CS 341 Algorithms

Keven Qiu

Contents

1 Introduction

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.

2 Order Notation

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: Pseudocode, 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 2is $O(n \log n)$. 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 $\Theta(n^2)$.

3 Divide and Conquer

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

1. Divide - break problem into small instances
2. Recurse - use recursion to solve smaller problems
3. Conquer - combine results of the smaller problems to solve initial larger 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 $T(k) \le c \cdot k \log k$ for all $k < n$ where $k \ge 2$.

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:

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

n odd:

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

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

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

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 $\{a_1, \dots, a_n\}$, 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. $r = \sum\limits_{a_j \in B} r_j$.
It would help if A and B are sorted. We can modify mergsort 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: 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 $2\delta$.
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 $\theta \leq 9$.

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

4 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: $Split[0] \leftarrow$ True
2: for $k \leftarrow 1$ to $n$ do
3:     $Split[k] \leftarrow$ False
4:     for $j \leftarrow 0$ to $k-1$ do
5:         if $Split[j]$ and $Word(j+1, n)$ then
6:             $Split[k] \leftarrow$ 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 $A[k]$. Runtime: $O(n^2)$.

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.

```
$LISe[0] \leftarrow 1$
for $k \leftarrow 2$ to $n$ do
    $LISe[k] \leftarrow 1$
    for $j \leftarrow 1$ to $k-1$ do
        if $A[k] > A[j]$ then
            $LISe[k] \leftarrow \max\{LISe[k], LISe[j]+1\}$
```

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 $M(n,m) \to OPT(n,m)$.

```
function OPT$(i,j)$
    if $M(i,j) = M(i-1,j)$ then
        $OPT(i-1,j)$
    else if $M(i,j) = M(i,j-1)$ then
        $OPT(i,j-1)$
    else
        output $i,j$
        $OPT(i-1,j-1)$
```

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 $\sum\limits_{i \in S} w(i)$ 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 $1, \dots, n$ by right endpoint. Intervals disjoint from i are $1, \dots, j$ 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 $1 \dots n$ by right endpoint and sort by left endpoint $l_1, \dots, l_n$.

```
1: $j \leftarrow n$
2: for $k$ from $n$ down to 1 do
3:     while $l_k$ overlaps $j$ do
4:         $j \leftarrow j - 1$
5:     $p(l_k) \leftarrow j$
```

Revised algorithm:

```
1: Sort by finish time
2: Sort by start time
3: Compute all $p(i)$
4: $M(0) \leftarrow 0$
5: for $i \leftarrow 1$ 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.

```
1: function $S - OPT(i)$
2:     if $i = 0$ then return $\emptyset$
3:     else if $M(i-1) \geq w(i) + M(p(i))$ then return $S - OPT(i-1)$
4:     else return $\{i\} \cup S - OPT(p(i))$
```

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 $I = \{1, \dots, n\}$ and probability $p_1, \dots, p_n$ where p_i is the probability that item i will be searched. Find: A BST that minimizes the search cost $\sum\limits_{i \in I} (p_i) \cdot \text{ProbeDepth}(i)$.

$\text{ProbeDepth}(i) = 1 + \text{Depth}(i)$. The root node has depth= 0 but ProbeDepth= 1; i.e. it takes 1 probe to reach it. Serach 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 $1, \dots, k-1$
- Right subtree is then the optimal BST on $k+1, \dot{;} n$

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

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

- One of the nodes $k \in i, \dots, j$ 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 $P[i] = \sum\limits_{t=1}^{i} p_t$ where $P[0] = 0$, so, $\sum\limits_{t=i}^{j} p_t = P[j] - P[i-1]$. Runtime: $O(n^2 \cdot n) = O(n^3)$

```
for $i \leftarrow 1$ to $n$ do
    $M[i, i] \leftarrow p_i$                                ▷ Single nod e tree
    $M[i, i-1] \leftarrow 0$                                ▷ Empty tree
for $d \leftarrow 1$ to $n-1$ do                            ▷ $d = j - 1$ from above
    for $i \leftarrow 1$ to $n-1$ do                        ▷ Find $M[i, i+d]$
        best $\leftarrow \infty$
        for $k \leftarrow i$ to $i+d$ do
            temp $\leftarrow M[i, k-1] + M[k+1, i+d]$
            if temp $<$ best then best $\leftarrow$ temp
        $M[i, i+d] \leftarrow$ best $+P[i+d] - P[i-1]$
```

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

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

- If $i \notin S \implies$ Optimal solution on $1, \dots, i-1$
- $i \in S \implies$ If we take i, what subproblem do we want?
  - Maximize $\sum$ values considering items $1, \dots, i-1$
  - 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 (4)

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.

5 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 $[x_1, \dots, x_i]$ where $i < n$ is a partial solutioni if no constraints are violated.

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

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

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 $\mathcal{I} = \{1, \dots, n\}$ 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 $f_1 \leq \cdots \leq f_n$
2: $S = \emptyset$
3: for $i \leftarrow 1$ to $n$ do
4:     if interval $i$ is pairwise disjoint with all intervals in $S$ then
5:         $S \leftarrow S \cup \{i\}$
```

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

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

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 $\implies a_1, a_2, b_3, \dots, b_k, b_{k+1}, \dots b_l$.

Induction step: Suppose $a_1, \dots, a_{k-1}, b_k, \dots, b_l$ 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 $a_1, \dots, a_k, b_{k+1}, \dots, b_l$ 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 $\{1, \dots, n\}$ 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 $d_1 \leq d_2 \leq \dots \leq d_n$.

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

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 $1, \dots, n$ with values $v_1, \dots, v_n$, weights $w_1, \dots, w_n$ and a capacity, W. These are all positive integers. Feasible solution: An n-tuple $X = [x_1, \dots, x_n]$ 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:     $x_i \leftarrow \min\{w_i, freeW\}$
5:     $freeW \leftarrow freeW - x_i$
```

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 sol;ution 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 mathced to each other, but prefer each other to their existing matches. A matching with this property is called a stable matching.

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

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

```
1: $Match \leftarrow \emptyset$
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:         $Match \leftarrow Match \cup \{(e_i, s_j)\}$
6:     else
7:         if $s_j$ prefers $e_i$ (over current $e_k$) then
8:             $Match \leftarrow Match\{(e_k, s_j)\} \cup \{(e_i, s_j)\}$
9:             Note: Employer $e_k$ now looking to hire again
   return Match
```

6 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. indegree(v) and 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 ndoe 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($v_0$) $\leftarrow \emptyset$
4: level($v_0$) $\leftarrow 0$
5: Queue.add($v_0$)
6: mark($v_0$) $\leftarrow$ discovered
7: **while** Queue is not empty **do**
8:     $v \leftarrow$ Queue.remove()
9:     **for** each neighour $u$ of $v$ **do**                    ▷ Explore($v$)
10:        **if** mark($u$)= undiscovered **then**
11:            mark($u$) $\leftarrow$ discovered
12:            parent($u$) $\leftarrow v$
13:            level($u$) $\leftarrow$ 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 $u \in V$ do
3:     if $v$ is undiscovered then
4:         DFS($v$)
5: function DFS($v$)
6:     mark($v$) $\leftarrow$ discovered
7:     for $u \in$ AdjacencyList($v$) do
8:         if $u$ is undiscovered then
9:             DFS($u$)
10:            parent($u$) $\leftarrow v$
11:            $(u, v)$ is a tree edge
12:        else
13:            $(u, v)$ is a non tree edge if $u \neq$ parent($v$)
14:    mark($v$) $\leftarrow$ 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).

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

Enhanced-DFS(v)

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