DecreaseMin`	$\mathcal{O}(\log(n))$

For Dijkstra, we would have $\mathcal{O}(|V|\log(|V|)+|E|)$. This is the most ideal runtime, and thus the runtime that we can assume.

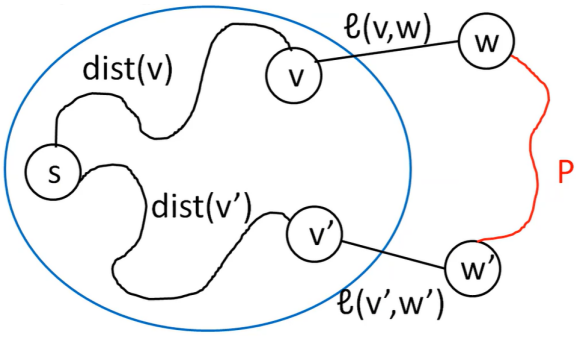
Therefore, with a Fibonacci Heap, Dijkstra's algorithm would have a runtime of $\boxed{\mathcal{O}(|V|\log(|V|)+|E|)}$.

[2]So, you might spend more time on a particular operation, but the overall runtime will be "consistent."

4.3 Problem: Finding Shortest Path in Unweighted Graph with Negative Edge Weights

So far, we've talked about non-negative lengths. However, depending on what we're representing as lengths, we might have *negative* lengths. That being said, the problem statement is the same - find the path with the smallest sum of edge weight.

Right now, Dijkstra's algorithm doesn't actually work on negative edge values. To see why this is the case, consider the following visualization:



Here, the bubble represents all correctly assigned distances. In other words, here, s and and v has edge length $\text{dist}(v)$, the shortest distance between the two. Then, from v (inside the bubble) to w (outside the bubble), if we consider the edge length $\ell(v, w)$, it follows that this must be the shortest path. The point was that this must be the right distance to w; any other way to get to w means that we would have to leave the bubble some other way (in our case, from s to v' to w'). Then, there has to be a path connecting w' to w, of which we denote P. So, it follows that:

$$\text{dist}(v) + \ell(v, w) \leq \text{dist}(v') + \ell(v', w') + \ell(P)$$

This correctly works if $\ell(P)$ is non-negative. However, if $\ell(P)$ is negative, we run into an issue.

Thus, the problem becomes: find the shortest path from one vertex to another when there are potnetially negative edge weights.

4.3.1 Negative Weight Cycles

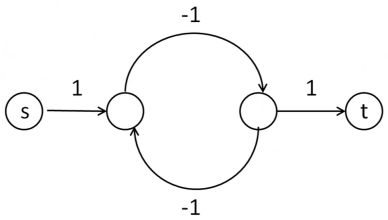
Definition 4.1

A **negative weight cycle** is a cycle where the total weight of edges is negative.

Remarks:

- If G has a negative weight cycle, then there are probably no shortest paths since we can go around the cycle over and over again.
- For an undirected graph G, a single negative weight edge gives a negative weight cycle by going back and forth on it. So, we usually don't talk about the negative edge weight in the context of an undirected graph.

For example, consider the following graph:



Here, the shortest path length from s to t is as small as we like it to be. So, really, the shortest path length is $-\infty$. This graph**also** answers an important question:

> What stops us from adding a constant to every edge length so every edge length is non-negative, thus letting us run Dijkstra's algorithm?

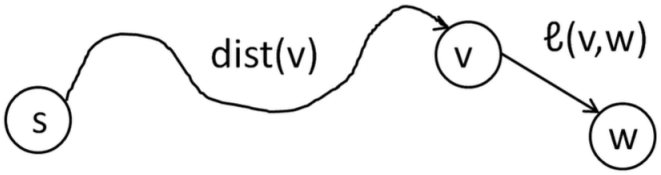
Well, if we added a constant to every edge length so that every edge length is non-negative, then there would only be one answer. For example, if we added 1 to each edge length, the shortest path becomes $2+0+2=4$. But since we added 1 to each edge length and we went through three edges, our final answer is 1. *However*, this completely avoids the fact that we could just loop around the negative cycle many times.

4.3.2 Fundamental Shortest Paths Formula

For any vertex w that isn't the source s, $w \neq s$,

$$\text{dist}(w) = \min_{(v,w)\in E} \text{dist}(v) + \ell(v,w)$$

This says that the distance of w is equal to the minimum over all edges v to w in E of the distance to v plus the length of the edge from v to w.



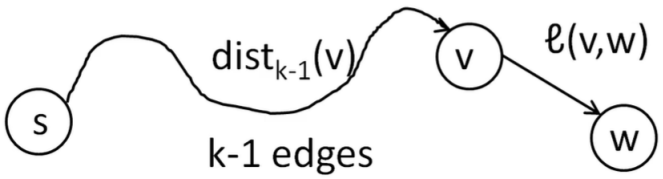
Looking at the visualization above, we're saying that the path from s to v has length $\text{dist}(v)$; this is the shortest path from s to v.

We can use a system of equations to solve for the distances from s to ever other vertex in the graph. When $\ell \geq 0$, Dijsktra gives an order to solve in. But, with negative edge weights, this order is no longer clear.

4.3.3 Algorithm Idea

Instead of finding the shortest paths, which may not exist due to a negative edge cycle, we instead find the shortest paths of length at most k edges. So, for $w \neq s$, we have:

$$\text{dist}_k(w) = \min_{(v,w)\in E} \text{dist}_{k-1}(v) + \ell(v,w)$$



If we look at a path from s to w with at most k edges, this is a path from s to v that uses at most $k-1$ edges for some v plus a single edge from v to w. The best length this path could have is $\text{dist}_{k-1}(v)+\ell(v,w)$, where our $\text{dist}_{k-1}(v)$ is minimized.

4.3.4 Bellman-Ford Algorithm

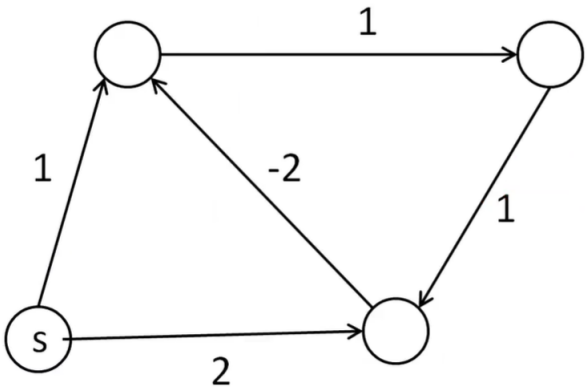
This formula gives rise to the *Bellman-Ford* algorithm.

```
Bellman-Ford(G, s, l)
    dist_{0}(v) = infinity for all v
    dist_{0}(s) = 0
    For k = 1 to n
        For w in V
            dist_{k}(w) = min(dist_{k}(w), dist_{k - 1}(v) + l(v, w))
        dist_{k}(s) = min(dist_{k}(s), 0)
```

We note that each iteration of the outer-loop is $\mathcal{O}(|E|)$ time. However, what value of k do we use?

4.3.5 Example: Applying the Bellman-Ford Algorithm

Find the shortest path from s to every other vertex in the graph shown below. For convenience, let the top-left vertex be denoted A, the top-right vertex be B, and the bottom-right vertex be C.



- First, when $k = 0$, $s = 0$ and everything else is assigned ∞; you can't get to anywhere else with no edges. Therefore, the distances are:

k	s	A	B	C
0	0	∞	∞	∞

- When $k = 1$, we consider all vertices that we can reach with one edge. So, you still can't reach vertex B since it needs at least two edges. But, we can get from s to A with path length 1, and we can get from s to C with path length 2. So:

k	s	A	B	C
0	0	∞	∞	∞
1	0	1	∞	2

  In other words, these are the shortest distances we can get from a path of length one to each of our vertices.

- When $k = 2$, we consider all vertices that we can reach with two edges. First, we can reach vertex B with just two edges (from s to A and from A to B). A has distance 1 so B must have distance 2. We can't do any better with s or C, but note that we can actually improve A (if we go from s to C and from C to A) by getting it down to distance 0. Therefore:

k	s	A	B	C
0	0	∞	∞	∞
1	0	1	∞	2
2	0	0	2	2

- When $k = 3$, we consider all vertices that we can reach with three edges. So, we can again reach every vertex. In particular, note we can update vertex B's distance. This is because if A is 0, then there is a path from A to B that has length 1 ($s \to C \to A \to B$). So:

k	s	A	B	C
0	0	∞	∞	∞
1	0	1	∞	2
2	0	0	2	2
3	0	0	1	2

- Past $k = 3$, we notice that the distances no longer change. In other words, the process has stabilized.

4.3.6 Analysis

Proposition. *If $n \geq |V| - 1$ and if G has no negative weight cycles, then for all v,*

$$dist(v) = dist_n(v)$$

This says that if we run the Bellman-Ford algorithm, there is a limit. Assuming there are no negative weight cycles[3], we only need to run the algorithm for $|V| - 1$ rounds for a final runtime of $\mathcal{O}(|V||E|)$.

Proof. We need to show that the shortest path has fewer than $|V|$ edges. Suppose that there is a path that has at least $|V|$ edges, then by the pigeonhole principle, it must contain the same vertex twice. This means that there is a loop. If we remove the loop (which we assume has non-negative total weight, i.e. not a negative weight cycle), then we get a shorter path. This new path has at most $|V| - 1$ edges since we can only hit each vertex at most once. Note that this path is no longer than the one with the loop since we're considering the *shortest* distance. □

4.3.7 Revised Bellman-Ford Algorithm

With this analysis in mind, our algorithm looks like:

```
Bellman-Ford(G, s, l)
    dist_{0}(v) = infinity for all v
    dist_{0}(s) = 0
    For k = 1 to |V|
        For w in V
            dist_{k}(w) = min(dist_{k}(w), dist_{k - 1}(v) + l(v, w))
        dist_{k}(s) = min(dist_{k}(s), 0)
    Return dist_{|V|}(t)
```

Which we now know computes the shortest paths *if no negative weight cycles* in $\mathcal{O}(|V||E|)$ time.

4.3.8 Detecting Negative Cycles

If there are no negative weight cycles, Bellman-Ford computes the shortest paths. Suppose there *are* negative weight cycles. Well, Bellman-Ford will calculate some distances which will probably be garbage values.

How do we know whether or not there are any negative weight cycles?

[3] If there is a negative weight cycle, there is probably no shortest path.

4.3.9 Negative Cycle Detection

Proposition. *For any $n \geq |V| - 1$, there are **no** negative weight cycles reachable from s if and only if, for every $v \in V$:*

$$dist_n(v) = dist_{n+1}(v)$$

So, to detect negative cycles, all we need to do is run one more round of Bellman-Ford and see if any distances change.

Proof. Suppose no negative weight cycles exist. Then, for any $n \geq |V| - 1$, $\text{dist}_n(v) = \text{dist}(v)$. So, $\text{dist}_n(v) = \text{dist}(v) = \text{dist}_{n+1}(v)$ by the transitive property (since $n + 1 \geq n \geq |V| - 1$).

Suppose $\text{dist}_n(v) = \text{dist}_{n+1}(v)$ for all v. Then:

$$\begin{aligned}\text{dist}_{n+2}(w) &= \min_{(v,w)\in E}(\text{dist}_{n+1}(v) + \ell(v, w)) \\ &= \min_{(v,w)\in E}(\text{dist}_n(v) + \ell(v, w)) \\ &= \text{dist}_{n+1}(w)\end{aligned}$$

We essentially apply the idea that if $\text{dist}_n(v) = \text{dist}_{n+1}(v)$, then the same idea holds for $n+1$. In other words, if the distances are the same for one round from n to $n + 1$, then it will be the same for $n + 1$ to $n + 2$ and so on. If the distance functions stabilize for one round, they will stabilize forever. So:

$$\text{dist}_n(v) = \text{dist}_{n+1}(v) + \text{dist}_{n+2}(v) + \text{dist}_{n+3}(v) + \ldots$$

However, if there were a negative weight cycle, the distances would decrease eventually. □

4.4 Problem: Finding Shortest Paths in DAGs (With Arbitrary Edge Weights)

We saw that shortest paths is harder when we needed to deal with negative weight cycles. For general graphs, we needed to use Bellman-Ford which is much slower than our other algorithms. In our case here, if we're working with a DAG, then there are faster algorithms that we can apply.

Recall that, for any vertex w that isn't the source s, $w \neq s$,

$$\text{dist}(w) = \min_{(v,w)\in E} \text{dist}(v) + \ell(v, w)$$

We can use topological ordering for DAGs to compute the shortest distance.

4.4.1 Algorithm

The algorithm is as follows:

```
ShortestPathsInDAG(G, s, l)
    TopologicalSort(G)
    For w in V in topological order
        If w = s
            dist(w) = 0
        Else
            dist(w) = min(dist(v) + l(v, w))
```

This has runtime $\mathcal{O}(|V| + |E|)$.

4.5 Shortest Path Algorithms Summary

Path Type	**Algorithm**	**Runtime**
Unit Weights, General Graph	`Breadth First Search`	$\mathcal{O}(\|V\| + \|E\|)$
Non-Negative Weights, General Graph	`Dijsktra`	$\mathcal{O}(\|V\| \log(\|V\|) + \|E\|)$
Arbitrary Weights, General Graph	`Bellman-Ford`	$\mathcal{O}(\|V\|\|E\|)$
Arbitrary Weights, DAG	`ShortestPathsInDAG`	$\mathcal{O}(\|V\| + \|E\|)$

5 Divide and Conquer

The idea behind divide and conquer is as follows:

1. Divide: Break the given problem into similar pieces (i.e. subproblems of same type). This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible.
2. Conquer: Recursively solve these pieces (sub-problems). This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.
3. Combine: Appropriately combine the answers. When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem.

5.1 Problem: Integer Multiplication

Suppose we're given two n-bit binary numbers and are asked to find their product.

5.1.1 Naive Algorithm

The naive algorithm is to do multiplication like we would from elementary school. This runs in $\mathcal{O}(n^2)$ time because we need to write down $\mathcal{O}(n^2)$ bits of numbers to add (addition is done in linear time and is omitted).

5.1.2 Improving the Algorithm: Two-Digit Multiplication

Suppose we tried to multiply ab by cd. This is normally done like so:

```
          a       b
      x   c       d
          ---------
          ad      bd
+   ac    bc       0
    ---------------
    ac    ad+bc    bd
```

This requires 4 one-digit multiplications and one addition. The **trick** is to compute ac, bd, $(a+b)(c+d)$. We note that (*digit-wise*):

$$bc + ad = (a+b)(c+d) - ac - bd$$

This requires *3* one-digit multiplications and 4 addition/subtractions.

5.1.3 Improving the Algorithm: Larger Base

We can essentially generalize the above to larger bases with more digits. For example:

```
      a1      a2      a3   ...a(n/2)      a(n/2 + 1)   ...an
x     b1      b2      b3   ...b(n/2)      b(n/2 + 1)   ...bn
      ------------------------------------------------------
      AC                      AD + BC                     BD
```

Where we can split the digits so that A takes the first half of the first number, B takes the second half of the first number, C takes the first half of the second number, and D takes the second half of the second number.

5.1.4 Formally

Suppose we wanted to multiply N by M. Then:

1. Let $X \approx \sqrt{N+M}$ be a power of 2.
2. Write $N = AX + B$ and $M = CX + D$. This can be done by just taking the high and low bits.
3. $NM = ACX^2 + (AD + BC)X + BD = ACX^2 + ((A+B)(C+D) - AC - BD)X + BD$, where the multiplication by X are just bit shifts.

5.1.5 Algorithm

```
ImprovedMult(N, M):
    Let X be 2^(log(N + M) / 2)
    Write N = AX + B, M = CX + D
    P1 = Product(A, C)
    P2 = Product(B, D)
    P3 = Product(A + B, C + D)
    Return P1 X^2 + [P3 - P1 - P2]X + P3
```

The first two lines take $\mathcal{O}(n)$ time, the three `Product` calls take $\mathcal{O}(n^2)$, and the last step takes $\mathcal{O}(n)$, for a total runtime of $\mathcal{O}(n^2)$. Despite there not being an asymptotic difference, note that this algorithm runs in $\mathcal{O}\left(\frac{3}{4}n^2 + n\right)$ time, whereas the naive algorithm runs in $\mathcal{O}(n^2)$ time, so there is a slight improvement.

5.1.6 Further Improvements

Our algorithm still uses the naive algorithm (`Product`). However, we don't actually need to use this; we can just use our own implementation! We introduce *Karatsuba's Algorithm.*

```
KaratsubaMult(N, M)
    If N + M < 99
        Return Product(N, M)
    Let X be 2^(log(N + M) / 2)
    Write N = AX + B, M = CX + D
    P1 = KaratsubaMult(A, C)
    P2 = KaratsubaMult(B, D)
    P3 = KaratsubaMult(A + B, C + D)
    Return P1 X^2 + [P3 - P1 - P2]X + P3
```

Here, the pre-processing and post-processing still takes $\mathcal{O}(n)$ time. The three recursive calls are a bit tricky.

5.1.7 Runtime Recurrence

Karatsuba's multiplication on inputs of size n spends $\mathcal{O}(n)$ time, and then makes three recursive calls to problems of approximately half the size. If $T(n)$ is the runtime for n-bit inputs, we have the recurrence:

$$T(n) = \begin{cases} \mathcal{O}(1) & n = \mathcal{O}(1) \\ 3T(n/2 + \mathcal{O}(1)) + \mathcal{O}(n) & \text{Otherwise} \end{cases}$$

This isn't exactly a clean recurrence that we can solve. That being said, this runs in roughly $\mathcal{O}(n^{\log_2(3)})$ time. We will explain this later.

5.2 Generalization & Master Theorem

We will often get runtime recurrences with divide and conquer looking something like:

$$T(n) = \begin{cases} \mathcal{O}(1) & n = \mathcal{O}(1) \\ aT\left(\frac{n}{b} + \mathcal{O}(1)\right) + \mathcal{O}(n^d) & \text{Otherwise} \end{cases}$$

Here, the second line is saying a subproblems of size $\frac{n}{b}$. Note that the recursive subcalls are a constant *fraction* of the size of the original. For example, if $T(n) = 2T(n-1)$, then $T(n) = \mathcal{O}(2^n)$.

5.2.1 Tracking Recursive Calls

We have:

- 1 recursive calls of size n
- a recursive calls of size $n/b + O(1)$
- a^2 recursive calls of size $n/b^2 + O(1)$
- ...
- a^k recursive calls of size $n/b^k + O(1)$

So, the total runtime is:

$$\begin{aligned} \text{Total Runtime} &= \sum_{k=0}^{\log_b(n)} a^k \mathcal{O}\left(\left(\frac{n}{b^k}\right)^d\right) \\ &= \mathcal{O}(n^d) \sum_{k=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^k \end{aligned}$$

There are several cases to consider.

1. $a > b^d$: Increasing geometric series dominated by last term. The runtime is dominated by recursive calls at the bottom level. The runtime would be $O(n^{\log)b(a)})$.
2. $a < b^d$: Decreasing geometric series is dominated by the first term. Rutime is mostly based on the cleanup steps at the top level. The runtime is $O(n^d)$.
3. $a = b^d$: Every level of the recursion does the same amount of work. The runtime is $O(n^d \log(n))$.

5.2.2 Master Theorem

Theorem 5.1: Master Theorem

Let $T(n)$ be given by the recurrence:

$$T(n) = \begin{cases} \mathcal{O}(1) & n = \mathcal{O}(1) \\ aT\left(\frac{n}{b} + \mathcal{O}(1)\right) + \mathcal{O}(n^d) & \text{Otherwise} \end{cases}$$

Then we have:

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

5.2.3 Example: Runtime

Suppose that a divide and conquer algorithm needs to solve 4 recursive subproblems of half the size and do $\mathcal{O}(n^2)$ addition work. What is the runtime?

Here, we have 4 subproblems ($a = 4$) of half the size ($b = 2$) and we need to do $\mathcal{O}(n^2)$ additional work ($d = 2$). So, our recurrence is given by

$$T(n) = \begin{cases} \mathcal{O}(1) & n = \mathcal{O}(1) \\ 4T\left(\frac{n}{2} + \mathcal{O}(1)\right) + \mathcal{O}(n^2) & \text{Otherwise} \end{cases}$$

Since $b^d = 2^2 = 4$ and $a = b^d$, it follows that the runtime is

$$\mathcal{O}(n^2 \log(n))$$

5.3 Problem: Matrix Multiplication

Suppose we wanted to multiply $n \times n$ matrices.

5.3.1 Recall

If $AB = C$, then

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

The naive algorithm computes this sum of n terms for each of n^2 entries, so the runtime is given by $\mathcal{O}(n^3)$.

5.3.2 Block Matrix Multiplication

If we divide the matrix into blocks, we can get the product of the full matrix in terms of the products of the blocks.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Here, $A, B, C, \ldots$ are $(n/2) \times (n/2)$ matrices.

5.3.3 Divide and Conquer Algorithm

We can compute 8 products of $(n/2) \times (n/2)$ matrices, and do some addition to get an answer. The runtime is

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

The Master Theorem states that this would give us $\mathcal{O}(n^3)$ runtime. And, this isn't hard to see why. If we unrolled what this algorithm was doing, we're essentially just doing the naive multiplication.

5.4 Problem: Sorting

Given a list L of numbers, return L in sorted order.

5.4.1 Algorithm Idea

An idea is to:

- Divide L into two parts, L_1 and L_2.
- Sort L_1 and L_2.
- Finally, *merge* both L_1 and L_2 together.

5.4.2 Merge Operation

One question we have is, how do we define the *merge* operation? Well, suppose we have

$$L_1 = [1, 3, 6, 10]$$

$$L_2 = [2, 4, 5, 7, 8, 9]$$

We can combine the two lists by comparing the two smallest elements an then putting the smaller of the two elements in the new list. So, for example, the first element in L_1 is 1 and the first element in L_2 is 2, so we can put 1 into the final list and somehow point L_1 to the second element so we don't consider 1 again. Then, we can consider 3 in L_1 and 2 in L_2; in this case, we put 2 into the final list and point L_2 to the second element so we don't consider 2 again. We can keep doing this until we go through every element in both parts of the list.

The algorithm can be described like so:

```
Merge(A, B):
    Let C be the list with the length being len(A) + len(B)
    // Kane uses one-indexing instead of zero-indexing.
    a = 1
    b = 1
    for c = 1 to len(C):
        if b > len(B)
            C[c] = A[a]
            a++
        else if a > len(A)
            C[c] = B[b]
            b++
        else if A[a] < B[b]
            C[c] = A[a]
            a++
        else
            C[c] = B[b]
            b++
    Return C
```

This runs in $\mathcal{O}(|A| + |B|)$ time, particularly due to the `for`-loop.

5.4.3 Merge Sort Algorithm

```
MergeSort(L):
    // Every divide & conquer algorithm needs a base case
    if len(L) == 1:
        return L

    Split L into approx. equal lists L1, L2
    return Merge(MergeSort(L1), MergeSort(L2))
```

To analyze the runtime, we note the following:

- Base Case: The base case runs in $\mathcal{O}(1)$ time; this is obvious (and is expected for any base case).
- Divide & Conquer: The divide and conquer part comes in several steps.
  - Divide: We divide our list L into approximately equal-sized lists L_1 and L_2. We assume that this takes $\mathcal{O}(1)$ time.

- Conquer: It takes $2T(n/2 + \mathcal{O}(1))$ time to recursively solve two subproblems, each of size $\frac{n}{2}$. The $\mathcal{O}(1)$ is due to the possibility that one of the two lists may have unequal sizes (e.g. one sublist has size 5 and another sublist has size 6). At the end of the day, we can just simplify this down to $2T(n/2)$.
- Combine: Merging an n-element sublist takes $\mathcal{O}(n)$ time. This term absorbs the $\mathcal{O}(1)$ time from the divide part of the algorithm.

We then have (by the Master Theorem):

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

Note that $a = 2$, $b = 2$, and $d = 1$, so the final runtime is given by the Master Theorem:

$$\mathcal{O}(n \log n)$$

5.5 Problem: Order Statistics

Given a list of numbers L, find the **median** or the **largest element** or the **10th smallest** element of L. Essentially, you want to find something based on the order in some way.

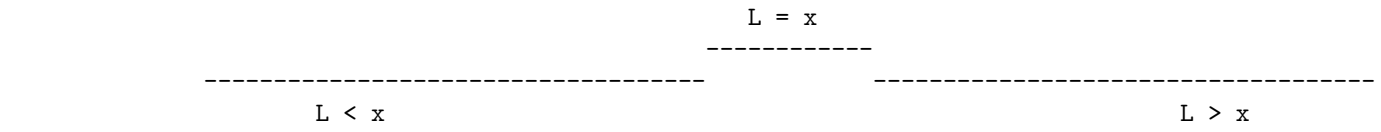
Let's focus on one particular problem: Given L and k, find the kth smallest element of L.

5.5.1 Easy Algorithm

The easy algorithm is to sort L and then return $L[k]$. The runtime is $\mathcal{O}(n \log n)$, but there are better ways to do this.

5.5.2 The Idea for the Divide Step

First, we pick a *pivot* $x \in L$ at random here, and we sort all elements in L relative to x. That is, every element to the *left* of x is less than x, every element to the *right* of x is greater than x, and every element in the center is equal to x. In other words:



In this sense, we can sort these elements into categories in $\mathcal{O}(n)$ time. Then:

- We note that the kth smallest element is less than x if and only if $|L_{<x}| \geq k$. If this is the case, then the answer is the kth smallest element of $L_{<x}$.
- The answer is x if and only if $|L_{<x}| < k$ and $|L_{<x}| + |L_{=x}| \geq k$.
- Otherwise, the answer is the $(k - |L_{<x}| - |L_{=x}|)$th smallest in $L_{>x}$.

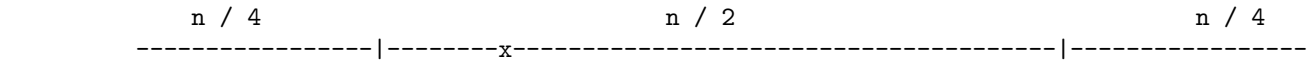
5.5.3 Divide and Conquer Algorithm

```
OrderStats(L, k)
    Pick x in L randomly
    Sort into L[<x], L[=x], L[>x]
    If len(L[<x]) >= k
        return OrderStats(L[<x], k)
    else if len(L[<x]) + len(L[=x]) >= k
        return x
    else
        return OrderStats(L[>x], k - len(L[<x]) - len(L[=x]))
```

The idea is that we use randomization to improve our runtime. If we didn't pick a random $x \in L$, then it's very possible that the x that we pick is the smallest or biggest x, which means we would essentially have to consider $n-1$ elements, and thus we would have a divide algorithm that takes $T(n-1)$ time (and so $\mathcal{O}(n^2)$ time). By randomly selecting an $x \in L$

- There is a 50% chance that x is selected in the middle half.
- If so, no matter where the answer is, the recursive call is at most $3n/4$.
- On average, we might need two tries to reduce calls.

The idea behind the $3n/4$ is that, when we pick this x, we either recurse on everything bigger than x or smaller than x. Because x is in this middle half, there are at least $n/4$ elements on either sides:



So, we have the runtime

$$T(n) = \mathcal{O}(n) + T(3n/4)$$

By the Master Theorem, we get the *expected*

$$\mathcal{O}(n)$$

5.6 Problem: Searching Problem

Given a sorted list L and a number x, find the location of $x \in L$.

5.6.1 Naive Algorithm

The naive algorithm is to iterate over each element in L and see where x is in L. This takes $\mathcal{O}(n)$ time as the element in question might be at the end of the list.

Remark: Usually, you cannot beat $\mathcal{O}(n)$ because any algorithm needs to read the entire input. *However*, since the list is guaranteed to be sorted, we can do better.

5.6.2 Algorithm Idea

First, we want to split L into two sublists. The idea is to take advantage of the fact that L is sorted to see which of the two sublists we should check. Essentially:

- If $L[i] > x$, then the location must be before i.
- If $L[i] < x$, then the location must be after i.
- If $L[i] = x$, then we found it.

Of course, we don't want to actually *split* the list; that would take too much time. Instead, we can have two indices i and j which determine which part of the list to check.

5.6.3 Divide and Conquer Algorithm

```
// Search between L[i] and L[j]
BinarySearch(L, i, j, x)
    // This is the base case.
    if j < i
        return 'error'
    k = (i + j) / 2
    if L[k] == x
        return k
```

```
if L[k] > x
    return BinarySearch(L, i, k - 1, x)
if L[k] < x
    return BinarySearch(L, k + 1, j, x)
```

To analyze the runtime, we note the following:

- Base Case: The base case runs in $\mathcal{O}(1)$ time; this is obvious (and is expected for any base case).
- Divide & Conquer: The divide and conquer part comes in several steps.
  - Divide: The divide part comes from the calculation of k, which determines what range we want to consider when recursively calling `BinarySearch`.
  - Conquer: We're only interested in running through a *subset* of L. Recall that, with binary search, while we're still working with the big list L, the range that we check will always be half of the previous range that was considered. With this in mind, it takes $T(n/2+\mathcal{O}(1))$ time to recursively solve one subproblem of size $\frac{n}{2}$. The $\mathcal{O}(1)$ is due to the possibility that one of the two ranges may have unequal sizes (e.g. one range covers 5 elements and another range has covers 6). At the end of the day, we can just simplify this down to $T(n/2)$.
  - Combine: Combining the answer is as simple as returning k.

We then have (by the Master Theorem):

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

Note that $a = 1$, $b = 2$, and $d = 0$, so the final runtime is given by the Master Theorem:

$$\mathcal{O}(\log n)$$

5.6.4 Puzzles & Applications

Binary search, or something related to that, occurs when we try to solve a problem with a finite number of possible answers and, every time we make a guess as to what the answer is, we eliminate a *constant fraction* of the possibilities. **Put it another way**, lots of puzzles have binary search-like answers; as long as you can spend *constant time* to divide your search space by some constant fraction, you can use binary search in $\mathcal{O}(\log(n))$ time.

Suppose you have 27 coins, one of which is *heavier* than the others, and a balance. You are asked to determine the heavy coins in 3 weighings. The idea is:

- After one weighing, you're down to 9 possibilities.
- After another weighing, you're down to 3 possibilities.
- After a third weighing, you're down to just 1 possibility.

5.7 Problem: Closest Pair of Points

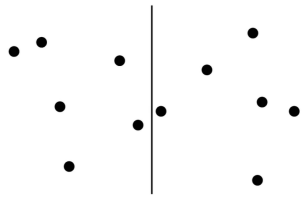
Given n points in the plane, $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair (x_i, y_i) and (x_j, y_j) whose Euclidean distance is as small as possible.

5.7.1 Naive Algorithm

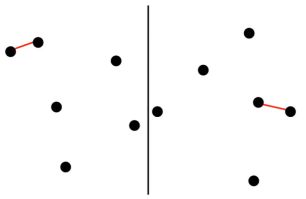
The naive algorithm is to simply try every pair of points. This runs in $\mathcal{O}(n^2)$ time; you have n points so n^2 possibilities.

5.7.2 Divide and Conquer Outline

In order to use divide and conquer, you need to find some way to divide the set of points into two equally sized subsets. One reasonable way to do this is to draw a line and cut the plane so that you can consider the points on the left and right of the line.



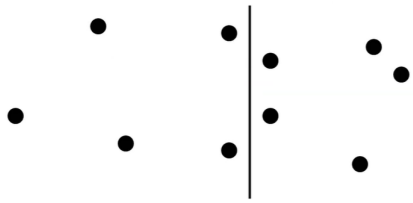
From there, you can compute the closest pair of points on each side.



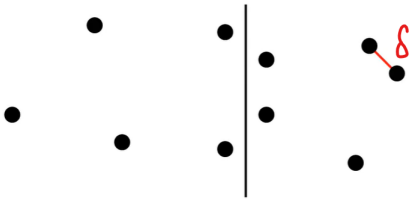
After we do this, we need to somehow recombine the answers to our subproblems so we can get the answer. **However**, we can't simply just return the better of the two pairs (denoted by the red line) because it's possible that the best pair of points actually *crosses* the line that we used to cut our problem in half (which we didn't consider since these two points are in separate subsets). So, what do we do?

5.7.3 Observation

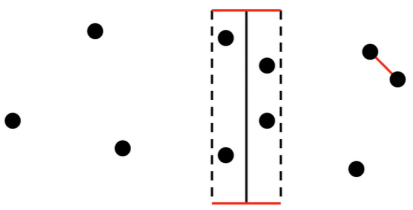
Consider the following set of points, with the dividing line:



Suppose that the closest pair of points on *either side* of the dividing line is at distance δ; that is, two points on the same side has distance δ.



At the very least, we only need to care about points *within* δ of the dividing line. So, we only need to consider a *strip* of width 2δ:



In our case here, we now only focus our attention on any points within this strip. We now need to know if there is some pair of points in this strip where the distance between the points in that pair is *less than* δ. In other words, is there a pair of points in this strip whose distance is *better* than the distance that we already found (i.e. δ)?

5.7.4 Main Idea

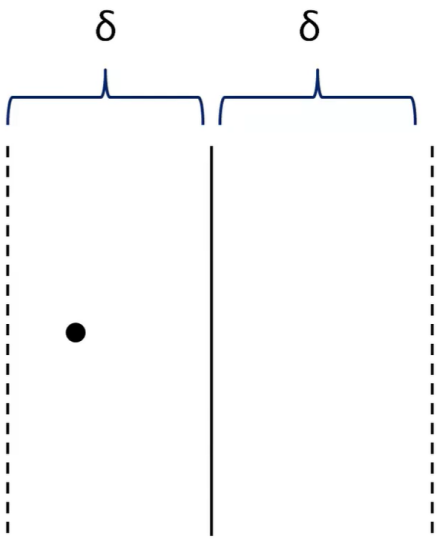
The main idea is that we know that any pair of points on the same side of the line has to be separated by at least δ. So, the points inside this strip must be reasonably spaced out vertically. So, if we want to find some pair of points that are closer to δ, there can't be that many other points in between them. If we sort the points vertically (by their y-coordinates), then two points that are super close to each other cannot be far off in that ordering.

Proposition. *Take the points within δ of the dividing line and sort them by y-coordinate. Any one of these points can only be within δ of the 8 closest points on either side of it.*

So, sort everything by y-coordinate. Then, look at the next 8 points[4] in y-coordinate above it and the next 8 points in y-coordinate below it. Those are the only possible points that can possibly be within δ of the point that we're considering.

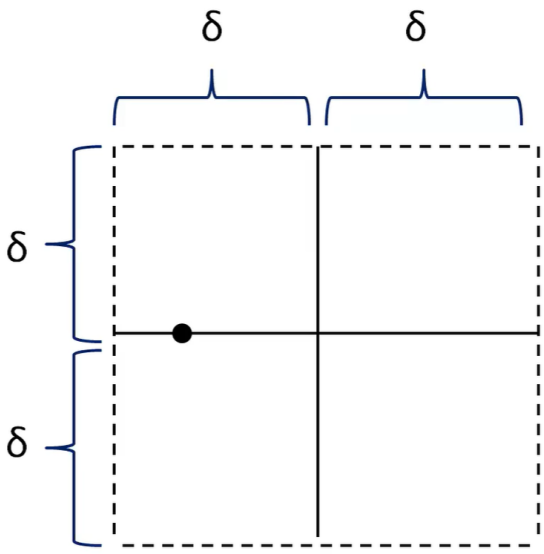
If the proposition is true, then this means we only need to check a few pair of points crossing the line. The *idea* is that the points on each side of the line are separated by at least δ. This is because we know that δ is the distance to the closest pair of points for any pair of points on either side of the line. This means that there isn't enough room to cram many points into a narrow region because this forces points to be spread out.

Proof. Suppose we have the dividing line, some δ, and a point.

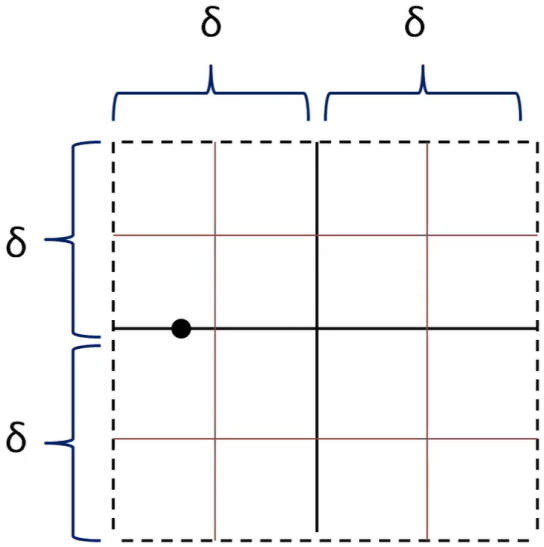


Any nearby point has to be within this strip and the y-coordinate of that point we're looking for must be within δ of the y-coordinate that we have. So, we can create a bigger box and every other point that we want to match this one point with must be somewhere in this box.

[4] What really matters is that this is **some constant**; it can be 5 or 10 or 127 or whatever.



So, we can subdivide this region into $\delta/2$-sided squares.



There are now 16 squares; 8 of them are above the point and 8 of them are below the point. The claim is that each of these little squares can have at most one point. This is because if we take any two points inside the same square, the distance between those points will be less than δ. In particular, since every square is either on the left or right side of the dividing line, we can't have two points in the same square with distance less than δ because then there would be points on the same side of the dividing line with distance less than δ.

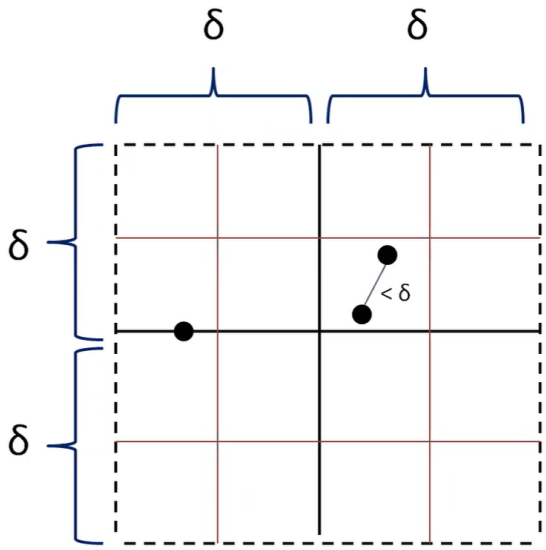


Figure: Two points in the same square with distance less than δ, a contradiction to our original claim that δ was the smallest distance between two points on the same side of the line.

So, the point is that there is at most one point in each square[a]. Thus, there are at most 8 points below and above our given point; any other points close to our given point must be farther than δ or outside of the strip. This completes the proof. □

[a]Note that there is a possibility that there is a point on the dividing line. In this case, just move the dividing line until there are no points on it. Alternatively, we can treat points on the line as being on the left/right of the dividing line.

5.7.5 The Algorithm

```
CPP(S)
    // Base case, brute force it.
    If len(S) <= 3
        return closest distance

    Find line L evenly dividing points.                    // (a)
    Sort S into S_{left}, S_{right}                        // (b)
    // D is delta
    let D = min(CPP(S_{left}), CPP(S_{right}))             // (c)
    let T be the points within D of L                      // (d)
    Sort T by y-coordinate                                 // (d)
    Compare each element of T to 8 closest on either side. // (e)
        Let min dist be D'
    Return min(D, D')
```

The runtime analysis is as follows:

(a) First, finding the line that evenly divides the points take $\mathcal{O}(n)$ time.

(b) Sorting the points into the two sets takes $\mathcal{O}(n)$ time.

(c) The two recursive calls take $2T(n/2)$ time.

(d) Finding and sorting T takes $\mathcal{O}(n \log(n))$ time.

(e) Comparing each element of T to 8 closest on either side takes $\mathcal{O}(n)$ time.

This gives us the recurrence:

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

This is not a runtime recurrence that we can use the Master Theorem for; this is particularly because the Master Theorem uses $\mathcal{O}(n^d)$ whereas we have $\mathcal{O}(n \log(n))$. However, by running a Master Theorem-like argument, we have that

$$T(n) = \mathcal{O}(n \log^2(n))$$

Alternatively, if we are more careful and have `CPP` take points already sorted by y-coordinate, we can reduce this runtime down to $\mathcal{O}(n \log(n))$.

6 Greedy Algorithms

A greedy algorithm is an algorithm where you want to build up some solution one step at a time. At every step, you come up with some notion of what the *best* choice you can make is, and make that choice; in other words, at every step, you make the *best available choice* until you can no longer make any choices.

A few things to keep in mind regarding greedy algorithms:

- They are very simple (almost trivial) and clean to write. *When they work*, they tend to be very good algorithms.
- It's not uncommon to write a greedy algorithm that seems to work, but actually fails. Thus, proving correctness is very important.

6.1 Problem: Making Change

Given some amount of money (in US dollars), you want to make the exact change for it using as few bills and coins as possible. For example, if we wanted to make change for a single dollar, you would use a single dollar bill instead of 100 pennies.

6.1.1 Greedy Algorithm

The algorithm is to repeatedly add the biggest denomiation still available.

6.1.2 Example: Making Change

What is the most optimal way to make change for \$12.73? Here, we define optimal as using the least number of bills and coins to make exact chamge.

Suppose we wanted to make change for \$12.73.

- We can begin with a change for \$10 bill, so we now need to make change for \$2.73.
- We can next use two \$1 bills, so we now need to make change for \$0.73.
- Next, we can use two quarters, so we now need to make change for \$0.23.
- Next, we can use two dimes, so we now need to make change for \$0.03.
- Finally, we can use three pennies.

At the end, we only needed 10 different bills and coins to make change.

6.1.3 Result

Proposition. *For standard US currency denomiations, this algorithm is always optimal.*

Proof. This proof is omitted. □

6.1.4 Non-Application to Other Currencies

This property does not necessarily hold for other currencies.

For example, suppose we have a country with currency W, where the denominations are $7W$, $5W$, and $1W$. What if we try to make change for $10W$?

- We first use $7W$, leaving us with $3W$ left to make change.
- Then, we use three $1W$ to cover the rest of the change needed.

This leaves us with 4 different bills and coins. However, the optimal solution is actually to use two $5W$ instead.

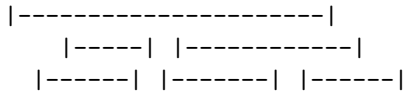
6.2 Problem: Interval Scheduling

You are trying to figure out your classes for the next quarter, and you are at a school where classes have weird times (e.g. 3:15 PM to 3:27 PM). You're not allowed to pick two classes that overlap. However, beyond this, you are a masochist and so you want to schedule as many classes as you possibly can; you don't care what classes you want to take, just that you can take them without having any overlaps.

More formally, given a set S of intervals $[x_i, y_i] = I_i$, you want to find a subset $T \subseteq S$ such that

1. No two intervals in T overlap.
2. Subject to condition 1, $|T|$ should be as large as possible.

For example, if we have the following intervals:



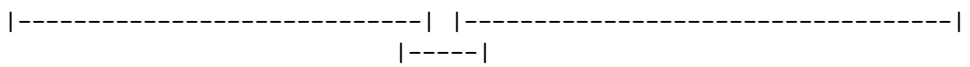
It's not hard to see that the answer would be the three bottom intervals.

6.2.1 Greedy Algorithm Idea

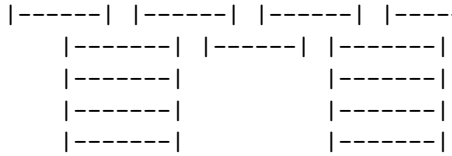
We want to build this schedule one interval at a time. Now, we want to figure out which intervals we should be adding. Some ideas include:

- Shortest intervals, because longer intervals are more expensive to add.
- Fewest overlaps.
- Ends soonest.
- Starts soonest.

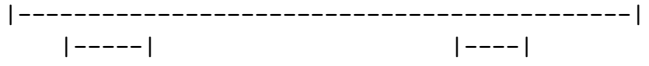
Picking the shortest interval doesn't work; consider this counterexample, where the two longest intervals represent the best possible solution:



Picking the intervals with fewest overlaps doesn't work; consider this counterexample, where picking the interval with the fewest overlaps locks ourselves from the best possible schedule:



Picking the intervals that start the soonest also doesn't work; consider this counterexample, where picking the interval that starts the soonest locks ourselves from the best possible answer:



However, picking the interval that ends the soonest works! We define the interval that ends the soonest to be the interval where the endpoint is minimal. In the example below, (a) ends the soonest because its endpoint is minimal (compared to (b)'s endpoint):

```
   (a)
|------|
    |------|
       (b)
```

Although not a proof, picking the intervals that ends the soonest in the above counterexamples gives us the best possible schedule.

6.2.2 Intuition

What does it mean for T to have no overlapping intervals? Well, if T has no overlaps, then we can sort T in some order; that is,

$$T = \{J_1, J_2, \ldots, J_k\}$$

such that

$$J_1 < J_2 < \cdots < J_k$$

When we say that J_1 comes before J_2, we essentially are saying that if $J_1 = [x_1, y_1]$ and $J_2 = [x_2, y_2]$, then $y_1 < x_2$. But, this is the only requirement on the interval J_1.

6.2.3 Formal Proof

We now want to formally prove that this is the case.

Proof. Suppose the greedy algorithm produces the set T such that

$$J_1 < J_2 < \cdots < J_m \qquad J_i = [x_i, y_i]$$

and each of $J_i \in T$. We will now prove, by induction on k, that <u>any</u> set of intervals

$$I_1 < I_2 < \ldots \qquad I_i = [x'_i, y'_i]$$

has I_k ending no sooner than J_k.

- <u>Base Case:</u> For $k = 1$, we know that J_1 ends the earliest and so

  $$y_1 \leq y'_1$$

- <u>Inductive Step:</u> Suppose that this is true for k; that is, $y_k \leq y'_k$. So, J_{k+1} has the smallest possible y for any interval with $x > y_k$. We know that $I_{k+1} > I_k$, which means that $x'_{k+1} > y'_k$. By the inductive hypothesis, we know that

  $$x'_{k+1} > y'_k \geq y_k$$

  This all implies that

  $$y_{k+1} \leq y'_{k+1}$$

And this completes the proof. □

6.2.4 Greedy Algorithm

```
BestInterval(S):
    let T = {}
    while S not empty:
        let J = S with minimum end time
        add J to T
        remove all intervals overlapping J from S
    return T
```

The `while`-loop takes $\mathcal{O}(n)$ time, and finding the minimum element and removing all intervals overlapping J takes $\mathcal{O}(n)$ time, so the runtime is approximately $\mathcal{O}(n^2)$ time.

To optimize the algorithm, we can sort S by the end time.

```
BestInterval(S):
    let T = {}
    Sort S by end time
    yMax = -inf
    For I in S in order:
        If start(I) > yMax:
            add I to T
            yMax = end(I)
    return T
```

In our optimized algorithm, it takes $\mathcal{O}(n \log(n))$ time to sort and then $\mathcal{O}(n)$ time to iterate over the intervals. Therefore, the optimized runtime is approximately $\mathcal{O}(n \log(n))$ time.

6.3 Exchange Argument: Proving Correctness of Greedy Algorithms

The exchange argument is a generic way[5] to prove correctness of a greedy algorithm. The idea for the exchange argument is as follows:

- Start from an arbitrary solution A.
- Slowly turn A into the greedy solution G, making it only better each step.
- Conclude that G is just as good as A, if not better. From there, you can conclude that since A is arbitrary, then G must be optimal.

6.3.1 Layout

Construct a sequence of solutions

$$A = A_0 \leq A_1 \leq A_2 \leq \cdots \leq A_n \leq G$$

where A_t agrees with the first t greedy choices. Then, given any A_t that agrees with the first t greedy choices, there exists an A_{t+1} that agrees with the first $t+1$ greedy choices and A_{t+1} is no worse than A_t, i.e. $A_{t+1} \geq A_t$.

Define an arbitrary A, we can say that $A = A_0$ (you made no choices so there's nothing to be consistent). From the layout above, we can construct an A_1. By induction, we can continue until we have an A_n, where n is the number of choices that you had to make for the greedy algorithm. Finally, we need to show that $A_n \leq G$.

So, really, the goal is to find a sequence of solutions

$$A = A_0, A_1, \ldots, A_n = G$$

such that

- $A_i \leq A_{i+1}$ (i.e. A_{i+1} is just as good as A_i, if not better).
- A_i agrees with $D_1, D_2, \ldots, D_i$, where D is the decision that the greedy algorithm made.

6.3.2 Example: Interval Scheduling Problem

We will show that the greedy solution from the interval scheduling problem is, indeed, optimal.

[5] Of course, it may not always work and it may not be the best.

Proof. Let the greedy solution be defined by

$$G = J_1, J_2, \ldots, J_n$$

such that

$$A_t = \underbrace{J_1 < J_2, < \cdots < J_t}_{\substack{A_t \text{ is some solution that agrees with the first } t \text{ decisions made by the greedy algorithm.}}} < \overbrace{I_{t+1} < I_{t+2} < \cdots < I_m}^{\substack{\text{After that, these can be any arbitrary intervals as long they don't overlap.}}}$$

We want to show that, given that the above is some valid set of intervals, we can find an A_{t+1} that is no worse than the above (i.e. no fewer intervals that A_t) but also includes the first $t+1$ intervals of the greedy algorithm. There are two cases to consider:

1. If $m = t$ (if A_t only has those t intervals), then there's a clear way to improve it - by adding the next interval. So,
$$A_t < A_{t+1} = J_1 < J_2 < \cdots < J_{t+1}$$

2. If $m > t$, then we can change the current solution to the solution which includes J_{t+1} by replacing I_{t+1} by J_{t+1}. That is,
$$A_{t+1} = J_1 < J_2 < \cdots < J_{t+1} < I_{t+2} < \cdots < I_m$$
It's clear that $A_{t+1} \geq A_t$. It's also clear that it agrees with the first $t+1$ decisions of the greedy algorithm. We just need to check whether A_{t+1} is valid (i.e. if we add A_{t+1}, then will the intervals overlap?). One thing we need to verify is whether the intervals are disjoint. Now, because G is valid, it's clear that
$$J_1 < J_2 < \cdots < J_{t+1}$$
and because A_t is valid, then
$$I_{t+2} < I_{t+3} < \cdots < I_m$$
So, we need to show that $J_{t+1} < I_{t+2}$. Recall that J_{t+1} has the smallest max among all intervals greater than J_t. Since $I_{t+1} > J_t$, this means that
$$\max(J_{t+1}) \leq \max(I_{t+1})$$
But because the I's don't overlap, that says that
$$\min(I_{t+2}) > \max(I_{t+1}) \geq \max(J_{t+1})$$
But, this shows that $I_{t+2} > J_{t+1}$, so we are done.

This concludes the proof. □

6.3.3 Summary

So, the end idea is that we started with an arbitrary sequence

$$A_0 = I_1 < I_2 < \cdots < I_m$$

such that none of the decisions agree with the greedy solution, we can transform them so that the first interval agree

$$A_1 = J_1 < I_2 < \cdots < I_m$$

and then the second interval

$$A_2 = J_1 < J_2 < \cdots < J_m$$

If we keep going with this, eventually we'll end up with

$$A_m = J_1 < J_2 < \cdots < J_m$$

Now, expanding upon this, we have

$$A_{m+1} = J_1 < J_2 < \cdots < J_m < J_{m+1}$$

and then eventually we'll reach the desired conclusion

$$A_n = J_1 < J_2 < \cdots < \cdots < J_n$$

In particular, $n \geq m$ and so the greedy solution is just as good as any other solution.

6.4 Problem: Optimal Caching

To give some background, we discuss the memory hierarchy in a very simplified form.

- Your computer has a CPU and a disk, which stores your memory.
- On old computers, your disk was often a spinning platter of metal. To read a random entry, you need to spin this metal until the head points to the appropriate location.
- Your disk operates on the order of milliseconds; however, your CPU operates on the order of nanoseconds.
- To solve this, you also have a cache, which is often stored on the same chip as the CPU.
- If we need to access some memory many times, instead of storing it on the disk, we can store it on the cache. If the memory is in the cache, it is very easy to look up since it is right there on the CPU on the chip.
- The cache is relatively small; we assume that the cache can store k words.

An unrealistic assumption that we'll make is that we know what memory accesses the program is going to make ahead of time. Consider the following example below, where $k = 2$ and we make another assumption that whenever we want to operate on some point in memory, we always load it in cache instead of in the disk.

```
time           ---------------------------------------------
memory access  a  b  a  b  c  a  d  e  c  b  c  a  c

cache 1        a  -  -  -  -  -  -  -  c  -  -  -  -
cache 2           b  -  -  c  -  d  e  -  b  -  a  -
```

This particular memory/cache schedule satisfies the sequence of memory accesses that are made. At every time when some location in memory needs to be accessed, it is already in one of the two locations in the cache. As for how expensive this is, suppose that the cost of the schedule is the number of times we need to load the new memory into cache. So, in our example above, this would be **8** cache misses (8 times when we needed to read something that wasn't in the cache and thus needed to be loaded from disk).

This leads us to the optimal caching problem. Given k and a sequence of memory accesses, find a consistent cache schedule[6] such that the number of loads from disk is minimized.

So, with the example above, the memory/cache schedule needed to load from disk 8 times. Can we do it in fewer times?

[6] Schedule of what memory location is in cache at what time in such a way that when we need to access that memory location, it is already in cache (maybe it was just loaded there, but it's there)

6.4.1 Observation

We should only load new memory on a cache miss. In other words, if we need to load some memory that isn't in cache, then we will. Now, the question becomes: when we load something new in memory, what do we throw out? i.e. if cache 1 has a and cache 2 has b, and we needed to load c, what would we throw out of cache?

- Least recently used: throw out the memory that was least recently used.
- Furthest in the future: since we have the sequence of memory accesses (including the future ones), we can see, in memory, how long it will take before that particular location is needed again, and the one that is furthest in the future is the one that we throw out.
- Least frequently used: throw out the memory that isn't being used frequently.
- Least frequently used in the future.

Here, *furthest in the future* works.

6.4.2 Proof that Furthest in the Future Works

Proof. We make use of the exchange argument. Here, we say that A_t agrees with G, the greedy solution, for the first t timesteps. That is, up to time t, both A_t and G do exactly the same thing; after time t, they do something else.

```
timestep ---------------------------- t -------------
A        xxxxxxxxxxxxxxxxxxxxxxxxxxxxx aaaaaaaaaaaaa
G        xxxxxxxxxxxxxxxxxxxxxxxxxxxxx ggggggggggggg
```

We want to show how to get from A_t to A_{t+1}. In other words, if we can find something that agrees with G for t timesteps, we can turn it into something that agrees G for $t+1$ timesteps and is no worse (no more extra memory loads than we had before). To do this, we need to show a bunch of different cases.

- Case 1: No cache misses at time $t+1$. In this case, G does nothing.
  - Case 1a: A_t also does nothing. In this case, $A_{t+1} = A_t$.
  - Case 1b: A_t does replace something. In this case, to get A_{t+1}, we put off those replacements by one timestep.

    ```
    timestep ------------------ t --- t+1 --- t+2
    A                           x     a
                                y     b
                                z     c
    G                           x               a
                                y               b
                                z               c
    ```

    So, if A_t replaces x, y, and z by a, b, and c, then because there wasn't a cache misses, then the memory access being asked for at timestep $t+1$ is none of a, b, or c, and so in particular we can put off replacing these by a, b, and c until timestep $t+2$.
- Case 2: Cache miss at time $t+1$. Here, we make a reduction that A_t only replaces the missed element. In other words, say we had to load up w but the cache already has a, b, and c; then, we can replace a with w and keep b and c.
  - Case 2a: Load into same location as G. Then, $A_{t+1} = A_t$ since we already agree with the greedy solution.

– Case 2b: Load into a different location than G. In other words, suppose at time $t+1$ we needed to load w, but the cache for both A_t and G had x, y, and z. Then, maybe the cache will replace y with w while the greedy will replace x with w; that is:

```
                                    (w)
timestep ------------------ t --- t+1
A                           x     a
                            y     w
                            z     c
G                           x     w
                            y     b
                            z     c
```

In this case, we need to consider several subcases.

* Case 2bi: A_t throws out x before using it. For example,

```
                                          (w)      (p)
timestep ------------------ t --- t+1 --- t+2
A                           x     -        p
                            y     w        -
                            z     -        -
A_{t+1}                     x     w
                            y     -        p
                            z     -
```

In this case, one way to think about it is that A_t throws out y and then x. Instead, we can throw out x and then y.

* Case 2bii: Need x first. Then, x is furthest in the future, so there must be some call y also early.

This concludes the proof. □

6.5 Problem: Huffman Codes

Suppose, for simplicity, that our alphabet was a, b, c, d. If we have a string of letters that we wanted to encode, say, abcdacbdadccb, we would need to store it in the computer as a bunch of `0`'s and `1`'s. So, we can let $a =$ `00`, $b =$ `01`, $c =$ `10`, and $d =$ `11`. Then, we can encode the string above like so

a	b	c	d	a	c	b	d	a	d	c	c	b
00	01	10	11	00	10	01	11	00	11	10	10	01

With this encoding, it's easy to decode as long as we have the mapping. This has an issue, though: suppose we have the string aaaaaaaaaaabaacaa. Then, it's obvious that we're encoding `a` many more times than the other three letters. So, is there a way to map `a` to one bit instead of two bits so we can code the `a`'s with fewer bits?

Suppose $a =$ `0`, $b =$ `1`, $c =$ `00`, and $d =$ `01`. Then, we have soem ambiguity. If we had the string `00`, then this would map to either `aa` or `c`. So, how do we make this work?

Now suppose $a =$ `0`, $c =$ `10`, $b =$ `110`, and $d =$ `111`. Then, if we have the string `000010001011000011110100`, how do we decode this? Well, if we start reading from the left, then we note that:

- We first read the `0`. This can only map to `a` as the other characters start with `1`.
- Doing this three more times, we end up with `aaa`. Now, we consider the character `1`. There are three mappings where `1` starts first: `b`, `c`, and `d`. So, we read in the next character, `0`. Here, we note that this can only be `c` as `c` maps to a string `10` but the other two mappings start with `11`.

- We can repeat this process until we're done reading the string.

At the end, the decoded string is:

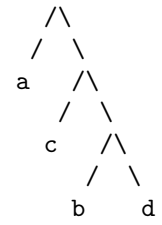
0	0	0	0	10	0	0	10	110	0	0	0	111	0	10	0
a	a	a	a	c	a	a	c	b	a	a	a	d	a	c	a

This is an example of **prefix-free encoding**: that is, no letters x, y is the encoding of x a prefix of the encoding of y.

Our problem statement is as follows: Given an alphabet where each letter x has a frequency[7] $f(x)$, we want a prefix-free encoding of the alphabet such that the encoding length $\sum_x f(x) \cdot |\text{encoding}(x)|$ is minimized.

6.5.1 Rewording the Problem

Our first observation is that there is a binary tree representation of the prefix-free encoding, where taking the left branch is the `0` bit and the right branch is the `1` bit. Then, you can place letters in locations corresponding to their encoding. For example, the tree representation of the above prefix-free encoding is:



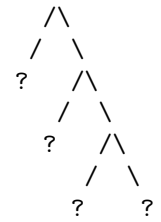
So, for a prefix-free encoding, the letters are at the leaves of the binary tree. So, we can rephrase the problem

$$\sum_x f(x) \cdot \text{depth}(x)$$

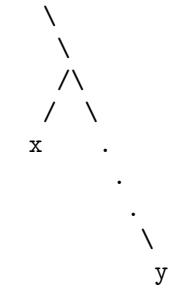
where the depth is the same as the length of the encoding of some x.

6.5.2 Easy Case

Suppose we fix the tree structure. That is, our tree must look like



We know that we have four letters where one letter has depth 1, one letter has depth 2, and two letters have depth 3. Now, the best way to assign letters is to assign the *high frequency* letters to the smaller depths. To see this, we note that



[7] The number of times it appears in the string.

To compare x and y, we want to compare

$$d_1 f(x) + d_2 f(y)$$

with

$$d_2 f(x) + d_1 f(y)$$

where d_i is the depth. Then, we can factor this out like so

$$(d_1 - d_2)(f(x) - f(y))$$

If x has lower frequency and it was stored at the smaller depth, then switching them would decrease the total expense. So, if we have a fixed tree, there is a greedy algorithm:

- Sort letters by frequency.
- Sort location by depth.
- Assign letters with higher frequencies to lower depth.

6.5.3 Observation

Two of the deepest elements are siblings. Note that the deepest elements correspond to the letters with little frequency. Thus, the key insight here is

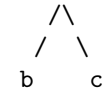
> The two least frequent letters in the alphabet might as well be siblings.

6.5.4 An Example

Suppose we had:

- 30 copies of `a`
- 15 copies of `b`
- 25 copies of `c`
- 50 copies of `d`
- 65 copies of `e`

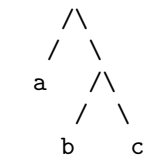
We can assume that `b` and `c` are siblings since they have the lowest frequencies. So:



Next, note that we can call the parent of `b` and `c` *`b` or `c`*, and thus treat it as a "different letter." We can also give this "different letter" a value of `40`, as `b` and `c` appear 40 times. Thus, we now have the alphabet:

- 30 copies of `a`
- 40 copies of `b` or `c`
- 50 copies of `d`
- 65 copies of `e`

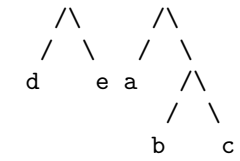
Notice that we can repeat the process again. In particular, notice that `b` or `c` *and* `a` are siblings since they have the lowest frequencies. So:



Again, we can call the parent of `a` and `b` or `c` *`a` or `b` or `c`*, and treat it as a "different letter" with the value being 70. Thus, we now have the alphabet:

- 70 copies of `a` or `b` or `c`
- 50 copies of `d`
- 65 copies of `e`

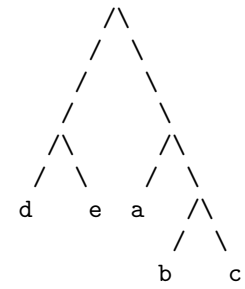
We now repeat this process. Here, we see that `d` and `e` are siblings since they have the lowest frequencies. So:



We call the parent of `d` and `e` *`d` or `e`* and give it a value of 115. Thus, we now have the alphabet:

- 70 copies of `a` or `b` or `c`
- 115 copies of `d` or `e`

But since there are only two "letters," we can just pair them up:



6.5.5 Algorithm

The algorithm is as follows, where

- S is the alphabet.
- f is the frequency function.

```
HuffmanCode(S):
    while |S| > 1:
        Let x, y be least frequent elements of S
        Create z with f(z) = f(x) + f(y)
        Make x, y children of z
        Remove x, y from S
        Add z

    Return element in S
```

If S has size n, then the runtime is $\mathcal{O}(n^2)$. This is because we need to iterate over the `while` loop, and for each iteration we need to do a linear scan. But, we can make use of a *priority queue* to optimize our algorithm.

```
OptimizedHuffmanCode(S):
    Insert S into priority queue Q
    while |Q| > 1:
        Let x = deleteMin(Q)
        Let y = deleteMin(Q)
        Create z with f(z) = f(x) + f(y)
        Make x, y children of z
        Q.Insert(z)

    Return deleteMin(Q)
```

This runs in $\mathcal{O}(n \log(n))$.

6.5.6 Takeaways

There are some takeaways from this.

- To find a greedy decision procedure, a good thing to do first is to find a safe first step. The safe first step in this algorithm is that the two elements might as well be siblings. Once you made that decision, you need to reduce the problem back to a copy of the original problem.
- Often, in order to turn something into a greedy algorithm, you might need to rearrange the way the problem is phrased; i.e. rephrase the problem. The original phrasing of this problem – finding the prefix-free encodings of letters of this alphabet which minimize the total encoding length – is awkward to phrase as a greedy algorithm. The idea that we can pair two elements as siblings would be incredibly awkward to phrase under the original problem statement.

6.6 Problem: Minimum Spanning Trees

Given a weighted graph G, find a minimum spanning tree of G.

6.6.1 Tree

Definition 6.1

A **tree** is a connected, *undirected* graph with no cycles.

Remark: This is different from a DAG.

6.6.2 Spanning Tree

Definition 6.2

A **spanning tree** for a graph G is a subset T of the *edges* of G such that

- The edges forms a tree.
- All vertices in G are covered in T; that is, the vertices in T are the same as the vertices in G.

6.6.3 Minimum Spanning Tree

Definition 6.3

Given a weighted graph G, a **minimum spanning tree** of G is a spanning tree where the sum of the weights of the edges are as small as possible.

6.6.4 Tree Properties

Lemma 6.1

For an undirected graph $G = (V, E)$, any two of the below imply the third.

1. $|E| = |V| - 1$.
2. G is connected.
3. G has no cycles.

Proof. Suppose we build G one edge at a time. Initially, we have $|V|$ many connected components with 0 edges. When we add an edge, this reduces the number of connected components by one unless the new edge is part of a cycle. To show this, suppose we have connected components A, B, C, and D.

- Case 1: Not Cycles. Suppose we have an edge connecting A to B. Since this edge is not part of a cycle, there are now three connected components.
- Case 2: Cycles. Suppose A has two vertices A' and A'', and we connect an edge from A' to A''. Since A' and A'' are in this connected component A, this implies that there was already a edge from A' to A''. Therefore, adding another edge creates a cycle. Clearly, this doesn't reduce the number of connected components.

If there are no cycles, then it will be the case that the number of connected components will be $|V|-|E|$. On the other hand, if there are cycles, the number of connected components will be bigger than $|V|-|E|$. From this alone, we can show that the lemma holds for two of the conditions.

- If we assume that (1) and (2) holds, then since G is connected, the number of connected components is 1. On the other hands, $|V| - |E|$ is also equal to 1, so there must be no cycles.
- If we assume that (1) and (3) holds, then since G has no cycles, then $|V| - |E| = 1$, so there is one connected component, and thus the graph is connected.
- If we assume that (2) and (3) holds, then again we know that the number of connected components is 1 since we're connected, so the number of edges must be one less than the number of vertices.

This proves the lemma. □

Corollary 6.1

If G is a tree, then $|E| = |V| - 1$.

6.6.5 Intuition for a Greedy Algorithm

The intuitive – and obvious – way to begin is to just pick the edge with the smallest weight. That edge will be a part of your tree. From there, it becomes somewhat unclear what we should do. However, this works!

Lemma 6.2

If e is a minimum weight edge in G, then there is a minimum spanning tree of G that contains e.

If e is a unique minimum weight edge in G, then all minimum spanning trees of G will contain e.

Proof. Take a minimum spanning tree T. If $e \in T$, then we're done. Otherwise, suppose we add e to our minimum spanning tree. Then, we get a cycle. So, we can find some other e' in this cycle and let $T' = T \cup \{e\} \setminus \{e'\}$. This is another spanning tree as the tree is still connected with the right number of edges. Then, the weight of T' is given by

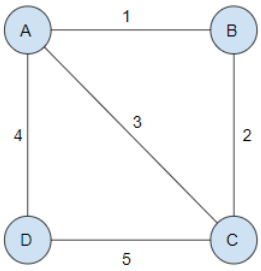
$$w(T') = w(T) + w(e) - w(e')$$

but since e is a minimum weight edge, and the weight of e must be no more than e', so it follows that

$$w(T') \leq w(T)$$

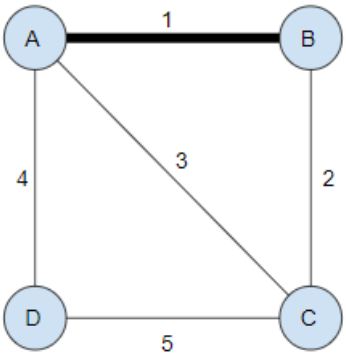
Additionally, if e is a unique minimum weight edge, then $w(T') < w(T)$. So, we are done. □

6.6.6 Example: MST

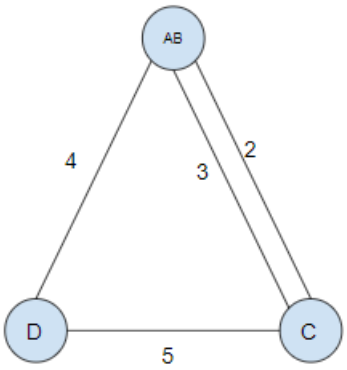


Consider the graph above. Find the minimum spanning tree.

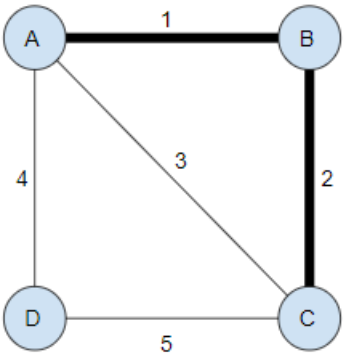
From our lemma, we know that the edge of weight 1 must be in the tree since it is minimal. So, we can select it.



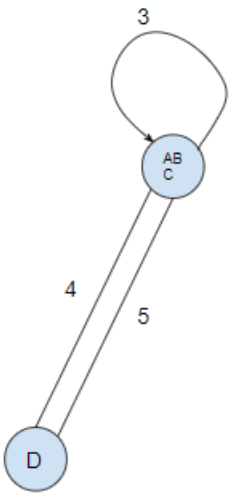
Our lemma doesn't say anything about what to do when we have to consider multiple edges. However, we can use a trick – something we know is that A and B essentially form a connected component, so we can "merge" them together.



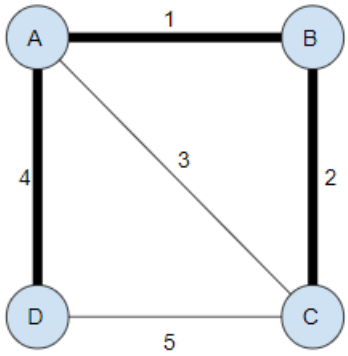
From here, we can see that the edge of weight 2 must be in the tree since it is minimal. So, we can select it.



Making use of the same trick that we described above, we have:



Since MSTs cannot include loops, the edge of weight 4 must be in the tree since it is minimal. So, we can select it.



Since we have reached every vertex in G, we are done. Indeed, if we try to use the trick that we described above, we get:

Here, it's obvious that the only edges we need to consider would just loop back, and since we don't allow loops, we are done.

So, the lemma only gave us the first step of the process. However, by contracting along the edge, we get a new version of the original problem, from which we can apply the lemma again. By repeating this process, we get the MST.

6.6.7 Kruskal's Algorithm

Kruskal's algorithm essentially says to repeatedly add the lightest edge that does not create a cycle. That is:

```
Kruskal(G):
    T = {}
    while |T| < |V| - 1:
        find lightest e such that T u {e} does not have a cycle
        add e to T

    return T
```

To check that $T \cup \{e\}$ doesn't create a cycle, we can check if the endpoints of e are in different connected components of T. What we mean by this is, if we're just adding the lightest edge every time, it's possible that we'll have several trees; for the sake of simplicity, call these two trees T_1 and T_2. Then, if one endpoint of an edge e is in T_1 and the other endpoint is in T_2, then it's not a cycle. If both endpoints e are in T_1 or T_2 (but not both), then this will form a cycle and so we don't consider this e.

The while loop runs in roughly $\mathcal{O}(|V|)$ time. For each iteration, we need to check if there is a cycle that can be formed by picking some e; this takes $\mathcal{O}(|V|)$ time. Going through all of the edges take $\mathcal{O}(|E|)$ time. Therefore, the final runtime is $\mathcal{O}(|V|^2|E|)$ time.

6.6.8 First Optimization of Krustal's Algorithm: Recomputing Edge Weights

Something to note is that, when we're checking edge weights, we're constantly going to see the same edges. If we, for example, add an edge $e = 5$ to our tree, then we don't need to consider the edges with weight less than 5. So, one optimization we can make is to simply not consider any edges that have weight less than the heaviest edge in the tree. So:

```
Kruskal(G):
    T = {}
    for e in E in increasing order of weight:
        if endpoints in different connected components:
            add e to T
    return T
```

Sorting takes $\mathcal{O}(|E|\log(|E|))$. The loop goes through $|E|$ iterations, and computing the connected components takes $|V|$ time. So, our new runtime is

$$\mathcal{O}(|E|\log(|E|) + |V||E|) = \mathcal{O}(|V||E|)$$

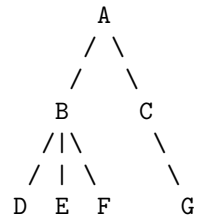
which is somewhat better.

6.6.9 First Optimization of Krustal's Algorithm: Union-Find

Something else to note is that, when checking connected components, we're essentially computing the same connected components over and over again, all while barely making changes to said connected components. So, we want a data structure that can keep track of a bunch of sets. The idea is that we want a data structure that can

- Create a new set: `v`
- Join two sets: `join(v, e)`
- Check representative element to see if v and w are in the same set: `rep(v)`

This is known as the *union-find* data structure. Each set is a directed tree with the representative element at the root. This might look something like[8]



where the representative is A. So, creating a new set takes $\mathcal{O}(1)$ time. Finding the representative takes $\mathcal{O}$(depth of tree). Joining two trees (which is as simple as having one representative point to the other representative) takes $\mathcal{O}$(depth of tree) time, since we need to find the representative of the given elements. If we maintain the depth correctly, then the runtimes become $\mathcal{O}$(log(number of nodes)).

So, with this in mind, our revised algorithm is as follows:

```
Kruskal(G):
    T = {}
    add each v to Union-Find data structure
    for e = (u, v) in increasing order of weight:
        if rep(u) != rep(v):
            add e to T
            join(u, v)
    return T
```

[8]Imagine each branch pointing upwards, and that each node can have more than 2 children (i.e. it's not a binary tree).

Sorting takes $\mathcal{O}(|E|\log(|E|))$. Adding $|V|$ vertices take $\mathcal{O}(|V|)$ times ($|V|$ `join` calls). Then, iterating through each edge takes $\mathcal{O}(|E|)$ time. For each edge, we need to perform $|E|$ `rep` operations, and it we have $|V|$ `join` operations. Overall, this takes $\mathcal{O}(|E|\log(|E|))$ time, assuming we have a reasonable union-find implementation[9].

6.6.10 Trees and Cuts

Lemma 6.3

Let G be a weighted graph, and let C be a cut (i.e. a partition of the vertices of G into two sets).

- Let e be a lowest weight edge crossing C. Then, there exists a MST of G that contains e.
- Let e be the unique lowest weight edge crossing C. Then, every MST of G contains e.

Proof. Suppose we have a cut C which splits the vertices V into two subsets V_1 and V_2. Let T be a MST; then, T's edges will have to cross this cut at some point. If $e \in T$, we're done. Otherwise, we can add the edge to T such that it crosses the cut. This creates a cycle R, which implies that R contains some other e' that crosses the cut C. Then, let

$$T' = T \cup \{e\} \setminus \{e'\}$$

Then, the weight of T' is given by

$$w(T') = w(T) + w(e) - w(e')$$

but since e is a minimum weight edge, and the weight of e must be no more than e', so it follows that

$$w(T') \leq w(T)$$

Additionally, if e is a unique minimum weight edge, then $w(T') < w(T)$, which is a contradiction as this implies that T wasn't a minimum spanning tree. So, we are done. □

6.6.11 Prim's Algorithm

Prim's algorithm relies on the above lemma. Let $b(v)$ be the lightest weight edge (for vertices that we have not reached) that we have discovered that allows us to reach v from the vertices that we have reached.

```
Prims(G):
    T = {}
    b(v) = inf
    b(s) = 0
    Insert all V's into priority queue Q
    while Q not empty:
        u = DeleteMin(Q)
        If u != s:
            Add (u, prev(u)) to T
        For all (u, v) in E:
            if v in Q and b(v) > l(u, v):
                b(v) = l(u, v)
                DecreaseKey(v)
                prev(v) = u
    return T
```

[9]Note that the above discussion of union-find is far from ideal. We can make use of, say, path compression to make this more ideal.

Basically, what's going on is that:

- For all v in the priority queue (which we haven't reached), $b(v)$ will store the cheapest length of an edge that connects it to some vertex that is already connected to s, and `prev(v)` will tell you the other end of that best edge.
- So, find the vertex in Q with the smallest with the smallest value of b, which is the lightest edge which connects some vertex in u which has been connected to s to some vertex that hasn't. We will add that edge to the tree, and then we need to do some updates.

This algorithm looks nearly like Dijkstra's algorithm, and in fact has the same runtime as Dijkstra's algorithm. That is, $\mathcal{O}(|V|\log(|V|) + |E|)$.

7 Dynamic Programming

Consider the Fibonacci sequence

$$F_n = F_{n-1} + F_{n-2}$$

The naive algorithm would just recursively call F_{n-1} and F_{n-2}, which is very inefficient, especially since we would be making duplicate calls. Instead, we can *tabulate* the answers, thus saving us a bunch of time.

So, what is a dynamic program? The idea is as follows:

1. Relate your answer to some family of similar subproblems. In the case of Fibonacci, we had to relate the nth Fibonacci number to all of the Fibonacci numbers before it; that is, we had to compute F_t for all $t \in [t, n] \subseteq \mathbb{Z}$.
2. There should be a recurrence relation that gives the answer to each subproblem in terms of answers to simpler subproblems.
3. Create a *table*, compute the answers to all subproblems, and then tabulate them (store the answers in the table so that you can refer back to the table when you need the answer from the previous iteration for your current iteration). This is done in the simplest to most complicated order.

Remark: You can *usually* look up entries in the table in constant time. Generally speaking, the table is done using an array or a hash map.

A few notes about dynamic programming.

- The general correct proof outline is to prove by induction that each table entry is filled out correctly. Essentially, you want to make sure the base case is correct, and then make use of the recurrence relationship for the inductive hypothesis.
- The runtime of dynamic programming is *usually* the number of subproblems *multiplied by* the time per subproblem.
- For finding the recurrence, which is very important in a dynamic programming algorithm, you often look at the first or last choice and see what things look like without that choice.

7.1 Problem: Longest Common Subsequence

Given a sequence $a_1 a_2 \dots a_n$, we can get a subsequence by removing some entries in the sequence. For example, if we have a sequence `ABCD`, a subsequence would be `ACD` (by removing the `B`). Given two sequences $a_1 a_2 \dots a_n$ and $b_1 b_2 \dots b_m$, then the common subsequence is just some $c_1 \dots c_k$ that is a subsequence of both $a_1 a_2 \dots a_n$ and $b_1 b_2 \dots b_m$.

So, the problem statement is as follows: Given two sequences, compute the longest common subsequences. That is, the subsequence with as many letters as possible.

7.1.1 Example: Longest Common Subsequence

Suppose $X =$ `ABCBA` and $Y =$ `ABACA`. Then, the longest common subsequence of X and Y is `ABCA`.

7.1.2 Case Analysis

How do we compute `LCSS`$(A_1 A_2 \dots A_n,\ B_1 B_2 \dots B_m)$? We need to consider several cases for the common subsequence.

1. It does not use A_n (it does not use the last letter in the sequence A). That is, if the common subsequence does not use A_n, it is actually a common subsequence of

   $$A_1A_2A_{n-1} \text{ and } B_1B_2 \dots B_m$$

   Therefore, in this case, the longest common subsequence would be

   $$\texttt{LCSS}(A_1A_2 \dots A_{n-1}, B_1B_2 \dots B_m)$$

2. It does not use B_m (It does not use the last letter in the sequence B). If the common subsequence does not use B_m, it is actually a common subsequence of

   $$A_1A_2A_n \text{ and } B_1B_2 \dots B_{m-1}$$

   Therefore, in this case, the longest common subsequence would be

   $$\texttt{LCSS}(A_1A_2 \dots A_n, B_1B_2 \dots B_{m-1})$$

3. It uses both A_n and B_m, and these characters are the same. If a common subsequence uses both A_n and B_m, then:

   - These characters must be the same.
   - Such a subsequence can be obtained by taking a common subsequence of $A_1A_2 \dots A_{n-1}$ and $B_1B_2 \dots B_{m-1}$ and adding a copy of $A_n = B_m$ to the end.
   - Therefore, the longest length of such a subsequence is given by

     $$\texttt{LCSS}(A_1A_2 \dots A_n, B_1B_2 \dots B_{m-1}) + 1$$

Remark: We can be in both case 1 and case 2.

7.1.3 Recursion

We've broken the longest common subsequence into three different cases, and for each of these cases we managed to compute the longest common subsequence of subsequences in that case by a longest common subsequence problem that is a little bit simpler than the one that we started at. Of course, the *best* longest common subsequence overall is just the best that we can get from any of these three cases.

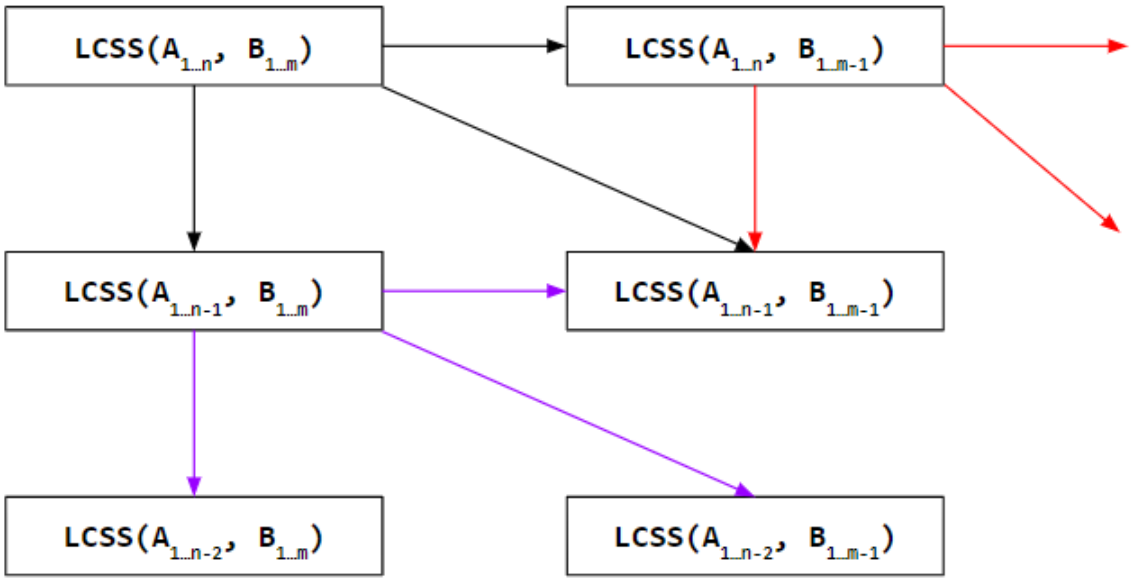
So, all we need to do is look at the three cases and take the *maximum* of the subsequences. That is

$$\texttt{LCSS}(A_1A_2 \dots A_n, B_1B_2 \dots B_m) = \max \begin{Bmatrix} \texttt{LCSS}(A_1A_2 \dots A_{n-1}, B_1B_2 \dots B_m) \\ \texttt{LCSS}(A_1A_2 \dots A_n, B_1B_2 \dots B_{m-1}) \\ \texttt{LCSS}(A_1A_2 \dots A_{n-1}, B_1B_2 \dots B_{m-1}) \end{Bmatrix}$$

where the last option is allowed only if $A_n = B_m$.

7.1.4 Analyzing Recursive Calls

Suppose we make a call to $\texttt{LCSS}(A_1 \dots A_n,\ B_1 \dots B_m)$. What recursive calls are made?



Here, many recursive calls are omitted. However, the key thing to note is that we're making a call to (LCSS)$(A_1A_2\dots A_{n-1}, B_1B_2\dots B_{m-1})$ three times (there are three different paths to get to that function call).

Important Note 7.1

This is the key difference between a dynamic programming algorithm and a divide and conquer algorithm.

- The recursive subcalls for a divide and conquer algorithm are significantly smaller than the original problem. So, you never have to compute that many recursive subcalls even if you do the entire recursion tree.
- The recursive subcalls for a dynamic program algorithm are almost as big as the original call. *However*, the same subproblems will show up multiple times in the recursion tree. So, rather than recomputing each subproblem, we can compute each subproblem once and then *store* the result so we can refer to it later if we need it.

In the case of our problem above, we only ever see $\texttt{LCSS}(A_1A_2\dots A_k, B_1B_2\dots B_\ell)$ for some k and ℓ.

7.1.5 Base Case

Our recursion also needs a base case. So, our base case is:

$$\texttt{LCSS}(\emptyset, B_1B_2\dots B_m) = \texttt{LCSS}(A_1A_2\dots A_n, \emptyset) = 0$$

7.1.6 Algorithm

To take advantage of the fact that our dynamic program makes multiple *repeated* subcalls (i.e. subcalls with the same inputs), we use *tabulation* to *store* the results of one subcall and then, when needed, we can retrieve the results of the subcall.

Thus, our algorithm is[10]:

[10]Because the `verbatim` environment is limiting, I'll represent $A_1A_2\dots A_n$ as $A[1..n]$ and $B_1B_2\dots B_m$ as $B[1..m]$.

```
LCSS(A[1..n], B[1..m]):
    Initialize Array T[0..n, 0..m]
    // T[i, j] will store LCSS(A[1..i, 1..j])
    For i = 0 to n:
        For j = 0 to m:
            // Base Case
            If i == 0 OR j == 0:
                T[i, j] = 0

            // Case (3)
            Else if A[i] == B[j]
                T[i, j] = max(T[i - 1, j], T[i, j - 1], T[i - 1, j - 1] + 1)

            // Case (1) or (2)
            Else:
                T[i, j] = max(T[i - 1, j], T[i, j - 1])

    Return T[n, m]
```

The runtime is as follows:

- The outer loop runs in n time.
- The inner loop runs in m time.
- Everything inside the inner loop runs in constant time.

This gives us the runtime $\mathcal{O}(nm)$.

7.1.7 Example: Longest Common Subsequence Redux

Suppose $X =$ `ABCBA` and $Y =$ `ABACA`. Find the longest common subsequence.

We will tabulate the process. Call the table T.

	∅	A	AB	ABA	ABAC	ABACA
∅						
A						
AB						
ABC						
ABCB						
ABCBA						

We will go through each table one row at a time.

1. If we start at `i = 0`, then regardless of the value of `j`, the entire first row will be `0` by the first `if`-condition. So:

	∅	A	AB	ABA	ABAC	ABACA
∅	0	0	0	0	0	0
A						
AB						
ABC						
ABCB						
ABCBA						

2. Now, when `i = 1`, then we need to consider the value of `j`.

- If `j = 0`, then the entry at `T[1, 0] = 0`.
- If `j = 1`, then we're comparing `A` with `A`, so we need to consider the second `if`-condition. In this case, we need to check the entry directly above (`T[i - 1, j]`), to the left (`T[i, j - 1]`), and directly adjacent in the top-left entry (`T[i - 1, j - 1] + 1`). Here, we see that the values are `0`, `0`, and `0 + 1 = 1`, respectively. So, we store the *maximum* of these values in `T[1, 1]`. Thus, `T[1, 1] = 1`.
- If `j = 2`, then we're comparing `A` with `AB`, so we need to consider the third `if`-condition. In this case, we check the entry directly above `T[i - 1, j]` and directly to the left (`T[i, j - 1]`), and take the maximum. In our case, we have `0` and `1` (which we computed from the previous bullet point), respectively. Thus, `T[1, 2] = 1`.
- If `j = 3`, then we're comparing `A` with `AB`, so we need to consider the second `if`-condition. In this case, we check the entry directly above, left, and adjacent (add one to the end result) of `[i, j] = [1, 3]`. We get the values `0`, `1`, and `0 + 1 = 1`, respectively. Thus, `T[1, 3] = 1`.
- Continuing on, we get `1`'s for the remaining cells in this row.

Therefore, the second row is:

	∅	A	AB	ABA	ABAC	ABACA
∅	0	0	0	0	0	0
A	0	1	1	1	1	1
AB						
ABC						
ABCB						
ABCBA						

Omitting the rest of the work, we see that the result is:

	∅	A	AB	ABA	ABAC	ABACA
∅	0	0	0	0	0	0
A	0	1	1	1	1	1
AB	0	1	2	2	2	2
ABC	0	1	2	2	3	3
ABCB	0	1	2	2	3	3
ABCBA	0	1	2	3	3	4

And so the answer is given by the bottom-right entry, or $\boxed{4}$.

7.1.8 Finding the Longest Common Subsequence

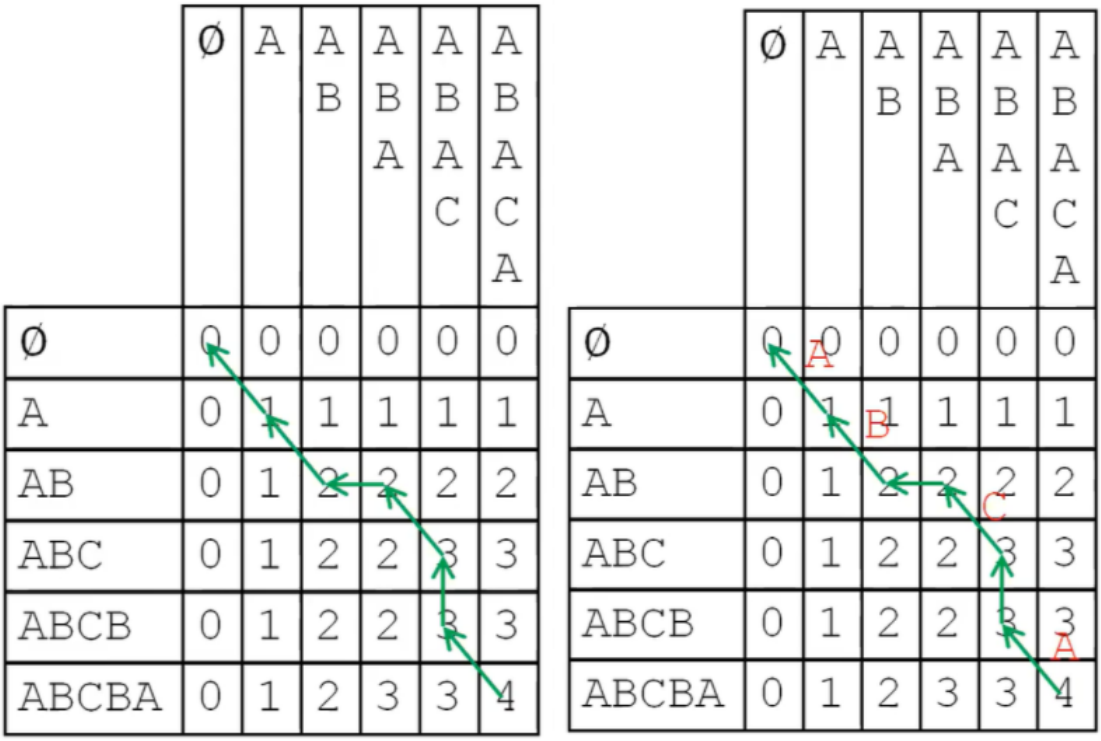
The above algorithm finds the *length* of the longest common subsequence. What if we wanted the actual subsequence? Well, we can *backtrack*. So, consider the table above.

	∅	A	AB	ABA	ABAC	ABACA
∅	0	0	0	0	0	0
A	0	1	1	1	1	1
AB	0	1	2	2	2	2
ABC	0	1	2	2	3	3
ABCB	0	1	2	2	3	3
ABCBA	0	1	2	3	3	4

We can use this trick, based on the recurrence, to find out what the subsequence is. If you're at `[i, j]`, consider the neighbors `[i - 1, j]`, `[i, j - 1]`, and `[i - 1, j - 1]`.

- If the values at all three of those entries are equal, then take the diagonal path; that is, take `[i - 1, j - 1]`.
- If `[i - 1, j - 1]` and `[i - 1, j]` are equal but `[i, j - 1]` isn't equal to any of those two, then take the `[i, j - 1]` path.
- If `[i - 1, j - 1]` and `[i, j - 1]` are equal but `[i - 1, j]` isn't equal to any of those two, then take the `[i - 1, j]` path.

Once you reach the beginning, then you backtrack again. For each *diagonal* path that was taken, take the last letter of one of the subsequences (the subsequences corresponding to the cell should end with the same letter). Once you reach back to the bottom-right square, then you can just concatenate the letters that you found.



7.1.9 Proof of Correctness

We now need to prove, by induction, that this algorithm is correct.

Proof. We will use induction on i, j to show that the value assigned to $T[i, j]$ is the correct value for $\texttt{LCSS}(A_1 \dots A_i, B_1 \dots B_j)$.

- Base Case: When $i = 0$ or $j = 0$, then we assign `0` since the empty string has no common subsequence with a string of some length.
- Inductive Step: Suppose the previous values are assigned correctly. We note that, because of the recursive relationship and the inductive hypothesis (since we have previously filled out $T[i-1, j]$, $T[i, j-1]$, and $T[i-1, j-1]$ correctly), that $T[i, j]$ gets the correct values.

This completes the proof. □

7.2 Problem: Knapsack

You are a burglar and are in the process of robbing a home. You have found several valuable items, but the sack you brought can only hold so much weight. What is the best combination of items to steal?

Some alternative formulations are:

- You are packing for a trip and your suitcase can only store so much stuff. You want to pack the most useful items for this trip.
- You're building a spacecraft, but launching a spacecraft is incredibly expensive based on its weight. So, you want to decide what are the best modules to put on a spacecraft.

Problem Statement: You have an available list of items. Each item has a non-negative integer weight. Your sack also has a capacity. The goal is to find the collection of items so that:

1. The total weight of all of the items is less than or equal to the capacity.
2. Subject to condition 1, the total value is as large as possible.

There are two slight variations of this problem:

- Each item can be taken as many times as you want.
- Each item can be taken at most once.

In our case, we'll assume that you can take each item as many times as you want.

7.2.1 Example: Knapsack

Suppose you have the following items:

Item	Weight	Value
A	1	\$1
B	2	\$4
C	3	\$3
D	4	\$5

Further suppose that the maximum capacity is 6. What is the best set of items to take, assuming you can only take one copy of each item?

We can brute-force this.

- With weights $1 + 2 + 3 = 6$ (A, B, C), you can get $1 + 4 + 3 = \$8$.
- With weights $4 + 2 = 6$ (B, D), you can get $5 + 4 = \$9$.

The other weights smaller than 6 do not give us the best choice. Thus, the best set of items to take is $\{B, D\}$ with a value of \$9.

7.2.2 Greedy Algorithms Don't Work

Problems like this one usually suggest greedy algorithms as a solution. However, greedy algorithms don't work. To see that this is the case, consider the following two counterexamples.

1. Most Valuable Item: Suppose the greedy algorithm defines "best" to be the most valuable item. Suppose you have a knapsack of capacity 6 and the following set of items that you can steal, where you can only take one of each item.

Item	Weight	Value
A	6	\$10
B	3	\$9
C	3	\$9

Then:

- The greedy solution would immediately go for A, since it is the most valuable item, which has value **\$10** and weight 6.
- The optimal solution is B and C, which has value **\$18** and weight 6.

2. Biggest Value/Weight: Suppose the greedy algorithm defines "best" to be the item with the highest value to weight ratio. Again, suppose you have a knapsack of capacity 6 and the following set of items that you can steal, where you can only take one of each item.

Item	Weight	Value
A	4	\$5
B	3	\$3
C	3	\$3

Then:

- The greedy solution would immediately go for A, since A has the highest value/weight ratio, which has value **\$5** and weight 4. Since the other items have weight 3 and $4 + 3 > 6$, we can't pick any of those items.
- The optimal solution is B and C, which has value **\$6** and weight 6.

7.2.3 Subproblems

Suppose we make *one* choice of an item to go into the bag. Then, what is left?

- The remaining items must have a total weight at most Capacity − Weight of Item.
- The total value of the items is

$$\text{Value of Item} + \text{Value of Other Items}$$

- Therefore, we want to maximize the value of the other items subject to their weight not exceeding the Capacity − Weight of Chosen Item.

Therefore, the subproblem is

$$\text{BestValue(c')}$$

where c' is the new capacity.

7.2.4 Recursion

What is BestValue(C), where C is the capacity?

- If there are *no items* in the bag, then

$$\text{Value} = 0$$

- If item i is in the bag, then the value is given by

$$\text{Value} = \text{BestValue}(C - \text{Weight}(i)) + \text{Value}(i)$$

So, the best attainable value for a sack with capacity C is the maximum of either:

- 0, *or*
- For the item i with weight such that $\text{Weight}(i) \leq C$ and $\text{Weight}(i)$ is maximal,

$$\text{Value}(i) + \text{BestValue}(C - \text{Weight}(i))$$

Therefore, the recursion is given by

$$\text{BestValue}(C) = \max(0, \max_{\text{Weight}(i) \leq C} (\text{Value}(i) + \text{BestValue}(C - \text{Weight}(i))))$$

Here, the 0 takes into account the possibility that the sack might be too small to fit any items.

7.2.5 Algorithm

The algorithm, which makes use of the recurrence above, can be described like so:

```
Knapsack(Wt, Val, Cap):
    Create Array T[0..Cap]
    // Consider all capacities from 0..Cap
    For C = 0 to Cap:
        // Accounts for the fact that the sack might be too small to
        // hold any items
        T[C] = 0
        // This is computing the maximum of all of the possible items
        For items i with Wt(i) <= C:
            If T[C] < Val(i) + T[C - Wt(i)]:
                T[C] = Val(i) + T[C - Wt(i)]
    Return T[Cap]
```

To see the runtime, we note that:

- There are $\mathcal{O}(\text{Cap})$ subproblems (the outer `for`-loop).
- For each subproblem, we need to (possibly) consider all of the possible items (the inner `for`-loop).

Therefore, the runtime is given by

$$\mathcal{O}(\text{Cap} \cdot \text{Num. Items})$$

7.2.6 Example: Knapsack Redux

Suppose you have the following items:

Item	Weight	Value
A	1	\$1
B	2	\$4
C	3	\$3
D	4	\$5

Further suppose that the maximum capacity is 6.

1. What is the highest possible value you can get, (assuming you can only take as many of the same item as you want)?

   We can make use of the algorithm above. We define the table T like so:

C	0	1	2	3	4	5	6
BestValue							

   - Start at $c = 0$. Then, it's trivial to see that you can't get anything of value since, well, your capacity is 0. So, $T[0] = 0$ and

C	0	1	2	3	4	5	6
BestValue	0						

- Now, we're at $c = 1$. By the algorithm we can initially set $T[1] = 0$. There are two choices that we can consider:
  - We can fit item A in and get \$1.
  - Or, we can fit nothing and get \$0.

  Clearly, the better choice is \$1, so we put that into $T[1]$.

C	0	1	2	3	4	5	6
BestValue	0	1					

- Next, we're at $c = 2$. By the algorithm we can initially set $T[2] = 0$. There are several choices we can consider.
  - We can fit item B in and get \$4.
  - Or, we can take A, along with $T[1]$, and get $\$1 + \$1 = \$2$.

  Clearly, \$4 is the better choice, so we put that into $T[2]$.

C	0	1	2	3	4	5	6
BestValue	0	1	4				

- Next, we're at $c = 3$. There are several choices to consider.
  - We can fit item C in and get \$3.
  - We can fit item A, along with what we already have in $T[2]$, and get $\$1 + \$4 = \$5$.
  - We can fit item B, along with what we already have in $T[1]$, and get $\$2 + \$1 = \$3$.

  Clearly, \$5 is the better choice, so we put that into $T[3]$.

C	0	1	2	3	4	5	6
BestValue	0	1	4	5			

- Next, we're at $c = 4$. There are, again, several cases to consider.
  - We can fit item D in and get \$4.
  - We can fit item C, along with what we have in $T[1]$, and get $\$3 + \$1 = \$4$.
  - We can fit item B, along with what we have in $T[2]$, and get $\$4 + \$4 = \$8$.
  - We can fit item A, along with what we have in $T[3]$, and get $\$1 + \$5 = \$6$.

  Here, we see that \$8 is the best choice, so we put that into $T[4]$.

C	0	1	2	3	4	5	6
BestValue	0	1	4	5	8		

- Next, we're at $c = 5$. There are several choices to consider.
  - We can fit item D, along with $T[1]$, and get $\$5 + \$1 = \$6$.
  - We can fit item C, along with $T[2]$, and get $\$3 + \$4 = \$7$.
  - We can fit item B, along with $T[3]$, and get $\$4 + \$5 = \$9$.
  - We can fit item A, along with $T[4]$, and get $\$1 + \$8 = \$9$.

  Here, we see that \$9 is the best choice. So, we put that into $T[5]$.

C	0	1	2	3	4	5	6
BestValue	0	1	4	5	8	9	

- Next, we're at $c = 6$. There are several choices to consider.
  - We can fit item D, along with $T[2]$, and get $\$5 + \$4 = \$9$.
  - We can fit item C, along with $T[3]$, and get $\$3 + \$5 = \$8$.

- We can fit item B, along with $T[4]$, and get $\$4 + \$8 = \$12$.
- We can fit item A, along with $T[5]$, and get $\$1 + \$9 = \$10$.

Here, we see that \$12 is clearly the better choice. So, we put that in $T[6]$.

C	0	1	2	3	4	5	6
BestValue	0	1	4	5	8	9	12

So, the answer is given by $T[6] = 12$.

2. What was the best possible set of items to collect?

Remember that our table T was populated to have the following values:

C	0	1	2	3	4	5	6
BestValue	0	1	4	5	8	9	12

So, we can backtrack to find out what items we collected.

- First, we note that, to get \$12, we had to pick item B along with whatever was in $T[4]$. So, we know that **B** was an item that we collected. But, since we had to use whatever was in $T[4]$, we consider that entry next.
- At $T[4] = 8$, we note that, to get \$8, we had to pick item B along with whatever was in $T[2]$. So, we know that **B** was an item that was collected. Additionally, since we had to use whatever was in $T[2]$, we consider that entry next.
- At $T[2] = 4$, we note that, to get \$4, we had to pick item B. So, we know that **B** was an item that was collected. Since we didn't have to depend on the maximum value at some other entry, we're done.

Therefore, we collected three **B**'s for a total of **\$12**.

7.2.7 Non-Repeating Items: Subproblems

Suppose we tried to do this process with non-repeating items. We can try to repeat what we did above (with the subproblems) for this case. That is, suppose we put one item into the bag.

- Then, the remaining items must have total weight at most Capacity − Weight of Item.
- The total value of the items is

$$\text{Value of Item} + \text{Value of Other Items}$$

- Therefore, we want to maximize the value of the other items subject to their weight not exceeding the Capacity − Weight of Chosen Item **and** such that the chosen item(s) cannot be picked again.

The recursion needs to now keep track of the remaining capacity, since the item that was picked cannotbe used again. This makes the problem a little harder. So, we can try to make several subproblems (where c' is the new capacity).

1. Suppose the subproblem we use is $\text{BestValue}_{\neq i}(c')$. In other words, the best value we can achieve without using item i that doesn't go over the capacity. However, we **cannot** make a recursion out of this. This is because after using item j, the remaining items cannot include i and j. However, we need to create a recursion such that a recursive subproblem can be solved in terms of other recursive subproblems.

2. Suppose the subproblem we include is $\text{BestValue}_{\neq i,\neq j}(c')$. Again, we cannot make a recursion for this because this would imply the need to exclude a third item.

3. Suppose the subproblem we use is

$$\text{BestValue}_S(c') = \max\left(0, \max_{i \in S}(\text{Value}(i) - \text{BestValue}_{S\setminus\{i\}}(\text{Cap} - \text{Weight}(i)))\right)$$

In other words, the best value achievable using only items from S with total weight at most Cap. This recursion *works*, but this doesn't give us a very good algorithm. This is because the number of subproblems is more than $2^{\text{Number of Items}}$ since we're effectively considering every possible subset $(S \setminus \{i\})$.

4. Instead of trying the above approaches, let's think of something else. Suppose we have items coming along a conveyor belt. You decide, one at a time, whether to add the item to your sack. Then, the question we can think about is – do we take the last item or leave it alone?

   - If we take the item, the recursion becomes

$$\text{BestValue}_{\leq n-1}(\text{Cap} - \text{Weight}(n)) + \text{Value}(n)$$

   - If we don't take the item, then the recursion becomes

$$\text{BestValue}_{\leq n-1}(\text{Cap})$$

   If we tried to convert this into a recursion, then we only need subproblems of the form

$$\text{BestValue}_{\leq k}(\text{Cap})$$

   In other words, what happens to the first k items? Somehow, by imposing this order, we don't need to deal with every possible subset of items.

7.2.8 Non-Repeating Items: Recursion

So, our recursion $\text{BestValue}_{\leq k}(\text{Cap})$ is defined by the highest total value of items with the total weight at most Cap using only items from the first k. Thus:

- Base Case: If $k = 0$, then we can't take any items. So:

$$\text{BestValue}_{\leq 0}(\text{Cap}) = 0$$

- Recursion: $\text{BestValue}_{\leq k}(\text{Cap})$ is the maximum of either:

  1. $\text{BestValue}_{\leq k-1}(\text{Cap})$: You don't take the item, so we're finding the best value of the $k-1$ items.
  2. If $\text{Weight}(k) \leq \text{Cap}$, then $\text{BestValue}_{\leq k-1}(\text{Cap} - \text{Weight}(k)) + \text{Value}(k)$: You take the item, add on the value, and then add on the best value of the $k-1$ items.

7.2.9 Non-Repeating Items: Example

Suppose you have the following items:

Item	Weight	Value
A	1	\$1
B	2	\$4
C	3	\$3
D	4	\$5

Further suppose that the maximum capacity is 6.

1. Find the highest possible value you can get, assuming you can only take one copy of each item.

Instead of a one-dimensional table, we have a two-dimensional table T.

Cap	0	1	2	3	4	5	6
$\emptyset$							
A							
AB							
ABC							
ABCD							

Here, we say that $T[\texttt{x}, n]$ refers to the cell with the row being `x` (the items that we can pick) and the column being n (the capacity).

- At the row with the $\emptyset$, we essentially have $k = 0$. In other words, we're not able to pick anything. So, it follows that, regardless of capacity, if there's nothing to pick, we'll end up with \$0.

Cap	0	1	2	3	4	5	6
$\emptyset$	0	0	0	0	0	0	0
A							
AB							
ABC							
ABCD							

- Suppose we're at the row with `A`; that is, we have $k = 1$ so we can pick one item, which in our case is A. Now, when the capacity is 0, clearly we cannot pick anything. But, as long as the capacity is greater than or equal to 1, we can pick A (value of 1). So:

Cap	0	1	2	3	4	5	6
$\emptyset$	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
AB							
ABC							
ABCD							

- Suppose we're at the row with `AB`; that is, we're at $k = 2$ so we can pick two items, which in our case is A or B. Now, when the capacity is 0, clearly we cannot pick anything. When the capacity is 1, clearly we can only pick A, so we can take $T[\texttt{A}, 1]$. When the capacity is 2, clearly the most optimal choice is B (value of 4). When the capacity is greater than 2, we can pick B and $T[\texttt{A}, c]$, where c is the capacity such that $c > 2$ (value of 5).

Cap	0	1	2	3	4	5	6
$\emptyset$	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
AB	0	1	4	5	5	5	5
ABC							
ABCD							

- Suppose we're at the row with `ABC`; that is, we're at $k = 3$ so we can pick three items, which in our case is A or B or C.
  - Now, when the capacity is 0, clearly we cannot pick anything.
  - When the capacity is 1, clearly we can pick $T[\texttt{AB}, 1]$ (value of 1).
  - When the capacity is 2, clearly the most optimal choice is $T[\texttt{AB}, 2]$ (value of 4). This is because there isn't any space for item C.
  - When the capacity is 3, clearly the most optimal choice is just $T[\texttt{AB}, 3]$. Note that if we picked C, then we would be worse-off.

- When the capacity is 4, clearly the most optimal choice is just $T[\texttt{AB}, 4]$. Note that if we picked C, then we would be worse-off.
- When the capacity is 5, we can pick C and also $T[\texttt{AB}, 2]$ (value of 7).
- When the capacity is 6, we can pick C and also $T[\texttt{AB}, 3]$ (value of 8).

Thus:

Cap	0	1	2	3	4	5	6
$\emptyset$	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
AB	0	1	4	5	5	5	5
ABC	0	1	4	5	5	7	8
ABCD							

- Suppose we're at the row with ABCD; that is, we're at $k = 4$ so we can pick all four items if we can, which in our case is A or B or C or D.
  - Now, when the capacity is 0, clearly we cannot pick anything.
  - When the capacity is 1, clearly we can pick $T[\texttt{ABC}, 1]$ (value of 1).
  - When the capacity is 2, clearly the most optimal choice is $T[\texttt{ABC}, 2]$ (value of 4).
  - When the capacity is 3, clearly the most optimal choice is just $T[\texttt{ABC}, 3]$.
  - When the capacity is 4, clearly the most optimal choice is just $T[\texttt{ABC}, 4]$. Here, we could also just pick D.
  - When the capacity is 5, the optimal choice is just $T[\texttt{ABC}, 3]$ (value of 7). Note that the other option is D and A, which is worse.
  - When the capacity is 6, we can pick D and also $T[\texttt{ABC}, 2]$ (value of 9).

  Thus:

Cap	0	1	2	3	4	5	6
$\emptyset$	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
AB	0	1	4	5	5	5	5
ABC	0	1	4	5	5	7	8
ABCD	0	1	4	5	5	7	9

Thus, the final answer is given by $T[\texttt{ABCD}, 6] = 9$.

2. Find the items that you took to get the maximum value found in the previous part.

Using the table T from the previous part:

Cap	0	1	2	3	4	5	6
$\emptyset$	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
AB	0	1	4	5	5	5	5
ABC	0	1	4	5	5	7	8
ABCD	0	1	4	5	5	7	9

We can backtrack to find out what items we collected.

- Starting at $T[\texttt{ABCD}, 6]$, note that, in order to get to this position, we picked $T[\texttt{ABC}, 2]$ and D. So, we know that **D** was an item that we collected, and we additionally consider $T[\texttt{ABC}, 4]$.
- Now that we're at $T[\texttt{ABC}, 2]$, note that, in order to get to this position, we picked $T[\texttt{AB}, 2]$. As we didn't pick up any other items along th way, we can just consider $T[\texttt{AB}, 2]$.

- Now that we're at $T[\texttt{AB}, 2]$, note that, in order to get to this position, we picked **B**. Since we didn't consider any other entries in the table, we're done.

Thus, the solution is **B** and **D**.

7.3 Problem: Chain Matrix Multiplication

How long does it take to multiply matrices? Suppose A is an $n \times m$ matrix and B is a $m \times k$ matrix. Then, for each entry of C (which is a $n \times k$ matrix), of which there are nk entries, we need to sum m terms. Therefore, the runtime[11] is just

$$\mathcal{O}(nmk)$$

Suppose, now, we want to multiply three matrices ABC. We can multiply it two different ways since matrix multiplication is associative. That is, we can do it either way:

$$A(BC) \qquad (AB)C$$

How long does it take to multiply each? Suppose A is a 2×3 matrix, B is a 3×3 matrix, and C is a 3×1 matrix.

- If we did $A(BC)$, then multiplying BC would take $3 \cdot 3 \cdot 1 = 9$ operations and multiplying $A(BC)$ would take $2 \cdot 3 \cdot 1 = 6$ operations. The total runtime is $9 + 6 = 15$.
- If we did $(AB)C$, then multiplying AB would take $2 \cdot 3 \cdot 3 = 18$ operations and multiplying $(AB)C$ would take $2 \cdot 3 \cdot 1 = 6$ operations. The total runtime is $18 + 6 = 24$.

The point is, from this simple example, if the matrices are complicated (not square), the order that you multiply the matrices may matter a lot.

Problem Statement: Find the order to multiply the matrices $A_1, A_2, \ldots, A_m$ that requires the fewest total operations. In particular, assume A_1 is an $n_0 \times n_1$ matrix, A_2 is an $n_1 \times n_2$ matrix, and generally A_k is an $n_{k-1} \times n_k$ matrix.

7.3.1 Recursion

In order to find some sort of a recursion so we can make a dynamic programming algorithm, we can again consider the last step. For some value of k, the last step is:

$$(\underbrace{A_1 A_2 \ldots A_k}_{M_1})(\underbrace{A_{k+1} A_{k+2} \ldots A_m}_{M_2})$$

That is, we've already computed the product of M_1 and M_2, so the last step is to multiply M_1 and M_2. So, if we want to compute the big product of matrices, what's the best runtime? The number of steps is as follows:

- We first need to compute the first half; that is, $\texttt{CMM}(A_1, A_2, \ldots, A_k)$.
- We next need to compute the second half; that is, $\texttt{CMM}(A_{k+1}, A_{k+2}, \ldots, A_m)$.
- Finally, we need to do the final multiplication $(M_1 M_2)$, which takes $n_0 n_k n_m$ operations.

Therefore, the recursion $\texttt{CMM}(A_1, A_2, \ldots, A_m)$ is given by

$$\min_k \left(\texttt{CMM}(A_1, \ldots, A_k) + \texttt{CMM}(A_{k+1}, \ldots, A_m) + n_0 n_k n_m\right)$$

Where we need to consider all possible chain matrix multiplication orders (so we need to take the minimum of all possible values of k).

[11] We'll ignore Strassen's algorithm for the time being.

7.3.2 Subproblems

What subproblems do we need to solve?

- Remember, we cannot afford to solve all possible chain matrix multiplication problems. So, in other words, while we *do* have a recursion, if we were to try every possible k, that would simply take too much time on its own.
- We know that $\texttt{CMM}(A_1, \ldots, A_k)$ requires $\texttt{CMM}(A_1, \ldots, A_k)$ and $\texttt{CMM}(A_{k+1}, \ldots, A_m)$ for various values of k.
- Suppose we wanted to compute $\texttt{CMM}(A_1, \ldots, A_k)$. We might need to break this down into pieces; some of those pieces will be $\texttt{CMM}(A_1, \ldots, A_{k'})$ $(k' < k)$ and some of the pieces will be $\texttt{CMM}(A_{k'}, \ldots, A_k)$; these are all new subproblems to deal with. When we break *these* down, we don't really get anything new; that is, if we tried to run $\texttt{CMM}(A_i, A_{i+1}, \ldots, A_j)$ and break these into two halves, we still have consecutive intervals of these matrices.
- So, the general recursive subproblem that we need to solve is of this form

  $$C(i, j) = \texttt{CMM}(A_i, A_{i+1}, \ldots, A_j)$$

  for $1 \leq i \leq j \leq m$; so there are fewer than m^2 total subproblems. Essentially, all we're doing is We're taking some consecutive collection of indices and we want to compute the chain matrix multiplication of that.

7.3.3 Full Recursion

We now talk about the components of this recursion.

- Base Case: For the base case, $C(i, i) = 0$. All this is saying is that you want to multiply the matrix A_i and that's it; so, there's only one matrix. With a single matrix, we don't need to do anything.
- Recursive Step: The recursive step is given by

  $$C(i, j) = \min_{i \leq k < j} (C(i, k) + C(k + 1, j) + n_i n_k n_j)$$

  If we have $i < j$, then $C(i, j)$ should be the *minimum* of all possible places we can break this product into two pieces; in other words, it's the minimum of the two recursive calls plus the final computation.
- Solution Order: We need to solve the subproblems with smaller $j - i$ first. This ensures that the recursive calls will always be in your table.

7.3.4 Example: Chain Matrix Multiplication

Suppose A is a 2×5 matrix, B is a 5×4 matrix, C is a 4×3 matrix, and D is a 3×5 matrix.

1. Compute the minimum number of operations needed to evaluate $ABCD$.

   We can build this table T:

	A	B	C	D
A				
B				
C				
D				

   Here, we're performing chained matrix multiplication, where we start at some matrix (denoted by the row) and finish at another matrix (denoted by the column). So, if we chose the row A and the column C, this means ABC. Thus, $T[X, Y]$ means that we start at matrix X and end at matrix Y. Thus, we're interested in finding the value of $T[A, D]$ since this means the number of steps

needed to compute $ABCD$.

- First, we note that the bottom-left diagonal calls make no sense. For example, we don't need $T[C, B]$ as this is saying the number of operations needed to compute CB, but we will never need a subproblem where this is the case. So:

	A	B	C	D
A				
B	X			
C	X	X		
D	X	X	X	

- Next, we start with the base case. That is, $T[A, A] = T[B, B] = T[C, C] = T[D, D] = 0$. So:

	A	B	C	D
A	0			
B	X	0		
C	X	X	0	
D	X	X	X	0

- Next, we look at $T[A, B]$. Recall that A is a 2×5 matrix and B is a 5×4 matrix. Further, there is only *one* way to break down AB: as itself. So it follows that AB will take $2 \cdot 5 \cdot 4 = 40$ operations to compute.

	A	B	C	D
A	0	40		
B	X	0		
C	X	X	0	
D	X	X	X	0

- Next, we look at $T[B, C]$. Again, there's only one way to break this down – as itself. So, it follows that BC will take $5 \cdot 4 \cdot 3 = 60$ operations to compute.

	A	B	C	D
A	0	40		
B	X	0	60	
C	X	X	0	
D	X	X	X	0

- For the same reason as above, $T[C, D]$ will take $4 \cdot 3 \cdot 5 = 60$ operations to compute.

	A	B	C	D
A	0	40		
B	X	0	60	
C	X	X	0	60
D	X	X	X	0

- Next, we look at $T[A, C]$. Since there are three matrices here, there are two ways to break this down.
  - $A(BC)$: Note that BC will be a 5×3 matrix. Since A is a 2×5 matrix, it follows that

$$2 \cdot 5 \cdot 3 + T[B, C] = 30 + 60 = 90$$

  - $(AB)C$: Note that AB will be a 2×4 matrix. Since C is a 4×3 matrix, it follows that

$$T[A, B] + 2 \cdot 4 \cdot 3 = 40 + 24 = 64$$

Taking the minimum of the total number of ways to break this problem down, we have **64**. So:

	A	B	C	D
A	0	40	64	
B	X	0	60	
C	X	X	0	60
D	X	X	X	0

- Next, we look at $T[B, D]$. Since there are three matrices here, there are two ways to break this down as well.
  - $B(CD)$: Note that CD is a 4×5 matrix. Since B is a 5×4 matrix, it follows that
  $$5 \cdot 4 \cdot 5 + T[C, D] = 100 + 60 = 160$$
  - $(BC)D$: Note that BC is a 5×3 matrix. Since D is a 3×5 matrix, it follows that
  $$T[B, C] + 5 \cdot 3 \cdot 5 = 60 + 75 = 135$$

  Taking the minimum of the total number of ways to break this problem down, we have **135**. So:

	A	B	C	D
A	0	40	64	
B	X	0	60	135
C	X	X	0	60
D	X	X	X	0

- Finally, we look at $T[A, D]$. Since there are four matrices here, there are three ways to break this down.
  - $A(BCD)$: Here, we know that $T[B, D] = 135$. Additionally, since BCD is a 5×5 matrix and A is a 2×5 matrix, it follows that
  $$2 \cdot 5 \cdot 5 + 135 = 185$$
  - $(AB)(CD)$: Here, we know that AB is a 2×4 matrix and CD is a 4×5 matrix. Additionally, $T[A, B] = 40$ and $T[C, D] = 60$. So, it follows that
  $$T[A, B] + 2 \cdot 4 \cdot 5 + T[C, D] = 40 + 40 + 60 = 140$$
  - $(ABC)D$: Here, we know that ABC is a 2×3 matrix and D is a 3×5 matrix. Additionally, $T[A, C] = 64$. So, it follows that
  $$T[A, C] + 2 \cdot 3 \cdot 5 = 64 + 30 = 94$$

  Taking the minimum of the total number of ways to break this problem down, we have **94**. So:

	A	B	C	D
A	0	40	64	94
B	X	0	60	135
C	X	X	0	60
D	X	X	X	0

Therefore, our solution is given by $T[A, D] = \mathbf{94}$.

2. What order did you have to multiply the matrices to get the lowest possible of computations?

We note that the least number of computations needed to evaluate $ABCD$ is given by 94. Like with the other examples, we can backtrack to figure out the order.

- In order to get $T[A, D] = 94$, we had to compute $(ABC)D$. So, we need to go back to $T[A, C]$, which is 64.
- Looking at $T[A, C] = 64$, we see that the we had to compute $(AB)C$. So, we need to go back to $T[A, B]$, which is 40.
- Looking at $T[A, B]$, there's only one way to multiply AB. So, we're done.

From this, it follows that the order in which we multiplied these matrices is

$$ABCD \implies (ABC)D \implies \boxed{((AB)C)D}$$

Remark: If we have a tie somewhere, this implies that there is more than one optimal solution. So, you can return one of the optimal solutions.

7.3.5 Runtime

The runtime of this algorithm is given by:

- Number of Subproblems: We needed to index through all possible (i, j) in the range $1 \leq i \leq j \leq m$. So, the total number of subproblems is at most m^2.
- Time per Subproblem: We need to check each $i \leq k < j$ as one of your terms for the maximum. Each check takes constant time (two table lookups and some multiplication and additions). We need to check m different values of k.

Thus, we have

$$\mathcal{O}(m^2 m) = \mathcal{O}(m^3)$$

7.3.6 Dynamic Programming Setup

Sometimes, there are many ways to create a dynamic programming algorithm for a given problem. How we set it up will have a large effect on the runtime.

7.4 All Pairs Shortest Paths

Problem Statement: Given a directed graph G with possibly negative edge weights, compute the length of the shortest path between every pair of vertices[12].

7.4.1 Naive Algorithm

The easy algorithm is to run Bellman Ford with source s for each vertex s. The runtime is given by $\mathcal{O}(|V|^2|E|)$.

7.4.2 Dynamic Program

Let $d_k(u, v)$ be the length of the shortest path from u to v that uses at most k edges.

[12]Bellman-Ford computes the single-source shortest paths. Namely, for some fixed vertex s, it computes all of the shortest path lengths $d(s, v)$ for every v. In *this* problem, we want to compute the shortest path from every vertex to every other vertex.

So, we can consider the last edge of that path; in our case, w to v. Then, we have the path with length $k-1$ from u to w followed by the edge w to v. So:

$$d_k(u,v) = \min_{w \in V}(d_{k-1}(u,w) + \ell(w,v))$$

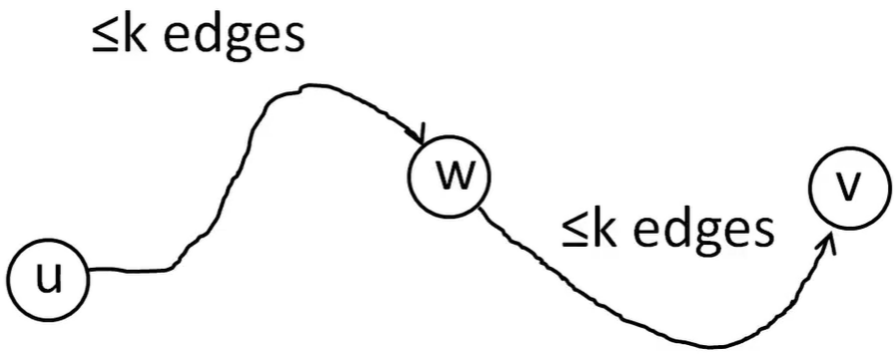
7.4.3 Matrix Multiplication Method

We know that Bellman Ford is slow in part because we can only increase k by one step at a time. This happens because we cut off only the last edge of the optimal path. However, for computation distances from every vertex to every other vertex, there's a better way to do this. What if, instead, we cut this path in the middle in the middle instead of just one edge off of the end like we do with Bellman Ford?

7.4.4 Recursion

Suppose we have a path u to v with at most $2k$ edges:

Then, we can always pick a vertex w so that there are at most k edges on either side of it, like so:



If we picked a particular vertex w, then the best length we will get is

$$d_{2k}(u,v) = \min_{w \in V}(d_k(u,w) + d_k(w,v))$$

It's not hard to see that $d_{2k}(u,v)$ is just the minimum over all vertices w that go in the middle. This gives us a somewhat different recurrence relation, and a totally different dynamic program for computing all pairs shortest path.

7.4.5 Algorithm

How does the algorithm work?

- Base Case: The base case is given by the path length if we only have 1 edge.

$$d_1(u,v) = \begin{cases} 0 & \text{if } u = v \\ \ell(u,v) & \text{if } (u,v) \in E \\ \infty & \text{otherwise (no edge)} \end{cases}$$

- Recursion: Given $d_k(u, v)$ for all u, v, we can compute $d_{2k}(u, v)$ using

$$d_{2k}(u, v) = \min_{w \in V}(d_k(u, w) + d_k(w, v))$$

- End Condition: We can compute $d_1, d_2, d_4, d_8, d_{16}, \ldots, d_m$ until $m > |V|$.

The base case takes $\mathcal{O}(|V|^2)$ time to fill the initial table. The recursion takes $\mathcal{O}(|V|^3)$; this is because we need to consider every pair, of which there are $|V|^2$ possibilities, and then taking the minimum of all third vertices. Finally, the end condition takes $\mathcal{O}(\log(|V|))$ time. Thus, the runtime is given by

$$\mathcal{O}(|V|^3 \log(|V|))$$

7.4.6 Floyd-Warshall Algorithm

We can look at this problem in a different way.

- First, we *label* the vertices $v_1, v_2, \ldots, v_n$, where n is the total number of vertices.
- Next, we define $d_k(u, v)$ be the length of the shortest path from u to w using only $v_1, v_2, \ldots, v_k$ as intermediate vertices. So, we are restricting what vertices we can use to get from one vertex to the other.

Thus, the algorithm is given by:

- Base Case: The base case is given by the fact that you aren't allowed to use any intermediate vertices. So:

$$d_0(u, v) = \begin{cases} 0 & \text{if } u = v \\ \ell(u, v) & \text{if } (u, v) \in E \\ \infty & \text{otherwise (no edge)} \end{cases}$$

- Recursion: Suppose we want to get from u to w using only vertices $v_1, \ldots, v_k$. We will consider several different cases depending on whether or not the shortest path uses v_k.
  - If the shortest path does not use v_k, then it only uses the intermediate vertices $v_1, v_2, \ldots, v_{k-1}$. Thus, the shortest path is given by $d_{k-1}(u, w)$.
  - If the shortest path does use v_k, then we can break this path down like so:

It has length

$$d_{k-1}(u, v_k) + d_{k-1}(v_k, w)$$

We can start at u, move through some intermediate vertices to get to v_k, and then move through more intermediate vertices to get to w. In either cases, the intermediate vertices can be v_1 to v_{k-1}.

In other words, for each u and w, we want to compute

$$d_k(u, v) = \min(d_{k-1}(u, w), d_{k-1}(u, v_k) + d_{k-1}(v_k, w))$$

- End Condition: The end condition is given by

$$d(u, w) = d_n(u, w)$$

where $n = |V|$.

The runtime is given by

- The base case takes $\mathcal{O}(|V|^2)$ time to fill up the table.
- The recursion takes $\mathcal{O}(|V|^2)$ time because there are $|V|^2$ pairs to consider.
- The end condition takes $\mathcal{O}(|V|)$ time.

The final runtime is then

$$\mathcal{O}(|V|^3)$$

7.4.7 Best Known Algorithm

The best known algorithm doesn't actually use dynamic programming.

1. Run Bellman-Ford *once* to compute $d(v)$.
2. Then, we reweigh the edges. We replace the length ℓ with ℓ'; ℓ' is given by (for every edge from u to w)

$$\ell'(u, w) = \ell(u, w) = d(u) - d(w) \geq 0$$

There are two things to note:

- $\ell'(u, w)$ will always be non-negative because $d(w) \leq d(u) + \ell(u, w)$.
- Computing the shortest path with $\ell'(u, w)$ is equivalent to computing the shortest path with $\ell(u, w)$.

3. Run Dijkstra's algorithm from every source.

The final runtime is given by

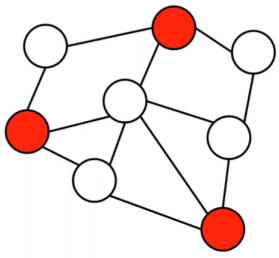
$$\mathcal{O}(|V||E| + |V|^2 \log(|V|))$$

7.5 Problem: Maximum Independent Set of Trees

Definition 7.1: Independent Set

In an undirected graph G, an **independent set** is a subset of the vertices of G, no two of which are connected by an edge.

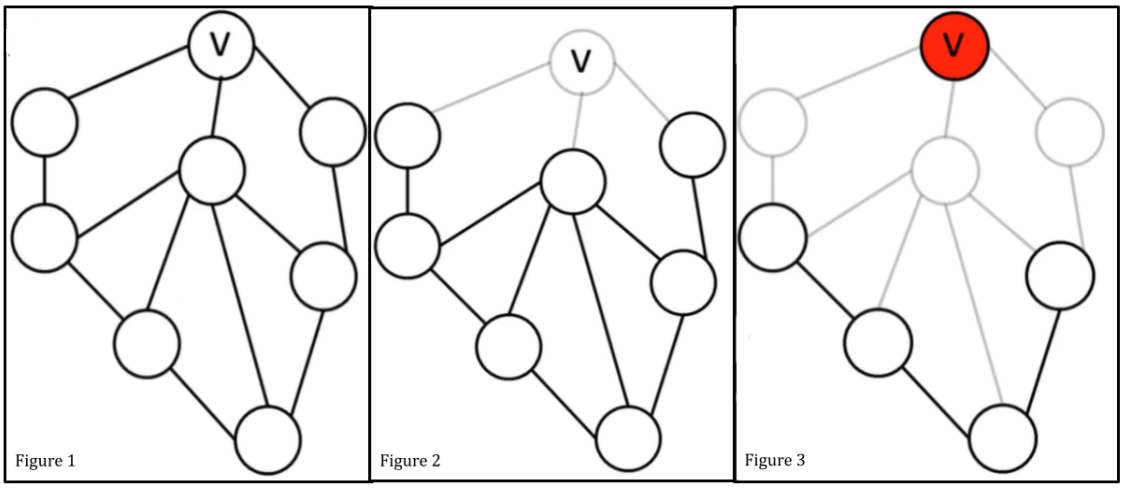
For example, the following is an independent set:



Problem Statement: Given an undirected graph G, compute the largest possible **size** of an independent set of G.

Call the result of the largest possible independent set $I(G)$; in the example above, $I(G) = 3$.

7.5.1 Simple Recursion



There is a reasonably simple recursion that can be used here. A simple question we can ask is: *is vertex v in the independent set?* We'll consider Figure 1 above as a baseline example.

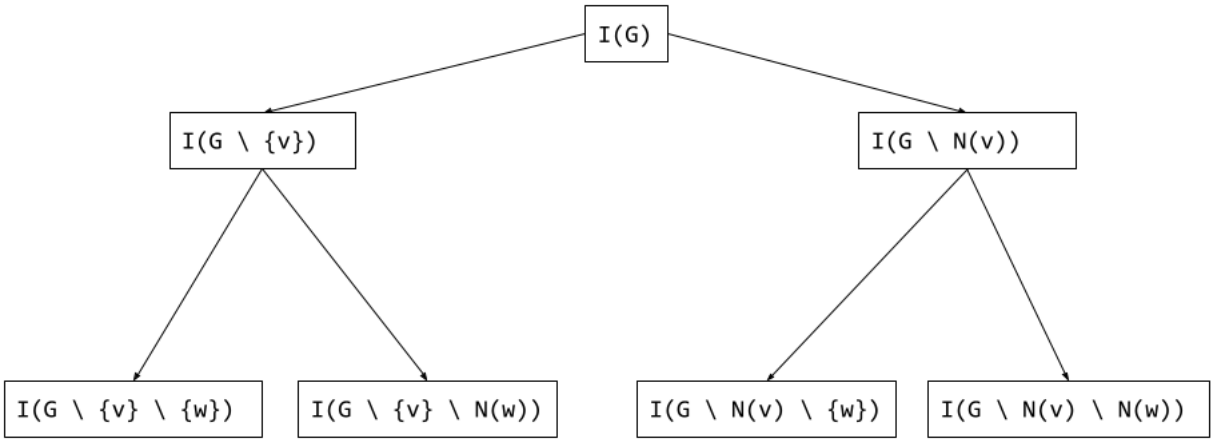
- If v is not in the independent set, then the maximum independent set is an independent set of $G \setminus \{v\}$. In other words, we have
$$I(G) = I(G \setminus \{v\})$$
For example, Figure 2 shows what this could look like.
- If so, then the maximum independent set is v plus an independent set of $G \setminus N(v)$, where $N(v)$ denote the set of vertices which are neighbors of v. In other words, we have
$$I(G) = 1 + I(G \setminus N(v))$$
For example, Figure 3 shows what this could look like.

Therefore, our recursion can be defined by

$$I(G) = \max(I(G \setminus \{v\}), 1 + I(G \setminus N(v)))$$

7.5.2 Subproblems

We now consider the subproblems generated by this recursion.



Here, it's not hard to see that, if we continued generating this recursion tree, we see very little subproblem reuse. In other words, we end up with *different subproblems* every time we get a new recursion. This is an issue for dynamic programming, as dynamic programming relies on subproblem reusability for efficiency.

In particular, for every subgraph G', we need subproblems $I(G')$. But, we'll end up with $2^{|V|}$ subproblems; each vertex can either be in G' or it can not be in G'. This further implies that the runtime of this algorithm will be exponential.

7.5.3 Hardness

Maximum Independent Set of any general graph is what is known as an **NP-Hard** problem. Basically, this means that people believe that there may well be no *efficient* algorithm for it (there is no sub-exponential runtime algorithm that solves this problem).

Instead, we now consider *special cases* of this problem where we can do better.

7.5.4 Independent Sets and Components

Lemma 7.1

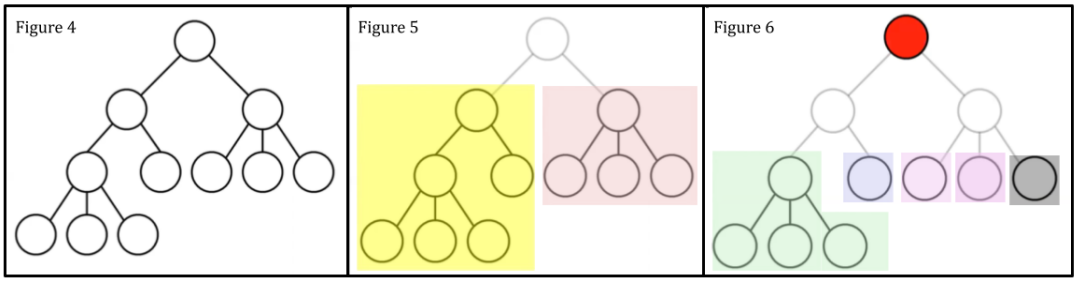
If G has connected components $C_1, C_2, \ldots, C_k$, then

$$I(G) = I(C_1) + I(C_2) + \cdots + I(C_k)$$

Proof. Since the components don't connect to each other, an independent set for G is exactly the union of an independent set for each of the C_i's. THen, we can pick the biggest set for each C_i. □

7.5.5 Independent Sets of Trees

One special case we can consider is independent sets of *trees*. The reason why we look at trees is because, when we remove vertices from trees, it splits the tree up into disconnected subgraphs. This is useful as we can consider the disconnected subgraphs separately.



Consider the above tree (Figure 4), and suppose we're focusing on the root of this tree r.

- If we do *not* include r in our maximum independent set, then we want to look at the maximum independent sets of $G \setminus \{r\}$. For this, see Figure 5.
- If we *do* include r in our maximum independent set, then we get r for free, remove all of the neighbors of r, and then we can consider the remaining subgraphs. For this, see Figure 6.

We note that the subproblems here are actually all *sub-trees* of the original tree. If we look at a subtree of a subtree, we note that this is simply just another subtree. Then, it follows that the set of all problems that we need to solve is just the set of all possible subtrees of the original tree.

7.5.6 Recursion of the Tree

We now consider several cases for our tree.

- Suppose the root is not used in the tree. Then, we end up with the subproblem of having to find the sum of the maximum independent sets of all of the children subtrees. That is,

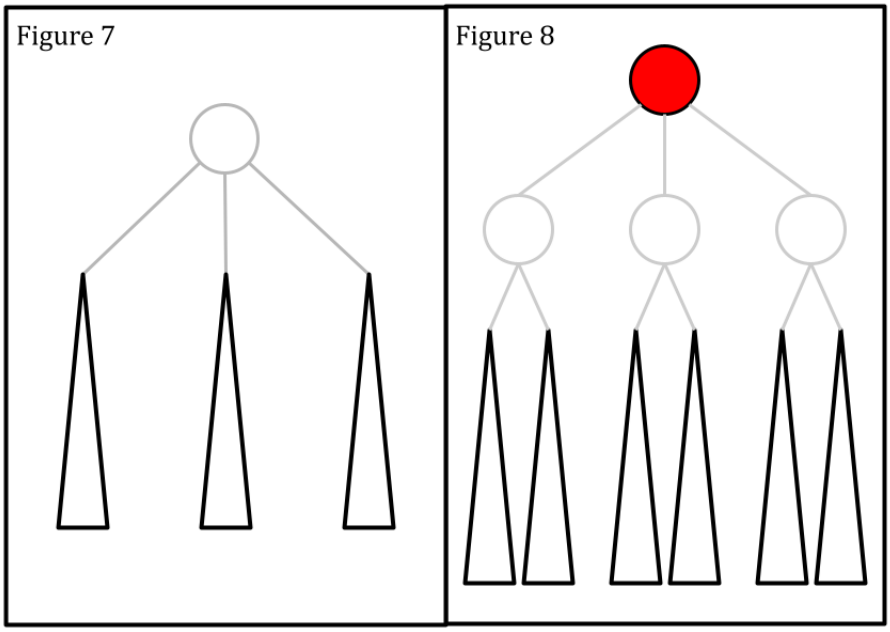
  $$I(G) = \sum I(\text{Children's Subtrees})$$

  This is shown by Figure 7 below.

- Suppose the root is used. Then, we end up with the subproblem of having to find the sum of all of the maximum independent sets of all of the grandchildren subtrees. That is,

  $$I(G) = 1 + \sum I(\text{Grandchildren's Subtrees})$$

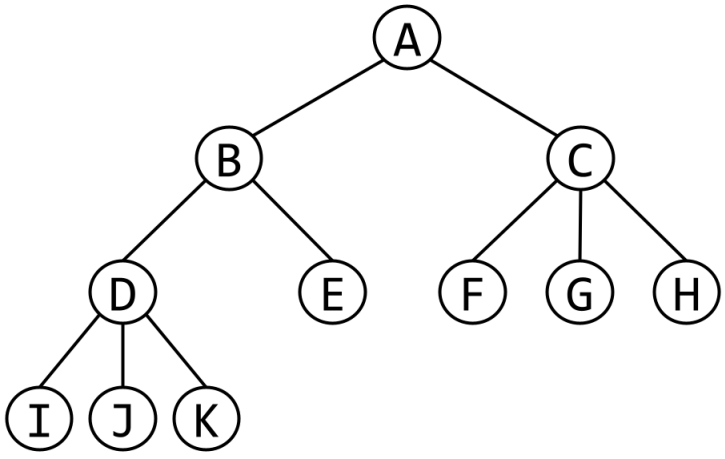
  This is shown by Figure 8 below.



Note: Triangles are arbitrary subtrees.

7.5.7 Example: Computing Maximum Independent Set of Trees

Consider the following tree:



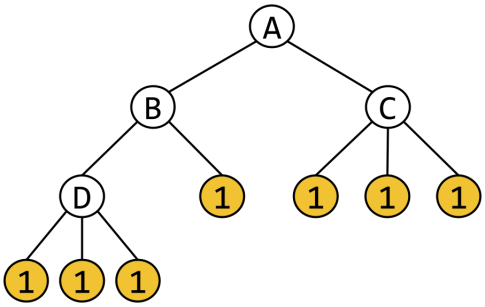
Compute the maximum independent set.

Remember, our subproblems are the subtrees. So, for each node, we associate that node to the subproblem that corresponds to what is the maximum independent set of that node's subtree. The formula that we have is that this will always be the maximum of either:

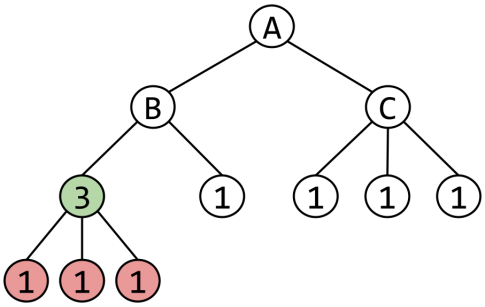
- The sum of the children's answers.
- One more than the sum of the grandchildren's answers.

So, we start at the leaves and make our way up.

- At the bottom level, each leaf (I, J, K, E, F, G, H) is going to have an answer of 1. If we just have that single node, the biggest independent set is going to be of size 1.

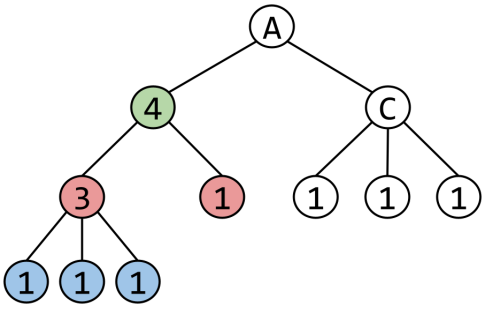


- Consider node D. We can either take the sum of the children $(1+1+1=3)$ or we can take one plus the sum of the grandchildren $(1+0=1)$. Clearly, the sum of the children is the best option, so D will be 3.



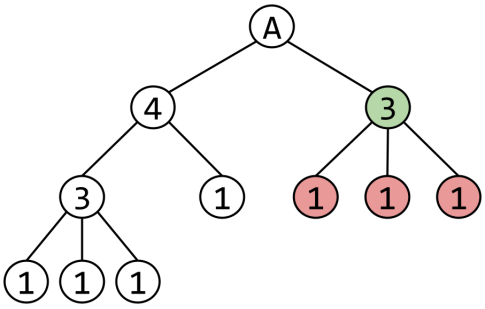
Note that the children nodes are *red* while node D is *green*.

- Consider node B. We can either take the sum of the children $(3+1=4)$ or we can take one plus the sum of the grandchildren $(1+(1+1+1)=4)$. Both options are good, so B will be 4.



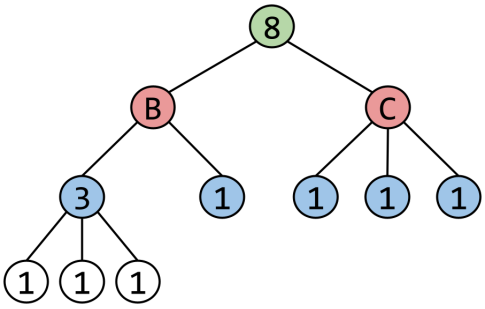
Note that the children nodes are *red*, the grandchildren nodes are *blue*, and node B is *green*.

- Consider node C. We can either take the sum of the children $(1+1+1=3)$ or we can take one plus the sum of the grandchildren $(1+0=1)$. Clearly, the sum of the children is the best option, so C will be 3.



Note that the children nodes are *red* and node C is *green*. Additionally, note that this is actually a subproblem that we've seen before.

- Consider node A. We can either take the sum of the children $(4+3=7)$ or we can take one plus the sum of the grandchildren $(1+(3+1+1+1+1)=8)$. Clearly, one plus the sum of the grandchildren is the best option, so A will be 8.



Note that the children nodes are *red*, the grandchildren nodes are *blue*, and node A is *green*.

Therefore, the maximum independent set of the *whole tree* is $\boxed{8}$.

7.6 Problem: Traveling Salesman Problem

In your job as a door-to-door vacuum salesperson, you need to plan a route that takes you through n different cities. In order to space things out, you do not want to get back to the start until you have visited all cities. You also want to do so with as little travel as possible.

Problem Statement: Given a weighted (undirected) graph G with n vertices, find a cycle that visits each vertex exactly once whose total weight is as small as possible.

7.6.1 Naive Algorithm

The naive algorithm is to try all possible paths and see which is the cheapest. There are $n!$ paths; in particular, if we assume that we have a complete graph where all edges are there, then:

- There are n possible options for the first city.
- There are $n-1$ possible options for the second city.

- There are $n-2$ possible options for the third city.
- ...

This gives us $n(n-1)(n-2)\dots(2)(1) = n!$. Therefore, the runtime of the naive algorithm is approximately $\mathcal{O}(n!)$.

7.6.2 Problem Difficulty

Like Maximum Independent Set, the Traveling Salesman Problem is widely considered to be a difficult problem. It is widely believed that there is no polynomial time algorithm for it. That being said, we note that there is an algorithm that beats the $n!$ naive algorithm.

7.6.3 Setup

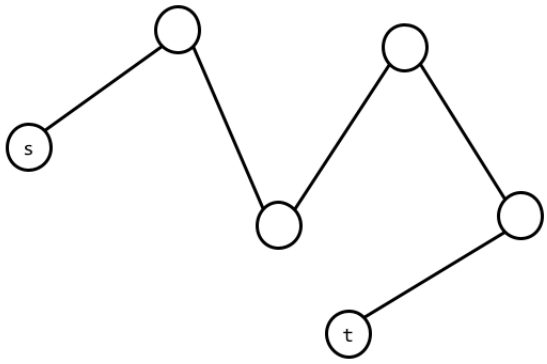
We need to find some notion of *partial solutions* for subproblems. In our case of cycles, we want to look for a path from vertex s to t such that, by adding one more vertex, we get a cycle. Therefore, the answer is given by

$$\text{Best}_{st}(G) = \ell(s,t)$$

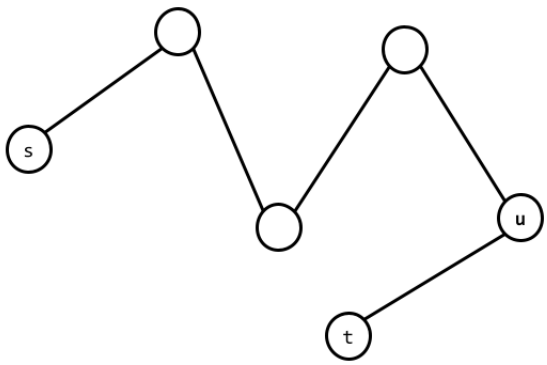
where $\text{Best}_{st}(G)$ is the best value of a path starting at vertex s and ending at vertex t that visits each vertex exactly once.

7.6.4 Recursion

Suppose we have this path from s to t.



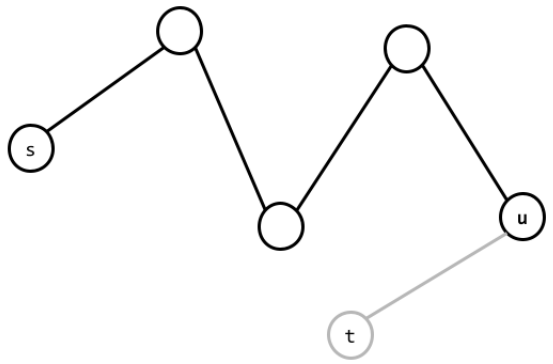
We need to find some recursion that relates the answer to this problem to some kind of subproblem. The standard way to do this is to ask ourselves: what happens if we undo the last step? In our case, our last step is some edge from u to t. That is:



The weight is, then, given by

$$\ell(u,t) + \text{Length of Rest of Path}$$

We want the best path from s to u that uses every vertex *except for* t.



Let's call this subproblem $\text{Best}_{su,-t}$; this is the minimum weight of the path from s to u that visits every vertex except for t exactly once. Thus, the formula is given by

$$\text{Best}_{st}(G) = \min_u \left(\text{Best}_{su,-t}(G) + \ell(u,t)\right)$$

For each t, we figure out what the best path is, add the length of the edge from u to t, and then we take the minimum over all the possible vertices u that have an edge to t. This will give us the best possible path length from s to t.

7.6.5 Recursion II

Now that we have a recursion that gives us the final answers in terms of subproblems, but we need a recursion that solves the subproblems. In other words, how do we solve for $\text{Best}_{su,-t}(G)$?

Again, we look at the edge from v to u. We need the best path that connects s to v that uses all vertices except for t and u. However, this makes the subproblems more complicated; all we're doing is avoiding more vertices.

Instead, we define $\text{Best}_{st,L}$ to be the best path length from s to t that uses exactly the vertices in L. That is, every vertex in L shows up in the path exactly once. The rest of the vertices not in L do not show up in the path.

The last edge is some $(v,t) \in E$ for some $v \in L$. Then, the cost of the path is given by

$$\text{Best}_{sv,L\setminus\{t\}} + \ell(v,t)$$

So, the full recursion is given by

$$\text{Best}_{st,L}(G) = \min_v \left(\text{Best}_{sv,L\setminus\{t\}}(G) + \ell(v,t)\right)$$

7.6.6 Runtime Analysis

Note that L can be any subset of the vertices; thus, there are 2^n possibilities. Additionally, s and t can both be any vertices; thus, there are n^2 possibilities. Thus, there are $n^2 2^n$ subproblems.

We also need to check this for every v (every vertex). This is because, for each subproblem, we need to find the minimum out of all vertices v. Thus, each subproblem takes $\mathcal{O}(n)$ time.

The final runtime is then given by $\mathcal{O}(n^3 2^n)$.

8 NP-Completeness

For almost every problem that we've seen in this class, there's a (usually bad) naive algorithm that just considers every possible answer and returns the best one. Some examples are:

- What is the shortest path from s to t in G?
- What is the longest common subsequence?
- What is the cloest pair of points?
- Does G have a topological ordering?

Of course, for nearly every algorithm we've discussed in class thus far, we were able to come up with a clever solution that runs in polynomial time. However, it's not generally the case that such (clever) algorithms will exist.

Note that, in this class, we define an *easy* problem to be one that can be solved in polynomial time and a *hard* problem to be one that cannot be solved in polynomial time.

8.1 Nondeterministic Polynomial (NP)

Such problems are said to be in **Nondeterministic Polynomial** time (NP). To think of this in a theoretical perspective, imagine some nondeterministic computer that can try every possibility in parallel and then return the best possible solution. This type of computer can solve these problems in polynomial time. For example, it is allowed to search for the shortest path by trying every single path in *parallel* and seeing which one is the shortest. There are two types of NP problems.

- **NP-Decision** problems ask if there is some object that satisfies a polynomial time-checkable property. For example, is there a path from s to t? If you write down a path from s to t, then it is easy for the computer to check, in polynomial time, whether this is actually a valid path from s to t. So, effectively, a decision problem is one whose answer is either yes or no.
- **NP-Optimization** problems ask for the object that maximizes (or minimizes) some polynomial time-computable objective. For example, instead of asking if there is a path from s to t, you might instead ask what the shortest path from s to t is. So, effectively, an optimization problem is one whose answer is a minimum or maximum value.

Essentially, NP is the set of problems for which you can *verify* the answer in polynomial time. Alternatively, it's the set of problems that can be solved in polynomial time by a *nondeterministic* computer.

8.2 Difference Between Decision & Optimization Problems

Now, note that NP-Decision and NP-Optimization problems are not too different.

- Every decision problem can be phrased as an optimization problem.
- Every optimization problem has a decision form.

For theoretical reasons, it's a good idea to understand the distinction. But, for practical purposes, it doesn't really matter.

8.3 Examples of NP Problems

Some examples of NP problems are:

- Formula-SAT (NP-Decision).
- Traveling Salesman Problem (NP-Optimization).

- Hamiltonian Cycles (NP-Decision).
- Generalized Knapsack (NP-Optimization).
- Maximum Independent Set (NP-Optimization).

8.3.1 Formula-SAT

Given a logical formula in a number of Boolean variables, is there an assignment to the variables that causes the formula to be true?

For example, consider the following formulas:

- Formula 1: If $x = \text{True}$, $y = \text{True}$, and $z = \text{False}$, then the following formula is satisfied:

$$(x \vee y) \wedge (y \vee z) \wedge (z \vee x) \wedge (\overline{x} \vee \overline{y} \vee \overline{z})$$

- Formula 2: No assignments of x, y, and z can satisfy the following formula:

$$(x \vee y) \wedge (y \vee z) \wedge (z \vee x) \wedge (\overline{x} \vee \overline{y}) \wedge (\overline{y} \vee \overline{z}) \wedge (\overline{x} \vee \overline{z})$$

There are some applications of SAT. In particular:

- Circuit Design: Given some circuit, you want to make sure it actually computes the function that it's supposed to compute. This is basically a satisfiability problem.
- Logic Puzzle: Most logic puzzle usually come down to some satisfiability problem; you want to find some settings or variables that satisfy some set of rules.

8.3.2 Hamiltonian Cycles

Given an undirected graph G, is there a cycle that can visit every vertex exactly once?

8.3.3 General Knapsack

Recall that knapsack has a number of items, each with a weight and a value. The goal is to find the set of items whose total value is as much as possible without the total weight exceeding some capacity.

The general knapsack problem essentially runs in polynomial time in the *weights*. Note that if the weights are allowed to be large (for example, written in binary), then you don't have a good algorithm.

8.4 Brute-Force Search

Every NP problem has a brute-force search algorithm (the naive algorithm). In this class, we have looked at problems with algorithms that substantially improve on these brute-force algorithms. The question, then, is: *does every NP problem have a better-than-brute-force algorithm?* Put it another way, *is it the case that every algorithm in NP has a polynomial time algorithm?*

- If this is the case, then every NP problem has a reasonably efficient solution. No matter what kind of complicated search or decision problem we want to solve, there is always a clever way to do it better than brute-force.
- If this is *not* the case, then some NP problems are fundamentally difficult. There are some problems which are easy to check but there is no straight-forward or efficient way to actually find the answer.

Note that P is the set of all problems that can be solved in polynomial time.

8.5 Reductions

In practice, at least some problems in NP appear to be hard. Despite decades of trying, people still don't know particularly good problems. Suppose you have a problem. How do you know if it's hard or not?

- You can search for an efficient algorithm to show that the problem is easy.
- You can try to prove that the problem is hard, but this is difficult.
- You can try to relate the difficulty of your problem to the difficult of other problems.

With the last bullet point, we'll introduce the notion of *reductions*.

8.5.1 What Is It?

Reductions are a method for proving that one problem is *at least as hard* as another problem. You can think of this as proving inequalities about problems. For example, problem A is as hard as problem B.

The way we do this is as follows: we can show that *if* there is an algorithm for solving A, then we can use this algorithm to solve B. Therefore, B is no harder than A.

To better understand this concept, consider the following statements:

- If problem X is no harder than problem Y, and if Y is easy, then X must also be easy.

  To get the intuition behind this, suppose you have two numbers x and y such that x is no greater than y; that is, $x \leq y$. Suppose y is negative. Then, x must also be negative (it cannot be positive).

  The same idea holds here. If we have two problems X and Y such that X is no harder than (or at least as easy as) Y and Y is easy, then X cannot be harder than Y; X has to be *as easy as, or easier than,* Y.

- If problem X is no harder than problem Y, and if X is hard, then Y must also be hard.

  Again, to get the intuition behind this, suppose you have two numbers x and y such that x is no greater than y; that is, $x \leq y$. Suppose x is positive. Then, y must also be positive (it cannot be negative).

  The same idea holds here. If we have two problems X and Y such that X is no harder than (or at least as easy as) Y and X is hard, then Y cannot be easier than X; Y has to be *as hard as, or harder than,* X.

8.5.2 Reducing Hamiltonian Cycle to Traveling Salesman Problem

We note that there is a natural reduction here. Particularly, they're related in the sense that we're visiting each vertex exactly once.

- Recall that the Traveling Salesman Problem asked: *Given a weighted (undirected) graph G with n vertices, find a cycle that visits each vertex exactly once whose total weight is as small as possible.*
- Recall that the Hamiltonian Path problem asked: *Given an undirected graph G, is there a cycle that can visit every vertex exactly once?*

To see this in action, consider the following example:

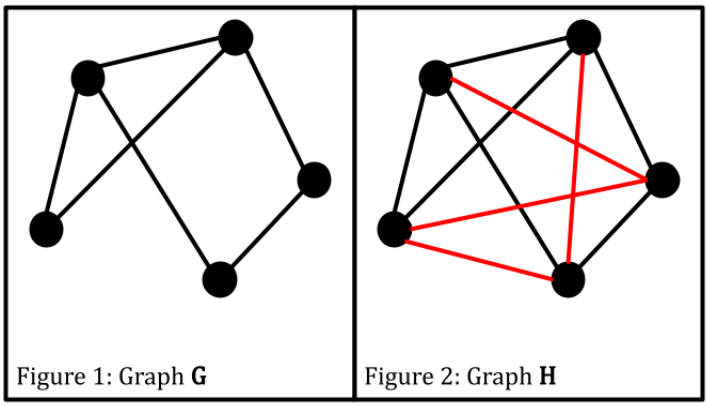


Figure 1: Graph **G** Figure 2: Graph **H**

Suppose we have an algorithm that solves the Traveling Salesman Problem efficiently. The idea is to convert a Hamiltonian Cycle problem to a Traveling Salesman problem. We do this by taking the original graph G from the Hamiltonian Cycle problem and "copying" it to the Traveling Salesman problem, which we'll call the copied graph H. For all edges in G, we can give the corresponding edges in H a weight of 1 (in Figure 1 and 2, the *black* edges). Then, any extra edges that we add to H (in Figure 2, the *red* edges) can be given a very expensive weight (say 1000).

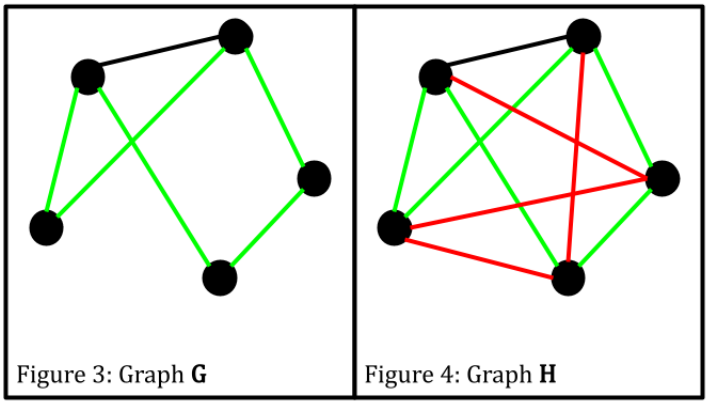


Figure 3: Graph **G** Figure 4: Graph **H**

Looking at Figure 4, we see that there is a Traveling Salesman path of cost 5. But, the path taken in the Traveling Salesman problem is the same path found in the Hamiltonian Cycle problem. So, the cheapest path we can hope for in the Traveling Salesman problem is the path consisting only of cost 1 edges. So, if we can find a path in the Traveling Salesman problem consisting only of cost 1 edges, then we have a Hamiltonian Cycle. *Otherwise*, we'll have to make use of some expensive edges, so that implies that no such Hamiltonian Cycle can exist.

To be more formal about it, suppose you have a Hamiltonian Cycle instance G with n vertices. Suppose that you have an efficient algorithm A that solves the Traveling Salesman Problem[13] Now, we create a Traveling Salesman problem instance H where:

- The edges in G have cost 1.
- The edges not in G have cost 1000 (or some expensive weight).

Then, using this Traveling Salesman Problem algorithm, we can solve the Traveling Salesman instance. We note that we have a path of cost n in H if and only if there is a Hamiltonian cycle in G. Therefore, we can use this answer to solve the initial problem.

[13] We don't need to know how this algorithm works, only that you *have* one.

- If there is a Hamiltonian Cycle in G, then the Traveling Salesman Problem will have a weight of n. If there is *no* Hamiltonian Cycle, then the best cost will be greater than n.
- Any Traveling Salesman Problem that uses these expensive-costing edges will be more expensive than if we decided to use the edges with cost 1.
- Therefore, we can run $A(H)$. If $A(H) \leq n$, then G has a Hamiltonian cycle. Otherwise, it does not.

From this, we get a new algorithm that can solve the Hamiltonian Cycle. This gives us a relation: the Hamiltonian Cycle problem is *no harder* than the Traveling Salesman Problem. Now, since Hamiltonian Cycle is in NP, then it follows that the Traveling Salesman Problem must also be in NP.

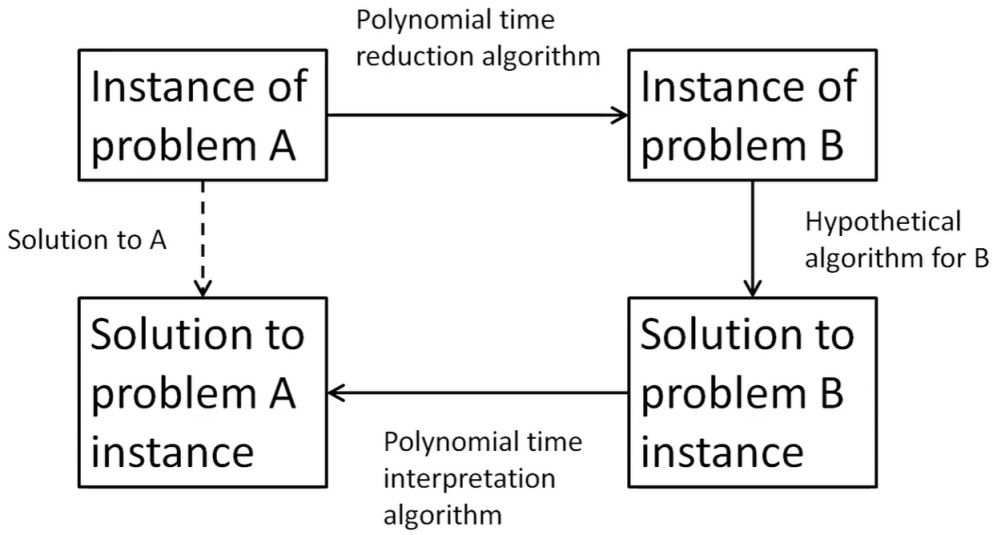
So, what we've shown is that if we have an algorithm that can solve the Traveling Salesman problem, then we can turn that into an algorithm that solves the Hamiltonian Cycle path.

8.5.3 Generalization

If we want to find a reduction from problem A to problem B, then what we want to show is:

- If we are given an algorithm to solve that solves problem B, we can turn that into an algorithm that solves problem A.
- Generally, we start with an instance of problem A. Then, with some polynomial time reduction algorithm, we can turn this instance of problem A into an instance of problem B. Then, with a *hypothetical* algorithm for problem B, we can find a solution to the problem B instance. Then, we need to turn this solution of the problem B instance into a solution for the problem A instance; this is done by using a polynomial time interpretation algorithm.

This process looks something like:



It's important that the reduction and interpretation algorithms run in *polynomial time* by construction. If you have a polynomial time algorithm for problem B, then this entire process runs in polynomial time (i.e. you have a polynomial time algorithm for problem A). If you don't have a polynomial time algorithm for problem B, then there might be some other way to solve problem A, but you can't use this particular method.

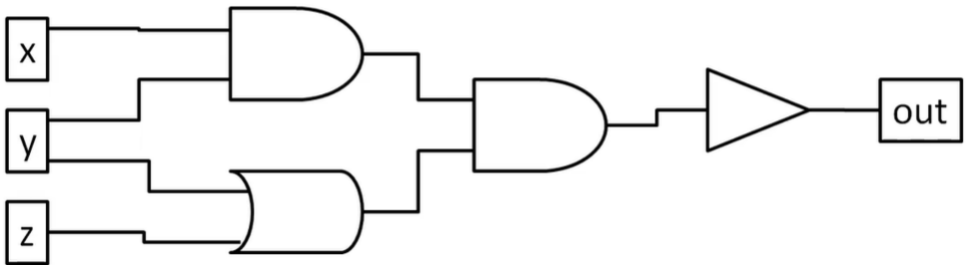
So, effectively, if we have algorithms for reduction and interpretation, then:

- Given an algorithm to solve problem B, we can turn it into an algorithm to solve problem A.
- This means that problem A might be *easier* to solve than problem B, but it cannot be *harder*.

8.6 Problem: Circuit SAT

Problem Statement: Given a circuit C with several Boolean inputs and one Boolean output, determine if there is a set of inputs that give the output `1`.

For example, the following circuit can be satisfied with inputs $x = y = z = 0$.



8.6.1 Important Reduction

Any NP-decision problem can be reduced to Circuit SAT. This is because any NP-decision problem is always of the same form; something is a NP-decision problem if it asks if there is some X that satisfies a polynomial-time checkable property.

Essentially, for some polynomial-time computable function F, is there an X such that $F(X) = 1$? We can create a circuit C that computes F. Then, the problem is equivalent to asking if there is an input for which C outputs 1. So, any NP-decision problem can be encoded as a Circuit SAT problem by taking the function that checks whether or not something satisfies your condition and encoding that as a Boolean circuit.

8.6.2 NP-Complete

Circuit-SAT is our first example of an **NP-Complete** problem. This is a problem in NP that is at *least* as hard as any other problem in NP. Therefore:

- If we can find a polynomial time algorithm that can solve Circuit SAT, we have a polynomial time algorithm for *all* NP problems.
- However, if any problem in NP is hard, then Circuit SAT is *hard*.

Remark: Decision problems can be NP-Complete. For optimization problems, we call them NP-Hard.

8.6.3 Other NP-Complete/Hard Problems

We note that the following problems are all NP-Complete or NP-Hard:

- Formula SAT.
- Maximum Independent Set.
- Traveling Salesman Problem.
- Hamiltonian Cycle.
- Generalized Knapsack.

We can show this by finding reductions from other NP-Hard or NP-Complete problems.

8.6.4 3-SAT

3-SAT is a special case of formula-SAT where the formula is an `AND` of clauses and each clause is an `OR` of at most 3 variables or their negations. The following is an example of a 3-SAT problem:

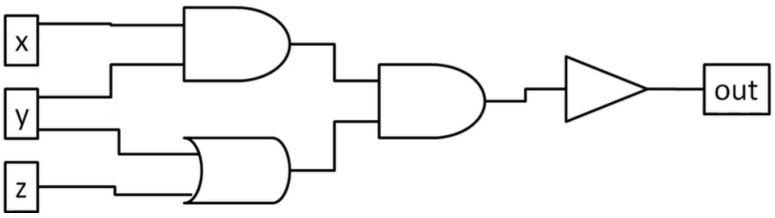
$$(x \vee y \vee z) \wedge (\overline{x} \vee u) \wedge (w \vee \overline{z} \vee u) \wedge (\overline{u} \vee w \vee \overline{z}) \wedge (\overline{y})$$

Here, each clause has *at most three* variables or their negations. Each clause only has `OR` operations, and each clause is separated by `AND` operations.

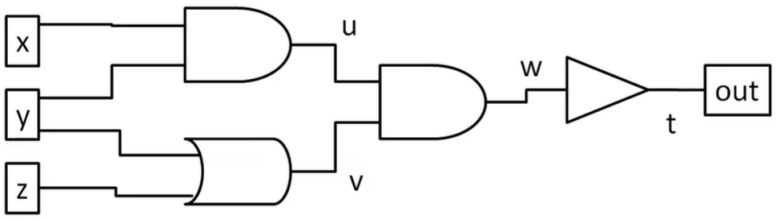
8.6.5 Showing that 3-SAT is NP-Complete

We will show that 3-SAT is NP-Complete. This is done by reducing Circuit-SAT to 3-SAT.

- Let's suppose we start with a circuit from a circuit-SAT problem.



- Next, we'll create new variables for each outgoing wire.



- For each variable, we'll create a formula for each gate and for the output. We know that output t must be true. Then, each clause is defined by its variables and the output after performing some operation (e.g. x and y are the inputs to the `AND` gate and u is the output). Thus, we can write the formula out like so:

$$(v \iff y \vee z) \wedge (u \iff x \wedge y) \wedge (w \iff u \wedge v) \wedge (t \iff \overline{w}) \wedge t$$

- Note that the formula above is *not* a 3-SAT formula. Effectively, we have 3-variable clauses that aren't 3-SAT clauses, so the goal is to write it in terms of them.

> To write thes 3-variable clauses in a way that represents 3-SAT clauses, we write out a truth table for each clause. Note that this is just an example of how you would do this.
>
> For example, consider a clause $(z \iff x \vee y)$. There are 8 different possibilities for all combinations of x, y, and z. For each combination of the original statement, we fill out the table like so.

x	y	z	$z \iff x \vee y$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

We note that when we impose a 3-SAT clause on three variables, it excludes exactly *one* combination of the inputs. We now describe how this works specifically. The idea is, given the original truth table of each clause:

- Find all rows where the result is `0`. In our initial example, we have

x	y	z	$z \iff x \vee y$
0	0	1	0
0	1	0	0
1	0	0	0
1	1	0	0

- Then, for each row, negate the correct inputs so that the result would be `0` in the parameters; then, `OR` each of them. So, for example, with the first row, you would have $x \vee y \vee \overline{z}$. This is because x's original value was `0`, so you would use x in the 3-SAT instance to "keep" the `0`. Likewise, since z's original value is `1`, you would use $\overline{z}$ to negate the `1` to a `0`.
- For each of the remaining rows, do the same thing. Combine the clauses with an `AND` operator. For example, with the second row, we would have $x \vee \overline{y} \vee z$. Then, we can combine this with the previous step by using the `AND` operator like so:

$$(x \vee y \vee \overline{z}) \wedge (x \vee \overline{y} \vee z)$$

  Eventually, you'll end up with the 3-SAT-equivalent of the above expression:

$$\underbrace{(x \vee y \vee \overline{z})}_{\text{1st Row}} \wedge \underbrace{(x \vee \overline{y} \vee z)}_{\text{2nd Row}} \wedge \underbrace{(\overline{x} \vee y \vee z)}_{\text{3rd Row}} \wedge \underbrace{(\overline{x} \vee \overline{y} \vee z)}_{\text{4th Row}} = (z \iff x \vee y)$$

- Repeat this process for each of the clauses that needs to be converted.

- This means that 3-SAT is also NP-Complete since any problem in NP can be reduced to Circuit-SAT, which in turn can be reduced to 3-SAT.

8.6.6 Another Look at 3-SAT

Lemma 8.1

A 3-SAT instance is satisfiable if and only if it is possible to select one term from each clause without selecting both a variable and its negation.

For example, suppose we have the 3-SAT instance

$$(x \vee y \vee z) \wedge (\overline{x} \vee y) \wedge (\overline{y} \vee \overline{z}) \wedge (\overline{x} \vee z)$$

For each clause, suppose we select one of the terms like so.

$$(x \vee \boxed{y} \vee z) \wedge (\boxed{\overline{x}} \vee y) \wedge (\overline{y} \vee \boxed{\overline{z}}) \wedge (\boxed{\overline{x}} \vee z)$$

We note that we did not select both a variable and its negation (e.g. we didn't select an x and $\overline{x}$). This shows that this 3-SAT instance is satisfiable. For this example, if $x =$ False, $y =$ True, and $z =$ False, then this statement will be True.

Proof. Suppose the instance was satisfiable. Then, at least one term in each clause must be true since each clause consists of `OR` of some number of terms. We select one such term from each clause. We note that they cannot contradict each other since all of those terms are consistent with the same satisfying assignment (it can't be true that x was true in one clause and $\overline{x}$ was true in another clause because x had to either be true or false).

Suppose we have a 3-SAT instance where we can select one term from each clause without selecting both a variable and its negation. Suppose we set those clauses to be true (for example, we can set x to be True and y to be False or whatever so each clause is true). Then, we set the other variables arbitrarily since we know that at least one variable in each clause is true. This causes the whole statement to be true. □

8.6.7 Reducing 3-SAT to Maximum Independent Set

This reduction is interesting especially since these problems appear to be unrelated. So, we start with a new formulation of 3-SAT. We want to select one term from each clause. We note that:

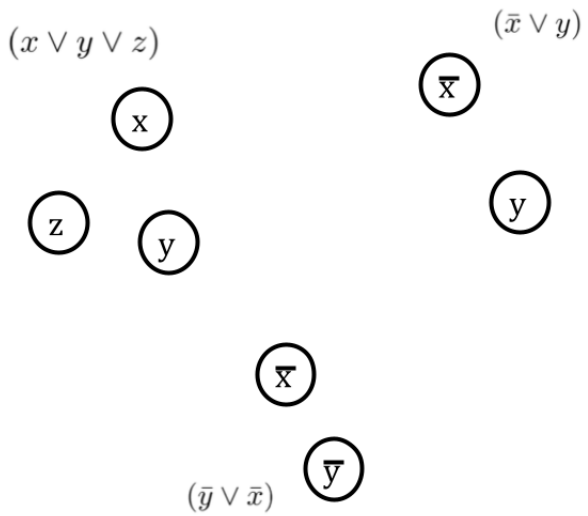
- For 3-SAT, we have a bunch of terms and we want to select some of them.
- For MIS, we have a bunch of vertices and we want to select some of them.

Suppose we want to reduce the following 3-SAT instance to a Maximum Independent Set instance.

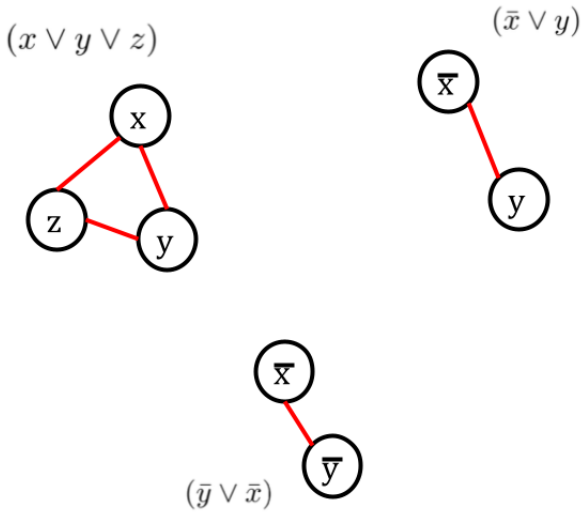
$$(x \vee y \vee z) \wedge (\overline{x} \vee y) \wedge (\overline{y} \vee \overline{x})$$

This is how you would go about it.

- First, for each term in each clause, create a vertex.

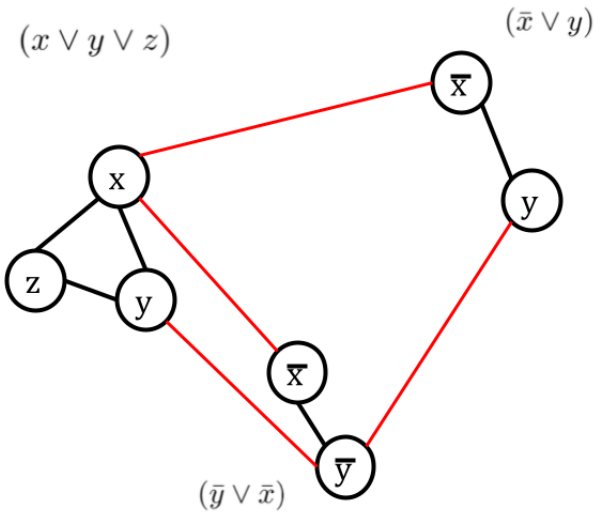


- We now need to add edges in the following way.
  - First, we want to add edges between terms in the same clause.



These edges ensure that, if we pick an independent set, we can only pick one term from each clause.

– We add edges between contradictory terms.



For example, we add an edge between the vertex x and a vertex $\overline{x}$.

This gives us the graph that we can use for the Maximum Independent Set instance.

We note that an independent set in this graph has:

- At most one vertex from each clause.
- No pair of vertices corresponding to contradictory terms.

This is because of the edges; an independent set is not allowed to have two vertices connected by an edge. Since two things in the same clause are connected by an edge, it can't have two things in the same clause. And since two contradictory things are connected by an edge, it cannot have two contradictory things. Other than that, it can have whatever vertices it wants.

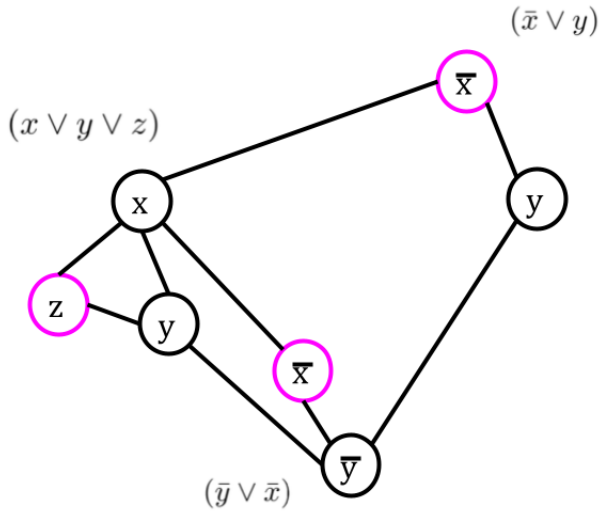
In particular, can we find an independent set such that its size is the same as the number of clauses in the original 3-SAT expression? If we can, then the only way to do that, since we can't select more than one

vertex per clause, is if we pick exactly one vertex per clause. Additionally, because of the second contradict, we had to have done this without any pair that contradicts each other. This is exactly what we need to find a satisfying assignment for the 3-SAT. The opposite direction works as well; if we can satisfy the 3-SAT, we can pick one vertex from each clause and that will give us an independent set of the corresponding graph of the appropriate size.

So, to summarize the previous paragraph, we note that we have an independent set of size c (where c is the number of clauses) if and only if you can select one term from each clause without a contradicton. Therefore, the size of the maximum independent set is equal to the number of clauses if and only if the 3-SAT has a solution.

8.6.8 Example of Reducing 3-SAT to MIS

Consider the graph we created in the previous section. If we select the following vertices (the pink vertices):



Then, as long as $x =$ False, $z =$ True, and y is arbitrary, then we found a satisfying assignment.

8.7 Problem: Zero-One Equations

To prove more complicated reductions, it is helpful to have the correct problem to prove the reductions from. Sometimes, you might need to build intermediate problems that are nice to work with, mainly for convenience.

One convenient problem is the **zero-one equation**. Given a matrix A with only `0`'s and `1`'s as entries and a vector b of only `1`'s, determine whether or not there is an x with `0` and `1` entries so that

$$Ax = b$$

The problem is clearly in NP. We will show that is NP-complete.

8.7.1 Example: Zero-One Equations

Find an x_1, x_2, x_3 such that

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

is satisfied. Equivalently, does there exist $x_1, x_2, x_3 \in \{0, 1\}$ such that

$$\begin{cases} x_1 + x_3 = 1 \\ x_1 + x_2 = 1 \end{cases}$$

holds?

It's not hard to see that this holds.

Generally, this is what a zero-one equation looks like; a bunch of sets of x_i's that need to add to `1`.

8.7.2 Reducing 3-SAT to Zero-One Equation

The idea behind reducing a 3-SAT formula to a zero-one equation is that we'll look at the formulation of 3-SAT and use one term from each clause.

We will create one variable for each term to denote whether or not it has been selected. Then, we'll add equations to enforce exactly one term from each clause such that no contradictory terms are selected.

To see this in action, we consider an example. Consider the following 3-SAT instance

$$(\underbrace{x}_{x_1} \vee \overbrace{y}^{x_2} \vee \underbrace{z}_{x_3}) \wedge (\overbrace{\overline{x}}^{x_4} \vee \underbrace{y}_{x_5}) \wedge (\overbrace{\overline{y}}^{x_6} \vee \underbrace{\overline{x}}_{x_7})$$

- We can translate this into the following zero-one equation instance by noting that we have one term per clause. That is:
$$\begin{cases} x_1 + x_2 + x_3 = 1 \\ x_4 + x_5 = 1 \\ x_6 + x_7 = 1 \end{cases}$$
- We now want to make sure we don't select any contradictory terms. For example, we can't select both x_1 and x_4. So, if we use inequalities, we get something like:
$$\begin{cases} x_1 + x_4 \leq 1 \\ x_1 + x_7 \leq 1 \\ x_2 + x_6 \leq 1 \\ x_5 + x_6 \leq 1 \end{cases}$$
So, if both x_1 and x_4 are `1`, then we have a problem. In any case, if we satisfy this system of inequalities of zero-one variables x_1 through x_7, then this is equivalent to having a solution to the original 3-SAT problem.
- We note that we do not allow inequalities. A zero-one equation is only allowed to have *equalities*. So, we need to tweak this system to make this equalities instead. An easy trick to do just this is to introduce a new variable. That is, if we had the inequality $a + b \leq 1$, then we can replace it with $a + b + c = 1$. The point is, if $a + b \leq 1$, then this will hold if and only if there is a zero-one variable c such that $a + b + c = 1$. To see what we mean, we note that if $a + b = 0$, then we let $c = 1$; likewise, if $a + b = 1$, then $c = 0$. And, as usual, if $a + b = 2$, then we have an issue either way.

  So, we have the system of *equalities*:
$$\begin{cases} x_1 + x_4 + x_8 = 1 \\ x_1 + x_7 + x_9 = 1 \\ x_2 + x_6 + x_{10} = 1 \\ x_5 + x_6 + x_{11} = 1 \end{cases}$$

In general, the construction is as follows:

- Create one variable per term.
- For each clause, create one equation.
- For each pair of contradictory term, create an equation with those two and a new variable, and set that equal to one.

This gives us a system of zero-one equations, which is satisfiable if and only if the original 3-SAT problem is.

8.7.3 Another Way of Looking at Zero-One Equations

Suppose we treat A as a bunch of column vectors. Recall that if $A = \begin{bmatrix} v_1 & v_2 & v_3 & \dots & v_n \end{bmatrix}$, then we have

$$Ax = x_1 v_1 + x_2 v_2 + \cdots + x_n v_n$$

So, let's suppose we have the following matrix declaration

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

and we wanted to solve for $Ax = \vec{1}$. Then, we can write

$$x_1 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} + x_3 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

We can rewrite this like so

$$\begin{aligned} &x_1 \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix} + \\ &x_2 \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix} + \\ &x_3 \begin{bmatrix} 1 & 1 & 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \end{aligned}$$

This is a bit suggestive because the way this was written implies that you could treat this as an addition of numbers. If we did this addition without performing any carries, this would almost be exactly the same computation as if we were adding numbers rather than adding vectors.

Let's pretend we actually did addition of numbers.

```
x1 *    1 0 0 1     +
x2 *    0 0 1 1     +
x3 *    1 1 1 0
---------------------
=       1 1 1 1
```

We have these three numbers and we want to see if there is some x_1, x_2, x_3 such that x_1 times the first number plus x_2 times the second number plus x_3 times the third number is equal to the solution. The only difference between adding numbers and adding vectors is that if we add numbers and we get big-enough terms, we might need to deal with carry-overs. For example, if we were doing base 2 addition, then we will get a carry over if we added 1 with 1. But, if we added these numbers in a high-enough base (e.g. base 10), we can't possibly get big-enough terms that will give us a carry over.

So, if the numbers are represented in a large enough base (e.g. base 1000) that carrying is impossible, we have a solution to the vector equation if and only if we have a solution to the number equation.

8.8 Subset Sum

What we just talked about actually relates naturally to the subset sum problem.

Problem Statement: Given a set S of numbers and a target number C, is there a subset $T \subseteq S$ whose elements sum to C?

Alternatively: Can we find an $x_y \in \{0, 1\}$ such that

$$\sum_{y \in S} x_y y = C$$

where

$$x_y = \begin{cases} 0 & \text{If } y \text{ is not in the subset.} \\ 1 & \text{If } y \text{ is in the subset.} \end{cases}$$

Essentially, we want to find some setting of these binary variables so that the sum of all y in the set of $x_y y$ is equal to the target.

8.8.1 Reducing Zero-One Equations to Subset Sum

The alternative point that we've made can actually be used to relate the problem to zero-one equations. The idea is that we replace all of these vectors with numbers in base n where n is big enough so that there is no carry over with the arithmetic, and since the base is big enough that we'll never have to do any carry over, the vector addition problem is exactly equivalent to the number addition problem here.

Asking whether or not there is some zero-one linear combination of these numbers that add up to another number is equivalent to asking if there is a subset of these numbers which add up to the target.

So, starting from the instance of a zero-one equation, we can reduce the instance of subset sum with the same answer, which gives us the desired reduction.

Therefore, subset sum is NP-complete.

8.9 Generalized (Non-Repeated) Knapsack

Subset sum is pretty closely related to the (non-repeated) knapsack. In particular:

- Subset sum wants to find a set of values so that the weights equal the capacity.
- Knapsack wants to find a set of values so that the weights are at most the capacity and the value is as large as possible.

There's a pretty easy way to get rid of this "value" idea.

8.9.1 Reducing Subset Sum to Knapsack

We can reduce subset sum to knapsack in the following way.

- We create a knapsack problem where, for each item, the value of the item is just the weight of the item. Then, this becomes a "the price is right" knapsack; you want to find some set of items where the total weight is as large as possible without going over the capacity.
- Maximizing the value is the same as maximizing the weight without going over the capacity.
- We can, then, achieve a certain value (which is the capacity) if and only if there is a subset of the items with toatl weight equal to the capacity. So, if we want to answer the question "is it possible to find a set of these items where the total weight is equal to the capacity?" then this is exactly the same as the subset sum problem.

So, this gives us a pretty natural reduction from subset sum to knapsack. Therefore, knapsack is NP-Hard.

8.9.2 Relation to Polynomial-Time Runtime Knapsack

Our dynamic programming algorithm that we discussed some time ago was polynomial time in the *total weight* (in terms of the numerical values of the weights). In this case, this is exponential.

8.10 Hamiltonian Cycles

We want to show that Hamiltonian Cycles (and thus Traveling Salesman Problem) are NP-Complete (and NP-Hard). We will use a reduction from zero-one equations to Hamiltonian Cycles to show this.

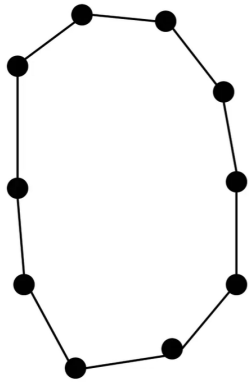
8.10.1 Strategy

Very often, to show that some problem is NP-Complete, we want to be able to show that it can simulate that logic somehow. This is difficuly for Hamiltonian Cycles as most graphs have too many options, so we will want to find specific graphs with clear, binary choices.

This is difficult for Hamiltonian cycle as most graphs have too many options in terms of the edges, so we want to find specific graphs with clear, binary choices.

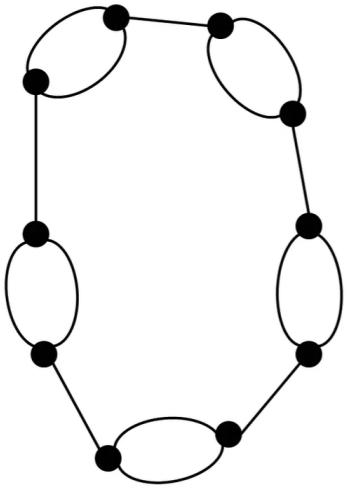
To see how this would work, let's consider an example.

- First, start with a cycle.



  There is clearly a Hamiltonian cycle here.

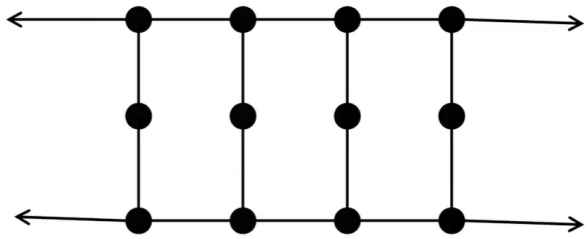
- Then, for some of these edges, we can double them up. That is, rather than



This graph still has a Hamiltonian cycle. But, now, there are multiple ways to find a Hamiltonian cycles (there are many binary choices that can be made here). At the moment, our construction has no constraints on these choices; we can pick whatever combinations of these binary choices we want and still get a Hamiltonian cycle. But, at least, we have a prototype of a graph where trying to find a Hamiltonian cycle on this graph involves making a specific collection of (in this case) binary choices, which is a good place to start if we want to make some NP-Completeness reductions. Now, at least, there is some *logic* involved. We just need to add some limitations as to what choices we can make.

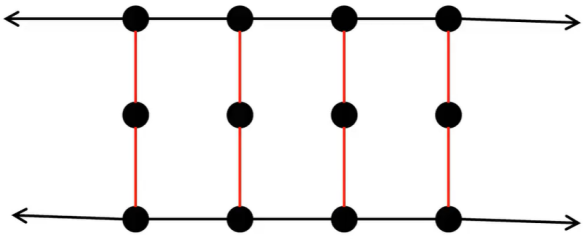
So, to sumamrize, in our cycle, we must pick one edge from each pair, which provides a nice set of binary variables.

- We need to add restrictions so that we can't just use any collection of choices. We need to make it difficult to figure out whether it is possible to make the right choices or not. To do so, we make use of a *gadget*.



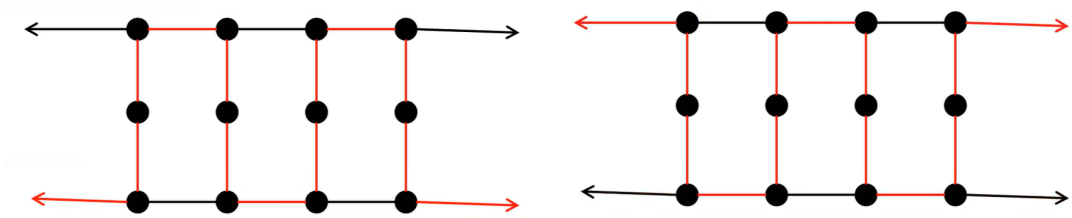
Gadgets are generally useful for proving NP-Completeness/Hardness reduction problems. It's not necessarily well-defined; it's essentially just a piece of construction which only has a few ways it can be filled out. It often allows us to enforce some condition or simulate some more complicated behavior. So, in our case here, this gadget will be a subgraph of the full graph we made. We have a collection of edges and vertices, and there are these four edges with arrows which connects the gadget to the rest of the graph.

If this gadget is part of the graph and there is a Hamiltonian cycle in this graph, then we note that, because of these degree 2 vertices, any Hamiltonian cycle has to visit these vertices. The only way it can do that is if it uses both edges on each of the columns (the red edges below).

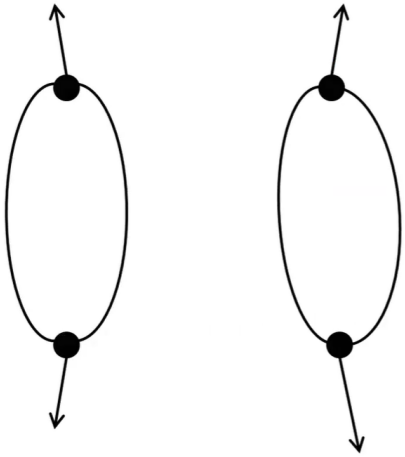


When we get to the top or bottom of one of these columns, the vertex at the top or bottom of the columns has to connect to exactly one of its left or right neighboring vertices. We can't just connect these vertices however we want, or else we might have a disconnected cycle in the middle.

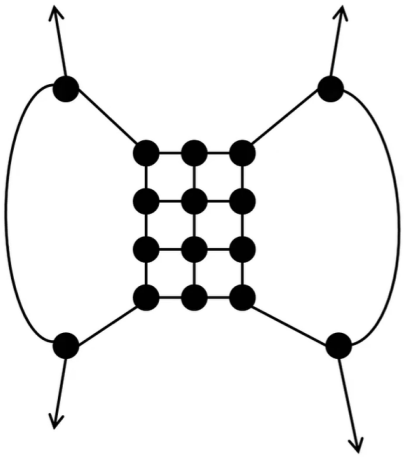
There are only two ways to fill these edges out and still be consistent with having a Hamiltonian cycle; both of those ways involve a zig-zag pattern (see the two graphs below).



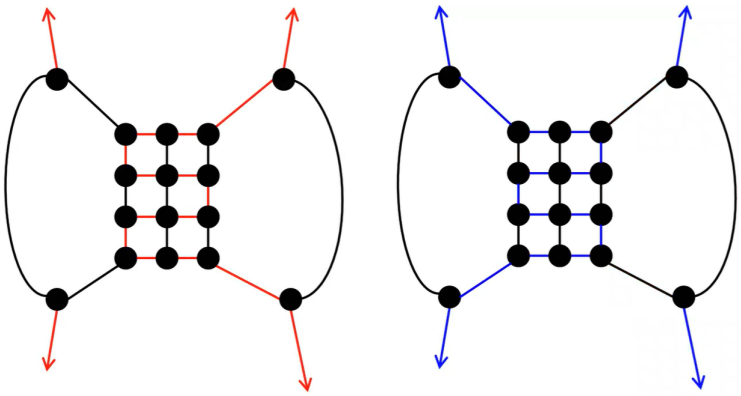
To see how this gadget used, suppose we have a giant cycle, and in the middle of this cycle we have these two binary choices that can be made.



We can then hook the gadget up between these pair of edges.



There are two ways to fill out ths gadget.



This allows us to force logic upon our choices. By doing this for several pairs of edges, we can construct a Hamiltonian cycle problem equivalent to the following:

- You are given a number of choices where you need to pick one from several options (of multi-edges).
- You have several constraints that say that of the two choices you could make, you must have picked exactly one of them.

With this in mind, we want to see if you can simulate a hard problem from this.

Now, we can talk about the zero-one equation.

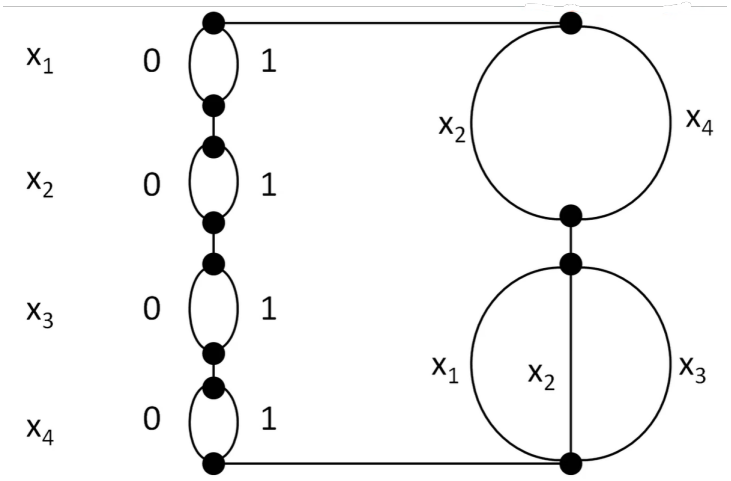
- For each variable, choose either 0 or 1. This will tell you the value of that variable.
- For each equation, choose one variable to be equal to 1.
- We also need to add some constraints. For each variable that appears in an equation, exactly one of the following should be selected. Either that variable is in the equation, or that variable is equal to 0.

8.10.2 Example: Zero-One Equation

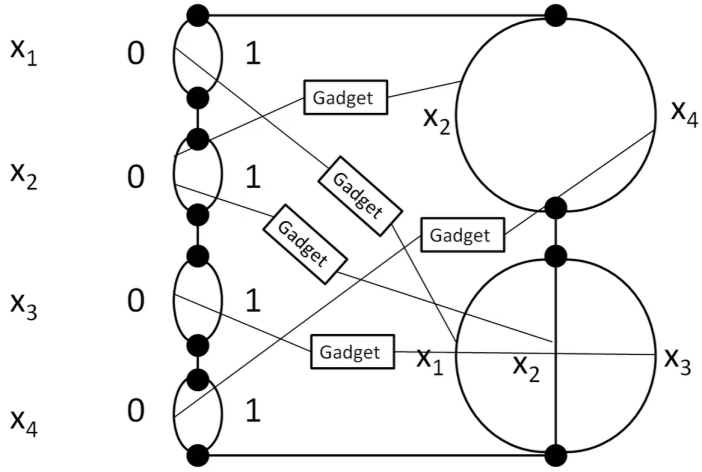
Suppose you have the following zero-one equation

$$\begin{cases} x_1 + x_2 + x_3 = 1 \\ x_2 + x_4 = 1 \end{cases}$$

First, we can build a graph for each gadget.



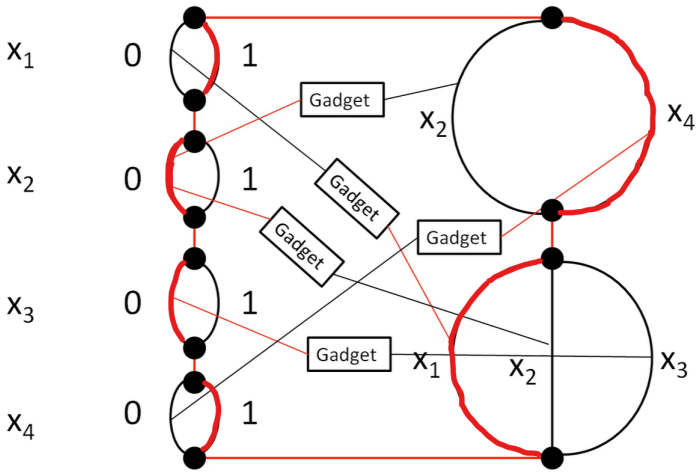
For each variable that appears in each equation, we have a gadget which enforces the constraint that we either select a variable or we set that variable equal to 0.



To see what this means, consider the top gadget. This is saying that either:

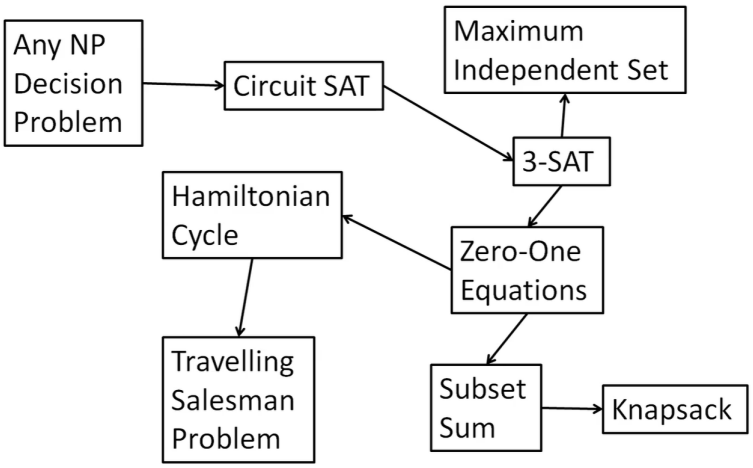
- Either x_2 will be selected from the equation.
- Or we set x_2 equal to 0.

This gives us a complicated Hamiltonian cycle problem, but this graph that we built has a Hamiltonian cycle if and only if the corresponding zero-one equation has a solution. Consider the following:



Here, we are saying that $x_1 = 1$, $x_2 = x_3 = 0$, and $x_4 = 1$.

8.11 Reduction Summary



9 Dealing with NP-Completeness

We now talk about how to deal with NP-Completeness. First, before solving a problem, we want to see if the problem is NP-Complete. If the problem is, then we probably won't find a very good solution for it. So, at least, you have an excuse for not having a better algorithm. However, if you need to solve the problem, well, you still need to find some way to solve it. In this section, we'll briefly talk about ways to solve such problems.

Consider the following statement: *if your problem is NP-Hard/NP-Complete, then unless $P = NP$, there is no algorithm that gives the exact answer to your problem on all instances in polynomial time.* Despite not being able to get around the fact that you might not be able to solve such problems efficiently, there are loopholes in this statement that can be exploted.

- "**all instances**": there are worst-case instances which can be very hard to solve. But, the instances that you're working on is easier. For example, maximum independent set is a hard problem, but maximum independent set of trees is not. So, see if you can make further assumptions.
- "**exact answer**": you sometimes do not need the *best* answer. It's okay to have a good answer. This leads to the notion of approximation algorithms.
- "**polynomial time**": for finite input sizes, or smaller input sizes, the runtime of your algorithm might not matter. So, an *efficient brute-force* algorithm might be good enough.
- "**P = NP**": If you can prove that $P = NP$, then you can solve whatever problems you would like in NP.

9.1 Sudoku

Consider the logic puzzle Sudoku, where you have a 9×9 grid of numbers with 1 - 9 so that:

- Each row has all numbers.
- Each column has all numbers.
- Each of the main 3×3 sub-squares has all numbers.
- Some entries are pre-filled.

For a fixed board size, you can't really say that this problem is in NP. However, there is an obvious generalization which is formally a NP-Hard problem.

9.1.1 Brute-Force

Generally speaking, you cannot do much better than brute-force saerch. However, we note that, even for a fixed board size of 9×9, a brute-force search would consider 9^{81} possibilities. This would essentially take longer than the lifespan of the universe. However, people can solve them while waiting for something. How is this the case?

9.1.2 Deductions

One way to make progress is to make deducations. In particular:

- We can use the rules to show that some square can only be filled out in one way.
- We can use that information to help fill out more squares.
- If you're lucky, you can keep going until the entire problem is solved.

To see an example of how this would work, consider the 3-SAT problem $x \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y} \vee z)$.

- For the whole statement to be true, each clause must be true. So, $x = \text{True}$.
- If we simplify this, we now get $y \wedge (\overline{y} \vee z)$. Therefore, $y = \text{True}$.
- Simplifying this further gives us z. Therefore, $z = \text{True}$.

So, instead of having to check every possible options, we can just make a sequence of logical deductions which led us to the answer.

9.1.3 Getting Stuck

If we just have simple rules like this, we'll get stuck pretty quickly. This especially applies if you have harder problems. So, how do you get unstuck?

- Use stronger deduction rules. In the case of Sudoku:
  - Find a square that only one number can fill.
  - Find a region with only one place for a given number to be put.
  - Find a pair of squares in the same row that must contain two numbers (which then cannot appear elsewhere in that row).
  - Find a rectangle whose corners must contain 2 copies of a number. That number cannot appear elsewhere in those rows/columns.
  - Find 3 rows and 3 columns whose intersections must contain 3 copies of a number. That number cannot appear elsewhere in those rows and columns.

  This helps cuts down a bunch of possibilities which can simplify your search space.
- Guess and check. For example, you can make a guess, and then use the deduction rules to see if you can reach a contradiction. If you reach a contradiction, then you know that the guess you made is wrong so you can guess again. Formally, this is known as *backtracking*.

9.2 Finding Exact Solutions to NP-Complete/Hard Problems

There are two strategies to find exact solutions.

9.2.1 Backtracking

The idea behind backtracking is as follows.

```
Search(Problem P, Search Space S):
    If you can find a contradiction:
        return 'no solution'
    Split S into subproblems S1, S2, ...
    For each i:
        Run Backtracking(P, Si)
    Return any solutions found.
```

Essentially, the idea is to try every possible solution in S. However, if we can reach a contradiction – especially early on – then we can cut our search space by a significant factor.

Another thing to consider is how you split your search space. Generally, you want to branch on variables that have a lot of deductive power – essentially, variables which may lead to a contradiction very early on.

Note that backtracking works well for decision problems. However, for optimization problems, not so much. So, we introduce the idea of *branch and bound*.

9.2.2 Branch and Bound

For optimization problems, we need to keep track of the best solutions so far. Then, instead of a contradiction, what you need to do is *bound* the best solution in S (the space you're searching over). Then, if it's *worse* than the best so far, then stop.

Essentially, the idea is as follows.

```
BranchAndBound(Best, Search Space S):
    If UpperBound(S) <= Best:
        Return 'no improvement'
    If S is a full solution:
        Return value of S
    Split S into subproblems S1, S2, ...
    For each i:
        New = BranchAndBound(Best, Si)
        Best = Max(New, Best)
    Return Best
```

9.3 Heuristic Search

Heuristic search is this idea where we relax one of the rules to find an answer. For example, we can relax the rule that we want the exact answer. Often times, especially for optimization problems, we don't necessarily need the best answer; we only need a "good enough" answer.

9.3.1 Hill-Climbing

The high-level idea is what is known as *hill-climbing*. The idea is that we find x, and then we try all y near x. If $f(y) > f(x)$, then we set x to be y. This idea would be repeated until you don't really see an improvement.

9.3.2 Getting Stuck

An issue is that you can get stuck, though. For example, suppose you're at the bottom of a hill. There are going to be a lot of local maximum points, which your algorithm might interpret as the maximum. However, we note that there could very well be an absolute maximum point that the algorithm can't find.

9.3.3 Getting Unstuck

There are several ways to get unstuck.

- Randomized Restart: If you try many starting points, you might be able to find the one that is the true maximum. In the hill-climbing analogy, this just means starting at different points in the hill.
- Expand Search Area: Instead of looking for changes in a small, restricted area, you could expand how far you saerch to hopefully find a better answer. The disadvantage of this is that the larger areas could take more work. However, at least it's harder to get stuck. In the hill-climbing analogy, this means that, rather than taking one step to see if you can find a higher point, you could take multiple steps around.
- Simulated Annealing: At the start of the algorithm, you can take big, random steps. This will hopefully get you to the right hill. Then, as the algorithm progresses, the step distance decreases and the algorithm starts to fine-tune more precisely. This works well, in practice, for a number of problems.

In either case, it's still no guarantee of finding the actual maximum in polynomial time.

9.4 Approximation Algorithms

An α-approximation algorithm is an algorithm that always obtains the optimal value up to a factor of α.

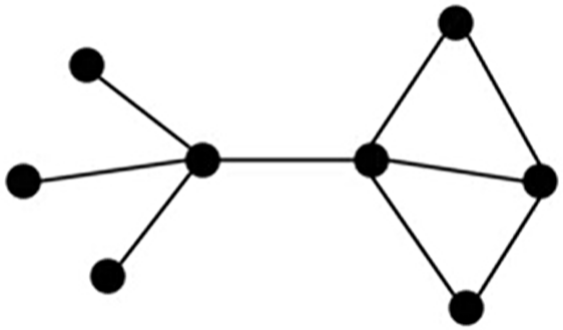
For example, a 2-approximation algorithm is an algorithm that obtains the optimal value up to a factor of 2; if you're maximizing, you're at least a half of the maximum value, and if you're minimizing, you're at most twice of the optimal value.

An approximation answer doesn't necessarily give you the best answer, but at least it gives you some rigorously proven guarantees about how good your answer is.

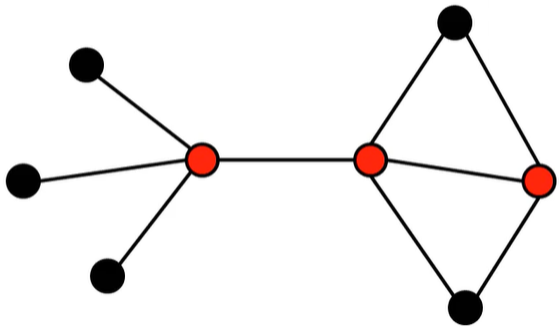
9.4.1 Problem: Vertex Cover

Given a graph G, find a set S of vertices so that every edge of G contains a vertex of S and so that $|S|$ is as small as possible.

For example, consider the graph below.



The minimal size is 3, and is given by:



9.4.2 Greedy Algorithm for Vertex Cover

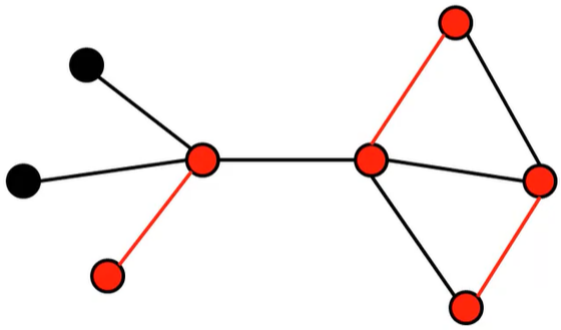
The greedy algorithm for the vertex cover problem is as follows:

```
GreedyVertexCover(G):
    S = {}
    While S doesn't cover G:
        (u, v) = some uncovered edge
        Add u and v to S
    Return S
```

We note that this is a 2-approximation algorithm.

9.4.3 Example of Algorithm on Graph

In the graph above, we note that this algorithm could potentially find the following vertices:



The algorithm finds k edges and returns $2k$ vertices. These edges are vertex-disjoint, i.e. no two edges could share a vertex since we added both vertices whenever we account for an edge. So, any cover must have at least one vertex on each of these edges. So, the optimal cover has size at least k, but we have one of $2k$.