CS 341 Algorithms

Keven Qiu
Instructor: Mark Petrick
Fall 2022

Contents

Chapter 1

Introduction and Order Notation

Convex Hull: Problem: Given n points in the plane, find convex hull. The smallest convex set containing the points.
$*$ Equivalently, the convex hull is a polygon whose sides are formed by lines l that go through at least 2 points and have no points to one side of l.

Algorithm 1: Find all lines such that $*$ is true. Choose any 2 points, form l. $O(n^2)$. Check the property: are all points on the same side of l. $O(n)$. Total run time is $O(n^3)$.

Algorithm 2 Jarvis March: Once we have 1 line l, there is a natural next line l'. Rotate l at point s until it hits next point t.

Finding l': compute all lines through s and another point. Find extreme one. This minimizes angle α created by original line l and new line l'. Find min α from set of angles. $O(n)$. Let k be number edges on convex hull. Total runtime is $O(nk)$. k could be n if all points are in convex set.

Algorithm 3 Reduction: Solve new problem by using a known alg.

Sort points by x-coordinate. Start at left most. Traverse points in order to find edges of convex hull.
Sorting $O(n \log n)$. Algorithm $O(n)$. Total runtime is $O(n \log n)$.

Algorithm 4 Divide and Conquer: Divide points in half and find convex hull of each side. Find upper bridge and lower bridge to connect.

$$T(n) = 2T(n/2) + O(n) \implies O(n \log n)$$

We can reduce to sorting. If we could find a better algorithm, then we could get a faster sorting algorithm, so in some sense no.

Given n points $x_1, \ldots, x_n$ to sort. Map x_i to (x_i, x_i^2) takes $O(n)$, call find C.H takes $f(n)$.

Obtain sorted points by starting leftmost, read right takes $O(n)$. So $O(n)+$ time to find C.H.

Pseudocode:

- $A[n] = \{0\}$. Initialize array to 0. Expect to be $O(n)$.

For a Fibonacci sequence, if `int` holds 64 bits, then $n = 94$ makes it overflow.

Computing $a \times b$ takes $O(\log(a) \times \log(b))$ using normal math.

Models of computation: Pseuodocode, random access machine, circuit family, turing machine, special purpose or structured models of computing.

Let $T_A(I)$ denote the running time of an algorithm A on instance I.
Worse-case complexity of an algorithm: take the worst I

- $T_A(n) = \max\{T_A(I), size(I) = n\}$
- Often simply say $T(n)$ (or $T(I)$) if A is understood

Average-case complexity
Output sensitive run-times, multiple variable run-times.

Reductions: Using a known algorithm to solve a new problem.

Suppose worst-case runtime of alg 1 is $O(n^2)$ and alg 2 is $O(n \log n)$. Which is better? O is an upper bound which might not be tight. Difficult to say. To compare, use Θ bounds. $\Theta(n \log n)$ is better than $\Theta(n^2)$.

Chapter 2

Divide and Conquer

Def Divide and Conquer: Divide and Conquer algorithms are broken into 3 basic steps:

1. Divide - Break problem into small sub-problems
2. Conquer - Solve sub-problems by calling recursively until solved
3. Combine - Combine the sub-problems to get the final solution of the whole problem

Examples: Binary search

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + c + c \\ &\vdots \\ &= T(1) + c + \cdots + c \in O(\log n) \end{aligned}$$

QuickSort: partition array based on pivot, $O(n)$ to divide. 2 subproblems.

MergeSort: Divide array into left and right halves, 2 subproblems, recurse on both, merge takes $O(n)$.

Draw a recursion tree, and the work done is sum of all levels in the recursion tree.

For merge sort, there are cn for each level and base case is 0.

$$nT(1) + c(n + 2\frac{n}{2} + 4\frac{n}{4} + \cdots + \frac{n}{2}\frac{n}{n/2}) = cn(\log n) + n(0) \in \Theta(n \log n)$$

Height is $n\left(\frac{1}{2}\right)^k = 1$. Take n and half it k times to get base case 1.
Precise recurrence:

$$T(n) = \begin{cases} T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + (n-1) & (n > 1) \\ 0 & (n = 1) \end{cases}$$

Solving Recurrences by Substitution: Guess the result and prove by induction.

Using the example: guess and prove by induction that $T(n) \leq c \cdot n \log n, \forall n \geq 1$.

Base case: For $n = 1$, $T(1) = 0$ and $c \cdot n \log n = 0$; i.e. $0 \leq 0$.

Induction Hypothesis: Assume that $T(k) \leq c \cdot k \log k$ for all $k < n$ where $k \geq 2$.

Induction Step: One method to give a rigorous proof is to separate even and odd cases. If n is even, we don't need floors and ceilings.
n even:

$$\begin{aligned} T(n) &= 2T(n/2) + (n-1) \\ &\leq 2c \cdot \frac{n}{2} \log\left(\frac{n}{2}\right) + (n-1) && \text{by I.H.} \\ &= cn \log\left(\frac{n}{2}\right) + (n-1) \\ &= cn\,(\log n - 1) + (n-1) \\ &= cn \log n - cn + (n-1) \\ &\leq cn \log n && \text{if } (c \geq 1) \end{aligned}$$

n odd:

$$\begin{aligned} T(n) &= T\left(\frac{n-1}{2}\right) + T\left(\frac{n+1}{2}\right) + (n-1) \\ &\leq c\left(\frac{n-1}{2}\right) \log\left(\frac{n-1}{2}\right) + c\left(\frac{n+1}{2}\right) \log\left(\frac{n+1}{2}\right) + (n-1) \end{aligned}$$

Fact: $\log\left(\frac{n+1}{2}\right) < \log\left(\frac{n}{2}\right) + 1,\ \forall n \geq 3.$

$$\begin{aligned} &\leq c\left(\frac{n-1}{2}\right) \log\left(\frac{n}{2}\right) + c\left(\frac{n+1}{2}\right)\left(\log\left(\frac{n}{2}\right) + 1\right) + (n-1) && (\forall n \geq 3)\text{Wrong after here} \\ &\leq cn \log n + c\left(\frac{n+1}{2}\right) + (n-1) && \left(n \log n \geq c\left(\frac{n+1}{2}\right) + (n-1)\right) \\ &\leq 2cn \log n && (c \geq 2, n \geq 2) \end{aligned}$$

Can't have a constant that is continuously growing. $2c$ is a growing constant. Fix? Do better algebra or add a lower order term.

Substitution - Changing Variables: Change the variable n in terms of another variable. Can also assign a different recurrence function for T.

Given 2 rankings, how do you compare similarity? We want to just compare the relative ordering in the 2 rankings.

Counting Inversions: Given 2 rankings of items $\{a_1, \ldots, a_n\}$, find the number of inverted pairs of items between the rankings; i.e. pairs where on ranking prefers a_i over a_j but the other prefers a_j over a_i.

Observe: If we draw edges between the same items in both rankings, the number of edge crossings is the number of inversions.

An equivalent formulation is to assign numbers to each item, then compare order of numbers. For simplicity, assign numbers in order to first ranking.

$$B\ D\ C\ A \implies 1\ 2\ 3\ 4$$

$$A\ D\ B\ C \implies 4\ 2\ 1\ 3$$

Problem of counting the number of inversions now becomes: count number of pairs that are out of order in the 2nd list.
Brute force: check all $\binom{n}{2}$ pairs, requires $O(n^2)$ time.
Divide and Conquer:

- Divide L into 2 lists at $m = \lceil \frac{n}{2} \rceil : A = a_1, \ldots, a_m$ and $B = a_{m+1}, \ldots, a_n$.
- Recursively count number of inversions in A and B, return counts r_A and r_B.
- Combine the results: $r_A + r_B + r$. r = number of inversions with one element in A and one in B; i.e. number of pairs (a_i, a_j) with $a_i \in A$ and $a_j \in B$ and $a_i > a_j$.

How to find r? Count for each $a_j \in B$ count the number of items, r_j, in A that are larger than a_j; i.e. $r = \sum_{a_j \in B} r_j$.
It would help if A and B are sorted. We can modify mergesort to compute r by modifying the merge process. When a_j is merged, $r_j \leq k$ for k items left in A to merge. So $r = \sum r_j$.

Algorithm: Sort-and-Count(L) returns a sorted L and number of inversions.

- Divide L at midpoint into A and B
- Sort-and-Count(A, acc) returns (sorted A, r_A). Sort-and-Count(B, acc) returns (sorted B, r_B).
- $r \leftarrow 0$. Merge(A, B) and when an element of B is chosen to merge, $r \leftarrow r+$ number of elements remaining in A, return (sorted $A \cup B, r_A + r_B + r$)

Common Recurrences: We often see recurrences of the form

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k$$

- $2T\left(\frac{n}{2}\right) + cn \in O(n \log n)$
- $T\left(\frac{n}{2}\right) + cn \in O(n)$
- $4T\left(\frac{n}{2}\right) + c \in O(n^2)$

Master Theorem Given $T(n) = aT\left(\frac{n}{b}\right) + cn^k$ where $a \geq 1, b > 1, c > 0, k \geq 0$, then

$$T(n) \in \begin{cases} \Theta(n^k) & (a < b^k, \text{i.e. } \log_b a < k) \\ \Theta(n^k \log n) & (a = b^k) \\ \Theta(n^{\log_b a}) & (a > b^k) \end{cases}$$

Multiplying Large Numbers: With basic way, multiplying two n-digit numbers $\in O(n^2)$.

Divide and Conquer method: Split numbers in half (by digits), multiply smaller components.
Analysis: $T(n) = 4T(n/2) + O(n)$, apply Master theorem, $a = 4, b = 2, k = 1$ and compare a and b^k. So $T(n) \in \Theta(n^{\log_b a}) \in \Theta(n^2)$.

Karatsuba's Algorithm: Avoid one of the four multiplications. Consider 0667×1234 where $w = 06, x = 67, y = 12, z = 34$, then

$$\begin{aligned} wx \times yz &\implies w|x \times y|z \\ &= (10^2 w + x) \times (10^2 y + z) \\ &= 10^4 wy + 10^2(wz + xy) + xz \end{aligned}$$

Don't need wz, xy individually, only the sum $(wx + xy)$.

$$(w + x) \times (y + z) = wy + (wz + xy) + xz$$

Algorithm (only 3 mults):

$$\begin{aligned} p &\implies wy \\ q &\implies xz \\ r &\implies (w + x) \times (y + z) \\ \text{return } & 10^4 p + 10^2(r - p - q) + q \end{aligned}$$

Analysis: $T(n) = 3T(n/2) + O(n)$. Master Theorem: $a = 3, b = 2, k = 1$ and compare a with b^k.

$$a = 3 > b^k = 2, \text{Case 3: } T(n) \in \Theta(n^{\log_b a}) \in \Theta(n^{\log_2 3})$$

If two numbers have different sizes, then the runtime is $O((n/m)m^{\log_2 3})$ or $O(nm^{0.585})$.

Matrix Multiplication: Instance: Two $n \times n$ matrices, A and B. Question: Compute the $n \times n$ matrix produce $C = AB$.

Naive algorithm takes $\Theta(n^3)$.

Simple Divide and Conquer: Divide into submatrices of size $n/2 \times n/2$. Requires 8 multiplications of $n/2 \times n/2$ matrices to compute AB. This still results to $\Theta(n^3)$.

Strassen's Algorithm: There are 7 subproblems instead of 8.

$$T(n) = 7T(n/2) + O(n^2) \in \Theta(n^{\log_2 7})$$

Summary Matrix Multiplication: Two $n \times n$ matrices can be multiplied in $O(n^\omega)$ where $2 \leq \omega \leq 3$.

Closest Pair Problem: Instance: A set of n distinct points in the plane. Find: Two distinct points p, q such that the distance between p and q.

$$d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

is minimized.

Brute force: Check all pairs, $O(n^2)$.
Special case: 1D: sort and compare consecutive pairs: $O(n \log n)$

Idea: Divide points in half, left half Q, right half R, dividing line L. Recursively find closest pair in Q, R. Combine - consider points with an endpoint on each side of L. It helps to sort by x-coordinate once. $O(n)$ time once they are sorted.

Let δ = min distance of 1. closest pair in Q 2. closest pair in R. Must check pair $q \in Q$, $r \in R$, looking for $d(q, r) < \delta$.

Candidates: $d(q, L) < \delta$ and $d(r, L) < \delta$.
Proof: If point is outside, distance $> \delta$.

Let S = points in this vertical strip of width 2δ. S may contain $O(n)$ or all the points. Hopefully, few points in the strip. This is similar to 1D problem - just a bit wider.

Sort by y-coordinate - only once. On the recursive subproblems, pull out relevant points in $O(n)$. Once extracted they are in sorted order.

Algorithm 1 closestPair(X, Y) (returns distance between closest pair of points)

1: $X \leftarrow$ points sorted by x-coord
2: $Y \leftarrow$ points sorted by y-coord
3: $L \leftarrow$ dividing line (middle of X)
4: Extract X_Q, X_R sorted by x-coord in region Q, R
5: Extract Y_Q, Y_R sorted by y-coord in region Q, R
6: $\delta_Q \leftarrow$ closestPair(X_Q, Y_Q)
7: $\delta_R \leftarrow$ closestPair(X_R, Y_R)
8: $\delta = \min\{\delta_Q, \delta_R\}$
9: Find S // vertical strip of width 2δ.
10: $Y_S \leftarrow S$ sorted by y-coord (extracted from Y)

Now, what do we do with S, Y_S? Hope: if $q, r \in S$ where $q \in Q, r \in R$ and $d(q, r) < \delta$, then q, r are near each other in Y_S. At most 8 points to check. To check for pair $< \delta$. For each $s \in Y_S$, check distances with next 7 points in Y_S.

QuickSelect: runtime based on where pivot falls.
Average case:

$$T(n) \leq \begin{cases} cn + \frac{1}{2}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) & (n \geq 2) \\ d & (n = 1) \end{cases} \in O(n)$$

Worst case: $T(n) = cn + T(n-1) \in O(n^2)$

BFPRT - Blum Floyd Pratt Rivest Tarjan: worst-case $\Theta(n)$, choose pivot so its close enough to the middle.
$n = 10r + 5 + \theta$ where $r \geq 1, 0 \leq \theta \leq 9$. blocks of size 5, odd number of them.

MOM - median of medians:

- Find the median of each of the blocks of 5. $O(n)$.
- Find median of these medians. r blocks have median less than MOM. 3 elements of each block $<$ MOM.
  $\implies$ $3r$ elements $<$ MOM.
  $\implies$ $+2$ from each block contain MOM
  $3r + 2$ total

$n - (3r + 2) - 1 = n - 3\left(\frac{n-5-\theta}{10}\right) - 2 - 1 = \frac{10n}{10} + \frac{-3n+15+3\theta}{10} - \frac{30}{10} = \frac{7n-15+3\theta}{10} \leq \left\lfloor \frac{7n+12}{10} \right\rfloor$ (Max size

subproblem) since $\theta \leq 9$.

$$T(n) \leq \begin{cases} T\left(\frac{n}{5}\right) + T\left(\left\lfloor \frac{7n+12}{10} \right\rfloor\right) + \Theta(n) & (n \geq 15) \\ d & (n \leq 14) \end{cases} \in O(n)$$

Chapter 3

Dynamic Programming

Def Dynamic Programming: The main idea is to solve the subproblems from smaller to larger (bottom up) and store results as you go.

Recursive Fibonacci: $T(n) = T(n-1) + T(n-2) + \Theta(1) \in O(2^n)$. There are $O(2^n)$ recursive calls.

A better approach is to use an iterative method and work up from base cases. Store the numbers in an array and loop from 1 to n.

Text Segmentation Problem: Instance: A string of letters $A[1 \dots n]$ where $A[i] \in \{A, \dots, Z\}$. Question: Can A be split into (2 or more) words?

Basic approach: Check shortest and longest prefix word, and check in between.

DP: Suppose we know Split(k) for $k = 0, \dots, n-1$ where

$$\text{Split}(k) = \begin{cases} \text{True } A[1 \dots k] \text{ is splittable} \\ \text{False} \end{cases}$$

Try Split(j) and Word($j+1, n$) for all $j = 0, \dots, n-1$. This runs in $O(n^2)$.

```
1: Split[0] ← True
2: for k ← 1 to n do
3:     Split[k] ← False
4:     for j ← 0 to k − 1 do
5:         if Split[j] and Word(j + 1, n) then
6:             Split[k] ← True
```

Longest Increasing Subsequence Problem: Instance: A sequence of numbers $A[1 \dots n]$ where $A[i] \in \mathbb{N}$. Find: The longest increasing subsequence

Let $LIS[k]$ = length of longest increasing subsequence of $A[1 \dots k]$. Not enough information to find $LIS[n]$ - length alone is not enough, need to know last number of subsequence to see if it can be extended by adding $A[n]$ or not.

Define $LISe[k]$ = length of longest increasing of $A[1 \dots k]$ that ends with $A[k]$.

To compute $LISe[k]$, consider all previous longest sequences that can be extended by $A[k]$. Run-

```
1: LISe[0] ← 1
2: for k ← 2 to n do
3:     LISe[k] ← 1
4:     for j ← 1 to k − 1 do
5:         if A[k] > A[j] then
6:             LISe[k] ← max{LISe[k], LISe[j] + 1}
```

time: $O(n^2)$.

Given $LISe[1 \dots n]$, how do you find the maximum length? Find maximum entry in $LISe$ or add dummy entry $A[n+1] = \infty$, then return $LISe[n+1] - 1$.

How do we recover the actual sequence itself? Need to also store which sequence j we extended by adding $A[k]$. Can then backtrack to recover. Runtime: $O(n^2)$ but $O(n \log n)$ is possible.

Longest Common Subsequence Problem: Instance: Two strings $x = x_1 \dots x_n$ and $y = y_1 \dots y_m$. $A[1 \dots n]$ where $A[i] \in \{A, \dots, Z\}$. Find: The longest common subsequence (common to x and y).

Let $M(i, j)$ = length of longest common subsequence of $x_1 \dots x_i$ and $y_1 \dots y_j$. How do we solve a subproblem using smaller subproblems? There are 3 possibilities: 1. Match x_i with y_j, $x_i = y_j$ 2. Skip x_i 3. Skip y_j

Base cases: $M(i, 0) = 0$ and $M(0, j) = 0$

$$M(i,j) = \max \begin{cases} 1 + M(i-1, j-1) & (x_i = y_j) \\ M(i-1, j) \\ M(i, j-1) \end{cases}$$

Solve subproblems in any order with $M(i-1, j-1), M(i-1, j), M(i, j-1)$ before $M(i, j)$. Runtime is $O(nmc)$. Find actual subsequence work backwards from $M(n, m) \to OPT(n, m)$.

```
1: function OPT(i, j)
2:     if M(i, j) = M(i − 1, j) then
3:         OPT(i − 1, j)
4:     else if M(i, j) = M(i, j − 1) then
5:         OPT(i, j − 1)
6:     else
7:         output i, j
8:         OPT(i − 1, j − 1)
```

Optimal Structure: Examine the structure of an optimal solution to a problem instance I, and determine if an optimal solution for I can be expressed in terms of optimal solutions to certain subproblems of I.

Def Elements of Dynamic Programming:

1. Optimal substructure: A problem exhibits an optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.

2. Overlapping subproblems: When a recursive algorithm revisits the same problem repeatedly, we say the optimization problem has overlapping subproblems.

Def Edit Distance: Another idea is to count the number of changes it would take to modify one string into the other. A change is one of

- add a letter (gap)
- delete a letter (gap)
- replace a letter (mismatch found)

This is called edit distance.

Edit Distance Problem: Instance: Two strings $x = x_1 \dots x_m$ and $y = y_1 \dots y_n$. Find: The edit distance between x and y, i.e. find the alignment that gives the minimum number of changes.

Subproblem: $M(i, j)$ = minimum number of changes to match $x_1 \dots x_i$ and $y_1 \dots y_j$.

Possible changes:

- match x_i to y_i at a replacement cost if characters are different
- match x_i to blank (delete x_i)
- match y_j to blank (add y_j)

Recurrence relation

$$M(i,j) = \min \begin{cases} M(i-1, j-1) & (x_i = y_j) \\ r + M(i-1, j-1) & (x_i \neq y_j) \\ d + M(i-1, j) & (\text{match } x_i \text{ to blank}) \\ a + M(i, j-1) & (\text{match } y_j \text{ to blank}) \end{cases}$$

where r is replacement cost, d delete cost and a add cost. Count number of changes: $r = d = a = 1$. Cost may be more sophisticated. Runtime: $O(mnc)$. Space: $O(mn)$.

Interval Scheduling Problem: Instance: A set of intervals I. Find: A maximum size subject of disjoint intervals.

Weighted Interval Scheduling Problem: Instance: A set of intervals I and weights $w(i)$ for each $i \in I$. Find: A set $S \subseteq I$ such that no two intervals overlap and $\sum_{i \in S} w(i)$ is maximized.

Subproblems: Let $M(i)$ = max weight subset of intervals $1 \dots i$. We can either choose interval i or not.

$$M(i) = \max \begin{cases} M(i-1) & \text{if we don't choose } i \\ w(i) + M(X) & \text{if we choose } i \end{cases}$$

Want X to be the intervals that disjoint from i but also to be labelled less than i (so they are smaller subproblems).
Can we somehow order the intervals? Yes, call this set $p(i)$.

$$M(i) = \max \begin{cases} M(i-1) & \text{if we don't choose } i \\ w(i) + M(p(i)) & \text{if we choose } i \end{cases}$$

Order intervals $1, \ldots, n$ by right endpoint. Intervals disjoint from i are $1, \ldots, j$ for some j.
For each i, let $p(i)$ = largest index $j < i$ s.t. interval j is joint from i.

Runtime: $O(n \log n)$ to sort n subproblems, each $O(n) \implies O(n^2)$.
Space: $O(n^2)$ to sort n sets of size $O(n)$.
Improvements: 1. Compute all $p(i)$ values first to save time 2. Compute S by backtracking to save space.

Compute all $p(i)$ first. Sort $1 \ldots n$ by right endpoint and sort by left endpoint $l_1, \ldots, l_n$.

```
1: j ← n
2: for k from n down to 1 do
3:     while l_k overlaps j do
4:         j ← j − 1
5:     p(l_k) ← j
```

Revised algorithm:

```
1: Sort by finish time
2: Sort by start time
3: Compute all p(i)
4: M(0) ← 0
5: for i ← 1 to n do M(i) = max{M(i − 1), w(i) + M(p(i))}
```

Recover S by recursive backtracking General idea: is an item in or out.

```
S − OPT(i)
1: if i = 0 then
2:     return ∅
3: else if M(i − 1) ≥ w(i) + M(p(i)) then
4:     return S − OPT(i − 1)
5: else
6:     return {i} ∪ S − OPT(p(i))
```

Maximum Weight Independent Set Problem: Instance: A set of elements I, weights $w(i)$ for each $i \in I$ and a set C of conflicts where $(i, j) \in C$ if elements i and j conflict. Find: A maximum weight subset $s \subseteq I$ with no conflicting pairs of items.

Constructing an Optimal Binary Search Tree Problem: Instance: A set of items $I = \{1, \ldots, n\}$ and probability $p_1, \ldots, p_n$ where p_i is the probability that item i will be searched. Find: A BST that minimizes the search cost $\sum_{i \in I}(p_i) \cdot \text{ProbeDepth}(i)$.

ProbeDepth(i) = 1 + Depth(i). The root node has depth= 0 but ProbeDepth= 1; i.e. it takes 1 probe to reach it. Search cost: # nodes$\cdot$ ProbeDepth $\cdot$ probability.

Dynamic Programming approach: Try all choices for root node,

- Suppose root will be some k

- Left subtree is then the optimal BST on $1, \ldots, k-1$
- Right subtree is then the optimal BST on $k+1, \ldots, n$

Subproblems: Let $M[i, j]$ be the optimal BST on items $i, \dot{,} j$.

$$M[i,j] = \min_{k=i \ldots j} \{M[i, k-1] + M[k+1, j]\} + \sum_{t=i}^{j} p_t$$

- One of the nodes $k \in i, \ldots, j$ will be the root so contributes $1 \cdot 1 \cdot p_k$
- $M[i, k-1]$ gives the search cost for this tree but doesn't consider it is the left subtree of k so we need to add $\sum_{t=i}^{k-1} p_t$
- Similarly, for right subtree, we add $\sum_{t=k+1}^{j} p_t$

Let $P[i] = \sum_{t=1}^{i} p_t$ where $P[0] = 0$, so, $\sum_{t=i}^{j} p_t = P[j] - P[i-1]$. Runtime: $O(n^2 \cdot n) = O(n^3)$

```
1:  for i ← 1 to n do
2:      M[i, i] ← p_i                                  ▷ Single node tree
3:      M[i, i − 1] ← 0                                ▷ Empty tree
4:  for d ← 1 to n − 1 do                              ▷ d = j − 1 from above
5:      for i ← 1 to n − 1 do                          ▷ Find M[i, i + d]
6:          best ← ∞
7:          for k ← i to i + d do
8:              temp ← M[i, k − 1] + M[k + 1, i + d]
9:              if temp < best then best ← temp
10:         M[i, i + d] ← best +P[i + d] − P[i − 1]
```

0-1 Knapsack Problem: Instance: A set of items $\{1, \ldots, n\}$ where item i has weight w_i and value v_i and a knapsack with capacity W. Find: A subset of items S such $\sum_{i \in S} w_i \leq W$ so that $\sum_{i \in S} v_i$ is maximized.

Dynamic programming approach: Consider items $1, \ldots, i$, is item i in or out?

- If $i \notin S \implies$ Optimal solution on $1, \ldots, i-1$
- $i \in S \implies$ If we take i, what subproblem do we want?
  - Maximize $\sum$ values considering items $1, \ldots, i-1$
  - Reduced weight (capacity left after taking i): $\sum$ weight $\leq W - w_i$

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

Algorithm 2 0-1 Knapsack

1: $M[0, w] \leftarrow 0$ for $w = 0, \ldots, W$
2: **for** $i \leftarrow 1$ to n **do**
3: **for** $w \leftarrow 0$ to W **do**
4: compute $M[i, w] \leftarrow M(i, w)$
5: **function** $M(i, w)$
6: **if** $w_i > w$ **then** ▷ Can't choose i, don't have weight
7: $M(i, w) \leftarrow M(i-1, w)$
8: **else** ▷ Option of choosing i
9: $M(i, w) \leftarrow \max \begin{cases} M(i-1, w) \\ M(i-1, w-w_i) + v_i \end{cases}$

Runtime: $O(nW)$. This is pseudopolynomial. W uses k bits so $k \in \Theta(\log W)$. Therefore, the runtime is $O(n \cdot 2^k)$. Thus, it is exponential in the size of the input. Runtime is polynomial in value of W rather than the size of W.

Recover items: 1. Backtracking - can you use M to backtrack (3)

Algorithm 3 Backtracking

1: **while** $i > 0$ **do**
2: **if** $M(i, w) = M(i-1, w)$ **then**
3: $i \leftarrow i - 1$
4: **else**
5: $S = S \cup \{i\}$
6: $i \leftarrow i - 1$
7: $w = w - w_i$

2. Store decision: When we set $M(i, w)$ also set $Take(i, w)$. Don't need to do the comparison, must still backtrack through subproblems

3. Store the set of items. Cost a lot of extra space, but fast to recover items taken.

Def Memoization: Is an optimization technique that stores the result of expensive function calls and return the stored result instead of recomputing.

- Use recursion, rather than explicitly solving all subproblems bottom-up
- When you solve a subproblem, store the solutions. Before resolving a problem, check if you have stored the solution. Solutions can be stored in a matrix or in a hash table.
  Some langugages help you implement memoization: `memoized-call(fact(n))` in Python, `option remember` in Maple

Advantages: maybe don't have to solve all the subproblems.
Disadvantages: harder to analyze runtime, recursion adds extra overhead - runtime stack, etc.

Chapter 4

Greedy Algorithms

Optimization Problems

Def Problem: Given a problem instance, find a feasible solution that maximizes (or minimizes) a certain objective function.

Def Problem Instance: Input for a specified problem

Def Problem Constraints: Requirements that must be satisfied by any feasible solution.

Def Feasible Solution: For any problem instance I, $feasible(I)$ is the set of all outputs for I that satisfy the given constraints.

Def Objective Functions: A function $f : feasible(I) \to \mathbb{R}^+ \cup \{0\}$. We often think of f as being a profit or a cost fucntion.

Def Optimal Solution: A feasible solution $X \in feasible(I)$ such that the profit $f(X)$ is maximized/minimized.

Making Change Problem: Instance: A set C of coins denominations for a coin system and a given amount M. Find: The minimum number of coins of denominations from C that sum to M.

Def Partial Solutions: A tuple $[x_1, \ldots, x_i]$ where $i < n$ is a partial solutioni if no constraints are violated.

Def Choice set: For a partial solution, we define the choice set

$$choice(X) = \{y \in \mathcal{X} : [x_1, \ldots, x_i, y] \text{ is a partial solution}\}$$

Def Greedy Algorithm: Starting with the empty partial solution, repeatedly extend it until a feasible solution X is constructed. A feasible solution may or may not be optimal.
There is no looking ahead and no backtracking. Often they consist of a preprocessing step, followed by a single pass through. Only one feasible solution is constructed. The execution of a greedy algorithm is based on local criteria.

Interval Scheduling or Activity Selection Problem Instance: A set $\mathcal{I} = \{1, \ldots, n\}$ of intervals, where for all $1 \leq i \leq n$, $i = [s_i, f_i)$ where s_i is start time and f_i is finish time. Find: A subset $S \subseteq \mathcal{I}$ of pairwise disjoint intervals of maximum size. i.e. $\max |S|$.

Select the activity with the earliest finishing time, i.e. the local evaluation criterion is f_i is the

optimal choice.

```
1: Sort intervals by finish time and relabel so f_1 ≤ ··· ≤ f_n
2: S = ∅
3: for i ← 1 to n do
4:     if interval i is pairwise disjoint with all intervals in S then
5:         S ← S ∪ {i}
```

Analysis: $O(n \log n)$ to sort + $O(n)$ to loop $\implies O(n \log n)$.
Correctness - 2 approaches: 1. Greedy always stays ahead 2. "Exchange" proof

Lemma The greedy algorithm returns a maximum size set A of disjoint activities.

Proof: Suppose greedy algorithm returns $a_1, \ldots, a_k$ sorted by endtime. Suppose an optimal solution is: $b_1, \ldots, b_k, b_{k+1}, \ldots, b_l$ where $l \geq k$ sorted by endtime.

a_1 ends before b_1 - greedy always chooses interval that ends first. $\implies a_1, b_2, \ldots, b_k, b_{k+1}, \ldots, b_l$ is optimal because $end(a_1) \leq end(b_1)$ so a_1 does not intersect with $b_2, b_3, \ldots$.

b_2 does not intersect with a_1 otherwise greedy would not have chosen it and greedy chose a_2 over b_2 so $end(a_2) \leq end(b_2) \implies$ intervals are disjoint $\implies a_1, a_2, b_3, \ldots, b_k, b_{k+1}, \ldots b_l$.

Induction step: Suppose $a_1, \ldots, a_{k-1}, b_k, \ldots, b_l$ is an optimal solution. b_k does not intersect a_{k-1} so the greedy algorithm could have chosen it; however, it chose a_k instead so $end(a_i) \leq end(b_k)$

a_k is then disjoint from all b_i for all $k+1 \leq i \leq l$. Thus, we can replace b_k with a_k.

This proves the claim. To finish proving, we argue $k < l$ then $a_1, \ldots, a_k, b_{k+1}, \ldots, b_l$ is an optimal solution. But then the greedy algorithm would have more choices after a_k.

Scheduling to Minimize Lateness Problem: Instance: A set of jobs $\{1, \ldots, n\}$ where job i requires time t_i to complete and has a deadline of d_i. Find: A schedule, allowing some jobs to be late but minimizing the maximum lateness.

Observation 1: Once you start a job, always complete it.
Observation 2: There is never any value in taking a break.

Do the jobs by deadline is optimal. Order jobs by deadline so $d_1 \leq d_2 \leq \cdots \leq d_n$.

Exchange proof: Let $1, \ldots, n$ be the ordering of jobs by greedy algortihm, i.e. $d_1 \leq d_2 \leq \cdots \leq d_n$.

Consider an optimal ordering. If it matches the greedy solution, then we are done and greedy is optimal. If not, there must be 2 jobs that are consecutive but in the wrong order. There are jobs i, j with the deadline of $d_j \leq d_i$.

Claim: Swapping i and j gives a new optimal solution. New optimal solution has fewer inversions. $l_G(j) \leq l_O(j)$, $l_G(i) \leq l_O(j)$ for $d_j \leq d_i$. Therefore, $l_G \leq l_O(j) \leq l_O$.

Def Exchange Proofs: Show how we can convert an optimal solution into the greedy solution.

- Let G be the solution produced by the greedy algorithm. Let O be an optimal solution
- If G is the same as O then greedy is also optimal. If $G \neq O$ then find a pair of items that are out of order in O when compared with G.

- Show that by exchanging the order of these two items, we create a new solution that is better (or at least no worse); i.e. resulting solutions remains optimal
- By making a number of exchanges we will obtain the greedy solution and since each exchange makes the solution no worse, the greedy algorithm is also optimal.

Knapsack Problem: Instance: A set of items $1, \ldots, n$ with values $v_1, \ldots, v_n$, weights $w_1, \ldots, w_n$ and a capacity, W. These are all positive integers. Feasible solution: An n-tuple $X = [x_1, \ldots, x_n]$ where $\sum_{i=1}^{n} w_i x_i \leq W$. In the 0-1 Knapsack problem, we require $x_i \in \{0, 1\}$. In the Fractional Knapsack problem, we require that $x_i \in \mathbb{Q}$. Find: A feasible solution X that maximizes $\sum_{i=1}^{n} v_i x_i$.

Possible greedy strategies:

1 Items in decreasing order of value (local evaluation criterion is p_i)

2 Items in increasing order of weight (local evaluation criterion is w_i)

3 Items in decreasing order of value divided by weight

0-1 Knapsack: None of the greedy choices are optimal.
Fractional Knapsack: Choosing highest value per weight is optimal.

---

x_i is the weight of item i taken

1: Sort items by value per weight and relabel so $\frac{v_1}{w_1} \geq \cdots \geq \frac{v_n}{w_n}$
2: $freeW \leftarrow W$
3: **for** $i \leftarrow 1$ to n **do**
4: $\quad x_i \leftarrow \min\{w_i, freeW\}$
5: $\quad freeW \leftarrow freeW - x_i$

Final weight: $\sum x_i = W$ (if $\sum w_i \geq W$)
Final value: $\sum \frac{v_i}{w_i} x_i$
Runtime: $O(n \log n)$ to sort, $O(n)$ to choose weights for each item

Greedy Proof of Correctness:
Claim: Greedy algorithm gives optimal solution to the fractional knapsack problem

Proof: Assume items are ordered by $\frac{v_i}{w_i}$. Let the greedy solution be

$$x_1, x_2, \ldots, x_{k-1}, x_k, \ldots, x_l, \ldots, x_n$$

and the optimal solution be

$$y_1, y_2, \ldots, y_{k-1}, y_k \ldots, y_l, \ldots, y_n$$

Suppose y is an optimal solution matches x on a maximum number of indices, say M indices. If $M = n$ then we are done, so assume $M < n$; i.e. implying the greedy solution is not optimal.

Contradiction: Show there exists an optimal solution that matches x on at least $M + 1$ indices.

Let k be the first index where $x_k \neq y_k$. Then $x_k > y_k$ since greedy always maximizes x_k since $\sum y_i = \sum x_i = W$. This implies there is a later item index l where $l > k$ such that $y_l > x_l$. So

$\frac{v_k}{w_k} \geq \frac{v_l}{w_l}$ (*) because of the ordering. Exchange Δ of item l for equal weight of item k in the optimal solution.

$$y'_k \leftarrow y_k + \Delta$$
$$y'_l \leftarrow y_l - \Delta$$
$$\Delta \leftarrow \min \begin{cases} y_l - x_l \\ x_k - y_k \end{cases}$$

Then $x_k = y'_k$ or $x_l = y'_l$.

Change in value: $\Delta\left(\frac{v_k}{w_k}\right) - \Delta\left(\frac{v_l}{w_l}\right) = \Delta\left(\frac{v_k}{w_k} - \frac{v_l}{w_l}\right) \geq 0$ by (*). y was optimal, so this can't be better. New optimal still optimal but matches on 1 more index (k or l).

Stable Marriage Problem: Instance: A set of n co-op students $S = \{s_1, \ldots, s_n\}$ and a set of n employers offering jobs, $E = \{e_1, \ldots, e_n\}$. Each employer has a preference ranking of the n students, and each student has a preference ranking of the n employers. $pref(e_i, j) = s_k$ if s_k is the jth preference of employer e_i and $pref(s_i, j) = e_k$ if e_k is the jth favourite employer of student s_i. Find: A matching of the n students with the n employers such that there does not exist a pair (s_i, e_j) who are not matched to each other, but prefer each other to their existing matches. A matching with this property is called a stable matching.

Gale-Shapley$(S, E, pref)$

```
1: Match ← ∅
2: while there exists employer e_i looking to hire do
3:     Let s_j be next student in e_i's preference list
4:     if s_j is unemployed then
5:         Match ← Match ∪ {(e_i, s_j)}
6:     else
7:         if s_j prefers e_i (over current e_k) then
8:             Match ← Match{(e_k, s_j)} ∪ {(e_i, s_j)}
9:             Note: Employer e_k now looking to hire again
   return Match
```

Overview of the Gale-Shapley Algorithm:

- Employers offer jobs to students.
- If a student accepts a job offer, then the pair are matched; the student is employed.
- An unemployed student must accept a job if they are offered one.
- If an employed student receives an offer from an employer whom they prefer to their current match, then they cancel their existing match and are matched with new employer; previous employer no longer has a match.
- If employed student receives an offer from an employer but they prefer job they already have, the offer is rejected.
- Employed students never become unemployed
- An employer might make a number of offers (up to n); the order of offers is determined by the employer's preference list

Chapter 5

Graph Algorithms

Def Graph: A graph $G = (V, E)$ where V is a set of vertices where $|V| = n$ and E is a set of edges, $E \subseteq V \times V$, where $|E| = m$ and $m \leq n^2$.

Def Directed Graph (Digraph): Edges can be undirected (unordered pairs) or directed (ordered pairs). A graph with directed edges is called a directed graph or digraph.

Def Neighbour: $u, v \in V$ are adjacent or neighbours if $(u, v) \in E$.

Def Incident: $v \in V$ is incident to $e \in E$ if $e = (v, u)$.

Def Degree: $\deg(v)$ is number of incident edges to v. $\text{indegree}(v)$ and $\text{outdegree}(v)$ are the number of incident edges direct into v and directed out of v.

Def Path: A path is a sequence of vertices $v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$.

Def Cycle: A cycle is a path that starts and ends at the same vertex.

Def Connectedness: An undirected graph is connected if there exists a path join all pairs $u, v \in V$.

Def Tree: A tree is a connected (undirected) graph with no cycles.

Def Connected Component: A connected component of a graph is a maximal connected subgraph.

Def Adjacency Matrix: The adjacency matrix of G is an $n \times n$ matrix A, requiring $O(n^2)$ space, which is index by V, such that

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

There are exactly $2m$ entries in A equal to 1. For a directed graph, there are m entries of A equal to 1.

Def Adjacency List: Representation of G consisting of n linked lists. Space: $O(n + m)$.

- For every $u \in V$, there is a linked list which is named $Adj[u]$.
- For every $v \in V$ such that $uv \in E$, there is a node in $Adj[u]$ labelled v.
- In an undirected graph, every edge uv corresponds to nodes in two adjacency lists: there is a node v in $Adj[u]$ and a node u in $Adj[v]$.

- In a directed graph, every edge corresponds to a node in only one adjacency list.

Operations

Operation	Adjacency Matrix	Adjacency Lists
Space	$O(n^2)$	$O(n+m)$
$(u,v) \in E$?	$\Theta(1)$	$O(1+\deg(u))$
List v's neighbours	$\Theta(n)$	$\Theta(1+\deg(v))$
List all edges	$\Theta(n^2)$	$\Theta(n+m)$

Exploring Graphs To explore a graph, we want to visit all vertices, or all vertices starting at some source. Two typical strategies are breadth first search and depth first search. To keep track of where we have been or go next, we mark vertices as undiscovered/discovered or unexplored/explored.

- A vertex is discovered meaning we have identified it as a place we want to go but have no yet visited.
- A vertex has been explored once we visit the node and perform the work needed to be done at the node.

Def Breadth First Search (BFS): Start at a specified vertex v_0. A cautious search: Spreads out from v_0 by checking everything one edge away, then two, etc.

First, from v_0 discover all neighbours of v_0. Next explore all neighbours of v_0 to discover the neighbours of neighbours. Mark all vertices as discovered. This process continues until all vertices have been explored.

Implementation: Use a queue to keep track of the vertices that have been discovered but must still be explored. Also useful to store parent and level information.

BFS

```
1: Initialization: Mark all vertices as undiscovered
2: Pick initial vertex v_0
3: parent(v_0) ← ∅
4: level(v_0) ← 0
5: Queue.add(v_0)
6: mark(v_0) ← discovered
7: while Queue is not empty do
8:     v ← Queue.remove()
9:     for each neighour u of v do                          ▷ Explore(v)
10:        if mark(u)= undiscovered then
11:            mark(u) ← discovered
12:            parent(u) ← v
13:            level(u) ← level(v) + 1
14:            Queue.add(u)
```

Runtime: Explore each vertex once and check all incident edges: $O(n+\sum_{v\in V}\deg(v)) = O(n+m)$. Note: $\sum_{v\in V}\deg(v) = 2m$ by handshake lemma.

Properties of BFS:

- The parent pointers create a directed tree (because each addition adds a new vertex u, with parent v in the tree).
- u is connected to v_0 if and only if BFS from v_0 reaches u.

Lemma The level of a vertex v = length of shortest path from v_0 to v.

Claim 1 v in level i implies there is a path v_0 to v of i edges.

Claim 2 v in level i implies every path v_0 to v has $\geq i$ edges.

Consequences

1. BFS from v_0 finds the connected component of v_0.
2. BFS finds all the shortest paths (number of edges) from v_0.

Applications:

- Find the shortest path from a root vertex v_0 to any node v = level of v.
- Test if a graph has a cycle.
- Test if a graph is bipartite

Def Bipartite Graph: A graph G is bipartite if V can be partitioned into $V_1 \cup V_2$ where $V_1 \cap V_2 = \emptyset$ such that every edge has one end in V_1 and one end in V_2.

Lemma G is bipartite if and only if G has no odd cycle.

Testing for Bipartite Let V_1 be the even layers and V_2 be the odd layers. Run BFS. For each edge $(u, v) \in E$ check if $u, v \in V_1$ or $u, v \in V_2$. If no such edge is found, then G is bipartite, otherwise if an edge is found it is not bipartite.

Since level(u) and level(v) differ by 0 or 1, if they are in the same vertex set, the difference must be 0 $\implies$ u, v are on the same level. That edge will create an odd length cycle.

Def Depth First Search (DFS): Start at a vertex v_0. A bold search: go as far away as you can. From v_0 discover a new neighbour and go explore it. Reach until you reach a vertex with no undiscovered neighbours. Backtrack and repeat.

Implementation: Marked a vertex as finished when all of its neighbours have been explored. Useful to store parent and also if an edge is a tree edge or non-tree edge.

DFS-Main

```
1: Initialization: Mark all vertices as undiscovered
2: for v ∈ V do
3:     if v is undiscovered then
4:         DFS(v)
5: function DFS(v)
6:     mark(v) ← discovered
7:     for u ∈ AdjacencyList(v) do
8:         if u is undiscovered then
9:             DFS(u)
10:            parent(u) ← v
11:            (u, v) is a tree edge
12:        else
13:            (u, v) is a non tree edge if u ≠ parent(v)
14:    mark(v) ← finished
```

Properties of DFS:

- Partitions G into separate trees (connected components)
- Gives an edge classification
- Vertex orderings: order of discovery, order of finishing

Lemma DFS(v_0) reaches all vertices connected to v_0.

Lemma All non tree edges join an ancestor and a descendant (vertices on same branch).

Enhanced-DFS(v)

```
1: time ← 1
2: mark(v) ← discovered
3: time ← time + 1
4: for u ∈ AdjacencyList(v) do
5:     if u is undiscovered then
6:         DFS(u)
7: mark(v) ← finished
8: finish(v) ← time
9: time ← time + 1
```

Def Cut Vertex: A vertex v is a cut vertex if removing v makes G disconnected. DFS can find cut vertices.

Claim The root is a cut vertex $\iff$ it has > 1 child.

Lemma A non-root v is a cut vertex $\iff$ v has a subtree T with no non-tree edge going to a proper ancestor of v.

How do we check if a vertex in a subtree T of v has a non-tree edge going to a proper ancestor of v or not? Use the discovery times.

Let u be the root of a subtree T of v, x a descendant of u, w is a proper ancestor. (x, w) is a non=tree edge. Define $low(u) = \min\{d(w)\}$. Compute $low()$ recursively.

$$low(u) = \min \begin{cases} \min\{d(w) : (u, w) \in E\} \\ \min\{low(x) : x \text{ a child of } u\} \end{cases}$$

You can enhance DFS or run DFS to compute discovery times then for every vertex u in finish time order, use the $low()$ above. Handle root node separately.

Classifying Edges

- An edge in the DFS tree is a tree edge.
- Forward edge: a non-tree edge (v, u) where u is a descendant of v.
- Back edge: a non-tree edge (v, u) where u is an ancestor of v.
- Cross edge: a non-tree edge (v, u) where u is not a descendant of v and v is not a descendant of u.

DFS(v) - on directed graphs

```
1:  mark(v) ← discovered
2:  discover(v) ← time
3:  time ← time + 1
4:  for u ∈ AdjacencyList(v) do
5:      if u is undiscovered then
6:          DFS(u)
7:          (v, u) is a tree edge
8:      else
9:          if u is not finished then (v, u) is a back edge
10:         else if discover(u) > discover(v) then (v, u) is a forward edge
11:         else(v, u) is a cross edge
12: mark(v) ← finished
13: finish(v) ← time
14: time ← time + 1
```

Lemma A directed graph has a (directed) cycle $\iff$ DFS has a back edge.

Proof: Suppose there is a directed cycle. Let v_1 be the first vertex discovered in DFS. Number of vertices in cycle is $v_1, \ldots, v_k$. Claim: (v_k, v_1) is a back edge. Because we must discover and explore all v_i before we finish v_1. When we test edge (v_k, v_1), we label it a back edge. v_1 is an ancestor of v_k.

Def Topological Sort: Topological sort of a directed acyclic graph. A directed edge (a, b) means a must come before b. Find a linear order of vertices satisfying all edge constraints. Reverse finish order.

One possible solution: Choose a vertex with no in-edges and remove it, then go again.

DFS solution: Use reverse of finish order. Proof: for every directed edge (u, v), $finish(u) > finish(v)$.

- Case 1: u discovered before v. Then because of $(u, v) \in E$, v is dicovered and finished before u is finished.
- Case 2: v discovered before u. G has no directed cycle, we can't reach u in $DFS(v)$. So v then must finish before u is discovered and finished.

Def Strongly Connected: For all vertices u, v, there is a path from u to v.
Let s be a vertex. For a directed graph, G is strongly connected $\iff$ for all vertices v, there is a path from s to v and a path from v to s.

Testing strongly connected in directed graphs: Test for a path $s \to v, \forall v$, call $DFS(s)$. To test for a path $v \to s$, reverse the edge directions and do $DFS(s)$.

Lemma Trees in the 2nd DFS when testing strongly connected are exactly the strongly connected components.

Proof: Vertices u, v are strongly connected iff they are in the same DFS tree in 2nd DFS.
$\implies$ Suppose u is discovered first, then there is a path from u to v in the reverse graph, v is discovered before u is finished.

$\impliedby$ Suppose u, v are in the same tree. Let r be the root. Claim: r and u are strongly connected. Proof: r is the root of tree containing u. So $\exists$ path from r to u in reverse graph, i.e. u to r in original. Show that $\exists$ path r to u in original graph. When we started the tree rooted at r, u was undiscovered. We picked r because it has a higher finish time in first DFS than u.

We have u to r path in original. If u discovered before r, then this implies r has a finish time that comes before u. Contradiction, this can't be true. So r is discovered before u and finished later. Implying u is a descendant of r implying a path r to u in original graph.

Minimum Spanning Tree (MST) Problem: Instance: Given a connected graph $G = (V, E)$ with weights $w : E \to \mathbb{R}$ on the edges. Find: A subset of the edges of size $n - 1$ that connects all the vertices and has minimum weight. The edge subset is called a minimum spanning tree.

Def Tree Property: Any connected graph on n vertices and $n - 1$ edges is a tree.

Kruskal's Algorithm

```
1: Order edges by weight: e_1, ..., e_m s.t. w(e_i) ≤ w(e_{i+1})
2: T ← ∅
3: for i ← 1 to m do
4:     if e_i does not make a cycle with T then
5:         T ← T ∪ {e_i}
```

Analysis: $O(m \log m)$ to sort edges but $m \leq n^2$, so $\log m \leq 2 \log n \in O(\log n)$, so runtime is $O(m \log n)$.

Correctness proof: Base case $i = 0$ is trivially true.

Assume by induction there is a MST M matching T on the first $i - 1$ edges. Alg

$$T = t_1, \ldots, t_{i-1}, t_i, \ldots, t_{n-1}$$

and MST

$$M = m_1, \ldots, m_{i-1}, m_i, \ldots, m_{n-1}$$

Let $t_i = e = (a, b)$ and let C be the connected component of T containing a. Note: when the alg considered t_i, all edges of weight $< w(t_i)$ have been considered and non of them go between C and $V - C$. In MST M, there is a path from a to b, it must cross from C to $V - C$ on edge e'. Then $w(e) \leq w(e')$ by ordering. So e' is later in the ordering in M.

Exchange: Let $M' = (M - \{e'\}) \cup \{e\}$. Claim: M' is a MST. This is enough since M' now matches T on i edges.
Proof: 1. M' is a spanning tree; replace e' with e ($a' \to b', a' \to a$, e, $b \to b'$) 2. $w(M') = w(M) - w(e') + w(e) \leq w(M)$ so M' is a MST.

Union-Find Problem: Maintain a collection of disjoint sets with operations:

- Find(x) - determine which set contains element x.
- Union - unite 2 sets.

Implementation: Array $S[1 \dots n]$, $S[i]$ = component of element i and linked list of elements in each set. We can find in $O(1)$. To union, we have to join two linked lists, they are both sets so it takes $O(1)$, but we need to update S which takes $O(|\text{set}|)$.

Every time we look for an edge to connect in Kruskal's, we do the find operation. For MST, elements are vertices and sets are connected components of T so far. Kruskal gives this ADT $O(m \log n)$ runtime.

$$\underbrace{O(m \log n)}_{\text{sort}} + \underbrace{O(m)}_{\text{find}} + \underbrace{O(n \log n)}_{\text{merge cost}}$$

Merge cost: We always use smaller set to merge into the bigger, and the new set is at least twice the size of the smaller set. Each set will only get merged at most $\log n$ times.

Prim's Algorithm

1: $C \leftarrow \{s\}$
2: $T \leftarrow \emptyset$
3: **while** $C \neq V$ **do**
4: Find a vertex $v \in V \setminus C$ such that $\exists u \in C$ with $e = (u, v)$ with min weight leaving C
5: $C \leftarrow C \cup \{v\}$
6: $T \leftarrow T \cup \{e\}$

Implementation: Need to find a vertex in $V \setminus C$ connected to C using a minimum weight edge. For $v \in V \setminus C$, define

$$weight(v) = \begin{cases} \infty & \text{if no edge } (u, v),\ u \in C \\ \min\{w(e) | e = (u, v)\} & u \in C \text{ otherwise} \end{cases}$$

Priority queue - all operations are $O(\log k)$, $k = |V \setminus C|$.

- Maintain a set $V \setminus C$ as an array in heap order according to $weight()$.
- ExtractMin() - remove and return vertex with min weight.
- Insert(v, $weight(v)$) - insert vertex v with $weight(v)$.

- Delete(v) - delete v from PQ.

Find an endpoint in C: $w((u,v)) = weight(v)$. Choose 1 to be tree edge. Any others in C, ignore. If $x \notin C$ and not in PQ - add to PQ. If $x \notin C$, but in PQ - check to update $weight(x)$.

Need a data structure to store where v is in PQ to find in $O(1)$. Create array $\bar{C}[1 \dots n]$

$$\bar{C}[v] = \begin{cases} -1 & \text{if } v \notin V \setminus C \\ \text{location of } v \end{cases}$$

Analysis: ExtractMin to add each v to C. Scan v's adjacency list to find $e = (u,v)$ with $w(e) = weight(v)$ - add to MST. Need to update/reduce weight of vertex v' such that (v,v') with $v' \in V \setminus C$. Delete and insert.

Size of heap is $O(n)$. $n-1$ ExtractMin's and $O(m)$ reduce weight operations delete and insert.

Overall: $O(m \log n)$.

Shortest Paths in Edge Weighted Graphs

Dijkstra's Algorithm Input: graph or directed graph $G = (V,E)$, $w : E \to \mathbb{R}^{\geq 0}$ and source vertex $s \in V$. Output: shortest path from s to every other vertex v.

Idea: grow tree of shortest paths starting s. General step: we have a tree of shortest paths to all vertices in B. Initially, $B = \{s\}$.

Choose an edge (x,y), $x \in B, y \notin B$ to minimize $d(s,x) + w(x,y)$ where $d(s,x)$ is the known minimum distance from s to x. Call this distance d. Greedy in a sense: always add vertex with

Dijkstra's Algorithm

```
1: d(v) ← ∞, ∀v ≠ s
2: d(s) ← 0
3: B ← ∅
4: while |B| < n do
5:     y ← vertex of V \ B with minimum d value
6:     B.add(y)
7:     if d(y) + w(y,z) < d(z) then
8:         d(z) ← d(y) + w(y,z)
9:         Parent(z) ← y
```

min weight distance from S.

Claim: d is the minimum distance from s to y.
Proof: Any path π from s to y consist of:

- π_1 = initial part in B; from $s \to u$
- $e = (u,v)$ = first edge leaving B
- π_2 = rest of path

$$w(\pi) \geq w(\pi_1) + w(u,v) \geq d(s,u) + w(u,v) \geq d$$

Using $w(\pi_2) \geq 0$. Proof breaks if we allow negative weight cycles.

By induction on $|B|$, the algorithm correctly finds $d(s,v)$ for all v.

Implementation: use min priority queue. Keep tentative distance $d(v), \forall v \notin B$. $d(v)$ = minimum weight path from s to v with all but the last edge in B.

Store the d values in a heap of size $\leq n$. Modifying a d value takes $O(\log n)$. Total time: $O(n \log n)$ to pick each vertex once + $O(m \log n)$ to adjust heap $\in O(m \log n)$ or $O((n+m)\log n)$.

Def Directed Acyclic Graph (DAG): A directed acyclic graph has no directed cycles.

Single Source Shortest Paths in a DAG Idea: use topological sort $v_1 v_2 \dots v_n$ so every edge (v_i, v_j) has $i < j$. If v comes before s, there is no path $s \to v$ so remove all such vertices, relabel, let $s = v_1$. Runtime: $O(n+m)$

1: $d_i \leftarrow \infty, \forall i$
2: $d_1 \leftarrow 0$
3: **for** $i = 1$ to n **do**
4: **for** every edge (v_i, v_j) **do**
5: **if** $d_i + w(v_i, v_j) < d_j$ **then**
6: $d_j \leftarrow d_i + w(v_i, v_j)$

Claim: This finds shortest paths from s. Use induction on i.

Dynamic Programming for Shortest Paths

How do we define a subproblem for shortest paths problem? Consider a uv path that goes through a vertex $x \implies$ consists of the shortest ux path + shortest xv path. In what sense do we consider these smaller?

- Fewer edges: try paths of ≤ 1 edge, ≤ 2 edges, etc. We will use this for the single source shortest paths algorithm.
- They don't use x. We will use this for all pairs shortest paths algorithm.

Single Source Shortest Paths Problem: Let $d_i(v)$ be the weight of the shortest path from s to v using $\leq i$ edges. Then,

$$d_1(v) = \begin{cases} 0 & \text{if } v = s \\ w(s,v) & \text{if } (s,v) \in E \\ \infty & \text{otherwise} \end{cases}$$

and we want to find $d_{n-1}(v)$. Why $n-1$ edges?

- A path with $\geq n$ edges would repeat a vertex giving a cycle.
- Every cycle has weight ≥ 0, remove the cycle is no worse.

Compute d_i from d_{i-1}:

$$d_i(v) = \min \begin{cases} d_{i-1}(v) & \text{use} \leq i-1 \text{ edges} \\ \min_u\{d_{i-1}(u) + w(u,v)\} & \text{use } i \text{ edges} \\ \infty & \text{otherwise} \end{cases}$$

Bellman-Ford Algorithm: Edge weights may be negative but cannot have negative weight cycles.

Bellman-Ford Algorithm

```
1: Initialize d_1(v) for all v
2: for i = 2 to n − 1 do
3:     for v ∈ V do
4:         d_i(v) ← d_{i−1}(v)
5:         for each edge (u, v) do // want edges directed into v
6:             d_i(v) ← min{d_i(v), d_{i−1}(u) + w(u, v)}
```

Runtime: $O(n(n+m))$

We can save space, reuse same $d(v)$ and simplify the code:

Bellman-Ford Algorithm-Improved

```
1: d(v) ← ∞ for all v
2: d(s) ← 0
3: for i = 1 to n − 1 do
4:     for (u, v) ∈ E do
5:         d(v) ← min{d(v), d(u) + w(u, v)}
```

All Pairs Shortest Paths Problem: Instance: A directed graph G with edge weights $w : E \to \mathbb{R}$ (but no negative weight cycle). Find: The shortest path from u to v for all u, v. Output the distances as an $n \times n$ matrix $D[u, v]$.

Idea: Use dynamic programming where intermediate paths use only a subset of the vertices.

Let $V = [n]$. Let $D_i[u, v]$ be the length of the shortest uv path using intermediate vertices in $[i]$.

Subproblems: Solve $D_i[u, v]$ for all u, v as i goes from 0 to n. Our final solution is then $D_n[u, v]$.

Base cases:

$$D_0[u,v] = \begin{cases} 0 & \text{if } u = v \\ w(u,v) & \text{use } (u,v) \in E \\ \infty & \text{otherwise} \end{cases}$$

The main recursive property, $i > 0$, is to use i or not:

$$D_i[u,v] = \min \begin{cases} D_{i-1}[u,i] + D_{i-1}[i,v] & \text{use vertex } i \\ D_{i-1}[u,v] & \text{don't use } i \end{cases}$$

Analysis: $O(n^3)$ runtime and $O(n^3)$ space

Floyd-Warshall Algorithm

Initialize $D_0[u, v]$ by base cases
for $i = 1$ to $n - 1$ **do**
    for $u = 1$ to n **do**
        for $v = 1$ to n **do**
            $D_i[u, v] \leftarrow \min\{D_i[u, v], D_{i-1}[u, i] + D_{i-1}[i, v]\}$

Floyd-Warshall Algorithm-Improved

Initialize $D_0[u, v]$ by base cases
for $i = 1$ to $n - 1$ **do**
    for $u = 1$ to n **do**
        for $v = 1$ to n **do**
            $D[u, v] \leftarrow \min\{D[u, v], D[u, i] + D[i, v]\}$

For a new initialization D (exercise), we can reduce the space requirement to $O(n^2)$:

Recovering the actual path: create a new array $Next[u, v]$ where each location stores the first vertex after u on a shortest path from u to v.

Suppose: a shortest uv path follows: u, x, y, z, v. Then, $Next[u, v]$ returns x, $Next[x, v]$ returns y, $Next[y, z]$ returns z.

Implementation: When we update $D[u, v] \leftarrow \min\{D[u, v], D[u, i] + D[i, v]\}$, also update $Next[u, v]$.

Chapter 6

Intractibility

All the "efficient" algorithms we have learned are in polynomial time.

Alternate options:

- approximations - often know error factor, quality of solution is based on error factor
- heuristics - often okay, but no guarantee on quality or runtime
- exact solutions - very expensive

Subset Sum Problem: Instance: Given elements $1, 2, \ldots, n$ with weights $w_1, w_2, \ldots, w_n$ and a target weight W. Question: Is there a subset $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in S} w_i = W$.

This is NP-complete. We have to explore all 2^n subsets, so $O(2^n)$.

A configuration $C = (S, R)$ where $S \subseteq \{1, \ldots, i\}$ and $R = \{i, \ldots, n\}$.

1: Let A be the set of active configurations
2: A starts with a single configuration
3: **while** $A \neq \emptyset$ **do**
4: $\quad C \leftarrow$ remove a configuration from A
5: $\quad$ Explore configuration C
6: $\quad$ **if** C solves the problem **then** Done
7: $\quad$ **if** C is a dead-end **then** Discard it
8: $\quad$ **else**
9: $\quad\quad$ Expand C to child configurations $C_1, \ldots, C_t$
10: $\quad\quad$ Add each child configuration C_i to A

Store A and Traverse Configuration Space

- Store A as a stack and use DFS of configuration space. $|A|$ = height of tree.
- Store A as a queue and use BFS of configuration space. $|A|$ = width of tree.

To reduce space, store A as a stack.
Subset Sum has a tree of width 2^n but height n.
An alternate approach is to store A as a priority queue, to store the most promising configuration first.

Backtracking in Subset Sum Configuration $C = (S, R)$ where $S \subseteq \{1, \ldots, i-1\}$, the set so far and $R = \{i, \ldots, n\}$, the remaining elements.

Maintain: $w = \sum_{i \in S} w_i$ and $r = \sum_{i \in R} w_i$. Then

- If $w = W$, success.
- If $w > W$, dead-end (don't expand this configuration).
- If $r + w < W$, dead-end.

Analysis: $O(2^n)$ runtime.

Subset Sum is similar to 0-1 Knapsack. If W is small, $O(nW)$ is better, but if W has n bits, then backtracking is better.

Def Decision Problem: Given a problem instance, answer a certain question with yes or no.

Hamiltonian Cycle Problem Given a graph, find a Hamiltonian cycle - a cycle that goes through every vertex exactly once.

Travelling Salesman Problem Given a graph with edge weights, find a Hamiltonian cycle such that the sum of its weights is minimized.

Def Branch and Bound: Exhaustive search for optimization problems. Explore most promising configuration first (rather than DFS). Keep the best found so far.

- Branch: Generate children
- Bound: compute a lower bound on the objective function for a configuration and discard configurations that can not be better than what has been found.

Branch-and-Bound for Travelling Salesman Problem

- Based on enumerating all subsets of edges
- Configuration $C = (N, X)$ where $N \subseteq E$ for included edges and $X \subseteq E$ for excluded edges.

Conditions used to detect dead-ends:

- $E - X$ is connected (actually biconnected)
- N has ≤ 2 edges incident to each vertex
- N contains no cycle except on all vertices

Branch: Choose an $e \in E - (N \cup X)$ to branch on - 2 children:

```
1: Let A be the set of active configurations
2: A starts with a single configuration
3: Best-cost ← ∞
4: while A ≠ ∅ do
5:     C ← remove most-promising configuration from A
6:     Expand C to C_1, ..., C_t                                  ▷ Branch
7:     for i = 1 to t do
8:         if C_i solves the problem then
9:             if cost(C_i) < Best-cost then
10:                Best-cost ← cost(C_i)
11:        else if C_i is a dead-end then
12:            Discard C_i
13:        else if lower-bound(C_i) < Best-cost then
14:            A.add(C_i)                                         ▷ Bound
```

- e in: $(N \cup \{e\}, X)$
- e out: $(N, X \cup \{e\})$

Bound: Given a configuration (N, X), we want to compute a lower bound on the min cost TSP that includes N and excludes X.

- Discard edges in X
- Assign (temporary) weight 0 to edges in N
- Find min spanning tree on vertices $2, \ldots, n$
- Add the two min weight edges incident to vertex 1

Then compute weight 1-tree (sum real weights).

Def 1-tree: A 1-tree is a spanning tree on vertices $2, 3, \ldots, n$ plus 2 edges incident to vertex 1. i.e. A spanning tree plus 1 extra edge incident to 1.

Claim: Any TSP tour is a 1-tree. Thus, the minimum weight of TSP $\geq$ minimum weight of 1-tree. This is our lower-bound.

```
1: Let A be the set of active configurations
2: A = {C(∅, ∅)}
3: min-weight ← ∞
4: while A ≠ ∅ do
5:     C = (N, X) ← remove most-promising configuration from A
6:     Choose e ∈ E − (N ∪ X)
7:     Expand C to C_1, C_x by choosing e in or e out                ▷ Branch
8:     for i = 1 to 2 do
9:         if C_i solves the problem then
10:            if weight(C_i) < min-weight then
11:                min-weight ← weight(C_i)
12:        else if C_i is a dead-end then
13:            Discard C_i
14:        else if min-weight-1-tree(C_i) < min-weight then
15:            A.add(C_i)                                            ▷ Bound
```

Suppose we have an algorithm A for problem P with runtime $T(n)$. To show it is the best, we need to show that any algorithm has worst case runtime of at least $T(n)$ asymptotically.

Def Polynomial Time: An algorithm runs in polynomial time if its runtime (asymptotically, worst-case) is $O(n^k)$ where n is input size and k is a constant.

Def Class P: The class of decision problems that have a polynomial time algorithm to solve the problem.

Def Reduction: Problem X reduces to Problem Y, written $X \leq Y$, if an algorithm for Y can be used to make an algorithm for X. X is easier than Y.

Can use subroutine for Y to solve X.

Def Turing Reduction: Problem X reduces in polynomial time to problem Y, written $X \leq_P Y$, if a polynomial time algorithm for Y can be used to make a polynomial time algorithm for X.

Important consequences:

- If X cannot be solved in polynomial time, then Y cannot be solved in polynomial time.
- In cases we don't have an algorithm for Y or a lower bound for X, if we can show $X \leq_P Y$ and $Y \leq_P X$ then we can show the problems are equivalently hard.

Hamiltonian Cycle Problem: Given a graph G, does G contain a Hamiltonian cycle?

Hamiltonian Path Problem: Given a graph G, does G contain a Hamiltonian path?

Lemma Hamiltonian Path Problem $\leq_P$ Hamiltonian Cycle Problem

Proof: Suppose we have a polynomial time algorithm A_{cycle} for Hamiltonian cycle. We want to use A_{cycle} as a subroutine to make a polynomial time algorithm for Hamiltonian path.

We need to modify G into G' such that: G has a Hamiltonian path $\iff$ G' has a Hamiltonian cycle.

Idea

1. Add an edge from one end of path to the other end. Problem, don't have the path.
2. G' adds new vertex adjacent to all vertices in G. G' has $n+1$ vertices and $m+n$ edges. A_{cycle} still runs in polynomial time since $2n + m + 1$ is linear in size compared with G.

Correctness: G has a Hamiltonian path iff A_{cycle} returns yes iff G' has a Hamiltonian cycle.

Proof: Suppose G has a Hamiltonian path $u_1, \ldots, u_n$. Then G' has a Hamiltonian cycle

$$v, u_1, \ldots, u_n, v$$

since all all vertices are adjacent to v.

Conversely, suppose G' has a Hamiltonian cycle, then delete v to get Hamiltonian path.

Decision Problems and Optimization Problems Decision problems and optimization problems are *usually* equivalent with respect to polynomial time.

- Given a number, is it prime? vs Find the prime factorization.
- Given a graph, does it have a Hamiltonian cycle? vs Find the cycle.
- Given an edge-weighted graph G and number k, does G have a TSP tour of length $\leq k$? vs Find the tour and find the minimum k.

Def Independent Set: An independent set in a graph is a set of vertices where no two are joined by an edge.

Max Independent Set Problem Find the maximum independent set in a graph.

Optimization: Find independent set of max size.
Decision: Given k, is there an independent set of size $\geq k$.

$$\text{decision} \leq_P \text{optimization}$$

Can check if the set returned from optimization has size $\geq k$.

$$\text{optimization} \leq_P \text{decision}$$

We want to use a subroutine from decision to solve optimization. Find the max k_{opt} by testing $k = 1, \ldots, n$ using decision algorithm. This gets us maximum k_{opt}, but it is still not the set itself (answer to optimization).

Delete vertices one at a time. If max independent set on $G - v$ is equal to k_{opt}, then v wasn't in the set, so $G \leftarrow G - v$ and run again.

Claim: At end, cannot delete any more vertices, G is the independent set of size k_{opt}.

Claim: This takes polynomial time assuming algorithm for decision is polynomial time.

Def Class NP: Nondeterministic polynomial. NP is the class of decision problems that can be verified in polynomial time. The certificate is a nondeterministic guess.

Def Certificate: Informally, a certificate for a yes-instance I is some extra information C which makes it easy to verify I is a yes-instance.

Def Certificate Verification Algorithm: Suppose that Ver is an algorithm that verifies certificates for yes-instances. Then, $Ver(I, C)$ outputs yes if I is a yes-instance and C is a valid certificate for I.
If $Ver(I, C)$ outputs no, then either I is a no-instance, or I is a yes-instance and C is an invalid certificate.

Def Polynomial Time Certificate Verification Algorithm: A certificate verification algorithm Ver is a polynomial time certificate verification algorithm if the complexity of Ver is $O(n^k)$.

Subset Sum is NP: Certificate: set S.
Verification (in polynomial time), show that $\sum_{i \in S} w_i = W$.

Travelling Salesman Problem is NP Certificate: Permutation of n vertices.
Verification:

- Is the set of vertices a permutation?
- Edges in G, do they exist?
- Sum edge weights $\leq k$?
- Check if it is a cycle

Claim P $\subseteq$ NP, i.e., if X in P, then X is in NP.

Proof: The certificate is empty. The polynomial verification algorithm is simply the polynomial algorithm for X.

Def coNP: coNP is the class of decision problems where the NO instances can be verified in polynomial time.

Primes Problem Given a number, is it prime? Primes $\in$ coNP.
Certificate: numbers $a, b \in \mathbb{N}$
Verification (of NO instance): verify $a, b \geq 2$ and $ab = n$.

As of 2002, Primes $\in$ P.

Properties of P, NP, and coNP:

- P $\subseteq$ NP
- P $\subseteq$ coNP
- Any problem in NP can be solved in $O(2^{n^t})$ time by trying all certificates one at a time.

Open Problems:

- P =? NP
- NP =? coNP
- P =? NP $\cap$ coNP

Def NP-Complete: A decision problem X is NP-complete if $X \in$ NP and for every $Y \in$ NP, $Y \leq_P X$, i.e. X is (one of) the hardest problems in NP.

Implications of X being in NP

- If X can be solved in polynomial time, then all problems in NP can also be solved in polynomial time. If $X \in$ P, then P = NP.
- If X cannot be solved in polynomial time, then no NP-complete problem can be solved in polynomial time.
- If $X \in$ coNP, then NP = coNP.

Proposition If $Y \leq_P X$ and $X \leq_P Z$, then $Y \leq_P Z$.

Prove Z is NP-Complete To prove a decision problem Z is NP-complete:

1. Prove Z is in NP
2. Prove $X \leq_P Z$ for some known NP-complete problem X

Satisfiability Problem (SAT): Instance: A Boolean formula made of Boolean variables and logical operands $\wedge, \vee, \neg$. Is there a truth assignment (True/False) to the variables that make the formula true?

SAT is in NP.
Certificate: Series of T/F assigned to variables $x_1, \ldots, x_n$.
Verifier: (I, C). Check you get n values in C. Verify that this truth assignment satisfies the formula.

SAT is NP-complete.

CNF-Satisfiability (CNF-SAT): Instance: A Boolean formula F in $x_1, \ldots, x_n$ such that F is the conjunction of m clauses, where each clause is the disjunction of literals. Is there a truth assignment such that F evaluates to True?

3-SAT is NP-complete.
2-SAT is in P.

Independent Set Problem Instance: A graph G and a number k. Quesiton: Does G have an independent set of size $\geq k$?

Proof: Independent Set is in NP. Reduction: 3-SAT $\leq_P$ Independent Set

Suppose we have a polynomial time algorithm that solves Independent Set.

Construct a graph G and choose a number k such that G has an independent set of size $\geq k \iff$ F is satisfiable.

Construction: For each clause C_i with literals l_1, l_2, l_3, make 3 vertices joined by 3 edges in G. If 2 literals are opposite, join them with an edge. Set $k \leftarrow m$.

Runtime: G has $3m$ vertices and at most $3m + n$ edges, so can be constructed in polynomial time in m and n.

Correctness: Prove G has an independent set of size $\geq k$ iff F is satisfiable.

($\Longleftarrow$) If F is satisfiable, then each clause has at least one True literal. Choose the corresponding m vertices of G - they are independent.

($\Longrightarrow$) If G has an independent set of size $\geq m$, there must be one vertex chosen from each triangle. Set the corresponding literals to True. This satisfies F.

Def Many-One Reduction: A many-one reduction $X \leq Y$ uses the algorithm for Y once and outputs its answer.

Correctness Proof: Answer for X is YES $\Longleftrightarrow$ answer of Y is YES.

Def Clique: A set of vertices where every pair are joined by an edge.

Clique Problem: Instance: A graph G and a number k. Does G have a clique of size $\geq k$?

Let G^C be the complement of G ($(u,v) \in G^C$ iff $(u,v) \notin G$). Observe that $C \subseteq V$ is a clique in G $\Longleftrightarrow$ C is an independent set in G^C.

Theorem Clique is NP-complete.

- Clique is in NP

  Certificate: The vertices of the clique.
  Verifier: Check $\geq k$ vertices. Check vertices are valid. Check edges between each pair in G to verify it is a clique.

- Independent Set $\leq_P$ Clique

Def Vertex Cover: A set $S \subseteq V$ such that every edge $(u,v) \in E$ has u or v in S.

Vertex Cover Problem: Instance: A graph G and a number k. Does G have a vertex cover of size $\leq k$.

Observe that $S \subseteq V$ is a vertex cover in $G \Longleftrightarrow V - S$ is an independent set in G.

Theorem Vertex Cover is NP-complete.

- Vertex Cover is in NP
- Independent Set $\leq_P$ Vertex Cover

  Assume a polynomial time algorithm for Vertex Cover. Give G, k as an instance of independent set. Construct G', k' from G, k to give as input to Vertex Cover. Run Vertex Cover with $G' = G$ and $k' = n - k$.

  Correctness: G has an independent set I of size $\geq k$. Then $V - I$ is a vertex cover, $|V - I| \leq n - k = k'$.

  $G' = G$ has a vertex cover S, $|S| \leq n - k = k'$. Then $V - S$ is an independent set of size $\geq k$.

Directed Hamiltonian Cycle Problem: Instance: A directed graph G. Does G have a directed Hamiltonian cycle?

Theorem Directed Hamiltonian Cycle is NP-complete.

- Directed Hamiltonian Cycle is in NP

- 3-SAT $\leq_P$ Directed Hamiltonian Cycle.

  G has a directed Hamiltonian cycle $\iff$ F is satisfiable.

  Create a variable gadget (a series of links). Create a vertex for each new clause and create a clause gadget. For each clause C_j, join at vertices $3j, 3j+1$.

  Clause gadget: the only way to visit C_j is by detouring off one of the variable paths.

  ($\Longleftarrow$) Suppose F is satisfiable. Traverse the variable paths in the True/False directions. For each clause C, at least one literal is True - take the detour from that path to vertex C. This gives a directed Hamiltonian cycle.

  ($\Longrightarrow$) Suppose G has a directed Hamiltonian cycle. The only way to visit C is by detouring off a variable path. If we take a detour using the edge at index $3j$ to C_j on the gadget for x_i, then we take the corresponding edge from C_j to the vertex at index $3j+1$ back.

  Thus, the Hamiltonian cycle must traverse each x_i in either a True or False direction and this corresponds to a satisfying truth-value assignment.

Theorem Hamiltonian Cycle is NP-complete.

- Hamiltonian Cycle is in NP
- Directed Hamiltonian Cycle $\leq_P$ Hamiltonian Cycle

  For a vertex u in directed G, make a new vertex u for all in edges and u' for each out edge and link the two using another vertex v.

Theorem Subset Sum is NP-complete (when weights are positive).

3-SAT $\leq_P$ Subset Sum

Construction: encode information in the bits we use for weights of subset sum.
Create a matrix with size $2(n+m) \times (n+m)$.
Encode the row as $x_1, \ldots, x_n, C_1, \ldots, C_m$ and the column as $x_1, \neg x_1, \ldots, x_n, \neg x_n, s_1, s_1', \ldots, s_m, s_m'$.

When you choose a weight for subset sum, you choose a row of the matrix. To meet target, sum rows. Sum each column and since it is in base 10, there are no carry overs. Clause is true if at least 1 literal is True.

Theorem Subset Sum $\leq_P$ 0-1 Knapsack is NP-complete.

- 0-1 Knapsack is in NP.
- Assume that we have a polynomial time algorithm for 0-1 Knapsack.

  Construction: Assign elements from Subset Sum to items in 0-1 Knapsack, weights from Subset Sum to weights in 0-1 Knapsack, and weights from Subset Sum to be the values in 0-1 Knapsack. Also, assign the target weight to be the maximum weight and k in 0-1 Knapsack.

  Return the output when you run this input on subroutine for 0-1 Knapsack.

Circuit Satisfiability Problem Instance: A circuit C. Is there an assignment of values to inputs such that the output is 1?

Cook's Theorem Circuit Satisfiability is NP-complete.

Circuit-SAT $\leq_P$ 3-CNF-SAT

Joining each level is not polynomial time since the size of the formula "doubles". Fix: create a variable x_u for node u in the circuit.
Note: $a = b$ means $(a \vee \neg b) \wedge (\neg a \vee b)$, so $x_u = x_v \wedge x_w$ means $(x_u \vee \neg(x_v \wedge x_w)) \wedge (\neg x_u \vee (x_v \wedge x_w))$.

Convert any clause with 2 literals into 3 literals by introducing a new variable x_n

$$(a \vee b) \implies (a \vee b \vee x_n) \wedge (a \vee b \vee \neg x_n)$$

This construction takes polynomial time and is polynomial in size.

Proof Circuit-SAT is NP-Complete being the first NP-complete proof.

For any Y in NP, there is an algorithm that maps any input y for Y to a circuit C such that y is a YES input if and only if C is satisfiable. There is a polynomial time verification algorithm A for Y.

A takes 2 inputs y, g where y is the instance of the problem and g is the certificate.
Property of A: y is a YES instance for Y if and only if there exists g of polynomial size such that $A(y, g)$ outputs YES. We have to convert algorithm A with known instance y and unknown input g to a circuit C with input variables = bits of g. Since we can write a program and compile it, at the hardware level, A is implemented with $\wedge, \vee, \neg$ gates and we get the circuit.

Inputs to C: bits of y and bits of g, internal nodes of circuit are memory locations after each step of algorithm A. Because the size of g is polynomial and A runs in polynomial time, the circuit has polynomial size.

Approximation Algorithms

Optimization Problems

1. Exhaustive Search (Backtracking, Branch and Bound) - It has exponential runtime in the worst-case, but it is good quality.
2. Heuristics - There is no guarantee of runtime or quality of solution.
3. Approximation - Aiming for polynomial time and some measure of the quality of solution. There is an approximation factor: $X \leq 2 \cdot$ optimum. $\forall \varepsilon > 0, \exists X \leq (1 + \varepsilon) \cdot$ optimum.

Vertex Cover A vertex cover is a set $S \subset V$ such that every edge $(u, v) \in E$ has u or v or both in S.

Claim: For algorithm 2, $|C| \leq 2 \cdot$ optimum

A matching M or independent edge set is a set of edges without common vertices and no 2 edges are incident. $|C| = 2\,|M|$. Any vertex cover must have at least one vertex from each edge in M. Therefore, $|M| \leq |C_{opt}|$ so $|C| \leq 2\,|C_{opt}|$.

Travelling Salesman Problem The Euclidean TS, we have a complete graph on points in the plane with weight equal to the Euclidean distance.

Greedy Algorithm 1

```
C ← ∅
while True do
    C ← C ∪ {vertex of maximum degree}
    Remove covered edges
    if No edges remain then break
return C
```

Greedy Algorithm 2

```
C ← ∅
F ← E // F is uncovered edges
while F ≠ ∅ do
    pick e = (u, v) from F
    add u, v to C
    remove (u, v) from F
    remove edges incident to u from F
    remove edges incident to v from F
return C
```

Approximation-TSP-Tour

```
Select a vertex r ∈ V to be a root vertex
Compute a minimum spanning tree T for G from root r using Prim's algorithm
Let H be a list of vertices, by walking around T, but if we visited a vertex, we take shortcuts
    to a new unvisited vertex
return Hamiltonian cycle H
```

Triangle inequality: $w(a,c) \leq w(a,b) + w(b,c)$.

Let t = length of tour found by this algorithm and let t_{TSP} = length of minimum TSP tour.
Claim: $t \leq 2t_{TSP}$. In polynomial time, we can find a tour within 2·optimum. Let t_{MST} = length of MST.

$$t_{MST} \leq t_{TSP}$$

deleting 1 edge of an optimal TSP tour gives a spanning tree. An MST has even less length.

$$t \leq 2t_{TSP}$$

MST (no short cuts) tour has length $2t_{MST}$. Then we can take shortcuts and could be shorter. This implies $t \leq 2t_{TSP}$.

Def NP-Hard: A problem X is NP-hard if there exist a NP-complete problem Y such that X $\leq_P$ Y.
Every NP-complete problem is NP-hard, but there exist NP-hard problems that are not NP-complete.

Def PSPACE: Decision problems solvable in polynomial space.

Proposition P $\subseteq$ PSPACE

Theorem NP $\subseteq$ PSPACE

Chapter 7

Undecidability

Def Undecidable: A decision problem is undecidable if it has no algorithm to decide it.

Def Unsolvable: A problem is unsolvable if it has no algorithm to solve it.

Program Equivalence Problem Given two programs, do they do the same thing?

Tiling Problem Instance: a finite set of tiles with coloured edges. Can they tile the plane where colours must match when tiles touch and you can use infinitely many copies of each tile.
This problem is undecidable.

Halting Problem Instance: an algorithm/program A and input w. Does A halt on w?

Theorem The Halting Problem is undecidable.

Proof: Suppose that there is a program H for the halting problem. Using H, we can make a new program H'. Does H' halt on input H'.

- If $H(H', H')$ returns YES, then H' loops forever
- If $H(H', H')$ returns NO, then H' halts

This is a contradiction, so assumption of H exists is wrong. There is no algorithm to decide the halting problem.

Undecidability Proofs To show problems are undecidable, use reductions. If X $\leq$ Y, then if Y has an algorithm, then so does X. The contrapositive is if X is undecidable, then Y is undecidable.

Def EXP: The runtime of the problem is $O(2^{\text{poly}(n)})$.

P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXP $\subsetneq$ Decidable $\subsetneq$ All decision problems