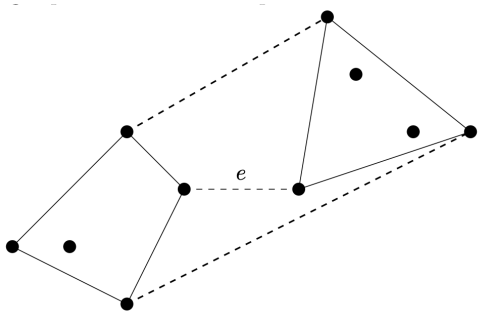
1 Introduction

Convex hull: Given n points in a plane, find the smallest convex set that contains all n points. The brute force $O(n^3)$ solution is to consider every pair of points, draw a line between them, then if the rest of the points are on one side of this line, add the line to the set of lines which will make up the convex hull. A better $O(n^2)$ solution is the Jarvis march: start with the leftmost point r, rotate a vertical line from r until it hits another point s, add the line (r, s) to the convex set, then rotate the line around s until it hits another point, add that line to the set, and repeat until we are back at r. In fact, Jarvis march is $O(nh)$, where h is the number of vertices of the convex hull. There is also a $O(n \log n)$ divide-and-conquer algorithm. Split the points into two halves based on x coordinate, find the convex hull of each half, then consider the edge e from the rightmost point on the left half to the leftmost point on the right half and rotate it up and down to create two bridges which will merge the two convex hulls into one. The recurrence here is $T(n) = 2T(n/2) + O(n)$. Any convex hull algorithm in a model where sorting numbers is $\Omega(n \log n)$ will be $\Omega(n \log n)$ because walking around the convex hull of the points $\{(a_1, a_1^2), (a_2, a_2^2), \ldots, (a_n, a_n^2)\}$ will sort the sequence $(a_1, \ldots, a_n)$. If the points are given sorted by x coordinate, we can achieve $O(n)$ using the Graham scan algorithm. Otherwise, we can achieve $O(n \log h)$, where h is the size of the convex hull using Chan's algorithm.



1.1 Definitions

An algorithm is a description of a process. An algorithm solves a problem if it returns a valid output for every possible input of the problem. The worst case time complexity of an algorithm is the maximum time complexity over all inputs of size n. Termination is an undecidable problem.

Asymptotics: $f(n) \in O(g(n))$ if $\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ such that $\forall n \geq n_0, f(n) \leq cg(n)$. Recall Ω, Θ as well from CS 240. And, $f(n) \in o(g(n))$ if $\forall c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}$ such that $\forall n \geq n_0, f(n) < cg(n)$.

Relationships:

1. $f(n) \in o(g(n)) \iff g(n) \in \omega(f(n))$
2. $f(n) \in o(g(n)) \implies f(n) \in O(g(n))$ and $f(n) \notin \Omega(g(n))$, and similarly for ω, Ω, O
3. If $f(n), g(n) > 0, \forall n \geq n_0$, then $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$ and $\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$. As well, $\max\{f(n), g(n)\} \in O(f(n) + g(n))$
4. $n^x \in o(a^n), \forall x > 0, a > 1$ and $(\log n)^x \in O(n^y), \forall x, y > 0$ and $\log(n!) \in \Theta(n \log n)$

Limits: If $f(n), g(n) > 0, \forall n \geq n_0$ and $L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists, then,

$$f(n) \in \begin{cases} o(g(n)), & \text{if } L = 0 \\ \Theta(g(n)), & \text{if } 0 < L < \infty \\ \omega(g(n)), & \text{if } L = \infty \end{cases}$$

However, it is possible that $f(n)$ is in one of the above orders of $g(n)$ but L does not exist.

Sums and series:

1. $\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2)$ if $d \neq 0$
2. $\sum_{i=0}^{n-1} ar^i = \begin{cases} a\frac{r^n-1}{r-1} \in \Theta(r^{n-1}) \text{ if } r > 1 \\ na \in \Theta(n) \text{ if } r = 1 \\ a\frac{1-r^n}{1-r} \in \Theta(1) \text{ if } 0 < r < 1 \end{cases}$
3. $\sum_{i=0}^{n} 2^i = 2^n - 1$
4. $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$ for $k \geq 0$
5. $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$

Math:

1. $\log(ac) = \log a + \log c$
2. $\log\left(\frac{a}{c}\right) = \log a - \log c$
3. $\log_b a = \frac{\log_c a}{\log_c b} = \frac{1}{\log_a b}$
4. $a^{\log_b c} = c^{\log_b a}$
5. $x^{-a} = \frac{1}{x^a}$
6. $\frac{d}{dx}\log_a x = \frac{1}{x \ln a}$
7. $\frac{d}{dx}a^x = a^x \ln a$

1.2 Models of Computation

Line cost model: Each execution of a line of an algorithm takes 1 time step. Inaccurate.

Bit cost model: Operations on a single bit take 1 time step. Accurate but too difficult to compute.

Word RAM model: On an input of size n, memory is broken up into words of length w, where typically $w = \lceil \log n \rceil$ (so that n fits into one word) and all elementary options such as read, write, add on words takes 1 time step. In general, the word RAM model will correspond exactly to the line cost model.

2 Reductions and Recurrences

2.1 Reductions

2SUM: Given an array of n integers and an integer m, return two indices i, j, not necessarily unique, such that $A[i] + A[j] = m$. In the word RAM model, every integer in A should fit into a single word in order to make conclusions about run time based off the number of comparisons and additions. The brute force algorithm is $O(n^2)$, or you can sort then either do binary search or a two pointer approach for $O(n \log n)$. Can be solved in $O(n)$ with high probability using a hash table but for whichever hash function is chosen, numbers can be chosen which do not work well with it. In those cases, $O(n)$ will not be reached.

3SUM: Same as 2SUM but returns three indices i, j, k. The brute force algorithm is $O(n^3)$. Alternatively, call 2SUM for each element in the array, which would be $O(n^2 \log n)$, assuming 2SUM is $O(n \log n)$. A better algorithm is to sort once first, then call 2SUM for each element, which would be $O(n \log n) + O(n^2) = O(n^2)$, since 2SUM on a sorted array is $O(n)$. Researchers have shown 3SUM can be solved in $o(n^2)$ but it is unknown whether it can be solved in $O(n^\gamma)$, for some $\gamma < 2$.

The idea behind reductions is to use known algorithms to solve new algorithms. For example, 2SUM can be reduced to binary search to find the second number, or 3SUM can be reduced to 2SUM. A reduction will involve calls to the unmodified, known algorithms. Correction proofs rely on the known algorithms to be correct. Recursion is a special case of reduction when the original problem is reduced to the same problem but on a smaller input.

2.2 Recurrences

Recurrences can be solved in several ways,

Recurrence tree: Draw a tree where the root is $T(n)$ and the leaves are the base case, usually $T(1)$. Then, sum up the time spent on every level of the tree. For example, for $T(n) = 2T(n/2) + O(n)$, the tree has height $\log n$ and $O(n)$ time is spent on each level. Thus, the total is $O(n \log n)$. Merge sort is the same asymptotically regardless of how many sub-arrays the original array is split into. For example, for $T(n) = 3T(n/3) + O(n)$, this is $O(n \log_3 n) = O(n \log n)$. If split into n arrays, each of size 1, $T(n) = nT(n/n) + O(n \log n) = O(n \log n)$, since k-way merging takes $O(n \log k)$ using a heap. Generally, assume n divides nicely (so the tree is full), unless otherwise specified. For example, if n is not a power of 2, we can take n' to be the smallest power of 2 larger than n. Since $n' \leq 2n$, then any proven asymptotic bounds for n' still hold for n.

Induction: Also known as the substitution method, consists of two steps,

1. Guess a tight upper bound on $T(n)$, ex: $T(n) \leq cn \log n$ for some constant c
2. Prove the guess is correct using strong induction, find the c

Typically a base case of $T(1) = 1$ is used but it does not matter. Inductive proofs also work well with floor and ceilings and do not require simplification, as the recurrence trees did. To be rigorous, sometimes it is useful to split induction steps into even n and odd n. Common mistakes with induction include,

1. Growing constants. If you want to prove $T(n) \leq cn$ but the induction step results in $T(n) \leq (c+1)n$, this is incorrect even though $c + 1$ is also a constant, because it is larger than the c from earlier.
2. Guesses which are not tight can still look right. The induction step will still work with a non-tight guess, so it is worth checking the guesses are as good as possible.

Using induction to prove a recurrence $T(n)$ is in $O(f(n))$ can be tricky. A good example to look at is A2Q1b. Key takeaways:

1. Make sure you prove enough base cases that your induction hypothesis can actually cover your induction step. In this question, we know that $T(1), T(2)$ do not work with the inequality so we want to prove that $T(n) \in O(n \log\log n)$ for all $n \geq 3$. But if we start the induction step with $n = 4$, then $\sqrt{4} = 2$, so we would not be able to apply the inequality as the induction hypothesis. Since 3 is the smallest n such that the inequality holds, we needed base cases up to $3^2 = 9$.
2. We can choose c ourselves. To satisfy the last inequality, we needed $c \geq 1$, that is okay!

Master Theorem: Solves recurrences of the form $T(n) = aT(n/b) + \Theta(n^c)$, for $a > 0, b > 1, c \geq 0$. The tree for $T(n)$ has height $\log_b(n)$, and summing up the work in each level,

$$\begin{aligned} T(n) &= n^c + a\Big(\frac{n}{b}\Big)^c + a^2\Big(\frac{n}{b^2}\Big)^c + \cdots + a^{\log_b n}\Big(\frac{n}{b^{\log_b n}}\Big)^c \\ &= n^c\left[1 + \frac{a}{b^c} + \Big(\frac{a}{b^c}\Big)^2 + \cdots + \Big(\frac{a}{b^c}\Big)^{\log_b n}\right] \\ &= n^c\left[\sum_{i=0}^{\log_b n}\Big(\frac{a}{b^c}\Big)^i\right] \end{aligned}$$

The value of this depends on the value of $\frac{a}{b^c}$, since this is a geometric series and simplifies to,

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } c > \log_b a \\ \Theta(n^c \log n) & \text{if } c = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } c < \log_b a \end{cases}$$

The general geometric series formula is quite useful,

$$\sum_{i=0}^{n-1} ar^i = \begin{cases} a\frac{r^n-1}{r-1} \in \Theta(r^{n-1}) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\frac{1-r^n}{1-r} \in \Theta(1) & \text{if } 0 < r < 1 \end{cases}$$

A useful technique for proving Ω bound is to let n_0 be the smallest value such that some inequality holds. For example, smallest value such that $r^{n_0} \leq \frac{1}{2}$.

Change of variables: Sometimes recurrences can be hard to work with but using a change of variables, can turn into an easier form. Ideally one solvable with Master Theorem. For example, $T(n) = 2T(\sqrt{n}) + \log n$. Using $m = \log n$, then taking $S(m) = T(2^m)$, this recurrence simplifies to $S(m) = 2S(m/2) + m = O(m \log m)$, which is solvable with Master Theorem. Then, substituting back for n, gives $O(\log n \log\log n)$.

3 Divide and Conquer

Divide and conquer algorithms come in three steps:

1. Divide: the original problem into one or more smaller subproblems
2. Conquer: each subproblem separately. Typically trivial and only involves recursive calls.
3. Combine: the results from the conquer stage back together. Typically the most difficult step but also may be trivial if the divide step produced only one subproblem, as is the case in binary search.

The correctness of a divide and conquer algorithm is usually proved by induction and the runtime is often determined by solving recurrence relations. Every recursive algorithm can be considered a divide and conquer algorithm, since the subproblems are getting smaller in each call. Often times, it is convenient to continually divide the input size by a constant (such as 2). In order to make division work nicely, define n' to be the smallest integer $\geq n$ that is a power of 2, then run the algorithm on n'. Since $n' \geq n \geq \frac{n}{2}$, the runtime in terms of n is still the same.

Merge sort: Split the array into two, sort each half, then merge together

Counting inversions: An inversion is a pair of indices (i, j) in an array A such that $i < j$ and $A[i] > A[j]$. The optimal $O(n \log n)$ solution involves a modified merge sort where the array is split into two, each half is sorted and the inversions within each half are counted, then the two arrays are merged together. In the process of merging, the inversions across the two half-arrays are counted. When an element from the right half is selected to be merged in, it must be smaller than all the remaining elements in the left half which have not been selected yet. So add the number of remaining elements in the left half to the running count. There are $O(\frac{n \log n}{\log \log n})$ and $O(n\sqrt{\log n})$ algorithms for this problem but they are outside the scope of this course.

Karatsuba multiplication: Given two n-bit positive integers x, y, output their product xy. The brute force algorithm involves grade-school multiplication techniques and is $O(n^2)$. Karatsuba multiplication makes use of the fact that you can split the bits of x into two halves, where x_L is the left half and x_R is the right half, then $x = 2^{(n/2)}x_L + x_R, y = 2^{(n/2)}y_L + y_R$. Re-arranging,

$$\begin{aligned} xy &= (2^{(n/2)}x_L + x_R)(2^{(n/2)}y_L + y_R) \\ &= 2^n x_L y_L + 2^{(n/2)}x_L y_R + 2^{(n/2)}x_R y_L + x_R y_R \end{aligned}$$

Note that multiplying something by 2^n or $2^{(n/2)}$ is $O(n)$ by bit shifting. The recurrence here is $T(n) = 4T(n/2) + O(n) = O(n^2)$. But since $(x_L + x_R)(y_L + y_R) = x_L y_L + x_L y_R + x_R y_L + x_R y_R$,

$$\begin{aligned} xy &= 2^n x_L y_L + 2^{(n/2)}x_L y_R + 2^{(n/2)}x_R y_L + x_R y_R \\ &= 2^n x_L y_L + 2^{(n/2)}((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R) + x_R y_R \end{aligned}$$

Thus, xy can be computed using three distinct multiplications of $n/2$ bit numbers. The recurrence becomes $T(n) = 3T(n/2) + O(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$.

k-piece Karatsuba: Instead of splitting the two n-bit integers x, y into two, what if they were split into k pieces, each of size n/k bits? $2k - 1$ multiplications are needed, so $T(n) = (2k-1)T(n/k) + O(n)$, or $T(n) \in O(n^{\log_k(2k-1)})$. This gets closer to $O(n)$ as k increases but the constant term gets very high and very complicated. Look at tutorial 3 for answer

Strassen matrix multiplication: Given two $n \times n$ matrices A, B, compute $C = AB$. The brute force algorithm is $O(n^3)$. Using block multiplication,

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

Then, each block of C is,

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$
$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$
$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$
$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

The recurrence is $T(n) = 8T(n/2) + O(n^2) = O(n^3)$, joining the blocks of C together is $O(n^2)$. This is not better than the brute force algorithm, but it is possible to do this with only 7 block multiplications, which will yield the recurrence $T(n) = 7T(n/2) + O(n^2) = O(n^{\log_2 7}) \approx O(n^{2.8})$. The specific multiplications are not important to look at in this course. There are better algorithms of approximately $O(n^{2.37})$ and it is conjectured an $O(n^2)$ algorithm exists.

Closest points: Given n points in 2D, find the min distance between any pair of points. The $O(n \log n)$ algorithm in 1D involves sorting then pairwise comparisons but this does not work in 2D. The brute force algorithm in 2D is to compare every pair in $O(n^2)$. The divide and conquer $\Theta(n \log^2 n)$ algorithm is:

Unique numbers: Given an array A of size n containing M unique numbers, where numbers are grouped together, such as $\{1, 1, 4, 4, 4, 2, 2, 9, 3, 3\}$, return the unique numbers in $O(M \log n)$.

Solution is to start at index 0, do binary search to find the index where the first number stops repeating. Then recursively call on this index (which has a new number). Recurrence is $T(M) \leq T(M-1) + \log n$, because as you throw stuff away the length to do binary search is less than n. This resolves to $O(M \log n)$.

Eddington number: See Assignment 2 question 2.

Min-weight interval for a point: See Assignment 2 question 3.

Fast fourier transform: See CS 370 notes.

4 Greedy

A greedy algorithm is one in which we:

1. Break down a problem into a sequence of decisions that need to be made
2. Make the decisions one at a time, each time choosing the option that is optimal at the moment and not worrying about later decisions

Typically the most difficult part is deciding how to select greedily. Correctness proofs can also be quite tricky and incorrect algorithms can seem correct. Common proof techniques:

1. Always ahead: show that at every point in the execution of the greedy algorithm, we can complete the partial solution we have so far into a valid solution for the original problem. Typically show the first decision does not disrupt later decisions, then use induction.
2. Exchange: given any optimal solution to the problem, we can convert it to the output of the greedy algorithm. Typically, it's easier to think about special cases where the input is of size 2 and argue why swapping the pair in a certain way yields a better solution. Then, use induction to apply the argument to the rest of the input, always doing one pairwise swap at a time from the optimal solution to the greedy solution.
3. Structural: does not involve induction and is a more direct proof. Might argue something like "by construction of the greedy algorithm, [this] holds, therefore [that]". An example is the proof for the interval coloring greedy algorithm.

Huffman codes and k-way merging are simple examples of greedy algorithms.

Interval scheduling: given n intervals of the form (s_i, f_i) where $s_i < f_i$, find a maximum subset such that no two intervals overlap. The greedy solution is to sort the intervals by their finish times, then repeatedly pick the interval which finishes first. The picking can be done in one pass through the sorted intervals since the next interval to be chosen is the first one which starts after the last one to be picked ended. This is $\Theta(n \log n)$ in total. The correctness proof is an "always ahead" argument. First, show that for any solution to this problem $I = \{i_1, \ldots, i_k\}$ sorted by finish time, you can replace i_1 with the first interval selected by the greedy algorithm (ie, the interval with the lowest finish time) to get another valid solution. Next, let I' be the set of intervals returned by the greedy algorithm and sorted by finish time. Do induction to show that the jth interval in I' can replace the jth interval in I to produce another valid solution. Then, conclude that the sizes of I', I are the same, so I' itself is also a valid solution so the greedy algorithm is correct.

Minimizing lateness: Given a set of n tasks with processing times $p_1, \ldots, p_n$, and deadlines $d_1, \ldots, d_n$, find an ordering such that the max lateness of any one task is minimized. The greedy algorithm is to sort the tasks by deadline then do the tasks with the earliest deadlines first. The proof of correctness is an "exchange" proof. Suppose you have an optimal solution S and a greedy solution G. G has the tasks sorted by deadline, so S must have some pair of tasks which are "out of order". That is, the first task has a later deadline than the second task. We prove that swapping these two tasks gives a smaller or equal max lateness. Then, by the correctness of Bubble sort, we can continue swapping these pairwise out of order tasks to give the greedy solution.

Interval coloring: given n intervals of the form (s_i, f_i) where $s_i < f_i$, find a coloring of the intervals such that the fewest colors are used and no two intervals which overlap have the same color. The greedy algorithm is to sort the intervals by start time, then for each interval, if there is a color c that was previously used and c is not used for any overlapping intervals (we can determine which colors are freed up by collecting all the colors used by intervals which are no longer overlapping that we've seen so far), assign c to the interval, otherwise assign a new color. This can be done in $O(n \log n)$ time using a priority queue. The proof is a structural one: suppose at most d intervals overlap at any point. Then, any valid coloring uses at least d colors. At the start point of any interval, at most $d - 1$ overlap with this start point, so the greedy algorithm

will either use the one color remaining if d were previously used, or choose a new color, to give a total of d. Since the minimum number of colors were used and no colors overlap, the greedy algorithm produces the optimal solution.

Fractional knapsack: given n items with positive weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack with max capacity W, find the amounts (weights) $x_1, \ldots, x_n$ of each item to fit in the knapsack that maximizes the total value $\sum_{i=1}^{n} \frac{x_i}{w_i} v_i$, where $0 \leq x_i \leq w_i$ and $\sum_{i=1}^{n} x_i \leq W$. The greedy algorithm is to sort the items by their "value-to-weight" ratios, $\frac{v_i}{w_i}$, in decreasing order then take as many of each item as you can in order. The proof of correctness is: Let $x_1, \ldots, x_n$ be the greedy solution, $y_1, \ldots, y_n$ be an optimal solution. Then, do induction on the number of differences between these two solutions. In the base case, there are 0 differences, so the greedy solution is the same as the optimal solution. Now, suppose there are $d \geq 1$ differences. Let $k \leq n$ be the smallest index such that $x_k \neq y_k$. Since the greedy solution maximizes x_k, then $x_k > y_k$. Since the sums of the weights are the same in every solution, then there is another index l where $l > k$ and $x_l < y_l$. Now, do an exchange where $y'_k = y_k + \delta, y'_l = y_l - \delta$, where $\delta = \min\{x_k - y_k, y_l - x_l\}$. Then, either $y'_k = x_k$ or $y'_l = x_l$, so by induction, since we exchanged more of a higher ratio item for a lower ratio item, this solution after exchanging is optimal.

Knapsack extension: what would happen if we allowed items with negative weights and values? Certainly, all items with negative weight and positive value should be taken and all items with positive weight and negative value should be discarded. What if we have an item i with negative weight w_i and negative value v_i? The solution is to subtract these weights from the total backpack capacity (thus increasing the capacity), then replace each item i with an item of weight $-w_i$ and $-v_i$ then do the above algorithm. Thus, these items have positive weight and value and taking it represents NOT taking the original item which had negative weight and value whereas not taking these items represents taking the original item. For example, if an item had weight -600 and value -3 and the other items had weight 200 and value 3 and weight 50 and value 1 and $W = 1$. Then, we consider the problem with $W = (1 - (-3)) = 4$ and the item now has weight 600 and value 3.

Offline caching (bonus):

5 Dynamic Programming

The idea is to break the problem into sub-problems, solve the sub-problems from smallest to largest, and store solutions along the way then use these solutions to solve the entire problem. Greedy also solves sub-problems but does so based on a simple local decision criterion whereas DP critically relies on previously done sub-problem to solve new problems. Two key attributes that indicate DP is applicable:

1. Optimal substructure: the optimal solution can be constructed from optimal solutions to its subproblems
2. Overlapping sub-problems: problems are broken down into sub-problems which are reused several times

DP solutions often arise naturally from recurrences. Eg, for fibonacci numbers, $f_n = f_{n-1} + f_{n-2}$, so the DP solution using an array (or even just two variables) is quite clear. Because of this recursive structure, you can often use induction to prove correctness. In general, the runtime to a DP algorithm is the number of sub-problems multiplied by the time to solve a single sub-problem.

Top-down dynamic programming (memoization) is recursive and starts from the original problem, checks whether the required sub problems have been computed (and compute them if not), then combines the answers into a solution to the original problem. In this way, memoization "lazily" computes sub problems because it only computes ones which are needed immediately. An advantage to top-down is that you might not need to solve all sub-problems but disadvantages are that it is harder to analyze the run-time and the recursion takes more overhead. On the other hand, bottom-up involves filling in a table from the base case up and solutions to sub-problems are guaranteed to have been solved when they are needed. The solutions discussed in CS 341 have been mostly bottom-up. The code for bottom-up looks completely different from top-down, even though both techniques are based on the same recurrence relation.

Text segmentation: Given a string s of n letters, and a function that returns in $O(1)$ time whether a given string of length m is a word (?), return whether s can be broken into valid words. There is no greedy solution, whether it repeatedly greedily takes the shortest valid word or the longest, from the beginning of s, works. The $O(n^2)$ DP solution is to construct an array A where $A[j]$ is if $s[1...j]$ can be split into words. Then, for each $k \in \{1, \ldots, n\}$, for each $j \in \{k-1, \ldots, 1\}$, if $A[j]$ is true, check if $s[j+1, k]$ is a word. Return $A[n]$ at the end.

Longest increasing subsequence: Given an array A of n numbers, return the length of the longest increasing subsequence. The $O(n^2)$ DP solution is to construct an array S where $S[j]$ is the length of the longest increasing subsequence that ends at $A[j]$ then for each $k \in \{1, \ldots n\}$, for each $j \in \{1, \ldots, k-1\}$ such that $A[j] < A[k]$, take $S[k] = S[j] = 1$. Return the max in S as the solution. There is also an $O(n \log n)$ DP solution, which makes use of binary search.

Longest common subsequence: Given two strings s, t, where s is length n and t is length m, return the length of the longest common subsequence. The $O(nm)$ solution involves an $n \times m$ DP matrix, where: $M[i][j] = \max(M[i-1][j], M[i][j-1])$, and max with $M[i-1][j-1] + 1$ if $s[i] = t[j]$. When the index is 0 (out of bounds), replace the number with a 0 since the length of the LCS between the empty string and any other string is 0. Then, we can extract the actual LCS using this matrix starting at the bottom right corner and working back up. Something interesting is that if we are given an array A of n numbers, copy A, then sort the copy, then find the LCS on A and the copied sorted array, this solves the longest increasing subsequence problem as well.

Edit distance: Given two strings s, t, where s is length n, t is length m, find the number of edit operations to transform s into t, where edit operations include adding a letter to s, deleting a letter, and replacing one letter with another. Note we are specifically keeping t constant and turning s into t. Transforming both strings until they are equal is equivalent. The $O(nm)$ DP solution involves an $n \times m$ matrix where $M[i][j]$ is the edit distance of $s[1...i], t[1...j]$. If $s[i] = t[j]$, then $M[i][j] = M[i-1][j-1]$. Otherwise, $M[i][j] = \min(M[i-1][j-1], M[i][j-1], M[i-1][j]) + 1$, where the respective operations are replacing $s[i]$ with $t[j]$, adding $t[j]$ to s right before $s[i]$, and deleting $s[i]$. The second case of adding a letter is quite

interesting because once we add $t[j]$ to s right before $s[i]$, then we can implicitly match this new letter with $t[j]$, so we now only need to consider $s[1...i]$ and $t[1...j-1]$ since $t[j]$ has been processed. If any indices go out of bounds (meaning one string is empty), then the edit distance is the length of the other string, since the edit distance between the empty string and any string of length r is r. Again, we can trace back the edit operations backwards starting from the bottom right in this matrix.

Weighted interval scheduling: Given a set of n intervals with start finish times $(s_i, f_i), 1 \leq i \leq n$ and positive weights $w_1, \ldots, w_n$, return a subset of intervals such that none overlap and the total weights is maximized. The $O(n \log n)$ DP solution is to first sort the intervals by finish time, then construct an array W where $W[j]$ is the max weight of a subset of non-overlapping intervals from the sorted intervals $I_1, \ldots, I_j$ (and $W[0] = 0$). Then, if we do not include I_j, $W[j] = W[j-1]$, if we do include I_j, $W[j] = W[k] + w_j$, where k is the interval with the largest finish time that is less than or equal to s_j. This can be found using binary search since the intervals are now sorted by finish time. Take the max of these two options. W contains total weights but at the end, we can trace back in W to find the actual subset of non-overlapping intervals which maximizes the total weight. As well, we can pre-compute $p(j)$, where p is a function that maps interval I_j to the interval which finishes as late as possible which not overlapping with I_j. This is equivalent to what we were trying to find with binary search, but $p(j)$ can be pre-computed in $O(n)$ for all $1 \leq j \leq n$, assuming we sort by both start and finish times separately first. Overall, still $O(n \log n)$.

Optimum BST: Given items $1, \ldots, n$ and corresponding weights $w_1, \ldots, w_n$, construct a BST that minimizes the search cost $\sum_{i=1}^{n} p_i \cdot \text{depth}_T(i)$. Recall the depth of a node is its height plus 1 (root has depth 1). Note that the Huffman codes greedy algorithm solves this problem, but does not necessarily create a BST. The $O(n^3)$ DP solution involves first sorting the items by increasing order then maintaining a $n \times n$ matrix where $M[i][j]$ is the optimum BST for items $i, \ldots, j$. To compute the optimum BST for items $i, \ldots, j$, we want k such that $i \leq k \leq j$ and $M[i][k-1] + M[k+1][j]$ is minimal. Then, $M[i][j]$ will equal this minimal value plus $\sum_{l=i}^{j} w_l$, because you are adding one to the depth of each node in the left and right subtrees and also creating a new root of depth 1. The most important takeaway of this question is the order of iteration. When computing $M[i][j]$, we need to make sure $M[i][k-1]$ and $M[k+1][j]$ is available, for all $i \leq k \leq j$. In order to achieve this, we need a third variable d to iterate from $1, \ldots, n-1$ and for each iteration, iterate i from $1, \ldots, n-1$, then set $j = i + d$. We want to iterate in order of $j - i$. First, solve find the optimum BSTs for 1 item, then 2 items, etc. Also note the additional weights $\sum_{l=i}^{j} w_l$ can be computed in $O(1)$ by first pre-computing an array A where index j stores $\sum_{l=1}^{j} w_l$, then $\sum_{l=i}^{j} w_l = A[j] - A[i]$.

0/1 Knapsack: Same as the fractional knapsack problem explored above in the greedy section, but each item is either selected or not selected. You cannot choose a fraction of an item. The $O(nW)$ solution is to construct a $n \times W$ matrix where $M[i][j]$ is the max value obtained by considering items $1, \ldots, i$ with a backpack of total weight j. Thus, each item i can either be added or not added, so $M[i][j] = \max(M[i-1][j], M[i-1][j-w_i]+v_i)$. When indices go out of bounds, it is equivalent to value 0. For example, if $w_i > j$, then it is not possible to have item i in a backpack of capacity j anyway. Iterate from $i = 1, \ldots, n$ and for each i, iterate from $j = 1, \ldots, W$. Again, if we preserve M, we can start at the bottom right corner and trace back up to find the exact items to put into our backpack of total weight W. $O(nW)$ is not polynomial because by "polynomial", we actually mean polynomial to the size of the input. In this case, the "input" is the number of bits to represent the problem, which is $n \log W$. Since nW is not polynomial to $n \log W$, then $O(nW)$ is not polynomial. However, notice $O(n \log W)$ is polynomial. The subset sum problem (see later) is very similar to this one.

Some common DP sub-problems for inputs $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$:

1. Consider $x_1, \ldots, x_i$, for all $1 \leq i \leq n$. For example, weighted interval scheduling.
2. Consider $x_i, \ldots, x_j$, for all $1 \leq i \leq j \leq n$. For example, optimum BST.
3. Consider $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$ for all $1 \leq i \leq j \leq n$. For example, edit distance.

Other classic problems: max rectangle under a histogram, shortest common supersequence, rod cutting.

6 Graphs

For some strange reason, these definitions are not the same as in CO 342, so be careful.

Definitions: A graph is a pair of vertices and edges, $G = (V, E)$, where the vertices are often called nodes and $E \subseteq V \times V$. By convention, $|V| = n, |E| = m$. An undirected/unordered graph is one with undirected edges, a directed/ordered graph is one with directed edges. Generally, assume graphs are undirected unless otherwise specified. Two vertices in an undirected graph are adjacent if there is an edge between them. A vertex is incident with an edge if it is one of it's endpoints. In directed graphs, the in-degree of vertex v is the number of edges coming into v. Opposite for out-degree. A vertex with in-degree 0 is a source vertex, a vertex with out-degree 0 is a sink vertex.

A path is a sequence of vertices and edges, where the vertices/edges possibly repeat. A simple path is a path with no repeats, except possibly at the start and end. A cycle is a simple path that starts and ends at the same vertex with at least three vertices. An undirected graph is connected iff every two vertices is connected by a path, a tree is a connected graph with no cycle, and a connected component is a maximal connected subgraph. Directed graphs have a similar notion in strong connectedness (described later). Loops and parallel edges are not allowed in this course.

Representations: An adjacency matrix is an $n \times n$ matrix where $A[i][j] = 1$ iff there is an edge from vertex i to j. Takes $O(n^2)$ space but can check adjacency between any two vertices in $O(1)$. An adjacency list is a set of n linked lists, each containing the neighbours of the node. In a directed graph, an edge from u to v is represented with v being in u's adjacency list. Takes $O(n + m)$ space, requires $O(n)$ to check adjacency between any two vertices, but can list all the neighbours of v in $O(\deg(v))$ whereas this would take $O(n)$ in an adjacency matrix. Instead of linked lists, if we had sophisticated hashing, we could represent adjacency using n hash tables which would allow us to check adjacency between any two vertices in $O(1)$ as well. In this course, assume we are using adjacency lists unless otherwise specified.

6.1 Breadth First Search (BFS)

Implemented with a queue and a status array (or status flag on each node) that indicates whether the node has already been visited. Time complexity is $O(n + m)$ since every vertex is pushed into the queue at most once and every adjacency list is visited at most once and $O(1)$ work is spent for each element of those lists. Sometimes it can be hard to keep track of which nodes have been visited. Something that may help is distinguishing between three states: undiscovered, discovered, and explored. Initially, all vertices are undiscovered, except the start vertex which is discovered. At any time in the BFS, the vertices in the queue are exactly those who are considered to be "discovered". When all neighbours have been visited (and pushed into the queue), the current vertex is marked as explored. BFS only pushes undiscovered vertices into the queue. The explored status is not used in the base BFS algorithm but may be useful in some other algorithms which build on BFS. Overall, vertices transition from undiscovered to discovered then discovered to explored then they are never considered again.

Single-source shortest path: Given a node in an unweighted graph, find the distance of the shortest path from this node to every other node in the graph. Apply BFS and use a separate counting array to keep track of the distances, where every value is initialized to ∞ except the distance for the start vertex, which is initialized to 0. Then, when about to push a new vertex onto the queue, update its distance to be one more than the distance of the current vertex whose neighbours we are iterating over. Can prove correctness using induction on d, where d is the distance using the fact that at some point in the algorithm, all nodes in the queue are distance d from the source, all nodes with greater distance have not yet been reached (thus distance ∞), and all nodes with distance $\leq d$ have already been processed. By the end of this algorithm, if the distance for a vertex is still ∞, then it is not connected to the start vertex. Also, if there is an edge between u and v, then the distances of u, v differ by 0 or 1.

Bipartiteness testing: Given a connected graph, determine if it is bipartite. Apply BFS, maintain a set which represents the vertices in A of the bipartition (A, B), where the start vertex is put into A, then as vertices are put into the queue, put them into A if the original vertex whose neighbours we are iterating over is not in A and vice versa. If a neighbour has been discovered/explored already and is in A and the original vertex is also in A (or both are not in A), then return false. To generalize this for possibly disconnected graphs, we can just test bipartiteness on all the components.

Spanning tree: Given a connected graph, output a spanning tree. Apply BFS. We can represent a rooted directed tree using an array of size n where the entry at index i is the parent to node i in the tree and this entry is null for the parent. Simply update this parent array when putting an undiscovered vertex into the queue.

6.2 Depth First Search (DFS)

In DFS, when a new vertex is discovered, all incident edges are explored before moving onto another vertex. The algorithm is the same as BFS except with a stack and visited array/flag instead of a queue and status array/flag. Could also use the three statuses (discovered/undiscovered/explored) but often visited/not visited is enough). Also, DFS can be implemented naturally using recursion, where the call stack replaces the stack data structure. The recursive structure looks like,

```
EXPLORE(G = (V, E), cur, parent):
    // initialized to false for all vertices, parent of start is false
    visited[cur] = True
    PreVisit(cur, p)
    for each neighbour w of cur:
        // cur is the parent of w in the DFS spanning tree
        if not visited[w] then EXPLORE(G, w, cur)
        // else, either w is a parent of cur or (cur, w) is a non-tree
        // edge in the DFS spanning tree
    PostVisit(cur, p)
```

In `Pre/PostVisit`, we can do things which are specific to the algorithm, such as printing the vertex, etc. In particular, we define the *pre*, *post* arrays for each vertex (referenced later), computed as follows using a global time counter which starts at 0,

```
PreVisit(v, p):
    pre[v] = clock
    clock++
```

And *post* is computed identically in `PostVisit`. The interval between *pre* and *post* now represents the time when the vertex's DFS call is on the stack. Lastly, unlike BFS, DFS will not solve the SSP problem (consider a triangle), unless vertices are visited more than once, possibly many more times than once, and this is messy. However, similar to BFS, DFS from a start vertex will visit every other vertex in its connected component. Runtime is also $O(n + m)$ by identical analysis to BFS.

Spanning tree: In `PreVisit(v,p)`, if p is not null (meaning that v is the start vertex), add the edge from v to p into the spanning tree calculated so far. Let T be the edges given by BFS on G and call edges in T tree edges and call the remaining edges in G non-tree edges. Also, when a vertex v is in the subtree rooted at w in T, call v a descendant of w and w an ancestor of v. T has can give interesting insights on the structural properties of G that cannot be found by the spanning tree from BFS. For example, every non-tree edge in a connected graph G connects an ancestor to one of its descendants. In other words, there are no non-tree edges which go horizontally "across" the spanning tree T. This is because for any vertex v, all unexplored vertices that can be reached by a path from v of only unexplored vertices end up as descendants of v in T

because they are examined before v finishes. So if there is a non-tree edge (u, v), either v is a descendant of u or vice versa.

Finding cut vertices: A cut vertex in a connected graph G is one where the graph with it removed G' is disconnected and a connected graph with no cut vertex is 2-connected (these are the same definitions as in CO 342). From the insight above regarding non-tree edges in the DFS tree T, we see that a vertex v is a cut vertex of G iff it satisfies either of these two conditions,

1. v is the root of T and has at least two children. If v has at most one child, then removing it leaves a spanning tree, so G' is connected. Otherwise, there is no edge between a vertex in the left subtree and a vertex in the right subtree, so v is a cut vertex.
2. v is not the root of T and has a child u such that the subtree of T rooted at u has no non-tree edge from one of its vertices to an ancestor of u. So, removing v isolates u from the ancestors of v and if v is a cut vertex then it isolates some vertices from the rest of the graph so one of its subtrees must be disconnected from the rest of the tree.

After T is computed, condition 1 is easy to check. Condition 2 can be checked using the *pre* times and a separate array that keeps track of the minimum time that any of its ancestors was reached. If a vertex is found to have a minimum time greater than or equal to the time that its parent was visited, then the parent is a cut vertex. The minimum time represents the furthest back up to the root that any non-tree edge or tree edge reaches from the current vertex. So if the furthest back up is below or equal to the parent, then the parent is a cut vertex.

6.3 DFS in Directed Graphs

Unlike in an undirected graph, DFS on any vertex might not give a spanning tree. Instead, repeatedly run DFS starting from an unvisited vertex and only visiting unvisited vertices. This will give a DFS forest and still takes $O(n + m)$ time since each vertex is visited at most once. Define tree edges, non-tree edges, ancestors, descendants as before and let forward edges be non-tree edges from an ancestor to a descendant, back edges be non-tree edges from a descendant to an ancestor, and cross edges to be non-tree edges from a vertex to another that is neither its ancestor nor descendant (possibly in different DFS trees of the DFS forest or within the same one). Recall cross edges do not exist in the DFS tree for undirected graphs.

Using the time-stamped `Pre/PostVisit` definitions, the DFS structure looks like,

```
EXPLORE(G = (V, E), cur, parent):
    visited[cur] = True
    PreVisit(cur, p)
    for each neighbour u of cur:
        if not visited[u] then EXPLORE(G, u, cur)
        // else, (cur, u) is a non-tree edge, consider three possibilities:
        // if PostVisit has not yet been called on u, (cur, u) is a back edge
        // else if pre(u) > pre(cur) then (cur, u) is a forward edge
        // else if pre(u) < pre(cur) then (cur, u) is a cross edge
    PostVisit(cur, p)
```

Directed acyclic graph: A cycle in a directed graph must respect the direction of the edges and possibly has length 2 (if there are edges in both directions between two vertices). A directed acyclic graph (DAG) has no such cycles and appear in many different contexts.

DAG detection: A directed graph has a cycle iff its DFS tree has a back edge. Clearly if there is a back edge, there is a cycle. If there is a cycle $v_0, v_1 \ldots, v_k, v_0$, where v_0 is the first vertex in the cycle to be discovered by the DFS, then each of $v_1, \ldots, v_k$ is a descendant of v_0 since the subtree rooted at v_0 includes

all vertices reachable from v_0 in G using only not-yet-visited vertices so the edge from v_k to v_0 is a back edge. We can determine whether an edge is a back edge using the *post* values (see above).

Topological sorting of DAGs: A directed graph can naturally correspond to a dependencies between tasks (vertices), where an edge from u to v means that u is a task that must be done before v. In a DAG, there are no circular dependencies. The goal of topological sort (which is possible iff the graph is a DAG) is to linearize the graph; give an ordering $v_1, \ldots, v_n$ of all the vertices such that for every directed edge v_i to v_j, $i < j$. This gives an ordering of the tasks such that a task is completed only when all the ones it depends on have been completed already as well. One way to do this is to find a source vertex (recall: a vertex with in-degree 0), remove it, and repeat (see Kahn's algorithm). This takes $O(n + m)$. Another method is to construct the DFS forest as described above then put the vertices in reverse order according to their *post* values (in `PostVisit`, add the node to a stack which will give the correct reverse order). This works because for each edge from u to v, $post[u] > post[v]$, since `PostVisit` is only called for u when all vertices reachable from u (including v) are explored. The sorted order can also be built on the go in a modified `PostVisit`. Since DFS is $O(n + m)$, so is this algorithm. Note that the first vertex given by the topological sort is a source vertex and the last vertex is a sink vertex.

Testing strong connectivity: A directed graph G is strongly connected if for every pair of vertices u, v, there is a path from u to v and a path from v to u. Alternatively, for a fixed vertex s, G is strongly connected iff for every v, there is a path from s to v and from v to s. So, running DFS from s gives the vertices reachable from s and running DFS on the reverse graph (every edge direction is reversed) from s gives the vertices which reach s. Both these sets of vertices contain every vertex iff G is strongly connected.

Finding strongly connected components: As in undirected graphs, directed graphs can be partitioned into strongly connected components (a maximal, strongly connected set of vertices), possibly with edges betwee components. Contracting every edge in each of these components gives an acyclic graph (or else if there were a cycle, the strongly connected components would be larger, a contradiction). Modifying the strong connectivity test to find strongly connected components is not as simple as modifying the connectivity test to find connected components in undirected graphs. The $O(n + m)$ algorithm (Kosaraju) is: run DFS on G and record the *post* finish times $f_1, \ldots, f_n$, then run DFS on the reverse graph with vertex order $f_n, f_{n-1}, \ldots, f_1$ to construct the DFS forest. The trees in this forest are exactly the strongly connected components. Alternatively, prepend the nodes to a list/push it onto a stack rather than record the *post* finish times. This prevents having to sort the *post* times in reverse order. For the proof, consider two strongly connected components C_1, C_2 with an edge from some vertex in C_1 to some vertex in C_2 (and no edge can exist in the other direction). Regardless of where the first DFS starts, the last vertex v to have its *post* time visited is in C_1. In the reverse graph, when DFS starts on v, no vertex in C_2 can be reached since the edge between the components now goes from C_2 to C_1 and v is in C_1. Since each of C_1, C_2 are strongly connected, this gives two trees, one for each of C_1, C_2, as expected. Using this fact, the proof can be done by induction on the number of strongly connected components.

6.4 Minimum Spanning Trees (MST)

Recall: a spanning tree T of a connected undirected graph $G = (V, E)$ is a tree with vertex set V and $n - 1$ edges of E. T gives a skeleton representation of G and can be used to find a path between any two vertices. T can also be stored in $\Theta(n)$ space, even though G might have $\Omega(n^2)$ edges in total.

A weighted graph (specifically, an edge-weighted graph) is one with a weight function $w : E \to \mathbb{R}$ on the edges. A MST of a weighted graph G is a spanning tree with minimum total edge weight and represents the "cheapest" cost to connect all the vertices of G.

Kruskal's algorithm: Sort the edges by increasing weight (lightest edge comes first). Then, for each edge, if its two ends are in different components, add the edge to the MST built so far. The overall runtime is $O(m \log m)$ which comes from the time to sort the edges. An efficient method for determining if two vertices

are in different connected components is required and can be done with the Union Find data structure (covered in CS 466).

We define a cut in a graph to be a partition of the vertices V into two sets $S, V \backslash S$ and an edge crosses the cut if one of its ends is in S and the other in $V \backslash S$. S can possibly be V but then no vertex crosses this cut. To prove correctness of Kruskal's, we first prove the cut property: if G is a connected graph and $X \subseteq E$ are edges in some MST and $S \subseteq V$ such that no edge in X crosses the cut $(S, V \backslash S)$ and e is the lightest edge that crosses this cut, then $X \cup \{e\}$ is part of a MST of G. The proof of this is to consider a MST T which has all the edges in X. If $e \notin E(T)$, then adding e gives a cycle containing e. Since e crosses the cut $(S, V \backslash S)$, there must be some other edge e' which crosses this cut in this cycle. Since no edge in X crosses the cut, $e' \notin X$; removing e' from T gives an MST which contains e, since e is the lightest edge that crosses this cut. In fact, it must be that $w(e) = w(e')$. The correctness for Kruskal's uses induction on this cut property, where X is the set of edges in the MST so far at each step and e is the edge which the algorithm finds and adds.

Prim's algorithm: Start the MST at a single vertex and grow the tree outwards by finding the lightest edge that connects a vertex in the tree to a vertex not in the tree. The proof follows from the cut property, again by induction on the edges in the tree so far (as X). The simple algorithm uses a min priority queue. Initially, if the MST starts at vertex v, the pq contains all edges incident with v. For each vertex u which gets added to the MST, add all edges incident with u. All edges in the pq are guaranteed to have one end in the MST but some have both ends in the MST and are not useful. So, at each step when querying the min value in the pq, if both ends are in the MST, throw it away and query again. Each edge is added twice to the pq (once for each end) and removed twice. Overall, this is $O(m \log m)$. However, with a Fibonacci heap, the final time complexity can be reduced to $O(m + n \log n)$. One thing to note is that since $m \in O(n^2)$, $\log m \in O(\log(n^2)) = O(\log n)$. So in the context of graphs, a logarithmic factor on m is the same as a logarithmic factor on n.

These are both greedy algorithms. The greedy selection is in choosing the lightest available edge to add to the tree at each step. Kruskal's algorithm chooses the edge such that no cycle is created (connects two different components) and Prim's algorithm chooses the edge with exactly one end in the tree built so far. One more correct algorithm is to start with the entire graph and greedily remove the heaviest edge such that the result is still connected.

Since Kruskal's algorithm is $O(m \log m) = O(m \log n)$ and Prim's can be implemented in $O(m + n \log n)$, it is a good idea to use Prim's when the graph has lots of edges. However, the typical graph is quite sparse and Kruskal's performs better because the data structure it uses is simpler (Union Find versus Fibonacci heap).

6.5 Single Source Shortest Path (SSSP)

In an unweighted graph (or graph where every edge has the same positive weight), BFS will solve the SSSP problem in $O(n + m)$. Now, we will see how SSSP can be solved in a weighted (directed or undirected) graph G, for a source vertex s. A valid solution is the minimum total path weight in G from s to any vertex $v \in V$.

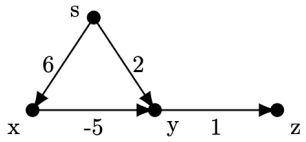
For small integer weights: If the weights are such that they are positive and bounded by a reasonably small integer c, then we construct G' by adding additional vertices and edges such that for any edge of weight k in G, $k - 1$ extra vertices are added along the edge and the edge can now be considered as split into k smaller edges, each with weight 1. BFS on G' will solve the problem in G. This works because the weights are "encoded" as distance in the form of additional vertices along a single edge. The runtime is $O(m' + n')$, where m', n' are the number of vertices and edges in G'. Since c is reasonably small, this is still roughly $O(m + n)$ but the complexity grows very quickly for larger weights.

Dijkstra: Suppose the weights are non-negative (and not necessarily integers). This algorithm greedily explores the vertex that is closest to the source s among all vertices that have not been visited yet. Implemented with a min priority queue,

```
dist[s] = 0
dist[v] = infinity for all other vertices v
PQ = min pq containing every vertex with key dist
while H is not empty:
    u = DeleteMin(PQ)
    for all neighbours v of u:
        if dist[u] + w(u, v) < dist[v]:
            dist[v] = dist[u] + w(u, v)
            DecreaseKey(H, v, dist[v])
return dist
```

Here, `DecreaseKey` bubbles the v entry up in the priority queue to its rightful spot (swap with parent while the parent has a smaller key). The reason why greedy selection works is because all weights are non-negative so adding more edges to a path cannot improve the total weight. So, the shortest path from s to v cannot go through a vertex u which is further away from s than v. Note that s will be the first element removed from PQ because every other vertex has initial distance ∞. The correctness of this algorithm can be proven by proving that $dist[v]$ is correct for every v not in PQ by induction on the number of vertices not in PQ (or, the number of visited vertices). The base case is when only s is removed from PQ. For the induction step, let u be the vertex removed from PQ; we wish to prove that $dist[u] = \min_{v \notin H}\{dist[v] + w(v, u)\}$. Let v^* be the vertex which attains this minimum. The upper bound is clear, since by the induction hypothesis, $dist[v^*]$ is correct and the path from s to v^* to u is a path from s to u. The lower bound uses the fact that every path P from s to u has some edge (x, y) where $x \notin H, y \in H$ since $s \notin H, u \in H$. By the induction hypothesis, $dist[u] \geq dist[x] + w(x, y)$. Also, since u was the next vertex to be removed from PQ, $dist[x] + w(x, y) \geq dist[v^*] + w(v^*, u)$ (because otherwise, $dist[y] < dist[u]$ and y would have been removed from PQ instead of u), so the lower bound holds. At each step, note that $dist[u]$ is the minimum weight path from s to u with every vertex besides u already visited (not in PQ).

The runtime is $\Theta(mT_1 + nT_2)$, where T_1 is the time for `DecreaseKey`, T_2 is the time for `DeleteMin`. Using a normal binary heap, both are $\log n$, so the runtime is $\Theta((m + n) \log n) = \Theta(m \log n)$. This can be improved to $\Theta(m + n \log n)$ if using a Fibonacci heap. This algorithm works on both undirected or directed graphs but does not work if there are negative weight edges as seen here,



The algorithm will give $dist[y] = 2$ but the actual shortest path has weight 1; Dijkstra's fails because after greedily selecting a vertex, it is considered visited and removed from contention but a path that originally is heavy then has a very negative edge might have given a lower weight path.

Bellman-Ford: Negative weight cycles are problematic because taking the cycle infinitely will give an unboundedly low total weight path, since paths can have repeat vertices in this course. Suppose the graph does not have negative weight cycles (but possibly has negative weight edges). Perhaps we can modify the graph by adding α to the weight of every edge, where α is the minimum weight edge to obtain a graph with non-negative edges then use Dijkstra's algorithm but this does not work because it unfairly biases paths with more edges (since a path with k edges now has $k\alpha$ added to its total weight). Instead, consider the following DP algorithm, where s is the start vertex,

```
d1[s] = 0
for every other vertex v:
    if (s, v) is an edge, d1[v] = w(s, v), else d1[v] = infinity
for i = 2, 3, ... n - 1:
    for every vertex v:
```

```
        di[v] = d(i-1)[v]
        for every vertex u:
            if (u, v) is an edge and d(i-1)[u] + w(u, v) < di[v]:
                di[v] = d(i-1)[u] + w(u, v)
```

In the end, d_{n-1} is returned. Here, $d_i[v]$ is the length of the shortest path from s to v that uses $\leq i$ edges. Since there are no negative weight cycles, the shortest path from s to any v cannot have more than n edges. The proof of correctness follows trivially by induction on the d_i arrays using the recurrence,

$$d_i[v] = \min \begin{cases} d_{i-1}[v] \\ d_{i-1}[u] + w(u, v) \text{ ; for each } u \text{ such that } (u, v) \text{ is an edge} \end{cases}$$

The runtime is $O(n^3)$, assuming $(u, v) \in E(G)$? is a constant time query. Also, rather than using $n-1$ arrays, this algorithm works with only one, and loops on edges rather than pairs of vertices,

```
initialize d[s] = 0 and d[v] = infinity for every other vertex v
for i = 1, 2, ... n - 1:
    for each edge (u, v):
        if d[u] + w(u, v) < d[v]:
            d[v] = d[u] + w(u, v)
return d
```

In undirected graphs, each edge will be iterated over twice: once when v is seen as a neighbour of u and once when u is a neighbour of v. The time complexity of this is $\Theta(nm)$ which is still possible $\Theta(n^3)$ if $m \in \Theta(n^2)$. This algorithm can also be extended to do several interesting things:

1. Update a parent pointer when $d[v]$ is updated so that we can recover the actual shortest paths backwards from each v to s
2. To detect negative weight cycles reachable from s, run one more iteration (at $i = n$) of the loop and see if any d values change. This works because Bellman-Ford assumed there were no negative cycles, and if there are none, then $n-1$ iterations guarantee the d values are correct. The additional iteration lets us consider paths with cycles, since any path with no cycle has at most $n-1$ edges, so if any d values change now, then it means there was a negative cycle which allowed some path to get a better result. And in fact if some d value changes, it will change again if the loop is run once more (at $i = n+1$), etc.
3. To detect negative weight cycles anywhere in the graph, add a new vertex s' and add edges from it to every other vertex with weight 0 then start the Bellman-Ford at the s' vertex. The reasoning for why this works follows from above.

Using topological sort: If the graph is a DAG, apply topological sort (using DFS) to get an ordering $v_1, \ldots, v_n$. Suppose $v_i = s$. There is no path from s to each of $v_1, \ldots, v_{i-1}$, so update *dist* to be ∞. Next, set *dist* to 0 for $v_i = s$ and ∞ for each $v_{i+1}, \ldots, v_n$ and,

```
for j = i, i + 1, ... n:
    for each edge from vj to vk:
        if dist[vj] + w(vj, vk) < dist[vk]:
            dist[vk] = dist[vj] + w(vj, vk)
return dist
```

This runs in $O(n+m)$, from both the topological sort and the nested loops above. The proof of correctness is by induction on j. Intuitively, since edges never go "backwards" in the ordering given by topological sort, iterating forwards once in this manner works.

6.6 All Pairs Shortest Path (APSP)

What if we wanted to compute the minimum distance between all pairs of vertices rather than just the distance to each vertex from a source vertex s? The following algorithms work for both directed or undirected weighted graphs but cannot contain negative weight cycles.

Bellman-Ford: In the naive implementation, we call the SSSP Bellman-Ford algorithm once for each vertex (and pass it in as the source vertex). This has time complexity $\Theta(n^2m)$, which can possibly be $\Theta(n^4)$ in edge-dense graphs. Another implementation is to consider all pairs of vertices simultaneously,

```
for all pairs of vertices u, v:
    if u = v, d[u, v] = 0
    else if (u, v) is an edge, d[u, v] = w(u, v)
    else d[u, v] = infinity
for i = 2, 3, ... n - 1:
    for every vertex u:
        for all edges (x, v):
            if d[u, x] + w(x, v) < d[u, v]:
                d[u, v] = d[u, x] + w(x, v)
return d
```

The recurrences here are the same, $d_i[u, v]$ is the shortest path from u to v using $\leq i$ edges and the implementation above is in the space-saving version (only one d matrix rather than $n-1$ matrices). However, the runtime is still $O(n^2m)$. This is about as efficient as Bellman-Ford can get for APSP.

Floyd-Warshall: Let the vertices of the graph be $v_1, \ldots, v_n$. The subproblems in this DP algorithm is paths which only contain a certain subset of vertices as intermediate vertices (those which are neither end of the path). Specifically, define $dist_k[u, v]$ to be the minimum weight of any path from u to v that only uses $v_1, \ldots, v_k$ as intermediate vertices, for $0 \leq k \leq n$. To compute $dist_k[u, v]$, either the shortest path from u to v uses v_k or it does not,

```
for every pair of vertices u, v: // three different base cases
    if u = v, dist0[u, v] = 0
    else if there is an edge from u to v, dist0[u, v] = w(u, v)
    else dist0[u, v] = infinity
for k = 1, 2, ... n:
    for every vertex u:
        for every vertex v:
            using_vk = dist(k-1)[u, vk] + dist(k-1)[vk, v]
            not_using_vk = dist(k-1)[u, v]
            distk[u, v] = min(using_vk, not_using_vk)
```

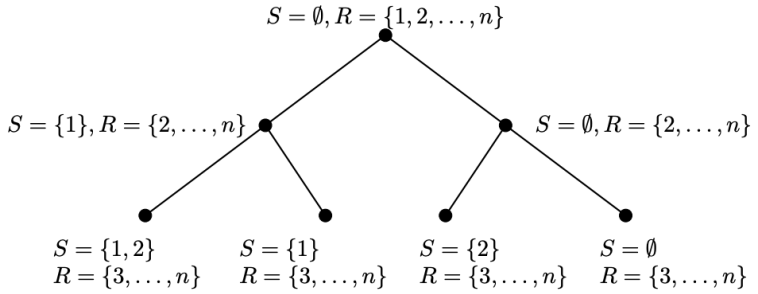
`distn` is returned at the end. The runtime is $\Theta(n^3)$ and $\Theta(n^3)$ space. Using only a single *dist* matrix works as well, for $O(n^2)$ space. The proof of correctness is by induction on k. To recover the actual paths, we can maintain a $next[u, v]$ matrix, which stores the first vertex after u on a shortest u, v path (see notes). Lastly, if any of the diagonal entries of the final *dist* matrix are negative, there was a negative cycle in the graph (see A7).

7 Exhaustive Search

In cases where there is no known efficient algorithm for the problem, there are three solutions: a heuristic argument that runs quickly but has no provable guarantee of correctness, an approximation algorithm that does not solve the problem optimally but gives some guarantees about its solutions, an exponential-time algorithm which is exactly correct but quite slow. This chapter focuses on the third option, **systematic search**, with special constraints that keep the algorithm as efficient as possible.

7.1 Backtracking

Subset sum: Given an array of n elements $1, 2, \ldots, n$ with positive weights $w_1, \ldots, w_n$ and a target weight T, determine if there is a $S \subseteq \{1, 2, \ldots, n\}$ such that the respective weights add up to T. We will see later that this problem is NP-complete and there is no known polynomial-time algorithm for it. Consider the set of all possible configurations, which each correspond to a full or partial solution (a set $S \subseteq \{1, \ldots, k\}$ of elements added to our solution and a set $R \subseteq \{k+1, \ldots, n\}$ of elements still to be considered, in a tree structure,



Each branch corresponds to either adding the current element to S or not (and in both cases, the element is removed from R). The tree is never explicitly built but the algorithm will explore these configurations, with additional rules to short circuit on a dead end: if the sum of the numbers in S exceeds T or if all the sum of the numbers in S and R is smaller than T. In either case, we can stop exploring this branch because all nodes underneath the current one will either overshoot (since all weights are positive) or undershoot (since even if everything in R is taken, it will not be enough to reach T) the target. So, the algorithm backtracks to the parent node (configuration) to continue other paths from there. The general backtracking structure looks like,

```
Initialize A to be the set of all active configurations
while A is not empty:
    C = next configuration in A
    if C is a solution return True
    if C is not a dead end:
        EXPAND(C) and add the resulting configuration to A
return False
```

The set of active configurations can either be kept in a queue or a stack. These correspond to exploring the configuration tree using BFS or DFS. Generally, DFS is the better option since only $O(n)$ configurations are active in memory at any time versus up to $\Omega(2^n)$ active configurations using BFS. However, memory aside, both should yield the same answer. In this specific example, `EXPAND(C)` is either adding the next element in R to S or not, corresponding to either taking it in the subset or not.

7.2 Branch and Bound

Travelling salesman problem (TSP): given a weighted, undirected graph with non-negative edge weights, find a cycle C that goes through each vertex in V exactly once (called a TSP tour) with minimum total weight. Like subset sum, this problem is also NP-complete and there is no known polynomial time algorithm. There are $|V|!$ total possible tours and instead of just the dead end technique as for backtracking, we now use a branch and bound technique which involves calculating a lower bound on any complete solution achievable from the current configuration and if it is not better than the current best, stop expanding.

```
Initialize A to be the set of all active configurations
Initialize best to infinity
while A is not empty:
    C = next configuration in A
    if C is a solution and VALUE(C) < best:
        best = VALUE(C)
    else if C is not a dead end and VALUELOWERBOUND(C) < best: // bounding
        EXPAND(C) and add the resulting configuration to A // branching
return best
```

In the TSP problem, a configuration is a set $I \subseteq E$ of edges included in the tour and a set $X \subseteq E$ of edges decided to not be in the tour. Initially, $I = X = \emptyset$. At each step, an edge can either be added to I or X. A configuration $C = (I, X)$ is a solution when I forms a TSP tour. To figure out whether C is a dead end, we check whether it can even be a valid tour. For example, if C is not a solution but already has a cycle, it is invalid. Note that in these algorithms, `EXPAND(C)` does not check validity, it is the recursive call which does. `VALUELOWERBOUND(C)` might check that the sum of all the edges in I so far is not larger than the current best, because the best lower bound is if all the remaining edges added have weight 0. Checking for dead ends and calculating a lower bound corresponds to bounding and `EXPAND(C)` corresponds to a branch. Typically, `VALUELOWERBOUND` is not a tight approximation to the actual minimum value. The more accurate this lower bound, the fewer unnecessary branches the algorithm has to explore but the cost of calculating the lower bound can itself be costly. So, there is a tradeoff there.

In summary, backtracking is suitable for decision problems (is there a subset such that...) and computes dead ends from where we can conclusively say whether all expansions from the current one will return a certain answer. Branch and bound is suitable for optimization problems (what is the best TSP tour...) and calculates a lower bound on any possible solution obtained from the current point to determine whether it is worth expanding, in addition to dead end checking. Even with the early short circuiting, both algorithms are still $\Theta(2^n)$ and it is easy to think of inputs where the algorithm explores $\Omega(2^n)$ nodes in the configuration tree before returning an answer. However, in practise, systematic search will be more performant than the naive search.

8 Efficiency and Tractability

An algorithm is considered reasonably efficient if it is a polynomial time algorithm. Or, there is a constant k such that for any input size n (where n is the number of bits to encode the input), the algorithm runs in $O(n^k)$. Problems which can be solved by polynomial time algorithms are called **tractable**. There is a distinction between polynomial and pseudo-polynomial (algorithms which run polynomial to the magnitude of the input but exponential in the size). For example, the DP solution to 0-1 knapsack is pseudo-polynomial, since the runtime is $\Theta(nW)$ and this is not polynomial to $n \log W$, which is the number of bits needed to encode the input).

Since we are concerned about polynomial runtime with respect to the input size, whether you calculate runtime using the RAM model or the bit cost model will yield the same results. For example, if you analyze binary search on an array of n numbers in the range $[1, n]$ using the bit cost model, the input size is $O(n \log n)$ and the time to compare two numbers is $O(\log n)$. Overall, we still have that this algorithm is polynomial to the input size.

Polynomial time algorithms are important because,

1. Strong Church-Turning thesis: the class of tractable problems is independent of the way we define algorithms. Any reasonable modification to the algorithm model will not affect the tractability of a problem.
2. If we can argue that a problem is intractable, then it cannot be solved by any efficient algorithm, under any reasonable definition of efficiency. Tractability is a very strong conclusion.
3. They can be composed together well. An algorithm which calls a polynomial time algorithm as a sub-routine a polynomial number of times is itself a polynomial time algorithm.
4. They increase in runtime algebraically to the input size, not exponentially.

For many problems (eg TSP), it is unknown whether there is a polynomial time algorithm nor can anyone prove otherwise. Instead, we wish to define classes of problems such that a polynomial time algorithm for one problem yields polynomial time algorithms for the rest. These decision problems are known as NP-complete (see later) and some well-known ones are:

1. Clique: Given a graph G and $k > 0$, determine if G contains a clique of size $\geq k$, a set of $\geq k$ vertices such that the subgraph induced by those vertices is complete
2. Independent Set (IndepSet): Given a graph G and $k > 0$, determine if G has an independent set of size $\geq k$, a set of $\geq k$ mutually non-adjacent vertices
3. Vertex Cover: Given a graph G and $k > 0$, determine if G has a vertex cover of size $\leq k$, a set of $\leq k$ vertices such that every edge has an end in at least one of them
4. Hamiltonian Path/Cycle: Given a graph G, determine if G has a Hamiltonian path/cycle, a path/cycle which touches every vertex exactly once
5. Subset Sum: Given an array of n elements and number t, determine if there is a subset whose elements sum to t
6. TSP: Given a graph G and k, is there a TSP tour (a path that visits every city exactly once and returns to the starting point; a Hamiltonian Cycle) that has total weight $\leq k$?

8.1 P

There are three main types of computational problems:

1. Decision problems: output is true or false (yes or no)
2. Optimization problems: want to find the value of the best solution
3. Search problems: want to find the best solution itself, not just its value

We focus on decision problems. Other problems can easily be turned into analogous decision problems. For example, instead of asking what the longest common subsequence between two strings is, ask if the LCS has length $\geq k$. However, this translation must be done with a polynomial number of calls. For example, turning multiplication into asking if the product of two n bit integers is k is invalid because you would need to ask this question for $k = 0, \ldots, 2^{2n}$, which is not polynomial to n. Instead, ask if the ith bit is a 1, for $i \in \{1, \ldots, 2n\}$. We only need to ask $2n$ times.

P: set of all decision problems that can be solved by polynomial time algorithms. Some examples are 2SUM, Karatsuba multiplication, DFS, binary search. But not the $O(\sqrt{n})$ primality test from the midterm. Since we are concerned with runtime relative to the input size (in bits), we calculate runtime using the bit cost model.

Reduction: a decision problem A is polynomial time reducible to the decision problem B, $A \leq_{\mathrm{P}} B$, if there is a polynomial time algorithm F that turns any input I_A for A into an input I_B for B such that I_A on A has the same answer as I_B on B. To prove the correctness of a many-one reduction, show that F is in polynomial time and that A on I_A always returns the same output as B on I_B. Informally, $A \leq_{\mathrm{P}} B$ means "A is easier than B".

What was described above is called a many-one reduction. In this reduction, the algorithm for B can only be called once on the transformed input. The other kind of reduction, which is more general, is Turning reductions (also called Cook reductions): $A \leq_{\mathrm{P}} B$ if a poly time algorithm for B can be used (as a black box) to solve A in polynomial time. This lets us call the algorithm for B a polynomial number of times. To prove correctness, prove that the use of an algorithm for B solves the problem A. There are also two main settings under which polynomial time reductions are useful:

1. Reductions between decision and optimization problems (see A8Q1i). Turing reductions are needed since a decision problem can never return the same answer as an optimization problem on any input. The equivalence of decision problems and optimization problems is not always known but they are generally equivalent with respect to polynomial time. But not always; the optimization version of TSP is NP-hard (not in NP) but the decision version is NP-complete.
2. NP-completeness proofs (described later). The more restrictive many-one reductions are sufficient.

Reductions are powerful because they can be proven even if polynomial time algorithms are unknown for either A nor B. For example, Clique is poly-time reducible to Independent Set and vice versa, by simply taking the complement graph where all present edges are removed and all non-existent edges are added.

A subtle point about reduction is that an algorithm for B is polynomial time with respect to its own input. So, if we want to conclude that an algorithm which uses B as a sub-routine is polynomial time with respect to the input of A, then the input given to B must be polynomial to the input given to A. In other words, start with the input given to A and construct inputs to B in polynomial time. Sometimes reductions can be on very different kinds of inputs. For example, transforming a graph into subsets of $\{1, \ldots, m\}$. As long as the transformation is polynomial, it's okay.

Not all reductions involve modifying the input. For example, to prove IndepSet $\leq_{\mathrm{P}}$ VertexCover, the converter algorithm simply takes the inputs to IndepSet, G, k, and turns it into $G, n - k$ for VertexCover. This works because a graph has an independent set of size k iff it has a vertex cover of size $n - k$; the compliment vertex set of any vertex cover is an independent set and vice versa.

Relationships: If A, B are two decision problems such that $A \leq_{\mathrm{P}} B$, then:

1. If $B \in \mathrm{P}$, then $A \in \mathrm{P}$ but if $B \notin P$, this does not tell us anything about A
2. If $A \notin \mathrm{P}$, then $B \notin \mathrm{P}$ (contrapositive of above)
3. If $B \leq_{\mathrm{P}} C$, then $A \leq_{\mathrm{P}} C$ by transitivity
4. It is possible that $B \not\leq_{\mathrm{P}} A$, the reduction does not always go both ways

As proof of Item 1, let β be any polynomial time algorithm for B, let F be the polynomial time input converter algorithm. Running F then β is polynomial time and solves A, so A is in P.

To show that two problems are equivalently hard, we can show that $A \leq_{\mathrm{P}} B$ and $B \leq_{\mathrm{P}} A$. Sometimes one direction is easy to prove and the other direction is very hard (unknown). Item 2 from above tells us that to show that a new problem is hard, we can reduce a known hard problem to the new problem. For many difficult problems (such as clique), it is not known how to prove that they are not in P. As seen later, there is a large class of decision problems not known to be in P but are all related by reduction; $A \leq_{\mathrm{P}} B$ for all A, B in this class. This means a polynomial time algorithm for one yields a polynomial time algorithm for them all. Some problems in this class are TSP, HamiltonianPath/Cycle, IndepSet, etc.

8.2 NP

Verifier: for a decision problem X, is an algorithm A that takes in as input an input x to X and an additional input y called a certificate such that two conditions are satisfied:

1. For any input x which X outputs `Yes` on, $\exists$ a certificate y that causes $A(x, y)$ to output `Yes`
2. For any input x which X outputs `No` on, $\forall$ certificates y, $A(x, y)$ outputs `No`

Polynomial time verifier: a verifier A that runs with time complexity polynomial to the input size x for X. This also implies that for `Yes` inputs x, the certificate y must also be polynomial size, or else the verifier would not even have time to read y.

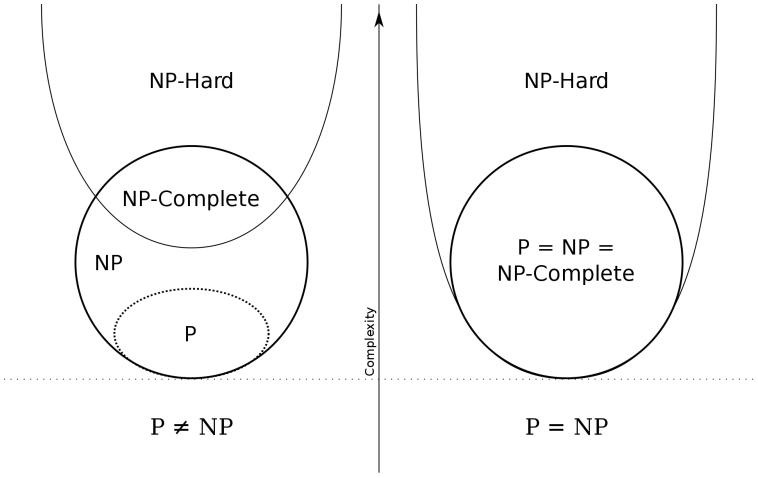
An example of a verifier for the clique problem is an algorithm which takes in the graph G, an integer k, and a set of vertices of size k as the certificate (note that we can enforce the certificate is a certain size), then returns whether all vertices in this set are mutually adjacent. To prove that a verifier is correct, we need to prove both the `Yes` case (on a "valid" certificate) and the `No` case for all certificates. For example, show that if G has a clique of size k, giving it as the certificate causes the verifier to return `Yes` and otherwise, there cannot be a certificate that gives a `Yes`, otherwise it is a clique.

What if we gave a trivial certificate which is a single bit that is 1 when X returns `Yes` on x and 0 otherwise? This does not work because on inputs where X answers `No`, the verifier must reject on all certificates but if we give a 1 as a verifier, the verifier will accept.

Another example of a verifier for the Travelling Salesman Problem (the decision problem version, ie: is there a TSP tour of weight $\leq k$) is one which takes in the graph, weights on the edges, and k, and a permutation of vertices (which are supposed to represent a TSP tour, if it exists) as a certificate then checks whether this tour is indeed a permutation, that the edges between consecutive vertices exist, and that the sum of the weights of the edges is $\leq k$. Note we cannot simply pass in a TSP tour and assume that the edges exist because there are graphs where this is not possible and the verifier must verify that it is indeed a tour. Also, the optimization version of TSP does not have a polynomial time verifier.

NP: (nondeterministic polynomial time), the set of all decision problems that have polynomial-time verifiers. Note that P is a subset of NP and the proof for this is to have a verifier that ignores the certificate and runs X on x. So, P $\subseteq$ NP. However, it is not known if all problems with polynomial time verifiers can be solved in

polynomial time (whether NP $\subseteq$ P thus P = NP is unknown). The two possibilities are pictured below, and as seen later, if P is not equal to NP, we can show that problems in a certain class of problems cannot be in P.



All problems in NP with a verfier with runtime $O(n^k)$ can at least be solved in $O(2^{n^k})$ by trying all certificates one by one.

Lastly, note that for some problems such as Clique and Subset Sum, it is easy to create a polynomial time verifier but the inverses are unknown. For example, it is unknown if the problem of "is there no subset of a certain total sum" or "does this graph not have a clique of size $\geq k$" is in NP. For the latter, causing the verifier to return `No` is easy (give a clique of size $\geq k$) but causing it to say `Yes` is unknown. Giving the verifier the "largest clique" in the graph then checking that its size is $< k$ does not work because the verifier must verify that it is the largest clique, nor does giving a subgraph of size k and showing that it is not a clique because a clique of size k could be elsewhere in the graph; the verifier cannot trust user information. These problems are not symmetric. **co-NP** is the set of decision problems where the "no" instances (counter-examples to what the problem is asking for) can be verified in polynomial time. Or, the set of decision problems where the complimentary problem is in NP. Certainly P $\subseteq$ co-NP but it is unknown if NP = co-NP or if P = (NP $\cap$ co-NP).

8.3 NP-Completeness

Ideally, we would like to show that problems such as Clique have no polynomial time algorithm but this is unknown. The next best thing is to show that problems such as Clique are the "hardest" problems in NP.

NP-complete: the decision problem X is NP-complete if $X \in$ NP and for every problem $A \in$ NP, $A \leq_{\text{P}} X$. Since problems in NP-complete are the "hardest" problems in NP, for any NP-complete problem X, if $X \in$ P, then P = NP (recall the relationships implied by a reduction from earlier) and NP = NP-complete. Otherwise, if $X \notin$ P, then P $\neq$ NP and no NP-complete problem can be solved with a polynomial time algorithm (see diagram above). These are very powerful conclusions and strongly indicate that X cannot be solved in polynomial time because if it could be, then every other problem in NP could be solved in polynomial time. The reason this definition requires that $A \leq_{\text{P}} X$ for all A in NP and not for some A in NP is because $X \leq_{\text{P}} X$ trivially and X is in NP so NP-complete would be the same as NP, which is useless.

However, proving $A \leq_p X$ for every $A \in$ NP sounds difficult. Instead, by transitivity, we only need to show that $A \leq_{\text{P}} X$ for any single NP-complete problem A. The Cook-Levin theorem states that satisfiability problems, SAT, (except 2SAT) are NP-complete. One particularly useful variant of SAT is,

3SAT: Given a boolean CNF formula (one constructed by AND'ing together a bunch of OR's) on n variables in which each clause has ≤ 3 distinct literals, where a literal is either x or $\neg x$ for some variable x, determine

if there is an assignment of `True/False` to the variables that satisfies the formula (makes it true). The alternate definition of 3SAT, which will be used on exams is that each clause has exactly 3 distinct literals.

Thus, to show a problem X is NP-complete, we only need to show that it is in NP and 3SAT $\leq_{\mathrm{P}} X$. Note that 2SAT is in P. This is because satisfying the clause $(x_i \vee x_j)$ corresponds to two implications: if x_i is not true, then x_j is true and if x_j is not true, then x_i is true (contrapositive). If this were not the case, then the clause would not be satisfied. So, the polynomial time algorithm follows these implications and if a contradiction is reached (such as starting with x_i being true then later concluding that it is false), the formula is not satisfiable, otherwise it is. This does not work with 3SAT because a 3SAT clause $(x_i \vee x_j \vee x_k)$ corresponds to the implication: if x_i is false then x_j or x_k is true but to consider these implications requires making a choice which results in exponential explosion.

Now, we prove that the following problems are NP-complete by proving that 3SAT is polynomial time reducible to these problems (the part of the proof that they are in NP is omitted),

Clique: Let φ be a CNF with c clauses, each of which has at most 3 literals. Let G be the graph where there is one vertex per literal (thus, up to $3c$ vertices) and two vertices are adjacent iff their literals are in different clauses and they do not contradict each other (x_i is never adjacent to $\neg x_i$). Now, if G has a clique of size c, assigning values to the variables such that each literal corresponding to each vertex of the clique is `True` gives a satisfying assignment of φ, since the vertices in the clique are all mutually adjacent, thus do not contradict each other. And if φ has a satisfying assignment, there is at least one literal in each of the c clauses which evaluates to `True` and each such `True` literal does not contradict those in other clauses so the c vertices corresponding to the `True` literals gives a clique of size c in G. Moreover, since Clique $\leq_{\mathrm{P}}$ IndepSet $\leq_{\mathrm{P}}$ VertexCover $\leq_{\mathrm{P}}$ SetCover, as shown earlier, these other problems are in NP-complete as well (proof that they are in NP ommitted). Another straightforward reduction to prove 3SAT $\leq_{\mathrm{P}}$ IndepSet is to create the graph with the same vertices as before but two vertices are adjacent iff their literals are in the same clause or they are contradictory (x_i adjacent to $\neg x_i$).

DirHamiltonianPath: This problem is the same as HamiltonianPath but on a directed graph.

HamiltonianPath:

HamiltonianCycle:

SubsetSum:

Other notes on NP-completeness:

1. NP-complete is not the same as NP-hard. A decision problem X is NP-hard if every problem A in NP satisfies $A \leq_{\mathrm{P}} X$. Alternatively, $A' \leq_{\mathrm{P}} X$, for some NP-complete problem A'. Unlike NP-completeness, NP-hard does not require that X is in NP. However, in most cases, all "hard" problems are in NP. Informally, NP-hard means "at least as hard as all NP-complete problems" and NP-complete is a subset of NP-hard. And note that if $A \leq_{\mathrm{P}} B$ and B is NP-hard, then no conclusions can be made about A.
2. There are many problems which are even harder than those in NP, such as undecidable problems: those which cannot be solved by any algorithm, regardless of efficiency.

9 Approximation Algorithms

For many optimization problems, such as TSP, Clique (find the largest clique), etc., even their simpler decision problem counterparts are NP-complete. However, there are simple, polynomial time algorithms for optimization problems for which there is no known polynomial time algorithm which might not always return the optimal solution but can provably return one that is "not much worse". For any $k \geq 1$ and minimization problem P, a **k-approximation** algorithm is one where for every instance x of P with optimal value OPT, it returns a solution with value at most $k \cdot$OPT.

Metric TSP: Given a complete weighted graph G with positive edge weights and $w(u, v) \leq w(u, x) + w(x, v)$ for every $u, v, x \in V$ (triangle inequality holds), find the length of the shortest TSP tour; one which visits every vertex in V once. This problem is also NP-complete but there is a polynomial time 2-approximation algorithm: construct a MST T of G, let L be the list of vertices visited in an in-order traversal of T with repeats (each edge in the MST is touched exactly twice), and let the tour returned be the one obtained by shortcutting paths in L that visit previously-seen vertices. Since deleting any edge from a TSP tour gives a spanning tree, then the total edge weight of the MST is $\leq$ the total edge weight of the optimal TSP tour. Moreover, the in-order traversal of T visited each edge of the tree twice but by the triangle inequality and the tour skipping previously-seen vertices using edges not in the MST, the total weight of the tour is $\leq$ twice the total weight of the MST which is $\leq$ twice the total edge weight of the optimal TSP tour, from above. These shortcut edges exist because the graph is complete.

Vertex cover: Recall the vertex cover problem attempts to find the size of the smallest vertex cover in a graph G. This is a 2-approximation algorithm,

```
C = empty set
for each edge (u, v):
    if neither u nor v is in C:
        add u and v to C
return C
```

For the proof of correctness, let M be the set of edges that caused the algorithm to add its ends into C. M is a matching because no two edges have a common end and $|C| = 2|M|$. However, $|M| \leq |C'|$, where C' is a min cover of G, since each edge in M must have at least one of its two ends in any cover. So, $|C| = 2|M| \leq 2|C'|$, so this is a 2-approximation. It is unknown whether there is a k-approximation for VertexCover for any $k < 2$.

TSP: If P = NP, then every NP-complete problem (including optimization ones) would have a polynomial time algorithm. However, if P $\neq$ NP, then for any $k \geq 1$, there is no polynomial time k-approximation algorithm for TSP. To prove this, we prove the contrapositive: if there is a $k \geq 1$ for which there is a polynomial time k-approximation algorithm A for TSP, then P = NP. This is because A can be used to design a polynomial time algorithm for HamCycle, which is NP-complete and thus P = NP. To do this, from the graph G, we construct the complete graph G' where if an edge is in G, then it has weight 1, else weight kn. Then, if the total weight of the TSP tour given by $A(G')$ is $\leq kn$, then there is a Hamiltonian Cycle in G, otherwise there is not. This is because G has a HamCycle iff there is a TSP tour with total weight n, and since A is a k-approximation, this is iff A finds a tour with weight $\leq kn$. Once the tour uses a single edge that is not in G (and has weight kn), the total weight of the tour can no longer be $\leq kn$.

10 Difficult Problems

10.1 Impossible Problems

Undecidability: A decision problem is undecidable if it has no algorithm. For more general problems, it is unsolvable if it has no algorithm.

Halting problem: Given the binary code for an algorithm and an input w, does it halt (after a finite number of steps) on w? This is a classic example of an undecidable problem and Turing proved it in 1936.

Collatz (1937): Given a positive integer x, while $x > 1$, if x is even then divide by 2, if it is odd, then set x to $3x + 1$. It is unknown whether this algorithm halts after a finite number of steps on every input x and is an example of why the halting problem is so difficult.

Goldbach (1742): Can every even number ≥ 4 be represented as the sum of two primes? This is an existence question that is open, so it is unknown if only the following algorithm halts,

```
n = 4
counterexample = False
while not counterexample:
    counterexample = True // assume it is a counterexample
    for x = 3, 5, 7, ... n - 1:
        if ISPRIME(x) and ISPRIME(n - x):
            counterexample = False
    n = n + 2
return "n is a counterexample"
```

This algorithm takes no input. So even if the halting problem is modified to determining whether an algorithm with no input halts, it is still impossible.

In fact, if X is a decision problem for which the reduction HALTING $\leq X$ holds, then X is undecidable. Note that $\leq$ rather than $\leq_{\mathrm{P}}$, the reduction does not need to run in polynomial time.

10.2 Very Difficult Problems

Totally quantified boolean formula (TQBF):

Halt in k steps:

10.3 Rather Difficult Problems

Clique:

10.4 Slightly Difficult Problems

3SUM:

11 Extra

This is bonus material that will not be covered on exams.

11.1 Median Finding

11.2 Heavy Hitters

11.3 Maximal Independent Set

12 Other

Theorem: The definition of little-o is valid with $\leq$ rather than $<$. Alternatively, if $\forall c > 0, \exists n_0 > 0$ such that $\forall n \geq n_0, f(n) \leq cg(n)$, then $\exists n_1 > 0$ such that $\forall n \geq n_1, f(n) < cg(n)$

Proof: Let $c > 0$. Then, $\frac{c}{2} > 0$ so $\exists n_0 > 0$ such that $\forall n \geq n_0$, $f(n) \leq \frac{c}{2} g(n) < cg(n)$.

Theorem: If $f(n) \in o(g(n))$ and $g(n) > 0, \forall n$, then n_0 must depend on c

Proof: Suppose for contradiction there was a constant n_0. Since $f(n) \in o(g(n))$, then $f(n_0 + 1) < cg(n_0 + 1)$ for any c. Choose $c = \frac{f(n_0+1)}{g(n_0+1)}$. But, $cg(n_0 + 1) = f(n_0 + 1)$ so this is a contradiction.

Theorem: If $f(n) \in O(g(n)), g(n) > 1$, then $\log f(n) \in O(\log g(n))$

Proof: Since $f(n) \in O(g(n))$, then $\exists c, n_0 > 0$ such that $f(n) \leq cg(n), \forall n \geq n_0$. Then,

$$\log f(n) \leq \log cg(n) = \log c + \log g(n)$$

Therefore, since $\log c + \log g(n) \leq \log c \log g(n) + \log g(n)$, taking $c' = (\log c + 1)$ gives $\log f(n) \in O(\log g(n))$. Note that if $g(n)$ doesn't have to be greater than 1, then $f(n) = 2, g(n) = 1$ is a counter proof to this theorem. We can also prove a similar result for Ω and Θ.

Theorem: Although $\lfloor \log n \rfloor \in \Theta(\log n), n^{\lfloor \log n \rfloor}$ is not necessarily in $\Theta(n^{\log n})$

Theorem: For any $1 < a < b, a^n \in o(b^n)$

Proof: Let $1 < a < b$. Then, for all $n > 0$, $a^n < a^{n+1} = \frac{a^{n+1}}{b^n} b^n$ and in order to satisfy the inequality $\frac{a^{n+1}}{b^n} b^n \leq cb^n$ for any c,

$$a\frac{a^n}{b^n} \leq c \to n \log\left(\frac{a}{b}\right) \leq \log\left(\frac{c}{a}\right) \to n \geq \frac{\log\left(\frac{c}{a}\right)}{\log\left(\frac{a}{b}\right)}$$

Therefore, taking n_0 as the max of this value and 1, $a^n \in o(b^n)$.

Theorem: If A, B are in P, then $A \leq_{\mathrm{P}} B$

Proof: To show that $A \leq_{\mathrm{P}} B$, we want to show a Turing reduction: given a polynomial time algorithm for B, P_B, create a polynomial time algorithm P for A. Since A is in P, it has a polynomial time algorithm for P_A. So, P can simply run P_A on A, ignoring P_B completely. However, a many-one reduction might not work here. For example, if A always returns `True` and B always returns `False`, there is no polynomial time input converter which can get the two algorithms to agree on the output.

Theorem: If $A \leq_{\mathrm{P}} B$ and B is in NP, then A is in NP

Proof: ?? see CS 360

Reminders:

1. NP-complete proofs must begin by proving that the problem is in NP, and this step is easy to forget.
2. $\binom{n}{k} = \frac{n!}{k!(n-k)!} \in \Theta(n^{k+1})$