CS 331 Algorithms and Complexity Notes

Jeffrey Wang

Fall 2020

Contents

1 Induction and loop invariants

1.1 Mathematical induction

Example: Climbing an infinite ladder
Suppose we have an infinite ladder.

1. We can reach the first rung of the ladder.
2. If we can reach a particular rung of the ladder, then we can reach the next rung.

From (1), we can reach the first rung. Then by applying (2), we can reach the second rung. Applying (2) again, the third rung, and so on. We can apply (2) any number of times to reach any particular rung, no matter how high up.

Principle of mathematical induction: To prove that proposition $P(n)$ is true for all positive integers n, we complete the following steps:

1. **Basis step** - show that $P(1)$ is true
2. **Inductive step** - show that $P(k) \rightarrow P(k+1)$ is true ($P(k)$ implies $P(k+1)$) for all positive integers k

To complete the inductive step, assuming the *inductive hypothesis* that $P(k)$ holds for an arbitrary integer k, show that $P(k+1)$ must be true.

Example: Show that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Define our proposition $P(n) =$ "$1+2+\cdots+n = \frac{n(n+1)}{2}$"

Base case: Let $n = 1$. $1 = \frac{1(1+1)}{2} = 1$. True.

Inductive step: Assume $1+2+\cdots+n = \frac{n(n+1)}{2}$ is true. That would mean $1+2+\cdots+n+(n+1) = \frac{(n+1)(n+1+1)}{2} = \frac{(n+1)(n+2)}{2}$. $(1+2+\cdots+n)+n+1 = \frac{n(n+1)}{2} + n + 1$ by our assumption. Then we can see that $= \frac{n(n+1)+2n+2}{2} = \frac{(n+1)(n+2)}{2}$. Thus the inductive step is true.

Hence we have proved this proposition.

Example: Prove that $n^3 - n$ is divisible by 3 for all positive integers.

Proposition definition: $P(n) =$ "$n^3 - n \bmod 3 = 0 \,\forall\, n > 0$"

Base case: Let $n = 1$. $1^3 - 1 = 0$ and 0 is divisible by 3. True.

Inductive step: Assume $n^3 - n$ is divisible by 3. $(n+1)^3 - (n+1) = n^3 + 3n^2 + 3n + 1 - n - 1 = n^3 - n + 3(n^2 + n)$. We know that $n^3 - n$ is divisible by 3 by our assumption. Any integer $n^2 + n$ is still an integer, and when multiplied by 3, it is clearly divisible by 3. Since each part is divisible by 3, we can then conclude that $(n+1)^3 - (n+1)$ is divisible by 3, so we prove this proposition true by induction.

1.2 Strong induction

Sometimes, we need more base cases than just $P(1)$. For instance, in the Fibonacci sequence, two previous values are needed. Therefore, we need to show

that, assuming $P(1), P(2), \ldots, P(k)$ are all true (rather than just $P(k)$), that $P(k+1)$ is true.

In some cases, the base case is extended past $P(1)$. We may need to prove $P(2), P(3), \ldots$ independently as well, depending on the nature of the induction.

Example: prove that postage amounts of 12 cents or more can be formed with just 4-cent or 5-cent stamps.

Base case: need 12, 13, 14, and 15.

Example: For $n \geq 0$, prove Fibonacci numbers $F_n < 2^n$.

Base case: $F_0 = 0 < 2^0 = 1$. $F_1 = 1 < 2^1 = 2$. $F_2 = 1 < 2^2 = 4$. All true.

Inductive step: Assume $F_0 < 2^0, F_1 < 2^1, \ldots, F_n < 2^n$. $F_{n+1} = F_{n-1} + F_n < 2^{n-1} + 2^n$. We know that $2^{n-1} + 2^n$ is less than $2^n + 2^n = 2^{n+1}$. Therefore we can prove $F_{n+1} < 2^{n+1}$ true.

1.3 Correctness of algorithms

An algorithm, a step by step set of instructions to solve a task, is described by:

1. Input data
2. Output data
3. Preconditions: specifies restrictions on input data
4. Postconditions: specifies what is the result

Example: Binary search:

1. Input data: a: int[], x: int
2. Output data: found: bool
3. Precondition: a sorted in ascending order
4. Postcondition: found is true if x is in a, otherwise found is false

Correctness: An algorithm is correct if:
for any **correct** input data, it (1) **stops** and (2) **produces correct output**.

If the algorithm does not stop but otherwise works, it is considered partially correct.

An algorithm is a list of actions.

Proving an algorithm is totally correct involves:

1. Proving it will terminate
2. Proving that the list of actions applied to the precondition imply the postcondition

Easy for simple sequential algorithms, complicated to prove for repetitive algorithms.

Example - sequential algorithm (swap): Precondition: x is a, y is b
Postcondition: x is b, y is a

temp := x implies temp is a
x := y implies x is b
y := temp implies y is a
Now postcondition is true due to what this algorithm implies.
Example - repetitive algorithm (sum of n numbers):
while i < length of array a: s = s + a[i]
Problem: cannot enumerate all actions in case of a repetitive algorithm. Solution: use loop invariants.

1.4 Loop invariants

A loop invariant is a logical predicate such that: if it is satisfied before entering any single iteration of the loop, then it is also satisfied after the iteration. The loop invariant is a predicate of the algorithm.

Example: loop invariant is the inductive hypothesis. At step i, s holds the sum of the first i numbers after adding the ith number ($a[i]$).

What must be shown true about loop invariant:

1. Initialization - it must be true before iteration.
2. Maintenance - if it is true before the iteration, it should still continue being true before the next iteration.
3. Termination - when the loop terminates, the invariant should still be true.

Example with sum of n numbers:
Loop invariant: s should contain the partial sum of numbers between the 0^{th} and i^{th} iteration.

- Initialization: Before the first iteration, $i = 0$ and $s = 0$. Sum should be zero, which it is.
- Maintenance: Assume s is the sum of the first i numbers. If we add $a[i+1]$ to s, our new s is equal to the sum of the first $i + 1$ numbers. Hence s is indeed equal to sum of first $i + 1$ numbers.
- Termination: Algorithm terminates when $i == n$, and s is the sum of all n numbers. Hence true.

Example of loop invariants for max-finding algorithm: Loop invariant: max contains the maximum of values in array A between indices 0 and $index$.

- Initialization: Before the first iteration, $i = 0$ and $max = A[0]$. Clearly max is the maximum of just $A[0]$ itself.
- Maintenance: Assume that max is the maximum value of $A[0]$ to $A[index]$. For the next iteration now max will either be $A[index + 1]$ or the original max value. We know the original max value must be true based on our assumption. If $A[index + 1]$ is the max between 0 and $index + 1$, then we know that $A[index + 1]$ must be the max instead. Thus we can see that max is indeed the maximum between 0 and $index$.

- Termination: When index $= n$, then the loop terminates. max should be the maximum between $A[0]$ and $A[n-1]$, which it is.

Iterative algorithms use loop invariants. They are the relationship between variables during loop execution.

Recursive algorithms use a hypothesis, so strong induction is suitable for them.

2 Stable marriage problem

Wanted a job recruiting process that is **self-enforcing**. If the process is self-enforcing, self-interest itself prevents offers from being retracted and redirected.

2.1 Definition of problem

Problem definition: given 3 men (x, y, z) and 3 women (A, B, C) i.e. two separate parties or groups, and the goal is to match each man with only one woman.

Idea: arrange marriages for n men and n women such that each man is paired with only one woman according to their preferences.

Each party should form a list of their preferences from most preferred to least preferred.

The result will be 3 matches. But we want to avoid the problem of instability (when man x prefers B over A even though x is matched with A and y is matched with B, and also B prefers x over y). Instability needs to be mutual.

Optimal solution: all parties have a match, there is no instability.

A perfect matching S' is a matching with the property that each member of M and each member of W appears in *exactly* one pair in S'.

2.2 Finding a solution to the problem

1. Develop a list of preferences.
2. Decide whether men or women propose. It is better to propose than to be the one receiving proposals.
3. Start with x who proposes to A. A accepts the proposal immediately. Now x and A are *engaged*. This is not a marriage yet, just an intermediary engagement.
4. y proposes to B and B accepts the proposal since she is free.
5. z proposes to A but A says no because she prefers x more than z. z also proposes to B but she also refuses because she is matched with y. Finally, z proposes to C and she accepts.
6. Now, there is no unmatched man, and there is no mutual higher preference, so these matches are stable.

2.3 Gale-Shapley algorithm

Precondition: There are the same amount of men and women.

Postcondition: Each man has a match and there does not exist any instability (i.e. no mutual higher preference).

Algorithm 1 Gale-Shapley Algorithm

```
1: procedure GaleShapley(M, W)
2:     M[0..n] ← ∅
3:     N[0..n] ← ∅
4:     while some man m ∈ M is free and hasn't proposed to every woman do
5:         m ← M   ▷ Choose man m who is free and hasn't proposed to every woman
6:         w ←1st woman on m's list to whom m has not yet proposed
7:         if w is free then
8:             arrange m and w to be engaged
9:         else if w prefers m to her fiance m′ then
10:            m engaged with w, m′ no longer engaged
11:        else
12:            w rejects m
13:        end if
14:    end while
15: end procedure
```

3 Basic runtime complexity

We are interested in algorithms that are efficient, deterministic, and correct.

The factors that affect efficiency are

1. the size of the input (number of input objects, usually, but sometimes the size in bits)

2. primitive operations (independent of the compiler used, architectural details, and optimization settings)

Big Oh: Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\mathcal{O}(g(n))$ if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$.

Little Oh: Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $o(g(n))$ if there are positive constants c and n_0 such that $f(n) < cg(n)$ for $n \geq n_0$.

Big Omega: Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Omega(g(n))$ if there are positive constants c and n_0 such that $f(n) \geq cg(n)$ for $n \geq n_0$.

Little Omega: Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\omega(g(n))$ if there are positive constants c and n_0 such that $f(n) > cg(n)$ for $n \geq n_0$.

Big Theta: Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Theta(g(n))$ if there are positive constants c and n_0 such that $f(n) = cg(n)$ for $n \geq n_0$.

Growth rates in ascending order: $1, \log n, \sqrt{n}, n, n \log n, n^2, n^3, n^c, 2^n, 3^n, c^n, n!, n^n$, where c is a scalar.

Comparing growth rates can be done using limits or plugging in numbers.

Take the limit:

$$\lim_{n \to \infty} \left| \frac{f(n)}{g(n)} \right|$$

Should the limit approach ∞, then $f(n)$ grows faster than $g(n)$. Should the limit approach 0, then $g(n)$ grows faster than $f(n)$. Otherwise, they grow the same.

$f(x)$ is $\mathcal{O}(g(x))$ if $\lim_{n \to \infty} |\frac{f(n)}{g(n)}| < \infty$.

$f(x)$ is $\Omega(g(x))$ if $\lim_{n \to \infty} |\frac{f(n)}{g(n)}| > 0$.

$f(x)$ is $\Theta(g(x))$ if $0 < \lim_{n \to \infty} |\frac{f(n)}{g(n)}| < \infty$.

4 Graphs

A graph is comprised of a set of vertices/nodes and a set of edges.

$$G = (V, E)$$

A graph can be either undirected or directed.

- **Undirected edges** are written: $a - b$ or $\{a, b\}$
- **Directed edges** are written: $a \to b$

Usually, we use $n = |V|$ and $m = |E|$ (cardinality/number of elements in V and E).

Complete graph: a graph whose nodes are each connected to all other nodes by a direct edge. There are $m = \frac{n(n-1)}{2}$ edges in a complete graph.

$\deg(n)$ is the number of neighbors of n.

Graphs can be sparse or dense.

For **sparse** graphs, **adjacency lists** are best. For **dense** graphs, **adjacency matrices** are best.

4.1 Graph representations

Adjacency matrix: For each row - column combination (a, b), if row a column b is 0, there is no edge, otherwise if it is 1 there is an edge. Space complexity is $\Theta(n^2)$ and time complexity is $\Theta(n^2)$.

Adjacency list: There is a list, one element for each node. For each element, all of the nodes it points to are the connected vertices. Space complexity is $\Theta(n + m)$ and time complexity is $\mathcal{O}(n + m)$. For an adjacency list, for a node n, $\deg(n) = 2m$.

4.2 Graph properties

Cycle: if there is a set of edges that connect unique nodes except the first and last node are the same.

Tree: an undirected graph if it is connected and does not contain any cycles.

Tree theorems: G is connected, G does not contain a cycle, and G has $n-1$ edges.

Rooted tree: given a tree T, choose a root node r to start with, where it is possible to reach all other nodes. The height of the tree is the number of edges in the longest path from the root. Root is at level 0, its descendants are at level 1, level 1's descendants are at level 2, etc.

Connectivity: given two nodes s and t, is there a path between s and t?

Shortest path problem: given two nodes s and t, what is the length of the shortest path between s and t?

4.3 BFS and DFS

Breadth first search: Explore outwards from s in all possible directions, adding nodes one "layer" at a time. Don't add nodes that were already added.

BFS is not $\Theta(n^2)$ because the test only checks $2m$ neighbors each time, thus its runtime complexity is $\Theta(m+n)$.

See Algorithm 2 for the BFS algorithm.

Algorithm 2 Breadth-first search

```
1:  procedure BreadthFirstSearch(G = (V, E))
2:      visited[0..|V|] ← false                    ▷ Mark all nodes as unvisited
3:      s ← some node in V                         ▷ Choose some starting node from V
4:      visited[s] ← true
5:      L ← a queue                                ▷ Create some queue L
6:      L ← enqueue s
7:      while L is not empty do
8:          v ← dequeue from L
9:          neighbors ← v's neighbors
10:         for each neighbor w do
11:             if w not in visited then
12:                 visited[w] ← true
13:                 L ← enqueue w
14:             end if
15:         end for
16:     end while
17: end procedure
```

Property: let T be a BFS tree of $G = (V, E)$ and let (x, y) be a property of G. Then the level of x and y differ by at most 1.

Depth first search: Explore downwards from s towards a leaf, one path at a time. This algorithm may add neighbors to the stack that have already been

visited. Runtime complexity is also $\Theta(m+n)$.

See Algorithm 3 for the DFS algorithm.

Algorithm 3 Depth-first search

```
1:  procedure DepthFirstSearch(G = (V, E))
2:      visited[0..|V|] ← false                    ▷ Mark all nodes as unvisited
3:      s ← some node in V                         ▷ Choose some starting node from V
4:      L ← a stack                                ▷ Create some stack L
5:      L ← push s
6:      while L is not empty do
7:          v ← pop from L
8:          if v not in visited then
9:              visited[v] ← true
10:             neighbors ← v's neighbors
11:             for each neighbor w do
12:                 L ← push w
13:             end for
14:         end if
15:     end while
16: end procedure
```

4.4 Connected components

Connected component: For undirected graphs, it is all nodes reachable between each other.

Connected components can be found using either BFS or DFS.

Application of connected components: bipartite graph testing.

Bipartite graph: can separate all nodes into two groups (say, red and blue nodes) such that red nodes only have edges connecting to blue nodes, and vice versa, but never an edge between two nodes of the same color/within the same group.

Lemma: Bipartite graphs cannot contain an odd length cycle.

Lemma: Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds:

1. No edge of G joins two nodes of the same layer, and G is bipartite.
2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

If there is an edge between two nodes of the same layer, then the graph is not bipartite.

4.5 Directed graphs

Mutual reachability: Nodes u and v are mutually reachable if there is a path from u to v and also v to u.

Strong connectivity: A graph is strongly connected if every pair of nodes is mutually reachable.

Directed acyclic graph (DAG): A directed graph that contains no directed cycles.

Topological ordering:

Lemma: G has a topological ordering if and only if it is a DAG.

Lemma: If G is a DAG, it must have a node with no incoming edges.

Finding a topological ordering of a DAG: See Algorithm 4

Algorithm 4 Topological ordering

```
1: procedure TopologicalOrdering(G = (V, E))
2:     Find node v with no incoming edges, order it first
3:     Delete v from G and add it to G', set v to w'
4:     while G is not empty do
5:         w ← TopologicalOrdering(G − {w})
6:         Add w to G', connecting it to the last added node w'
7:         w' ← w
8:     end while
9: end procedure
```

5 Greedy algorithms

A greedy algorithm is, generally but not always, one where a series of choices are made, and each choice is locally optimal, and that the final result is globally optimized.

Proof techniques:

1. **Stays ahead**
2. **Exchange argument**

5.1 Interval scheduling

Problem definition: We have some set of jobs/intervals, each with a starting time s_j and ending time e_j.

Goal: Find the maximum number of mutually compatible jobs which can be scheduled at one time.

Potential greedy options:

- **Earliest start time first** - doesn't work when the earliest job is the longest and overlaps with multiple shorter jobs that start later

- **Shortest job first** - doesn't work when it is between two other jobs that are longer
- **Fewest conflicts** - does not work when there are four jobs that are optimal, but the fewest conflict job is one that is between jobs 2 and 3
- **Earliest finish time** - is optimal

See Algorithm 5.

Algorithm 5 Interval scheduling

```
1: procedure IntervalScheduling(J = ((s_1, f_1), ..., (s_n, f_n)))
2:     SJ ← Sort jobs J by finishing time f_1 ≤ ··· ≤ f_n
3:     A ← ∅
4:     for j ← 1 to n do
5:         if SJ[j] compatible with A then
6:             A ← A ∪ {SJ[j]}
7:         end if
8:     end for
9: end procedure
```

Runtime complexity: $\mathcal{O}(n \log n)$

Proof (using stays ahead method):

Assume O is an optimal solution (i.e. O is the optimal set of intervals which can be scheduled).

Assume G is the solution given by the greedy algorithm 5.

We need to show:

- The greedy algorithm is as good as the optimal solution
- The greedy algorithm has the same size as the optimal solution

Let $i_1, i_2, \ldots, i_k$ be jobs selected by the greedy algorithm.

Let $j_1, j_2, \ldots, j_k$ be jobs selected by the optimal algorithm.

Proof by induction for accuracy of solution.

Base case. $f(i_1) \leq f(j_1)$ is true, since there is only one way to schedule one interval, so the greedy algorithm G MUST be at least as efficient as the optimal solution O.

Inductive hypothesis. $f(i_{r-1}) \leq f(j_{r-1})$ for $j_r > 1$

Since the optimal solution must pick intervals which are compatible (i.e. not overlapping), it must be the case that $f(j_{r-1}) \leq s(j_r)$. The inductive hypothesis applied implies $f(i_{r-1}) \leq f(j_{r-1}) \leq s(j_r)$.

If this is true, it means j_r was available when the greedy algorithm picked i_r. It picked i_r over j_r because the greedy algorithm always picks interval with the earliest finish time. Therefore, by transitivity, it must be the case that $f(i_r) \leq f(j_r)$.

Proof by contradiction for size of solution. Assume that the optimal solution has one more interval than the greedy algorithm. Then that extra

interval should've been placed after the last interval of the greedy algorithm, but it isn't chosen. This would mean that the extra interval does not exist, which causes a contradiction.

5.2 Scheduling to minimize lateness

For jobs with time to complete t, deadline d, and lateness $l = \max(\text{finishtime} - d, 0)$.

One bad arrangement: $t = 1, d = 7; t = 2, d = 4; t = 3, d = 3$. Lateness $= 3$.

Best arrangement would be one that minimizes lateness: $t = 3, d = 3; t = 2, d = 4; t = 1, d = 7$. Lateness $= 1$.

Pick the earliest deadline first.

Sort by deadline, find running sum of times. The final running sum minus the latest deadline is the maximum optimal lateness.

5.3 Dijkstra's algorithm

Single source shortest path algorithm for weighted graphs with nonnegative edges only. See algorithm 6. Runtime with regular queue is $\mathcal{O}(|V||E|)$ or $\mathcal{O}(mn)$.

Loop invariant: for all members $u \in S$, the correct shortest path is stored in $dist[u]$.

Algorithm 6 Dijkstra's algorithm

```
1:  procedure DIJKSTRAS(G = (V, E), w, s)
2:      for u ∈ V do
3:          dist[u] ← ∞
4:      end for
5:      dist[s] ← 0
6:      Q ← the set V with each u ∈ V having the dist value from line 3
7:      S ← ∅                                   ▷ Set of nodes already visited
8:      while Q ≠ ∅ do
9:          remove vertex u with minimum dist value from Q and add to S
10:         for v ∈ u's neighbors do
11:             if v ∈ Q then
12:                 dist[v] ← min{dist[v], dist[u] + w(u, v)}
13:             end if
14:         end for
15:     end while
16: end procedure
```

Improved algorithm with priority heap has runtime $\mathcal{O}(m \log n)$.

Dijkstra's may fail for graphs with negative weight edges because it may add a node v to S before the shortest distance to v is fully discovered, especially when the negative weight edge is visited after v is already added to S.

Proof of Dijkstra's algorithm.

Induct on S, set of explored nodes. Base case $|S| = 1$ is trivially true. $dist[s] = 0$ obviously.

Let $u \in S, v \in V - S$. Let $L(u)$ denote the shortest path from s to any node u.

Inductive hypothesis. Assume holds true for $|S| = k$. So $dist[u] = L(u)$ for every $u \in S$.

Inductive step. Show that when S grows to $k + 1$, still holds true. In other words, let's say the $k + 1$th element of S is called v. Then we need to say that $dist[v] = L(v)$ by the time $v \in S$.

$\ell(u, v)$ is the length of edge $(u, v) \in E$.

Let us have a shortest path from s to u to v called $P_{Dijkstra}$, which has length $L(v)$. Assume there is another path from s to x to y to v called P_{alt}, where $x \in S$ and $y \in V - S$.

Within P_{alt}, we know that:

- The length of the path from s to x is at least the shortest path from s to x (this means $L(x) \geq$ shortest path from s to x). By the inductive hypothesis, since $x \in S$, the shortest path from s to x MUST be $dist[x]$ (meaning shortest path from s to x = $dist[x]$). So $L(x) \geq dist[x]$.
- The distance between x and y is $\ell(x, y)$. (x, y) is one of the edges that Dijkstra's considered as a candidate edge for $dist[y]$.
- The distance between y to v must be nonnegative, by the basic assumption of Dijkstra's.

We know that $P_{Dijkstra}$'s length is: $L(P_{Dijkstra}) = dist[u] + \ell(u, v)$. We also know that the alternative length $L(v)$ of the path from s to v (P_{alt}), is at least $dist[x] + \ell(x, y) +$ the nonnegative length from y to v. This means that:

$$dist[v] \leq dist[u] + \ell(u, v) \leq dist[x] + \ell(x, y) \leq L(x) + \ell(x, y) \leq L(P_{alt})$$

Breakdown of what this means:

- $dist[v] \leq dist[u] + \ell(u, v)$ by definition, because it's most likely that $dist[v] = dist[u] + \ell(u, v)$; however, it could theoretically be less than as well.
- $dist[u] + \ell(u, v) \leq dist[x] + \ell(x, y)$ because $dist[u] + \ell(u, v)$ is the length of $P_{Dijkstra}$ and $dist[x] + \ell(x, y)$ is the shortest possible length of the alternative path P_{alt}, assuming that $\ell(y, v)$ is 0, which is highly unlikely but possible. Even if $\ell(y, v) = 0$, the path $P_{alt} = \{s - x - y - v\}$ should be longer than or equal to $P_{Dijkstra} = \{s - u - v\}$ in length, hence we get $dist[u] + \ell(u, v) \leq dist[x] + \ell(x, y)$.
- $dist[x] + \ell(x, y) \leq L(x) + \ell(x, y)$ because, remember, from the first bullet point of P_{alt}, the length of the path from s to x is at least the shortest path from s to x (this means $L(x) \geq$ shortest path from s to x). By the

inductive hypothesis, since $x \in S$, the shortest path from s to x MUST be $dist[x]$ (meaning shortest path from s to x = $dist[x]$). So $L(x) \geq dist[x]$.

- $L(x) + \ell(x, y) \leq L(P_{alt})$ because $L(x) + \ell(x, y) + \ell(y, v) = L(P_{alt})$. If $\ell(y, v) = 0$, then $L(x) + \ell(x, y) = L(P_{alt})$. If $\ell(y, v) > 0$, then $L(x) + \ell(x, y) < L(P_{alt})$. Therefore, $L(x) + \ell(x, y) \leq L(P_{alt})$.

We can see that $dist[v] \leq L(P_{alt})$ by the long inequality series above. That proves the inductive step, which proves Dijkstra correct.

5.4 Minimum spanning tree

Given a graph $G = (V, E)$, find a subset of edges T such that $T \subseteq E$ so that G is connected in a way that $\sum w(E)$ is minimized.

Assumptions:

1. G connected
2. All edges have nonnegative weights
3. G is undirected
4. All edges have distinct weights (no two edges have the same weight)

In a MST, there is no guarantee that there is the shortest distance between any two nodes, but there is a guarantee that the total cost is minimum.

Observations:

1. T is connected
2. T has no cycles

Cut property lemma: Assume that the cost of all edges is distinct. Let S be a subset of nodes that is neither empty nor equal to all of V and let $e = (v, w) \in E$ by a minimum cost edge with one node in S and another in $V - S$. Then every minimum spanning tree must contain edge $e = (v, w)$.

In other words, for all cut edges between S and $V - S$, the minimum weight cut edge MUST be in ALL minimum spanning trees.

Proof by contradiction of cut property lemma. Suppose for the sake of contradiction that T is a MST of G such that edge $e = (v, w)$ be a minimum cost edge that crosses from S to $V - S$ but $e \notin T$. If this is true, then there must be an alternative path in T from v to w which does not cross (v, w). Therefore, there must be some other edge e' that goes from S to $V - S$ that connects v to w. Since T is a MST and e is a minimum cost edge, it must be the case that $w(e') > w(e)$ (weight of e' and e). Consider replacing e' with e, resulting in a tree $T' = T + \{e'\} - \{e\}$. We would get a new MST $T' > T$ (total sum cost of T less than T'), which would mean T' would not be a MST. Thus, contradiction.

MST generating algorithms:

1. Prim's algorithm - node-based

Algorithm 7 Generic MST algorithm using cut property lemma

```
procedure GenericMST(G = (V, E), w)
    T ← ∅
    while |T| ≠ n − 1 do
        Find a cut (S, V − S) such that no edge in T crosses this cut.
        e ← a light edge crossing cut (S, V − S)
        T ← T ∪ {e}
    end while
    return T
end procedure
```

2. Kruskal's algorithm - edge-based; add minimum cost edge one by one

3. Reverse delete algorithm

Kruskal's algorithm: see Algorithm 8. Sort the edges in increasing order and pick them until a MST is formed. Runtime: $\mathcal{O}(m^2)$ unless using union-find data structure, which reduces runtime to $\mathcal{O}(m \log n)$.

Input. $G = (V, E)$ with weight function $w : E \leftarrow R$.
Output. A set A that contains the edges in an MST of G.

Algorithm 8 Kruskal's algorithm

```
1:  procedure Kruskals(G = (V, E), w)
2:      A ← ∅
3:      Sort E in nondescending order of w(e) as e_1 = (u_1, v_1), ..., e_m =
        (u_m, v_m)
4:      for 1 ≤ i ≤ m do
5:          if u_i and v_i are not connected in the graph F = (V, A) then
6:              A ← A ∪ {e_i}
7:          end if
8:      end for
9:      return A
10: end procedure
```

Proof of correctness of Kruskal's algorithm.

Claim: Kruskal's algorithm produces a MST. Let $e = (v, w)$ be some edge to be added to the MST. We need to show e is the minimum crossing edge, since it must be in the MST by definition of a minimum crossing edge.

Let there be sets S (set of all nodes reachable from v) and $V - S$ (set of all nodes reachable from w).

By the cut property lemma, it is consistently adding the minimum crossing edge, and therefore Kruskal's will properly find the minimum spanning tree.

Prim's algorithm is similar to Dijkstra in that we add to a set of visited nodes S. See Algorithm 9.

Correctness follows directly from the cut property lemma.

Runtime is same as Dijkstra's: $\mathcal{O}(m^2)$ naive, but $\mathcal{O}(m \log n)$ using min heap.

Algorithm 9 Prim's algorithm

```
1: procedure PRIMS(G = (V, E), w)
2:     S ← {u} where u is the arbitrarily-chosen starting node
3:     T ← ∅
4:     while not all nodes of V are added to S do
5:         choose node v ∉ S such that e = (u, v) is the minimum weight edge
   out of S                                  ▷ Note that u is already in S.
6:         add v to S
7:         add e to T
8:     end while
9:     return T
10: end procedure
```

Kruskal's vs. Prim's. Kruskal's builds MST by considering the edges in a non-decreasing order of weights and adding one edge at a time. The edge is only added if it does not create a cycle and has not been included before. Prim's builds a MST by adding one vertex at a time. The next vertex to be added is always the one nearest to set of the vertices already discovered but the cheapest.

Prim's vs. Dijkstra's. Dijkstra's may or may not add (u, v) depending on whether it is in the shortest path from s to v where s is the source. Prim's simply chooses the minimum cost edge (u, v) where u in S and v is NOT in S. v is then added to S and (u, v) added to T.

6 Divide and conquer algorithms

Divide and conquer algorithms are defined by **recurrence relations**, which is a function defined by itself. This can also determine the divide and conquer algorithm's time complexity. The recurrence relation mirrors the behavior of the algorithm.

Divide and conquer algorithms comprise of:

1. **Divide** n input problems into subproblems of the same type with smaller sizes n/b, $b > 1$.
2. **Conquer** by solving the subproblems *recursively*.
3. **Combine** the subproblems' solutions to get our final answer.

6.1 Merge sort

Merge sort is an example of a divide and conquer algorithm.

The items are first split into buckets, then they are sorted in their individual buckets and merged bucket by bucket. The merger operates on sorted arrays, making the operation only $\mathcal{O}(n)$ rather than $\mathcal{O}(n^2)$. There are $\log(n)$ splits and merges. Therefore, the runtime of divide and conquer is $\mathcal{O}(n \log n)$.

Algorithm 10 Merge sort

```
1: procedure MergeSort(L)
2:     Assume L is not empty
3:     if |L| = 1 then
4:         return L
5:     else
6:         Divide L into two lists of roughly equal size L_1, L_2
7:         return Merge(MergeSort(L_1), MergeSort(L_2))
8:     end if
9: end procedure
```

Algorithm 11 Merge helper

```
1: procedure Merge(L_1, L_2)
2:     Assume L_1, L_2 are sorted
3:     if L_1 empty then
4:         return one element of L_2
5:     else if L_2 empty then
6:         return one element of L_1
7:     else
8:         Compare top element of L_1, L_2
9:         minTop ← min of top element
10:        return minTop
11:    end if
12: end procedure
```

Claim: Merge (algorithm 11) returns an ordered list of the two ordered lists, L_1 and L_2. **Proof by induction** on $|L_1| + |L_2|$.

Base case. $|L_1| + |L_2| = 1$.

Inductive hypothesis. Claim is true for $|L_1| + |L_2| = k$

Inductive step. Show claim is true for $|L_1| + |L_2| = k + 1$

If we start with size $k + 1$, by the next step of the loop, we end with e_1, the smallest element, plus Merge (algorithm 11) with two lists of size k. We know Merge with two lists of size k is already true by the inductive hypothesis. Since e_1 is the smallest element and we add the sorted list after it, the resulting list must also be sorted. QED.

Runtime complexity proof. Let recurrence relation $T(n)$ be the worst case number of comparisons of merge sort on an array of size n.

The recurrence relation will be $T(n) = T(n/2) + T(n/2) + n + c$ for merge sort, where n represents the sorting done in MERGE (cost of merge + split) and c represents constant time operations at each step.

This can be rewritten as $T(n) = 2T(n/2) + n$.

The base case is how many comparisons we make when we have just one element being merge sorted, which is 0. So $T(1) = 0$.

Solve recurrence relation by **unrolling/tree method**.

Recurrence	**Level**	**Problem size**	**Total time**
$T(n)$	0	n	n
$T(n/2) + T(n/2)$	1	$n/2$	$(n/2) \cdot 2$
$T(n/4) + T(n/4) + T(n/4) + T(n/4)$	2	$n/4$	$(n/4) \cdot 4$
$T(n/2^k) \cdot 2^k$	k	$\frac{n}{2^k}$	$(n/2^k) \cdot 2^k$

To find k, we assume $k = 1$ and set it equal to the problem size.

$$1 = \frac{n}{2^k}$$

$$k = \log_2 n$$

Now we try to find $\sum_{i=0}^{\log n} \frac{n}{2^i} \cdot 2^i$. But rather than painfully finding the summation, we can use a shortcut. At each level, the total time is n. There are k levels. Multiplying the total time by levels gets us the total runtime. Since $k = \log n$, then the total runtime is $\mathcal{O}(n \cdot k) = \mathcal{O}(n \log n)$.

In general, when we use the tree method, we take the total time at each level and multiply it by the number of levels.

$$\text{Runtime} = \text{Total time per level} \times \text{Number of levels}$$

Another way to look at the runtime complexity is: the number of comparisons at each part is $\mathcal{O}(n)$, but we have $\mathcal{O}(\log n)$ levels in the tree. Therefore, the runtime of merge sort is $\mathcal{O}(n \log n)$.

6.2 Unrolling/tree method

Example:

$$T(1) = 450$$

$$T(n) = 4T(n/3) + n/2$$

This means $T(n/3) = 4T(n/9) + n/6$. $T(n/9) = 4T(n/27) + n/18$.

Recurrence	**Level**	**Problem size**	**Total time**
$T(n)$	0	n	$n/2$
$4T(n/3)$	1	$n/3$	$(n/6) \cdot 4$
$16T(n/9)$	2	$n/9$	$(n/18) \cdot 16$
$4^k \cdot T(n/3^k)$	k	$\frac{n}{3^k}$	$4^k(\frac{n}{2 \cdot 3^k})$

$$k = 1 = \frac{n}{3^k}$$

$$k = \log_3 n$$

$$T(n) = \sum_{k=0}^{\log_3 n - 1} 4^k \left(\frac{n}{2 \cdot 3^k}\right) + 450 \cdot 4^{\log_3 n}$$

$$T(n) = n^{\log_3 4}$$

In general, this can be stated as:

$$T(n) = \sum_{k=0}^{\text{total \# of levels} - 1} \text{formula} + \text{\# of leaves} \cdot T(1) = \frac{n}{c} \sum_{k=0}^{\log_3 n - 1} \left(\frac{a}{b}\right)^k + \text{\# of leaves} \cdot T(1)$$

When our recursive formula is in the form $T(n) = aT(n/b) + n/c$:

- Number of leaves is $a^{\log_b n} = n^{\log_b a}$.
- Number of levels is $\log_b n$.
- Formula is $a^k \left(\frac{n}{c \cdot b^k}\right)$.

6.3 Simple master theorem

For any recurrence in the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$

where $a, c \geq 1$, $d \geq 0$, and $b > 1$. Then:

- $T(n)$ is $\Theta(n^d)$ if $a < b^d$
- $T(n)$ is $\Theta(n^d \log n)$ if $a = b^d$
- $T(n)$ is $\Theta(n^{\log_b a})$ if $a > b^d$

6.4 Polylogarithmic case of master theorem

For any recurrence in the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^d \cdot \log^k n)$$

where $a \geq 1$, $d, k \geq 0$, and $b > 1$. Then, $T(n)$ is $\Theta(n^d \log^{k+1} n)$ if $a = b^d$.

When $k = -1$, then $T(n) = \Theta(n^d \log \log n)$. When $k < -1$, then $T(n) = \Theta(n^d)$.

6.5 Combined master theorem

For any recurrence in the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^d \cdot \log^k n)$$

where $a \geq 1$, $d \geq 0$, and $b > 1$. Then:

1. If $a < b^d$: $T(n)$ is $\Theta(n^d)$
2. If $a = b^d$ and:
   (a) if $k > -1$: $T(n)$ is $\Theta(n^d \log^{k+1} n)$
   (b) if $k = -1$: $T(n)$ is $\Theta(n^d \log \log n)$
   (c) if $k < -1$: $T(n)$ is $\Theta(n^d)$
3. If $a > b^d$: $T(n)$ is $\Theta(n^{\log_b a})$

Note that case 1 is root-heavy and case 3 is leaf-heavy.

7 Dynamic programming algorithms

Dynamic programming breaks up a problem into a series of overlapping subproblems and builds up a solution to larger subproblems.

The basic intuition is drawn from intuition behind divide and conquer, where a problem is divided into small subproblems. However, in DP, the subproblems are dependent on each other, unlike DP, where the subproblems are independent.

7.1 Weighted interval scheduling

We can't use a greedy algorithm to solve this. Instead, we'll need to use a dynamic programming approach.

Let our optimality function be $\textsc{opt}(j)$, the value of optimal solution using intervals i through j.

Let's say that each interval has weight v_c, where $c = 1, \ldots, n$.

Base case: $\textsc{opt}(0) = 0$.

General case: $\textsc{opt}(j)$ = either we include j or exclude j.

- Include j: $v_j + \textsc{opt}(p_j)$, where p_j is the first interval that precedes j but does not overlap with j.
- Exclude j: $\textsc{opt}(j-1)$

This can be rewritten as a maximization function.

$\textsc{opt}(j) = \max(v_j + \textsc{opt}(p_j), \textsc{opt}(j-1))$

Note that in this problem, finding each p_j for one j would have a linear runtime $\mathcal{O}(n)$ when implemented naively. So, for all j's, finding all p's would cause the algorithm to have a quadratic runtime $\mathcal{O}(n^2)$.

Proof by induction: We need to prove the claim that $\text{CO}(j)$ computes $\text{OPT}(j)$ for each $j = 1, \ldots, n$.

By definition, $\text{OPT}(1) = 0$.

Base case. when $j = 1$, both $\text{CO}(j)$ and $\text{OPT}(j)$ produce the same result.

Inductive hypothesis. $\text{CO}(j) = \text{OPT}(j)$.

Inductive step. $\text{OPT}(j+1) = \max(v_{j+1}+\text{OPT}(p_{j+1}),\text{OPT}(j))$
$= \max(v_{j+1}+\text{CO}(p_{j+1}),\text{CO}(j))$ by the inductive hypothesis
$= \text{OPT}(j+1)$

7.2 Memoization

At the core of dynamic programming is the fact that it's a problem whose solution is defined recursively. This causes the problem space to become exponentially large. However, an important piece of insight is that dynamic programming will encounter many duplicate subproblems, and it need not be solved repeatedly. Instead, these subproblems can be calculated just once, stored/cached, and used later on for solving future subproblems. A naive dynamic programming implementation will solve the same subproblem exponentially many times, but caching the values of subproblems, called **memoization**, will reduce the problem's runtime exponentially, down to a polynomial runtime (likely linear).

We can prove that in a memoized solution, that a dynamic programming solution will only calculate the recursive function n times; the other times, it will retrieve the memoized value, which is $\mathcal{O}(1)$. Therefore, the runtime for a standard memoized dynamic programming algorithm is $\mathcal{O}(n)$.

7.3 Subset sum problem

Given a set of items with nonnegative weights $S = \{w_1, w_2, \ldots, w_n\}$ and a max weight W; choose a set $s \subseteq S$ so as to maximize $\sum_{i \in S} w_i$ under the constraint $\sum_{i \in S} w_i \leq W$.

The brute force solution is to generate all 2^n subsets of n items, calculate weights, and choose the set of jobs such that $\sum_{i \in S} w_i \leq W$.

In the subset sum problem, accepting n does not immediately imply that we have to reject another request.

$\text{OPT}(i, w)$: the value of the optimal solution using a subset of the items $\{1, \ldots, i\}$ using the maximum allowed weight w. Not only are we considering i, we're considering how much weight we can afford to take, w.

Again, when defining our dynamic programming recurrence relation, we can either include i or exclude i.

- Include i: $\text{OPT}(i, w) = w_i + \text{OPT}(i-1, W - w_i)$
- Exclude i: $\text{OPT}(i, w) = \text{OPT}(i-1, W)$ (weight doesn't change because we excluded, and we include the previous solution)

Therefore, our recurrence equation for the subset sum problem is as follows:
$\text{OPT}(i, w) = \max(w_i + \text{OPT}(i-1, W - w_i),\text{OPT}(i-1, W))$

For an iterative approach to solving this problem, we need a two-dimensional approach for this problem. The two dimensions are the number of items and the upper bound on weight.

Therefore, our dynamic programming caching array $dp[]$ should be sized as upper bound on weight$\times$number of items; in other words, dp[upper bound on weight][number of items].

We can fill in the first row; for any weight upper bound when there are 0 items picked, we know that the current weight of the chosen items is 0. This forms our base case.

We'll have two loops; an outer loop to iterate over the number of items, and an inner loop to iterate over the running sum of the weight of the chosen items.

Algorithm 12 Subset sum

```
1:  procedure SubsetSum(n, W)
2:      M[0...n][0...W]
3:      initialize M[0, w] = 0 for each w = 0, 1, ..., W
4:      for i ← 1 to n do
5:          for w ← 0 to W do
6:              if w < w_i then
7:                  M[i, w] = M[i − 1, w]
8:              else
9:                  M[i, w] = max(M[i − 1][w], w_i + M[i − 1][w − w_i])
10:             end if
11:         end for
12:     end for
13:     return M[n, w]
14: end procedure
```

Time complexity: $\mathcal{O}(n \cdot W)$ - pseudo-polynomial time

This time complexity is not polynomial because W can grow exponentially.

n represents the number of elements, which is the size of inputs. The size of inputs is the actual size in bits. In sorting, we can get away with this as the number of objects rather than the size in bits because the size of input grows linearly as the input grows linearly, as the increase is constant. But in this problem, when we increase the input by one bit, the number of possible weights doubles each time, so the increase is variable. Therefore, this problem is pseudo-polynomial.

Pseudo-polynomial algorithms can be reasonably efficient for when W is small enough. But it becomes less practical as W or n grow larger.

$\mathcal{O}(nW) = \mathcal{O}(n2^s)$

Proof by strong induction. We know that $i \leq j + k \leq k$. Base case is trivial. Inductive hypothesis is $j + k = k + 1$. We need to prove two cases, $j, w + 1$ and $j + 1, w$.

7.4 Knapsack problem

Given a set of items with nonnegative weights $S = \{w_1, w_2, \ldots, w_n\}$ where each weight has a value v_i and a max weight W, choose a subset $s \in S$ so as to maximize $\sum_{i \in S} v_i$ under the constraint $\sum_{i \in S} w_i \leq W$.

In the subset sum problem, we maximized weights. In the knapsack problem, we're maximizing the value instead.

Recurrence:

If $w < w_n$, then $\text{OPT}(n, w) = \text{OPT}(n-1, w)$.

Otherwise, $\text{OPT}(n, w) = \max(v_n + \text{OPT}(n-1, w - w_n), \text{OPT}(n-1, w))$

Fractional knapsack problem: If we can take fractions of our items, it can be solved with a greedy algorithm.

Simply sort the items by ratio of value vs. weight, $p_i = v_i / w_i$. Larger v_i is better, smaller w_i is better.

7.5 Sequence alignment problem

Suppose we are trying to align two strings, $X =$ `mean` and $Y =$ `name`, in the globally optimal way.

We can insert spaces before, in between, and after the strings to make the alignment. We want to find the best possible alignment.

We need to measure how good the alignment is by measuring with alignment cost, which we need to define. There might be a higher penalty for matching consonants versus vowels, or they might be the same. There is also a cost of mismatch δ (penalty), which is when a letter matches with a space.

Let's say that $\delta = 2$, $\alpha_{v-v,c-c} = 1$, $\alpha_{v-c,c-v} = 3$. When we match the same letters, the cost is 0; only when we match two different vowels or two different consonants is the cost 1.

There are two ways to proceed: advance down X or advance down Y. We can't advance down both at the same time; we want to only do so one at a time.

Let $\text{OPT}(i, j)$ denote the minimum cost of alignment between strings X and Y, where i is the last symbol of X and j is the last symbol of Y.

There are two cases:

- $(i, j) \in M$; the last two symbols match
- $(i, j) \notin M$; the last two symbols do not match

When $(i, j) \in M$, $\text{OPT}(i, j) = \alpha_{x_i y_j} + \text{OPT}(i-1, j-1)$, where $\alpha_{x_i y_j}$ is the cost of mismatching.

When $(i, j) \notin M$, either X is longer than Y or Y is longer than X. In these cases, we pay a gap cost of δ.

- If the i^{th} position of X is not aligned, $j > i$, $\text{OPT}(i, j) = \delta + \text{OPT}(i, j-1)$.
- If the j^{th} position of Y is not aligned, $i > j$, $\text{OPT}(i, j) = \delta + \text{OPT}(i-1, j)$.

Our full recurrence relation is:

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

Time complexity: $\mathcal{O}(m \cdot n)$

See Algorithm 13 for the sequence alignment dynamic programming algorithm.

Algorithm 13 Sequence alignment algorithm

```
1:  procedure SequenceAlignment(X, Y)
2:      A[0...m][0...n]
3:      Initialize A[i, 0] = iδ for each i
4:      Initialize A[0, j] = jδ for each j
5:      for i = 1, ..., m do
6:          for j = 1, ..., n do
7:              A[i, j] = min(a_{x_i y_j} + A[i − 1, j − 1], δ + A[i − 1, j], δ + A[i, j − 1])
8:          end for
9:      end for
10:     return A[m, n]
11: end procedure
```

Our A matrix should look like:

n	4δ				
a	3δ				
e	2δ				
m	δ				
-	0	δ	2δ	3δ	4δ
	-	n	a	m	e

Tracing back the actual matching. While this algorithm finds the minimum cost of sequence alignment, it does not find the actual alignments for the optimal alignment.

Note that any solution ends at the very last array entry. We should always start at the end and trace back the matching path by picking which min value was chosen, then jumping to that location in the array and repeating the process.

Proof by induction.

Base case. $i + j = 0$. In this case, $i = j = 0$, and by definition $f(i,j) = \text{OPT}(i,j) = 0$.

Inductive hypothesis. $i' + j' < i + j$.

Inductive step: $i + j$.

Applying the inductive hypothesis, the last edge on the shortest path to (i,j) is either $(i-1, j-1)$, $(i-1, j)$, or $(i, j-1)$.

Therefore, $f(i,j) = \min(a_{x_i y_j} + f(i-1, j-1), \delta + f(i-1, j), \delta + f(i, j-1))$ $= \min(a_{x_i y_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1))$

$= \text{OPT}(i, j)$.

7.6 Bellman-Ford algorithm

The Bellman-Ford algorithm is a single-source shortest path algorithm that, unlike Dijkstra's algorithm, can provide an optimal solution for graphs with negative weight edges, as long as there are no negative weight cycles.

We need an algorithm that can handle negative weights because it is not feasible to scale up weights to contain no negative weight edges and run Dijkstra's on them, as this changes the identity of the graph and does not provide a correct solution.

We need to assume that the input of Bellman-Ford does not contain a negative weight cycle, because that would make a shortest path impossible to compute.

A generic Bellman-Ford algorithm is available as Algorithm 14, in a manner resembling Dijkstra's algorithm, which starts at the source node s and proceeds by relaxation. Bellman-Ford can also be implemented with dynamic programming, which starts at the destination node t and gradually approaches the source node s.

Runtime complexity of an efficient implementation of Bellman-Ford is $O(n \cdot m)$.

Algorithm 14 Generic Bellman-Ford algorithm

```
1:  procedure BellmanFord(G = (V, E), w, s)
2:      for v ∈ V − {s} do
3:          distance[v] ← ∞
4:      end for
5:      distance[s] ← 0
6:      for n − 1 iterations do
7:          for (u, v) ∈ E do
8:              Relax(u, v)
9:          end for
10:     end for
11:     return {distance[v] | v ∈ V}
12: end procedure
13:
14: procedure Relax(u, v)
15:     w ← weight of u, v
16:     if distance[u] + w < distance[v] then
17:         distance[v] ← distance[u] + w
18:         predecessor[v] ← u
19:     end if
20: end procedure
```

Bellman-Ford's application of dynamic programming is not straightforward. Graphs do not naturally have a sequence of objects, so we need to create our own

ordering of objects using the path from a source to vertex. The shortest path from smaller shortest paths is suboptimal, but combining these subsolutions will provide us with a globally-optimal solution.

How are these smaller subproblems defined? By using the number of edges. The number of edges in the path will control how many edges we're allowed in paths from source to destination. We want to define the number of edges we're allowed to traverse from nodes to a certain destination. We start off with a "budget" of 0 edges, and that just contains the destination node itself. Then, as we increase the number of edges to 1, 2, 3, and so on, we will increase the number of nodes that can reach the destination. Each time we increase the number of edges, we will update the shortest path between the nodes to the destination by adding the new edges into our calculations.

In Bellman-Ford, we want to start from the end destination t, gradually add intermediate nodes v, until we reach our starting source s.

Given a G with no negative cycles, is there a shortest path from v to t (simple) with at most $n-1$ edges. Yes. By definition, a simple path should contain at most $n-1$ edges in the path. Therefore, we only need to relax at most $n-1$ times in our algorithm.

Recurrence relation:

$\text{OPT}(i, v)$: The minimum cost of a $v - t$ path using at most i edges. (i is the "budget" of how many edges we are relaxing at this point, and v is the intermediate node we are attempting to reach from t).

There are two cases:

1. P_{path} uses at most $i-1$, which is $\text{OPT}(i-1, v)$ (exclude the edge i away from v)

2. P_{path} uses at most i, which is $c_{vw} + \text{OPT}(i-1, w)$ (include the edge i away from v)

At iteration i, we have already calculated the most efficient path with at most $i-1$ edges (in other words, $\text{OPT}(i-1, v)$ was already calculated, so we already know the most efficient subpath with at most $i-1$ edges). Now, we need to consider adding edges exactly i away from v. Should we add them or not? That, of course, depends on whether adding it would be optimal for that subproblem or not. We'd need to consider the weight between v and immediately connected node w, (v, w), which is c_{vw}, and use the optimal values from $i-1$ and w. Note that there are multiple w's we need to consider, and we would take the minimum weight between all of the (v, w) edges.

Thus, our optimal solution's recurrence relation is, for $i > 0$:

$\text{OPT}(i, v) = \min(\text{OPT}(i-1, v), \min_{w \in V}(\text{OPT}(i-1, w) + c_{vw}))$

The runtime complexity of Algorithm 15 is $\mathcal{O}(n \cdot m)$; when the graph is dense, $n = m^2$, so the runtime complexity becomes $\mathcal{O}(n^3)$ at worst.

Detecting negative weight cycles. We usually run Bellman-Ford $n-1$ times. However, we can run it an extra time (in total n times) to check for cycles. Here's how: the graph has a negative weight cycle if and only if there are different values in the DP array for the $n-1^{th}$ iteration and the n^{th} iteration.

Algorithm 15 Dynamic programming Bellman-Ford algorithm

```
1: procedure BellmanFordDP(G = (V, E), s, t, c)
2:     Array M[0 ... n − 1, v]
3:     Define M[0, t] = 0 and M[0, v] = ∞ for all other v ∈ V
4:     for n − 1 iterations do
5:         for v ∈ V in any order do
6:             M[i, v] = min(M[i − 1, v], min_{w∈V}(M[i − 1, w] + c_{vw}))
7:         end for
8:     end for
9:     return {distance[v] | v ∈ V}
10: end procedure
```

8 Network flow

Graphs are sometimes used to model transportation networks. In such networks, edges carry some sort of traffic and nodes act like switches passing the traffic between edges. For example, in a network, the edges are links that carry the data and the nodes are switches. Network flow are used to model such transportation networks.

8.1 Maximum flow

Problem. Given a directed graph $G = (V, E)$:

- Each edge has a capacity $c_e > 0$, an integer value
- G has a single source node $s \in V$ such that there are no edges going into s
- G has a single sink node $t \in V$ such that there are no edges going out of t
- Flow is generated at s and consumed at t

Goal: Need to find the maximum flow between s and t at equilibrium (continuous flow out of s into t) through all internal nodes.

In other words:

Nodes have a conservation condition: there are no fluids flowing into our out of the intermediate notes, only the starting source node and ending drain node.

We want to find the max $s - t$ flow at equilibrium.

Definitions:

- c_e: Capacity at edge e, $c_e \in \mathbb{Z} > 0$
- $f(e)$: The amount of flow carried by edge e, $f(e) \in \mathbb{R} \geq 0$ (in reality, $f(e) \in \mathbb{Z} \geq 0$ if $c_e \in \mathbb{Z} > 0$ too)
- $0 \leq f(e) \leq c_e$: Capacity condition

- $\forall v \neq s, t : \sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$: Conservation condition
- $v(f)$: the value of flow f, $v(f) = \sum_{e \text{ out of } s} f(e)$

(Note that the capacities must be rational values in order for any network flow algorithm to terminate. Irrational values cause the algorithm to never terminate.)

Network flow has two conditions:

- Capacity condition: the flow must be within capacity for that edge
- Conservation condition: what flows in flows out

8.2 Ford-Fulkerson algorithm

To solve the max flow problem, we can use the Ford-Fulkerson Algorithm, Algorithm 16.

Algorithm 16 Ford-Fulkerson algorithm

1: **procedure** FORDFULKERSON($G = (V, E), c$)
2: **for** $e \in E$ **do**
3: $f(e) \leftarrow 0$
4: **end for**
5: Find a simple $s - t$ path p
6: Choose the minimum capacity b on p and assign $f(e) = b$ for all e on p
7: Build a residual graph G_f.
8: – For every $e = (u, v)$ with $f(e) > 0$ in G (nonempty flow), place a backwards edge in G_f with residual capacity $= f(e)$.
9: – For every $e = (u, v)$ with $f(e) < c_e$ in G (have not reached capacity yet), place a forward edge in G_f from u to v with residual capacity $= c_e - f(e)$.
10: **while** there is a simple $s - t$ path in G_f called p **do**
11: Find the minimum residual capacity on the path p in G_f, call it b
12: Augment the flow of the corresponding path in G: ▷ Note: After augmentation, we will have a new value for flow.
13: – For every forward edge in p, assign $f(e) = f(e) + b$
14: – For every backward edge in p, assign $f(e) = f(e) - b$
15: Update G_f
16: **end while**
17: **return** f
18: **end procedure**

Maxim: Cannot reach the maximum flow by simply pushing until capacity. There may need to be some reversals in decisions. This is done by adding to the residual graph, G_f, a guideline that tells you to back off some of the fluid to open up other routes in order to increase the maximum $s - t$ flow.

Ford-Fulkerson does not specify how to find the augmenting path. Another algorithm, Edmonds-Karp, is a complete implementation that specifies how to find such a path using BFS.

Hint for finding a path for the algorithm to quickly converge: pick the edges with the "largest" bottleneck (largest capacities).

8.3 Properties and proofs

We can partition each max flow graph into cuts. The minimum cut will result in the minimum flow being identified, leading to the maximal flow.

Lemma: Each flow f' in the augmented path on residual graphs G_f are flows in G.

To prove some flow f' (augmented path on the residual graph G_f) as a valid flow in G, we must verify:

1. Capacity conditions: $0 \leq f(e) \leq c_e$
2. Conservation conditions: $0 \leq f'(e) \leq c_e$

The difference between f and f' is only on edges on the $s-t$ path that was chosen. In other words, the difference is b, the minimum residual capacity on our residual graph G_f. $b \leq$ residual capacity of any edge e.

If we choose some arbitrary edge e on P on G_f, it may be either a forward or backward edge. If the edge is forward, its residual capacity is $c_e - f(e)$. If the edge is backward, its residual capacity is f_e. We know that $b \leq c_e - f(e)$ for forward edges and $b \leq f(e)$ for backward edges.

We know by our forward edge residual capacity that: $f(e) + b \leq c_e - f(e) + f(e)$. $f(e) + b = f'(e)$, and $f'(e) \geq f(e)$ since b is nonnegative. So we can say that: $0 \leq f(e) \leq f'(e) \leq c_e$.

Lemma: At every intermediate stage of Ford-Fulkerson, the flow values $\{f(e)\}$ and the residual capacities in G_f are integers.

Why are the flows necessarily integers if capacities are integers? It depends on how they are modified by the Ford-Fulkerson algorithm. At each step of the algorithm, we only modify the flows through addition and subtraction, and these operations are closed under $\mathbb{Z}$. Thus, we can never get non-integer flows if our capacities are all flows.

Lemma: Let f be a flow in G and let p be a simple $s-t$ path in G_f, then $v(f') = v(f) + b$ since $b > 0$, we have $v(f') > v(f)$. ($v(f)$ is the value of the flow.)

$$v(f) \leq C = \sum_{\text{out of } s} c_e$$

Lemma: Suppose that all capacities in the flow network G are integers. Then Ford-Fulkerson terminates in at most C iterations of the while loop. This is because we must at least increase the flow capacity by 1 (lower bound), and of course, it is bounded by C on the upper bound.

Lemma: Suppose that all capacities in the flow network G are integers. Then Ford-Fulkerson can be implemented to run in $\mathcal{O}(mC)$ time.

Runtimes of each step of the Ford-Fulkerson algorithm:

- Residual path: $\mathcal{O}(m)$
- $s-t$ path: $\mathcal{O}(m+n)$, but m (number of edges) dominates, so $\mathcal{O}(m)$
- Augmentation: $\mathcal{O}(n)$, as a path with $m-1$ edges will have n nodes.

So in each iteration, the runtime is $\mathcal{O}(m)$. As there are C iterations, the total runtime is $\mathcal{O}(mC)$.

8.4 Minimum cut

A cut is a partition of nodes in a graph into two sets, A and B, such that $A \cup B =$ all nodes.

The $s-t$ cut is a cut where $s \in A$ and $t \in B$.

The minimum cut is an $s-t$ cut that gives us the minimum capacity.

Theorem: Let f be any $s-t$ flow, and (A, B) be any $s-t$ cut. Then $v(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$, where $f^{\text{out}}(A)$ is the total sum of capacity (i.e. flow) out of A and $f^{\text{in}}(A)$ is the total sum of capacity (i.e. flow) into A.

Corollary: Let f be any $s-t$ flow, and (A, B) be any $s-t$ cut. Then $v(f) = f^{\text{in}}(B) - f^{\text{out}}(B)$.

Theorem: Let f be any $s-t$ flow and (A, B) any $s-t$ cut. Then, $v(f) \leq C(A, B)$, where $C(A, B)$ is the capacity of the cut. (This is just saying that the flow must be less than or equal to the capacity, which is an obvious property established a while ago. It'll be used again for sanity checks in later theorems.)

Theorem: If f is an $s-t$ flow such that there is no $s-t$ path in the residual graph G_f, then there is an $s-t$ cut (A^*, B^*) in G for which $v(f) = C(A^*, B^*)$. Consequently, f has the maximum flow of any flow in G and (A^*, B^*) has the minimum cut of any $s-t$ cut in G.

8.5 Max-flow min-cut theorem

The max-flow min-cut theorem links together the concepts of a maximum flow and a minimum $s-t$ cut. It states that the value of the max flow of a graph is equivalent to the value of the minimum $s-t$ cut of that graph. In any flow graph G, there must exist a flow f and a cut (A, B) so that $v(f) = C(A, B)$.

f must be a maximum $s-t$ flow because if there were another flow f' of greater value, f' would exceed the capacity of (A, B), which contradicts the capacity condition $v(f) \leq C(A, B)$. Furthermore, the cut (A, B) must be a minimum cut; i.e., no other cut can have a smaller capacity, because if there is another cut (A', B') of smaller capacity, it would be less than the value of f, which contradicts the capacity condition $v(f) \leq C(A, B)$.

9 Complexity theory

This section is also known as: NP-computational intractability and the P-NP problem.

Recall that an algorithm is defined as efficient if it has a polynomial running time.

There are three properties of algorithms we have dealt with so far:

1. Provably correct
2. Deterministic
3. Worst case running time is bounded by a polynomial function of the input size

There are some algorithms that do not observe all three of those properties. Not observing the worst case running time is especially the property that is not observed.

For a certain class of problems, there are no known polynomial time algorithms to solve them. (In other words, no polynomial time algorithm has been found, yet there is also no proof that no polynomial time algorithms exist.)

A large class of problems have been characterized and has been proven to be equivalent in the following sense: a polynomial-time algorithm for any one of them would imply the existence of a polynomial-time algorithm for all of them. These problems are known as **NP-Complete** problems.

Complexity classes. We can categorize problems into specific complexity classes depending on what the fastest algorithm is to solve a problem, and whether it has been proven that there are no faster algorithms than what we know works.

Reduction is a technique that is key to understanding computational intractability. Let X and Y be two problems. $Y \leq_P X$ denotes Y can be reduced to X in polynomial time if and only if an arbitrary instance of problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a *black box* which solves X (i.e. how many times you call the black box). It also denotes that X is at least as hard as Y.

Explanation of the definition of reduction: to solve an instance of Y, you can do polynomial amount of work (regular kind of algorithm to create an instance of X) but you are able to call a black box that can solve instances of X.

The black box is sometimes known as the oracle (not a realistic model of computation). The oracle submits the question and receives the answer. Question must be asked in a yes or not format. This is what we call a decision version of a problem. Instead of returning the complete solution, we simply return whether a solution exists or not.

To clarify, the steps to take for reduction are:

1. Polynomial reduction (go from I_Y to $R(I_Y) = I_X$, in polynomial time)
2. Polynomial number of calls to black box of X

where I_Y is an arbitrary instance of Y and $R(\cdot)$ is a transformation.

When analyzing reduction, problems need to be recast as **decision problems** (yes/no classes) in order to be analyzed.

Decision problems are in contrast to optimization problems. Rather than looking for a minimum or maximum value, we are simply seeing if some property is true or false (i.e. when we make a call to the oracle, is it going to return true or false?).

Example for interval scheduling (decision problem). We are looking for the maximum of non-overlapping intervals. For this, we should define the decision problem as taking in $(intervals, value)$, and see if the value is the maximum number of intervals for this specific case or not. We need to see if there are only a polynomial number of queries made.

9.1 Independent set and vertex cover

These are two $\mathcal{NP}$-complete graph problems commonly used for reduction of other problems.

Independent set.

Problem definition: given a graph $G = (V, E)$, we say that a set of nodes $S \subseteq V$ are independent if no two nodes in S are joined by an edge (i.e. nodes in S are not adjacent).

Note that finding small independent sets in a graph is easy but finding the largest independent set is hard.

Goal: find the largest independent set. (i.e. find the maximum number of nodes such that no two nodes are joined by an edge)

We need to rephrase our problem as a decision problem so that it can communicate with the oracle.

Independent set, decision version. Does G contain an independent set of size at least k? (Note that our decision problem simply checks for the feasibility, not the optimality.)

If $k = 2$, then the answer is yes.

If $k = 4$, then the answer is no.

To illustrate the basic strategy for relating hard problems to one another, we consider another fundamental graph problem for which no efficient algorithm is known.

Vertex cover.

Problem definition. Given a set $G = (V, E)$, we say that a set has a vertex cover if every edge $e \in E$ has at least one end in S.

Note that finding the largest vertex cover in a graph is easy, but finding the smallest vertex cover set is hard.

Goal: Minimize the number of vertices used to cover E. (i.e. minimize the number of nodes which can successfully account for every edge in the graph)

Vertex cover, decision version: Does G contain a set of vertices of size at most k that touches all edges?

If $k = 3$, then the answer is yes.

If $k = 4$, then the answer is also yes.

Relationship between independent set and vertex cover. Let us use reduction between the independent set problem IS and the vertex cover problem VC. To see whether the following is true:

$$IS \leq_P VC$$

See if (1) I_{IS} has polynomial calls to $R(I_{IS})$ and (2) if the black box of VC is only called a polynomial number of times.

Lemma. Let G be a graph. Then S is an independent set if and only if $V - S$ is a vertex cover. (This lemma denotes the relationship between these two algorithms.)

- Forward direction: If we say S is an independent set, then we need to say $V - S$ is a vertex cover. Proof: Assume edge $e(u, v)$ is an arbitrary edge. Can both u and v be within S? No, because by the definition of an independent set, if S is an independent set, then u and v cannot be connected. So one of the edges must be in S and the other must be in $V - S$.
- Backwards direction: If we say $V - S$ is a vertex cover, then S is an independent set. Proof: Can both u and v be in $V - S$? By the definition of a vertex cover, they could be.

In order to argue a reduction between VC and IS, we need to come back to the definitions.

$$|VC| + |IS| = V$$

1. First, we need to argue that the reduction is polynomial.
   - Come up with an arbitrary instance of VC, a graph G and a target K.
   - Compute $K' = |V| - K$ (Taking the reduction is just one operation, so it is constant time.)
   - Call black box of VC
   - Yes for VC implies yes for IS
2. Second, we need to show that there are only a polynomial number of calls to the black box.

Lemma 8.1: Suppose $Y \leq_P X$. If X can be solved in polynomial time, then Y can also be solved in polynomial time.

This makes sense, since we have a polynomial number of steps in our reduction and a polynomial number of calls to X, then it stands to reason that for some P; $P^A \cdot P^B \cdot P^C = P^{A+B+C}$, where P^A is the time taken for the reduction, P^B is the number of calls to X, and P^C is the time taken for X to run **IF** X runs in polynomial time.

Lemma 8.2: Suppose $Y \leq_P X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time. (Contrapositive of Lemma 8.1)

Since $Y \leq_P X$ is equivalent to "X is at least as hard as Y", then it must be the case that if Y cannot be solved polynomially, then X cannot be solved.

Lemma 8.3: Let $G = (V, E)$ be a graph, then S is an independent set if and only if $V - S$ is a vertex cover.

9.2 SAT, the satisfiability problem

Given a set of boolean variables $X = x_1, x_2, \ldots, x_n$ and a formula comprised of clauses $F = c_1 \wedge c_2 \wedge \cdots \wedge c_m$. Each clause $c_i = (x_1 \vee \bar{x_2} \vee \cdots \vee \bar{x_n})$ must have disjunction of literals (not variables); it can only contain either x_i or $\bar{x_i}$ because having both would trivially make the clause c_i true.

For instance, $F = (x_1 \vee x_2 \vee \bar{x_3}) \wedge (x_4 \vee \bar{x_5} \vee x_{10} \vee x_{100})$.

The question is: can we find a satisfying assignment to make F true, by assigning boolean values to variables to make F true?

In this case, yes, of course. We could let the variable x_1 be true. Note the distinction between variables and literals. Making some variable x_i true would make the literal x_i true and the literal $\bar{x_i}$ false.

3-SAT. A variant of the SAT problem where each clause must have exactly 3 literals. (There is no restriction on the number of clauses.)

An example 3-SAT formula: $F = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x_1} \vee x_2 \vee x_{100}) \wedge \ldots$

Note that this does not limit the literals to being x_1, x_2, x_3; it only requires there to be 3 literals in each clause.

Reducing 3-SAT to Independent Set. We need to show that an arbitrary instance of 3-SAT, I_{3-SAT}, can be transformed into an instance of the independent set problem I_{IS} via a reduction transformation $R(I_{3-SAT})$.

We can build an independent set graphically. Let's therefore represent the 3-SAT problem with a graph, so we can represent an instance of 3-SAT in terms of independent sets.

$F = (x_1 \vee \bar{x_2} \vee x_3) \wedge (\bar{x_1} \vee \bar{x_2} \vee \bar{x_4}) \wedge (x_2)$

Let us convert each clause into a triangular graph where each literal is a node and the nodes of each clause are completely connected by edges. Then, place edges between nodes of the same variable that are of the *opposite* value. For instance, the x_1 literal in the first clause should connect to the $\bar{x_1}$ literal in the second clause.

Since we want F to be true, we need to somehow make each clause true. We need to guarantee that at least one of the variables within each clause will be true for the entire clause to be true. Then, we can place each node into as many independent sets as possible, and if every triangle group has at least one independent set (or in other words, if we have as many independent sets as the number of clauses), then we know 3-SAT to be true.

Claim. A 3-SAT formula is satisfiable if and only if the graph G has an independent set of size k, where k is the number of clauses.

1. Use the independent set graph representation to show 3-SAT.

- Forward direction: if we have 3-SAT, justify how we can get an independent set of size k. (Use proof by contradiction.)
- Reverse direction: if we have an independent set of size k, how can we get to a 3-SAT problem. (Again, use proof by contradiction.)

2. Check how many times we call the black box and justify it is only called a polynomial number of times. Our black box takes in the independent set and the k, and the black box returns yes or no. Since we only call this black box once in this problem, it is indeed polynomial.

9.3 Classes of complexity

Lemma (transitivity of reduction). If $Z \leq_P Y$ and $Y \leq_P X$, then $Z \leq_P X$. This can be easily proven by the fact that this would result in composition of functions.

Warning: $Y \leq_P X$ does NOT imply that $X \leq_P Y$.

Classes of complexity:

- $\mathcal{P}$: a class of problems that can be solved in polynomial time (solvable). The set of all decision problems which are solvable by an algorithm whose worst case running time is bounded by some polynomial function of the input size.
- $\mathcal{NP}$: the problems which we do not know if they can be solved in polynomial time (since we don't know whether $\mathcal{P} = \mathcal{NP}$), but can be *verified* using a polynomial time algorithm (verifiable). Verifiable means that given a certificate (i.e. a solution), we could certify (verify) that the certificate is correct in polynomial time.

Lemma. $\mathcal{P} \subseteq \mathcal{NP}$. Any problem that can be solved in polynomial time can also be verified in polynomial time.

Lemma. Is there a problem in $\mathcal{NP}$ that does not belong to $\mathcal{P}$? Does $\mathcal{P} = \mathcal{NP}$?

We don't know. The general belief is that $\mathcal{P} \neq \mathcal{NP}$, though there is no actual evidence to support this.

$\mathcal{NP}$ stands for non-deterministic polynomial (it does NOT mean "non-polynomial").

Since we do not know whether $\mathcal{P} = \mathcal{NP}$, we need to break down $\mathcal{NP}$ problems into two classes by the following question: what are the hardest problems in NP?

$\mathcal{NP}$-complete: a subclass of $\mathcal{NP}$, the hardest problems in $\mathcal{NP}$. A set of hard problems such that all other $\mathcal{NP}$ problems can be reduce to polynomial time. A problem X is $\mathcal{NP}$-complete if and only if:

- $X \in \mathcal{NP}$, which can be verified in polynomial time

- for all $Y \in \mathcal{NP}$, $Y \leq_P X$. (i.e. every problem in $\mathcal{NP}$ can be reduced to X; X is very expressive, powerful (another word for X is $\mathcal{NP}$-complete). If you have a problem $Y \in \mathcal{NP}$, then you can solve it using a black box for problem X.)

Another way to say X is $\mathcal{NP}$-complete is saying that X is powerful or expressive or "superhero" or some other similar adjective.

Corollary. If X and Y are both $\mathcal{NP}$-complete, then $X \leq_P Y$ and $Y \leq_P X$ (i.e. $X =_P Y$).

Lemma: Suppose X is an $\mathcal{NP}$-complete problem. Then X is solvable in polynomial time if and only if $\mathcal{P} = \mathcal{NP}$.

Proof:

Suppose that $\mathcal{P} = \mathcal{NP}$. Then X can be solved in polynomial time since $X \in \mathcal{NP}$. (Any $\mathcal{NP}$ problem would be reducible in polynomial time to any $\mathcal{NP}$-complete problem.)

Suppose X is solvable in arbitrary time. Given that X is also an $\mathcal{NP}$-complete problem, take an arbitrary problem $Y \in \mathcal{NP}$, then by definition $Y \leq_P X$. This means Y would also be able to be solved in polynomial time.

Consequence: If $\mathcal{P} \neq \mathcal{NP}$, then there are no polynomial time algorithms to find the solutions to $\mathcal{NP}$-complete problems.

This means these $\mathcal{NP}$-complete problems can **either be ALL solved in polynomial time (if $\mathcal{P} = \mathcal{NP}$) or NONE can be solved in polynomial time (if $\mathcal{P} \neq \mathcal{NP}$)**. Also, if there is one $\mathcal{NP}$-complete problem that can be shown to be not be able to be solved in polynomial time, then no other $\mathcal{NP}$-complete problem could be solved in polynomial time either.

$\mathcal{NP}$-hard: problems that are at least as hard as $\mathcal{NP}$ problems. They do not have to be $\mathcal{NP}$, nor do they have to be decision problems.

How to show that a problem $X \in \mathcal{NP}$-complete?

- Show that $X \in \mathcal{NP}$
- Choose a $\mathcal{NP}$-complete problem Y and show $Y \leq_P X$

Questions about ramifications:

- Is there an $\mathcal{NP}$ problem that is NOT in $\mathcal{P}$? Unknown. If so, $\mathcal{P} \neq \mathcal{NP}$.
- Is there an $\mathcal{NP}$ problem that is NOT $\mathcal{NP}$-complete? Graph isomorphism is not known to be in $\mathcal{NP}$-complete.

Corollary. $X \in Co - \mathcal{NP}$ if and only if its complementary problem $\bar{X}$ belongs to $\mathcal{NP}$.

e.g. is the SAT formula unsatisfiable?

i.e. the set of decision problems for which "no" answer has a polynomial time verifier.

Notes:

- If $Y \leq_P X$, that does not necessarily mean $X \leq_P Y$.

- e.g. interval scheduling $\leq_P$ weighted interval scheduling, but not the other way around.
- e.g. nurses scheduling $\leq_P$ independent set, but not the other way around.

9.4 Graph coloring

Given $G = (V, E)$, find the smallest number of different colors to assign to all nodes in V so that no two nodes with the same color share an edge. (i.e. find the chromatic number of a graph)

Lemma (review): A graph is 2-colorable if and only if it's a bipartite graph.

Theorem (review): A graph is bipartite if and only if it does not contain an odd length cycle.

Corollary: A graph's 2-colorability can be solved by running BFS on a graph.

Difficulty of colorability: 2-coloring is "easy", but 3-coloring and up are "hard" problems.

Graph coloring, decision version: Given a graph G and a bound k, does G have a k-coloring? ($\mathcal{O}(n^2)$ runtime to verify)

Claim: 3-coloring $\in \mathcal{NP}$-complete.

This can be proven by showing 3-coloring $\leq_P$ 3-SAT problem.

For each variable in the 3-SAT clause, we need to build two nodes: v_i and v_i', connected by an edge. Then, build three special nodes, B, T, and F, all connected to each other in a triangle. Think of them as the three colors, or true/false/don't care. Then, connect B to all of the v_i and v_i' nodes. v_1 and v_1' need to get different colors and need to get a different color than B, so assign v_1 to v_1' to T and F respectively - this means v_1 is set to true.

Our "gadget" will return yes or no, depending on whether the graph is 3-colorable or not.

10 Approximation algorithms

The gold standard of algorithms are ones that can solve the problem in an exact way. Specifically, ones that are:

- Correct
- Efficient
- Deterministic

So far, we have dealt with completely correct algorithms which were efficient (polynomial runtime) and deterministic (no random output).

$\mathcal{NP}$ problems relax the **efficiency** requirement. No polynomial algorithm has been found so far to solve such problems.

Rather than hoping for $\mathcal{P} = \mathcal{NP}$, in the meantime, we can create **approximate algorithms** by trading accuracy/correctness for efficiency. The end result

is an algorithm that is fast but not exact. We can guarantee such algorithms are *close* to optimal to some defined degree rather than being able to provide an exactly optimal output.

Challenge: Find a polynomial time algorithm for $\mathcal{NP}$-complete problems that is **close** to optimal. We need to show a solution's value is close to optimal without knowing the optimal.

To do so, we will use the concept of **bounding** to do this.

Load balancing problem. This problem can be solved by an approximation algorithm.

Assume you are given a set of jobs $n = \{2, 3, 4, 6, 2, 2\}$, where each entry is t_i, the processing time for job i. In this case $i = 0, 1, \ldots, 5$. We also have a few machines $M = \{m_1, m_2, m_3\}$. n is a list of jobs. M is a list of machines.

We need to place jobs on machines such that we minimize the **makespan** (maximum load on any machine).

Let's place jobs like so:

- m_1: $2, 6$
- m_2: $3, 2$
- m_3: $4, 2$

Define $A(i)$ to be the list of jobs assigned to machine i (for instance: $A(1) = \{2, 6\}$, the list of jobs on machine 1), and $T_i = \sum_i A(i)$ (for instance: $T_1 = 2 + 6 = 8$). Let $T_{max} = \max(T_i)$. Then we can rewrite that to be:

- $A(m_1)$: $\{2, 6\}$
- $A(m_2)$: $\{3, 2\}$
- $A(m_3)$: $\{4, 2\}$

Can we change the arrangements so that we can minimize T_{max}? Yes, let's try this:

- $A(m_1) = \{6\}$, $T_1 = 6$
- $A(m_2) = \{3, 2, 2\}$, $T_2 = 7$
- $A(m_3) = \{4, 2\}$, $T_3 = 6$

In this arrangement, $T_{max} = 7$, which is better.

Load balancing is $\mathcal{NP}$-complete. So let's find a polynomial solution that solves it approximately, but we need to prove about how far we are from an optimal solution.

For instance, we could find an algorithm that is off at most by 2 factors.

Greedy approximation algorithm (Algorithm 17)

Algorithm 17 Load balancing greedy approximation algorithm

1: Start with no jobs assigned
2: Set $T_i = 0$ and $A(i) = \emptyset$ for all m_1
3: **for** $j = 1, \ldots, n$ **do**:
4: Let m_i be a machine that achieves the minimum $\min_k T_k$
5: Assign job j to machine m_i
6: $A(i) \leftarrow A(i) \cup \{j\}$
7: $T_i \leftarrow T_i + t_j$
8: **end for**

Claim: This greedy algorithm is no more than 2 factors away from the optimal solution. (T: greedy solution, T^*: optimal solution). In other words:

$$T \leq 2T^*$$

We can provide the following bounds for the optimal case T^*:

- $T^* \geq \frac{1}{m} \sum t_j$ (the optimal T^* should be at least the average load per machine—the total jobs divided by the number of machines)
- $T^* \geq \max_j t_j$ (the optimal T^* should be at least the largest job)

Say we place job t_j on M_i, such that M_i now has the maximum makespan of all machines. T_i is the load on M_i after t_j is placed.

What was the load on M_i right before t_j was placed? $T_i - t_j$. From this, we can say the load on every *other* machine was at least $T_i - t_j$.

If we add up the load on all machines before t_j was placed (k jobs in total): $\sum_k t_k \geq m(T_i - t_j)$ because of the observations we made. Divide both of them by m and we get $\frac{1}{m} \sum_k t_k \geq T_i - t_j$.

We can use these conclusions to get the system of equations:

$$T_i - t_j \leq T^*$$
$$t_j \leq T^*$$

Adding the two equations together, they cancel out to get $T \leq 2T^*$, thus proving our 2 factors away bound.

Improving the bound. By sorting the jobs in descending order, we can reduce the bound down to $T \leq \frac{3}{2}T^*$.

Lemma. If there are m jobs each with the same time, then $T \geq 2t_{m+1}$ (eq. 1). This is because for t_{m+1}, we can choose any machine to add it on, and it cannot be greater than the largest job placed so far, because the jobs placed so far must be greater than or equal to job t_{m+1}.

We also know that the last job t_j placed after t_{m+1} must be less than t_{m+1}. Thus: $t_j \leq t_{m+1}$ (eq. 2).

Since $t_j \leq t_{m+1} \leq \frac{1}{2}T^*$, we get $t_j \leq \frac{1}{2}T^*$ (eq. 3).

Combining eq. 1 through 3, we get:

$$T \leq \frac{3}{2}T^*$$

10.1 Vertex cover as linear programming

There are several approximations for vertex covers. Some are simple, some are harder. An example of a difficult approximation for linear programming is provided below.

Vertex cover is $\mathcal{NP}$-complete, but linear programming can be solved in polynomial time. Vertex cover cannot itself be reduced to linear programming (that would imply $\mathcal{P} = \mathcal{NP}$). But vertex cover $\leq_P$ integer linear programming. It turns out integer linear programming is $\mathcal{NP}$-complete. Thus, we can take a vertex cover, translate it into integer linear programming, and then relax $\{0, 1\}$ discrete values into continuous values (regular linear programming).

Definition of linear programming. Given a $M \times N$ matrix A and vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^m$, find a vector $x \in \mathbb{R}^n$ to solve the following optimization problem: $\min(c^T x$ such that $x \geq 0$ and $Ax \geq b)$

(In this case, $c^T x$ is the optimization function.)

How do we transform a vertex cover problem into an instance of integer linear programming?

1. Decision variable: $x_i = 0$ if $i \notin VC$ and $x_i = 1$ if $i \in VC$
2. Formula for VC - we're looking for the minimum connection: $\min \sum_{i \in V} x_i$ (objective function)
3. Constraints: $x_1 + x_2 \geq 1$ (so we can ensure vertex cover for nodes x_1 and x_2; it's ok for both nodes to be in our VC set, but not ok if neither are in the set)

Relaxation. Allowing $0 \leq x_i \leq 1$ (i.e. fractional values).

Rounding. Round up if $x_i \geq \frac{1}{2}$, down otherwise. Feasible for $x_i + x_j \geq 1$. Optimality? What about approx value?

If we round up if $x_i \geq \frac{1}{2}$, then a solution can at most differ by $\frac{1}{2}$, so our relaxation at worst doubles each x_i.

A similar problem in Kleinberg and Tardos *Algorithm Design* works with the weighted VC problem, where nodes are chosen by their weights.

11 PSPACE

PSPACE is a complexity class that treats space as the fundamental limited computational resource. In other words, PSPACE is the set of all computational problems that can be solved by an algorithm with polynomial space complexity (i.e. an algorithm that uses an amount of space that is polynomial in the size of its input).

Lemma 9.1. $\mathcal{P} \subseteq$ PSPACE.

A Turing tape moves around left and right and will thus only consume a polynomial amount of space.

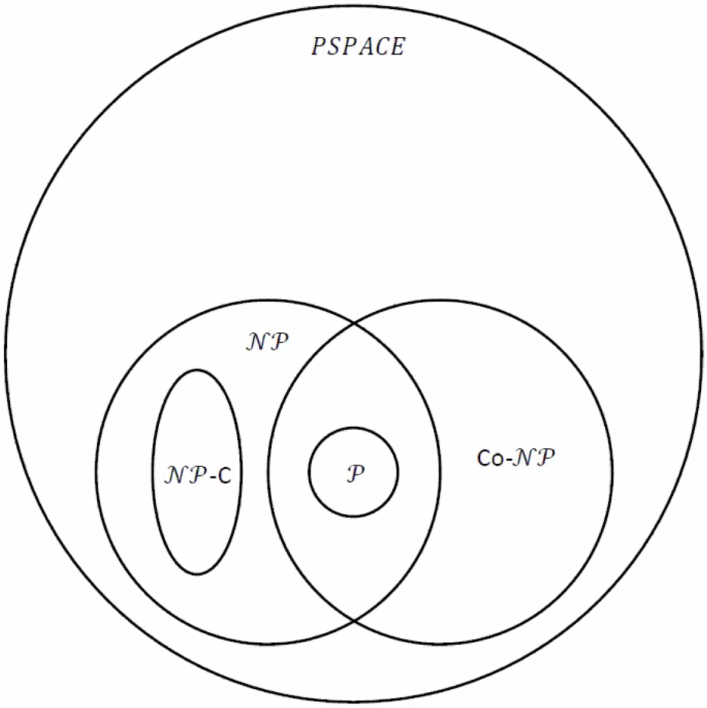
We don't know whether PSPACE $\subseteq \mathcal{P}$ yet.

Example. Let's say we are given a binary counter that counts from 0 to $2^n - 1$, where n is the number of bits of representation. If you have 2 bits, you can count from 0 to 3 (which is 4 numbers). Since our bound is $2^n - 1$, this is exponential relative to our input, the number of inputs. This means our time complexity is exponential. However, the space complexity is polynomial, because we are only dealing with n bits, which is a polynomial number. (We just need a vector that has n bits and reuses the space each time we do these exponential operations.)

Lemma. 3-SAT $\in$ PSPACE.

In 3-SAT, we can enumerate all 2^n possibilities using a counter. This counter iterates through our formula from x_1 to x_n, so there's only n amount of space used at any given time.

P, NP, NP-Complete, and Co-NP are all within PSPACE. It is still an open question whether $\mathcal{NP}$ = PSPACE, but we suspect it is not true. (Co-NP means negating the question: is it impossible to find a solution?)



An example of a problem in PSPACE but is not known to be in $\mathcal{NP}$ is called Quantified SAT or QSAT. This expands the definition from SAT, by adding $\forall$ and $\exists$ into the formula to quantify each variable x_1 through x_n. In SAT, if one variable is true in the clause, the entire clause is true. However, we can't say the same for QSAT. This makes the problem computationally more difficult, though the space used is still polynomial. How do we know that the space is polynomial? Let us create a graph. For each variable, we create a level in our tree, where the

nodes represent either AND or OR, and there's two edges extending from each non-leaf node: one representing true and the other representing false. Such a tree is used to determine whether a certain combination results in the overall formula evaluating as true or false. This tree is going to have approximately 2^{n+1} nodes (2^n leaf nodes and almost 2^n non-leaf nodes). But this tree is to calculate the result; despite the tree not being in polynomial space, the problem is still in PSPACE because we only need one vector, with a slot for each variable, which is definitely polynomial.

PSPACE-Complete. Problem $Y \in$ PSPACE-Complete if $Y \in$ PSPACE and for every problem $X \in$ PSPACE, $X \leq_P Y$.

3-SAT has been proven to be PSPACE-Complete.

A problem X is complete iff:

- $X \in$ PSPACE and
- choose a known PSPACE-Complete problem Y and show that $Y \leq_P X$.

12 Turing Machines, decidability, and halting problem

A Turing machine is an abstract model of computation invented by Alan Turing in 1936. A Turing Machine can do everything that a real computer can do. A TM has a **control** that contains a **state** and **transitions**. It also has a **tape** that has an end on the left side but infinite on the right side. The input is initially written on the tape but the TM can also write onto the tape (it's a read/write tape). The TM has a read/write head which is located over some particular character on the input tape. In a single move, the TM can choose to write or not, and can move left or right. The TM moves on the tape and can move bidirectionally.

Key features:

1. Contains an **infinite tape** that contains an input string and is blank everywhere.
2. Has a **read/write head**.
3. Has a **control**. Control represents the program in modern computers. (Note: the control is an example of a finite automaton, a state transition without a tape.)

To put it more abstractly, we are working with some language. The tool (our TM) takes in a string, and outputs whether the string is in the language or not in the language. We will show that this is potentially undecidable.

On the tape, each part of the tape either contains a 0, 1, or a cup (underscore $\sqcup$, denotes end of program/empty).

Theorem: No program can be written to take in a program and an input for this program, to determine whether the execution of this program will terminate or not. This is called undecidability.

Example: Creating a TM to recognize the language $B = \{0\ 1^*\ 0\}$. (i.e. Starts with only one zero, ends with only one zero, and has some amount of ONLY ones in the middle.)

Our control is comprised of states and transitions. For instance, we might start at q_0, and if the first bit is 0, we transition to q_1.

Let's create our control. Start at q_0. If we receive a 0, move tape right and go to state q_1. If we receive a 1, move tape wherever you want and go to state q_{reject} (terminate). From q_1, if 0, move tape wherever and go to state q_2, but if 1, move tape and loop q_1 to itself. From q_2, if cup/nothing else is the next symbol, change to state q_{accept}, but if there is either 0 or 1 after this, go to state q_{reject}.

Each time we read in this example, we will write back what was on the tape. Writing back is required, as it is the output of the program, even if it is the same thing.

Formal definitions for deterministic Turing Machines:

- Σ – input alphabet e.g. $\{0, 1\}$
- Q – the state e.g. $\{q_0, q_1, \ldots, q_n, q_{accept}, q_{reject}\}$
- Γ – tape alphabet e.g. $\{0, 1, \sqcup\}$
- δ – transition e.g. $\delta : Q \times \Gamma \implies Q \times \Gamma \times \{L, R\}$

Transitions for this example:

Read $0 \to R$ between q_0 and q_1 are written as $\delta : q_0 \times 0 \to q_1 \times R$

Write $0 \to \sqcup, R$ between q_0 and q_1 written as: $\delta : q_0 \times \sqcup \to q_1 \times \sqcup \times R$

A Turing Machine is a decider if it eventually always halts.

A language L is considered **recognizable** if the Turing Machine runs through it and for input $x \in L$, for any $x \in L$ halts and accepts and for any $x \notin L$ either runs forever or halts and rejects.

A language L is considered **decidable** if the Turing Machine runs through it and always halts eventually, and for input $x \in L$, for any $x \in L$ halts and accepts and for any $x \notin L$ only halts and rejects.

A Universal Turing Machine is a Turing Machine capable of simulating any other Turing Machine.

Let us have a language $A_{TM} = \{\langle M, w\rangle \| M \text{ is } TM \text{ that accepts } w\}$, where w is the input.

$\langle\rangle$ means description of.

$\langle M\rangle$ = comprised of state and transition (M is a Turing Machine).

Given a $TM(M)$, alphabet Σ, and w over Σ, does M always halt on w? (In other words, does w always end in a state of acceptance/rejection?)

Is there a TM that decides A_{TM}? No, since the halting problem is undecidable.

Proof by contradiction. Suppose there is a $TM(H)$ that can decide A_{TM}. H takes in $\langle M, w\rangle$, and outputs accept if M accepts w and reject if M does not accept w.

Now, construct a different Turing Machine $\langle D \rangle$ which takes in $\langle M \rangle$. The arbitrary input in D is the description of the machine, which is $\langle M, \langle M \rangle \rangle$. D contains H, and overall H will be used to make D's output. D accepts $\langle M \rangle$ if H rejects $\langle M, \langle M \rangle \rangle$, and D rejects $\langle M \rangle$ if H accepts $\langle M, \langle M \rangle \rangle$.

The last step is feeding $\langle D \rangle$ into itself; this means we have input $\langle D, \langle D \rangle \rangle$, and D accepts $\langle D \rangle$ if H rejects $\langle D, \langle D \rangle \rangle$, and D rejects $\langle D \rangle$ if H accepts $\langle D, \langle D \rangle \rangle$.

Now submit $\langle D, \langle D \rangle \rangle$ as input to H. H accepts $\langle D, \langle D \rangle \rangle$ if H accepts $\langle D \rangle$, and H rejects $\langle D, \langle D \rangle \rangle$ if H does not accept $\langle D \rangle$.

Transitively, this essentially is saying D rejects if D accepts. This is a contradiction. Hence, the Halting Problem is undecidable.