CS 180 – Introduction to Algorithms and Complexity

Fall 2023 – Professor Majid Sarrafzadeh

<u>Lecture 1: Pair Test, Celebrity Problem</u>

3 switches and lightbulbs

- Google

Algorithm: Method of doing or determining something

- Corrasme: Astronomical algorithms
- Interested in interesting problems and solving a large set of problems
    - Sorting
    - Lower bound analysis: Covering base cases/be universally good and portable in nature

Problem:

Half the work is understanding the problem

Definition of a famous person

- Specific: Everybody knows this person
    - Not Clear: A lot of people knows this person

Condition 1: This person doesn't know anybody

Condition 2: Everybody knows this person

Self-knowing is undefined

Minimize or maximize a resource

- Time in video production quickly, Limit number of questions

Models of Computation do simple steps

A – B

I ask Person B if they know Person A

Person B can say Yes or No. (no maybes)

Asking everybody is inefficient (within this model of computation)

- Minimize the number of questions asked
- Note: Asking Vice versa is not answering the question

Worst Case Analysis

If there are n amount of people,

- There are n – 1 questions (I ask everybody except the one person)
  - 1 person says idk him, he is not famous
- Ask another n – 1 question this time from potential candidate about everyone else
  - If he says he knows someone, the candidate is not famous
- Total computation of 2(n – 1)

No more work required to find multiple famous people after finding first one

Case 1: Nobody knows anybody, nobody is famous

Case 2: There is one famous person

There is no case 3

2n(n – 1) “about” n^2

This is not optimal

Lower Bound Analysis creates Optimal Solutions

Model of Computation

Pair Test

Ask if A knows B

- Yes: A not famous
- No
  - A <u>can</u> be famous, B cannot be famous

- Hmm (not allowed)

If A knows B, can A be famous

- No, if you know somebody you are not famous

Random requires a LOT of resources (makes more questions, more complications)

Approach:

- ~~Randomly~~ Arbitrarily pick 2 people, A and Bs
- Do a Pair-Test (A,B)
- Eliminate either A or B
- Anyone eliminated will not be a famous person (for sure due to nature of Pair Test)
- Repeat from first step n – 1 times
    - This does not confirm that we find a famous person
    - <u>**This can confirm that we only have 1 candidate to be famous with n-1 questions**</u>
- Confirm if candidate is famous with another n-1 questions by asking everyone if they know the candidate and asking the candidate if they know everyone, which is a complexity of 2(n-1)
- Total computation of 3(n-1) questions, about n questions

One problem can become unsolvable with changing of one word
Solutions should be general and simple

Model of Computation
Serial (parallel) Model of Computation

- Input Output
    - About 1 unit of time to process data
- Registers
    - Constant number of registers
- ALU: Arithmetic Logic Unit

    - Computations (basic ones) take about 1 unit of time
- Memory
    - Read into memory takes 1 unit of time

Load and add is 2(n-1) (2 is 1 to load, 1 to add)

2(n – 1) + 1, outputting number takes 1 unit of time

2n “about n” time it takes to add n numbers in serial model of computation

Logn→n→2n→n^2→n^100→2^n→n!

Exponential up to n^100, 2^n and beyond is exponential

Order Notation

F(n) = O(g(n)) if there exists a constant c0, n0

Such that f(n) ≤ c * g(n) for all n ≥ n0

BOUNDED FROM ABOVE

Constants dont matter

Lecture 2: Gale-Shapely Algorithm

Last Lecture:

Finding a problem, understanding a problem (we still don't know what a problem is)

Universal Problems: Sorting, Matching

Two Groups

Group 1 has n elements, Group 2 has n elements

We want to match every element of Group 1 with <u>exactly</u> one element of Group 2

Group 1 : {1, 2, 3, 4, … n}

Group 2: {a, b, c, d, … z} (n elements)

Matching: One-to-One Matching

Complete Matching: All elements of Group 1 have exactly 1 match on the right

How to match all elements of Group 1 with Group 2

- Complete Matching

Every element on Group 1 has ranking of all elements of Group 2

- Ex: element 1 from Group 1 has ranking {b, c, … x}
    - Ranking is arbitrary
- Ranking is discrete, there is no all or nothing thinking

Every element of Group 2 also has complete ranking of all elements of Group 1

Stable Matching

- Must be complete matching (everyone has 1-2-1 matching with everyone else)
    - Arrow defines ranking
- No two matches must have conflicting priority rankings (two matches occur where a more optimal solution exists)

S = (m’, m)

M’ = (s, s’)

Algorithm

- Arbitrarily pick students or med-schools (The problem is symmetrical/the same regardless) [lets pick students for the sake of this example]
- WHILE THERE ARE UNMATCHED STUDENTS
- Arbitrarily pick one student
- Look at student’s med-school rankings, take highest med-school in ranking <u>that you have not tried to get into yet</u>
- Check med-school to see if it has existing match
    - If not, match student with med-school
    - If so,
        - if current student is ranked below existing match, ignore current student
        - if current student is ranked above existing match, match med-school with current student instead

If all students are matched, all med-schools are matched

Question: Does this loop/program terminate? (Is it solvable)

- Worse case, each student asks each college
    - n students * n colleges = n^2

Properties for the algorithm

- Once school gets match, the match will remain throughout the entire algorithm
- School’s matching gets better and better in time, worse case it stays the same
- Student’s matching gets worse and worse in time, best case it stays the same

One to One: If one student is unmatched, by definition a college must be unmatched also

- The only time med schools will not match with a student is when a student is rejected for a better match

STABLE

The Algorithm produces a stable output

BWOC: By Way of Contradiction (Lets assume our algorithm produces an unstable output)

S proposed to m’ at some point

- Yes: Contradiction of property that med schools only pick better options, med school picks a worse student
- No: Contradiction, m’ must already have been asked if matched with m

Lecture 3: Using Data Structures in the Stable Sorting Problem

Stable Matching Problem

Iteration: student asks med school, med school makes a decision

- Student never asks a med school twice
- Global Analysis, Worse Case: O(n^2) iterations
- Runtime is function T: T(n) ?= O(n^2)
  - T(n) denotes that the runtime's only parameter is n
  - To prove T(n) = O(n^2), each iteration must take constant runtime

Data Structure must (in constant runtime):

1. Identify a free student (assign/unassign students)
2. For each student s, find highest rank m that has not been approached
3. For med school m, we must know if it has an assignment/match, we must identify the student assigned
4. Med school m needs to find the relative ranking of s (current) and s' (matched)

Approach:

Students stored in a linked-list

1

- Assume students are labeled 1 through n
- Initialize each student into linked-list
- Take first element in linked list of available students (all students have not approached)
  - Takes constant time
- If current student has better priority than matched student
  - Reassign and move newly unassigned student to either front or back of linked list
    - Constant Runtime, go for simpler moves as they are often faster

2

- Each student has an array of med-schools (which represent priorities)
- Assume each med-school is labelled 1 through n
- Student has array pointer

3

- Each med school m stores its current assignment, assume that 0 or -1 means no assignment (all med schools start without any assignments)
- Med school m can change its assignments, note that med school matches on get better over time

4

- Med school m has an array of size n, denoting its priority list of all students
- Pre iteration/rank computation O(n^2): Create table of med school m's priorities by iterating through all students and indexing

Space Complexity:

- Stable Matching problem requires a minimum of O(n^2) storage to simply process the problem (n students * n rankings)
- Thus, our "limit" should be n^2, aka we should not have any linear runtimes during any step **during** our algorithm, as that would increase our complexity by one order, however steps of n^2 complexity **before** and **after** our algorithm are ok

Election Problem (Actual Midterm Question)

We have a bunch of candidates and we want to vote for them

m candidates (can be 1 in some countries, some countries might like 18)

- Labelled 1, 2, … m

n votes

- Labelled 1, 2, … n

Each person votes for **one** candidate

- Candidates cannot choose not to vote, each candidate **must** vote

Majority is candidate that appears over n/2 times across all votes

- We have at most one candidate in a group of votes, no majorities is possible

- We don't care about cases where there is a tie and we must pick the candidate with the largest amount of votes due to the definition of a majority (that is a different problem)

Solve the problem in constant storage

Start with array of all votes:

- Count all votes required to become majority for each candidate
- We consider 2 **different** votes, we eliminate it from the array
    - In both cases, where one is majority and one isn't, the new, shortened list will maintain the older majority
    - You *can* create a majority by eliminating two votes, however this isn't relevant to our claim (we cannot use it as a counterargument)

Store current majority candidate and count in variables

Temp majority

- If current vote is different, we decrement the current majority candidate vote count
    - If vote count is 1 and next candidate is different, set both majority candidate and vote count to 0 and 0
- If current vote is same as current majority candidate, we increment the current majority candidate vote count

How do we confirm majority?

- Our majority candidate does not prove majority due to the arithmetic
- Another linear scan to double check that majority candidate has more than n/2 votes

Lecture 4: Priority Queues, Graph Algorithms

Elements are assigned priorities, find element with highest priority

Heaps

- Normally organized as a binary tree, element with highest priority is at top, higher priority means higher priority
- Minimum and maximum values are same problem
- Pointer to root, sort elements
- Any subtree: The mean value must be at the root
    - Every other element in subtree must be larger than top node
    - No relationship between two subtrees
- Array implementation of heap
    - Children of node are at 2i and 2i+1
    - Parent from floor of index of children
- Height of tree is O(logn)

ROOT IS MEAN VALUE

Non-decreasing order

Heapifying

- Remove root
- Replace root with Min(l, r)
- Continue until tree is finished
- Swap from bottom to top to get min

Nlogn sorting

Balanced tree is logn time to search

Graphs

- Consists of vertexes/nodes
- Edge/link connects two vertices
    - Directed
        - A can reach B, B is neighbor/neighbor of A
        - Vice versa does not hold, two edges are required
    - Undirected
        - B is adjacent/neighbor/accessible by A
        - A is adjacent to B
        - Two Ways
- G = (V, E)
    - If E = 10; {(1,5), (2, 6), …}
    - |V| = n
    - |E| = m

Connected graphs

- Unconnected
    - If two components of a graph share no edges
- Biconnectedness is the "connectedness" of directed graphs

Path is series of points connected with edges

Cycle: Beginning vertex and ending vertex are same

Simple Path: no same vertex is reused in same path

Simple cycle: only repeating vertex is the beginning/end vertex

Simple path and simple cycle

Matrixing tree

- Normally is there is no edge between a and a
- Linear Time

Linked List

- Constant time for one neighbor
- Linear otherwise

Consider an undirected, connected graph

- Number of vertices is n
- Minimum number of edges is n – 1
- Maximum number of edges is n(n+1)/2
    - O(n^2) edges
    - Every vertex can be connected with every other vertex

Dense graph, sparse graph, Tree structure

Scheduling Problems

We are given a bunch of tasks and we only have one processor for all tasks (we cannot overlap/parallel)

Greedy Paradigms

- Take the best decision every time
    - Fast, but often hard to prove

Sort all endpoints
Take first ending event
Eliminate all overlapping events

High level proof: Scheduling

- Optimal: Number of intervals the schedule produces is maximum (There can be multiple)
- Given a S and S' schedule, the algorithm must produce a more optimal schedule for S

Lecture 5: BFS

Undirected, May be unconnected

Directed changes how edges are connected/adjacent

Breadth First Search

- Look closeby first rather that deeply first
- Linked List representation can allow constant accessing of neighbors
- True edges connect between layers
    - Forms tree structure (not necessarily binary)
- BFS → Finding Distance
- DFS → Finding Cycles

<u>Discussion 2</u>

Bipartite Graph: Graph divided into two sets of vertices, No set has any edges within itself, edges only connect between the sets

Checking for Bipartite

- Idea: BFS: for each level, if each set is bipartite, then each level should obey certain traits
    - All the edges are between vertices on different levels
    - Ex: Set 1 has odd numbers, Set 2 has even numbers

Odd cycles result in non-bipartite graphs

- This implies there is an edge between two nodes on the same "set"
- Defined as odd cycle as you can loop back to root node in odd amount of distance

BFS trees **<u>ALWAYS</u>** have shortest distance to root

Once a node is explored, the value is never visited again

BFS: O(v + e) + O(e) → v + e to check each node's edge (we cannot confim birpartiteness with v + e alone), e to double check edges

- While there are unvisited nodes:
    - Pick arbitrary node
    - If edge connects node with node on same level:
        - Odd cycle, not-bipartite
    - Else
        - Is ok

Sparse BFS → Linked List Approach

Dense BFS → Pointer Array Approach

<u>Lecture 6: Graph Connectivity (3.4, 3.5)</u>

Connected Graphs with No Cycles

Bipartite Graphs (2 Colorable)

- Vertices are placed in two sets, left and right
- "Triangles" make graphs non-bipartite
  - Two vertices of same set share an edge
- C sub 3 is Cycle of length 3, P sub 3 is Path of length 4

Odd Cycle Property

- If a Cycle of odd length greater or equal to 3 exists in graph, the entire graph is not bipartite

G = (V, E) undirected, one component/connected

- BFS: Find all neighbors in one shot
  - All neighbors in bipartite graph must be a different set
  - Take arbitrary vertex, apply BFS to create tree

Bipartiteness:

- No graph can have an edge between two vertices on same level
- Graph must have no edge between vertices a and b on same level

Assume A,b on same level,

Assume a, b shares no edges

Strongly Connected → Directed Edge

For <u>all</u> pairs (u,v) that are vertexes, there is at least one path from u to v and one path from v and u (Directed Graph acts like Undirected)

Strongly connected graphs

- Pick an arbitrary vertex s
- From s, I run BFS on G (BFS is O(v + e) runtime)
  - MUST RUN THROUGH ALL VERTEXES IN G

- From s I run BFS on G^reversed
- If conditions 2-3 are satisfied, S is connected to all vertexes, and vice versa (we are still in linear time)

Lecture 7: Shortest Path: Dijkstra's

In a graph:

- If there a vertex which by removing it, it disconnects the graph, it is called an articulation or a vertex that can disconnect the graph
- A – b – c
    - B is an articulation, A is not an articulation
- Algorithm: (Find a vertex which is not an articulation in linear time)
    - Run breadth first search
        - Leaves of tree will not be articulations
- Proof:
    - Start from B, any vertex will be a leaf, start from A, any vertex will not be a leaf

Shortest Path

- In an unweighted, undirected graph:
    - Starting from a vertex and performing BFS, I will know the shortest path
        - Proved as distances are no more or no less than they are
    - BFS gives distances of vertexes to other vertexes for free !!!
- BFS **fails** with weighted edges
    - No correlation between weight and length path

Minimum weight path:

- Short = weight **NOT LENGTH** (we already did that problem)
- Famous Problem Paradigm: Each iteration, we eliminate one candidate permanently
- Greedy Paradigm: Look at problem locally, solve it, extend it from there
    - Interval Schedule

Either s is given to me or it is arbitrary, but it will be the root

Assume all weights are positive (the algorithm breaks otherwise)

Start at s

- Look at all neighbors of s
- Of all neighbors of s, one is closest to s

- Find minimum weight neighbor
- Claim that current neighbor is shortest path to s
    - By contradiction, assume a different path holds a shorter weight
        - Assume vx is first vertex in this graph, meaning the distance from ws→wvx + some other numbers >= wmin
        - If vx >= vmin, then it is impossible to get a shortest path
        - All other weights are positive or zero, meaning the whole weight is always greater
- Analyze next neighbor of smallest weight

Create three sets, PROCESSED, INTERMEDIATE and UNPROCESSED
All vertexes start in UNPROCESSED

- Of all neighbors of S, pick minimum weight distance neighbor B, which is the shortest distance as proven previously by contradiction
- Look at all other neighbors of processed vertices (vertices in INTERMEDIATE), look at all neighbors of B (move them to INTERMEDIATE)

Look at everyone's distance (that have temporary weight/in INTERMEDIATE) to those whose been fixed (in PROCESSED)
Compare weights to find minimum
Update weights, complete until everyone is in PROCESSED
Vertexes that are processed have visited all INTERMEDIATE vertexes or are leaves

Time Complexity

- N steps, each step modifies 1 vertex
- O(e) runtime as constant runtime for each of e edges

Midterm question:

- Assume a graph has positive and negative edges

- Take the most negative weight, add absolute value of it to every edge
- Now all weights become positive, one of them is zero, but all others are positive
- Apply Dijkstra's algorithm to this path. Is this the minimum weight path for the original graph?
- No, because it unfairly unbalances the weight of densely [populated paths, aka this is mixing the weights and the paths

Lecture 8: Minimum Spanning Tree (4.5)

Given G(V,E) = A – B – C – D

- B – C can be considered a subgraph

Possible subgraph properties

- Path Graph
- Tree Structure
- Connected/Not connected

Spanning tree:

- Subgraph of graph such that:
    - It is a tree (no cycles)
    - It is connected
    - It must touch every vertex

Find spanning tree:

- Perform BFS/DFS tree

Weighted spanning trees:

- Number of spanning trees is not unique

MST Minimum Weight Spanning Tree:

- Finding MST
    - If graph has m edges, number of edges to remove is m – (m – 1) ← n-1
    - If less than m – m - 1 edges (empty graph), add until n – 1 edges

Start with tree

- Add edge (create a cycle)
- Remove certain edge → Get another tree back
- Removing other edges may create disconnected parts

Problem: Come up with X mimimum spanning trees

With MST as input:

- Assume every edge is unique (their weight is unique)
- A (bipartite) partition of a graph is defined where every vertex only are connected to vertexes of both sides, both sides must be non-empty
- Look at all edges connecting both bipartite sets
    - There will ALWAYS be one edge connecting the sets (assuming graph is already connected as you cannot create a spanning tree with an undirected path)
    - Pick bipartite edge with minimum weight
        - Because edges are unique, we KNOW this is the minimum weight edge

MST Theorem:

- Idea: For every MST, one eMIN exists in the graph
    - There is no way to force eMIN to not be eMIN (this implies that eMIN can vary for a given graph)
- Proof by Contradiction: MST has eMIN in it
    - Assume eMIN is not in MST:
    - This means that we still have a spanning tree
    - Because graph is connected, for a given two vertices v and w, there MUST be a path
    - Given MST has no eMIN, we will add an eMIN edge (thus creating a cycle)
    - Pick arbitrary edge connecting both sets of bipartitions that is not eMIN, we will receive a tree back
    - The total weight will be <= to the original contradiction MST
        - This is because eMIN is less than every edge in terms of weight
    - Therefore you did not start with an MST, a contradiction

Pick eMIN from systematic partitions, after n – 1 iterations, we will have an MST

Prim's Algorithm

Given a bipartite graph:

- Take an arbitrary vertex, denote it as v1, place it as only vertex in set A
- Place v2 through vN in set B
- All vertices of v2 are neighbors of v1
- Every MST has an eMIN, thus we will select an eMIN for our given partitions
- Next iteration: Create new partition where vertices connecting eMIN are in same set, pick new eMIN

If all edges are unique weight, this will always return the same spanning tree

Lecture 9: Minimum spanning tree and union find (4.5, 4.6)

Minimum Spanning Trees:

- Assume all weights are unique (to make arguments simpler)
- Prim Algo: n on one side, n-1 on other side, move unexplored edge set to explored set based on each relative eMIN, build minimum spanning tree
- Proof is same as Djikstra's Algo
    - There is O(n^2) and O(eloge) algo versions

Linear sort: Refers to special cases, such as sorting binary

Kruskal's Algorithm: How to find if new edges will create a cycle (as DFS on each node is quite inefficient)

- Prim is vertex-centric (you deal with vertices)
- Kruskal directly attacks edges

Algo:

- Assume all edges are unique
- Take all edges
- Sort edges (eloge time)
- Iterate through edges:
    - If current edge does not create cycle:
        - Add to MST
    - If current edge does create cycle:

- Take minimum weight edge (e1)
- Create vertex bipartition where both sets are only connected by e1
- Assumption: e2 is different from e1
- Case: e1 and e2 shares no vertices
    - Create bipartition based on one vertex of e2
- Case e1 and e2 share a vertex
    - Create bipartition based on unconnected vertex (provable)

Assume an edge connects two existing MSTs (creating a cycle):

- Whenever ei creates a cycle, we cannot create a bipartition based on the MST theorem (one edge must be deleted)

Union Finding:

- Assume unique values
- Of n elements, partition into k sets
- Operation: Find
    - Input: 2 elements
    - Operation: Check if in same set
        - Of two sets, select two nodes as roots
        - Make all set vertices point to root
        -
    - Output: T/F
- Operation: Union
    - Input: 2 sets
    - Operation: Create union of two sets (permanently delete old sets)
        - Take first element of set, point to last element of other
        - Constant
    - Output: 1 set
- Idea: Create tree of O(logni)
    - Establish each set as a tree with a height of logn and explicit root
    - Unionizing will never increase the greatest height, as we will only point root of smaller tree to root of bigger tree
    - Finding will take

Kruskal's Algorithm:

- Place each vertex in set, represented by tree
- Adding edges now means unionizing both vertex trees
- Cycle Check:

    - Take two nodes, follow to root

Runtime of Kruskal:

- Sort edges → O(mlogm)
- Look at each edge e:
    - Call Find (to check for cycles)
        - Yes (Creates Cycle) → Ignore e
        - No → Add e to MST, perform a union

Apply MST

Book SDeletion Algorithm:

- Delete Largest Edge

Lecture 10: Divide and conquer (5.1), Master theorem (5.2), Counting Inversion (5.3)

Divide and Conquer Paradigm

- Disassemble problem into smaller pieces
- Operate on each piece
- Assemble smaller solutions into whole solution

Sorting:

We want to put numbers in non-decreasing order

- We can sort using heap sort, → O(nlogn)

Assume we are given an order to sort:

9 3 2 4 8 1 6 7

Divide and Conquer:

Divide

- Partition into two parts (each part is roughly n / 2 given n numbers)
- Partition each part into two parts (each part has 2 numbers in it now)
- Partition each quarter into two parts (each part is 1 number)
    - By definition, numbers of length 1 are sorted

Merge: If lists p and q exist and are in non decreasing order, how to combine them into p + q which is also non decreasing?

- Two pointers, one pointing at p head, other pointing at q head
- Heads are the smallest number of each list because theyre already sorted

O(pq) for x on left, I am doing at most p comparisons, thus O(pq)

Runtime is O(p + q) for whole list

time(sort n elements) = 2 * time(sort n/2 elements) + O(n) [for merging]

T(1) = O(1)

Time(n) = 2Time(n/2) + C(n) [C means constant]

- = 2[2T(n/4) + C*n/2] + Cn
    - [Plug in Time(n) into Time(n/2)]
- = 2^2 * T(n/2^2) + 2Cn
- = 2^3 * T(n/2^3) + 3Cn
- …
- = 2^i * T(n/2^i) + iCn

We know that sorting at 1 or 2 is constant

When is n/2^I = 1? Or 2^i=n?

This is when I = logn (base2)

- = 2^logn * T(n/2^logn) + Cnlogn
- = n * T(1) + C * n * logn
- = n + cnlogn
- T(n) = O(nlogn)

Master Theorem :p

T(n) ≤ qT(n/2) + Cn

T(2) ≤ C

q>2

T(n) = O(log2q) [u don't need to memorize this lol]

Use as argument for Divide and Conquer

Past Midterm Question:

- Two parallel lines
- Cords connect top and bottom lines at different points
- Come up with an algorithm that counts the number of crossings
    - O(n^2) is trivial (you can look at each pair of cords)

Approach

Divide And Conquer

- Count bottom lines, match with top lines
- Whenever two numbers are out of order, a crossing has occurred
    - Out of order refers to decreasing

- Divide into pairs, count number of crossings/out or order
    - Apply merge sort
    - For each out of order (decreasing) pair of numbers, add number of remaining numbers

$T(n) = 2T(n/2) + Cn + Cn$

Given a bunch of points, given x and y axes, find the closest two points

Defining Distance

- Euclidean distance $((x1-x2)^2 + (y1-y2))^{1/2}$
- L1/Manhattan Distance → Take horizontal and vertical

Brute Force Approach:

- For each point, get every other point's distance
- $O(n^2)$

Best Possible is $O(n)$ [not that it is optimal]

Famous/Celebrity Problem:

Algorithm:

Pair Test

- Ask if A knows B
    - Yes: A not famous
    - No: A can be famous, B cannot be famous

- Arbitrarily pick 2 people, A and Bs
- Do a Pair-Test (A,B)
- Eliminate either A or B
- Anyone eliminated will not be a famous person (for sure due to nature of Pair Test)
- Repeat from first step n – 1 times
    - This does not confirm that we find a famous person
    - **This can confirm that we only have 1 candidate to be famous with n-1 questions**
- Confirm if candidate is famous with another n-1 questions by asking everyone if they know the candidate and asking the candidate if they know everyone, which is a complexity of 2(n-1)

Proof:

Time Complexity:

- Total computation of 3(n-1) questions, about n questions

If there are n amount of people,

- There are n – 1 questions (I ask everybody except the one person)
    - 1 person says idk him, he is not famous
- Ask another n – 1 question this time from potential candidate about everyone else
    - If he says he knows someone, the candidate is not famous
- Total computation of 2(n – 1)

Stable Matching/Gale Shapley:

Algorithm:

- Arbitrarily pick students or med-schools (The problem is symmetrical/the same regardless) [lets pick students for the sake of this example]
- WHILE THERE ARE UNMATCHED STUDENTS
- Arbitrarily pick one student
- Look at student's med-school rankings, take highest med-school in ranking <u>that you have not tried to get into yet</u>
- Check med-school to see if it has existing match
    - If not, match student with med-school
    - If so,
        - if current student is ranked below existing match, ignore current student
        - if current student is ranked above existing match, match med-school with current student instead

If all students are matched, all med-schools are matched

Proof:

Properties for the algorithm

- Once school gets match, the match will remain throughout the entire algorithm
- School's matching gets better and better in time, worse case it stays the same
- Student's matching gets worse and worse in time, best case it stays the same

One to One: If one student is unmatched, by definition a college must be unmatched also

- The only time med schools will not match with a student is when a student is rejected for a better match

The Algorithm produces a stable output

BWOC: By Way of Contradiction (Lets assume our algorithm produces an unstable output)

S proposed to m' at some point

- Yes: Contradiction of property that med schools only pick better options, med school picks a worse student
- No: Contradiction, m' must already have been asked if matched with m

Time Complexity:

- Worse case, each student asks each college
    - n students * n colleges = n^2

Heap Sort/Priority Queues:

Algorithm:

Heapify:

- Remove root
- Replace root with Min(l, r)
- Continue until tree is finished
- Swap from bottom to top to get min

Proof:

Note: Priority Queues are implemented using binary trees with a height of logn

- Induction Proof of Heapifying:
- Base Case: i = 1
  - Nothing to prove, i is already a heap
- Inductive Case:
  - If we have a heap for n nodes, we can maintain a heap in O(logn) time at n + 1 nodes
- Proof:
  - Add new value into bottom of priority queue
    - If new value is biggest, heap is maintained
    - If new value is not biggest, heapify up until heap exists, for a total of logn time

Time Complexity:

- Insertion, extraction is O(logn)
- For n nodes, it is total O(nlogn)

BFS:

Algorithm:

Add starting vertex to queue

Mark starting vertex as discovered

While queue is not empty:

- Dequeue the top vertex from the queue, denote as c
- Process c
- For each vertex v directly reachable from c:
    - If v has not yet been discovered:
        - Mark v as discovered
        - Insert v into queue

Proof:

Time Complexity:

O(V + E)

<u>DFS:</u>

<u>Algorithm:</u>

Add starting vertex to stack

Mark starting vertex as visited

While stack is not empty:

- Pop top, store in s
- Process s
- Add all unvisited neighbors to stack

<u>Proof:</u>

<u>Time Complexity:</u>

Bipartite Check:

Bipartite means the graph is divided into two distinct sets

If there exists no odd cycle, then there must exist an edge between two nodes in the same set (a cycle)

Odd cycle check:

Use BFS, If an edge connects a node from the same level, there is an odd cycle

Topological Sort:

- Count indegrees of all nodes
- While there are nodes with 0 indegree(source nodes):
- add that node to the output, delete the node from the graph, delete edges originating from that nodes, and decrement node's indegrees appropriately.
- if we encounter a node whose indegree was change from 1 to 0, add it to the set of source nodes.

How do I find the sources in a graph?

- Go to each vertex, count each edge
- For n edges, computing all in-degrees can take O(n^2) for worse case
- O(e) → e is number of edges, count in-degrees alongside processing edges
- Go through all vertexes, check if in-degree is zero, if so it is a source (O(n))
- Thus, finding sources is O(e) + O(v) →O(n)
    - O(e) → Iterate through each edge to count in-degrees
    - O(v) → Iterate through each vertex to check for in-degrees of 1

Deletion of vertex from tree: Constant time per edge

- Have list of in-degrees
- Deleting node → Decrement in-degree in list
- If in-degree is now 0, it is a source

DAG always has a source node:

- assume a DAG has no source node. Then every node has indegree > 0, then you could trace back n+1 times, you are guaranteed to have a cycle.
- start from a source node, all edges from this source node, must point forwards in the topological sort
- if we remove a source node, the resulting graph is still a DAG, thus there is guaranteed to be another source node to use.
- repeat until all nodes are output in the ordering.

Proof:

- if we can make a top ordering, then we have a DAG => contrapositive, if you have a non-DAG, then you can’t make a topological ordering
- assume: assume we have a non-DAG that has a top sort.
- the graph has a cycle in it, and it has a topological sort.
- if there is a cycle, there must exist a node in the cycle that is last in the top ordering(out of the other nodes in the cycle). Thus that last node must have an outgoing edge pointing backwards, a contradiction.

Greedy Paradigm:

- sort by end times
- for every interval sorted by end time:
- take first element with earliest end time
- check the start time, only accept that interval into the solution if its start time is later than the last interval’s end time

Dijkstra's:

Start at s

- Look at all neighbors of s
- Of all neighbors of s, one is closest to s
    - Find minimum weight neighbor
    - Claim that current neighbor is shortest path to s
        - By contradiction, assume a different path holds a shorter weight
            - Assume vx is first vertex in this graph, meaning the distance from ws→wvx + some other numbers >= wmin
            - If vx >= vmin, then it is impossible to get a shortest path
            - All other weights are positive or zero, meaning the whole weight is always greater
    - Analyze next neighbor of smallest weight

Create three sets, PROCESSED, INTERMEDIATE and UNPROCESSED

All vertexes start in UNPROCESSED

- Of all neighbors of S, pick minimum weight distance neighbor B, which is the shortest distance as proven previously by contradiction
- Look at all other neighbors of processed vertices (vertices in INTERMEDIATE), look at all neighbors of B (move them to INTERMEDIATE)

Look at everyones distance (that have temporary weight/in INTERMEDIATE) to those whose been fixed (in PROCESSED)

Compare weights to find minimum

Update weights, complete until everyone is in PROCESSED

Vertexes that are processed have visited all INTERMEDIATE vertexes or are leaves

process root node, assign it a final min distance of 0.

- assign all other nodes temp_min distances of infinity
- for all neighbors of the root, and give them a distance of min(temp_min, cost to get to last node processed + cost of edge between last node processed to the neighbor)

- put all nodes and their temp_min distances into a heap
- while there are nodes we haven't finalized a min distance for:
- pick the node with minimum temp_min by popping the heap's top, and finalize that node's distance
- for all neighbors of that node give them a distance of:
- min(current temp_min, cost to get to last node processed + cost of edge between last node processed to the neighbors
- output the min_dist of the target node

O(ElogV)

Proof:
idea: assume we have finalized min distances of a subset
of nodes:

- because we are taking min of the intermediates, by contradiction if there exists another path, then we have to go through another intermediate node, by definition that path must be greater than theminimum of the intermediates.

Prim's:

Given a bipartite graph:

- Take an arbitrary vertex, denote it as v1, place it as only vertex in set A
- Place v2 through vN in set B
- All vertices of v2 are neighbors of v1
- Every MST has an eMIN, thus we will select an eMIN for our given partitions
- Next iteration: Create new partition where vertices connecting eMIN are in same set, pick new eMIN

If all edges are unique weight, this will always return the same spanning tree

O(ElogV)

Dense Graphs

Kruskal's:

- Assume all edges are unique
- Take all edges
- Sort edges (eloge time)
- Iterate through edges:
  - If current edge does not create cycle:
    - Add to MST
  - If current edge does create cycle:

- for each edge we perform "find" from union-find:

O(logE)

- we may "union" groups together: O(1)

=>algo has time complexity of O(eloge)

- Take minimum weight edge (e1)
- Create vertex bipartition where both sets are only connected by e1
- Assumption: e2 is different from e1
- Case: e1 and e2 shares no vertices
  - Create bipartition based on one vertex of e2
- Case e1 and e2 share a vertex
  - Create bipartition based on unconnected vertex (provable)

MST:

- Idea: For every MST, one eMIN exists in the graph
    - There is no way to force eMIN to not be eMIN (this implies that eMIN can vary for a given graph)
- Proof by Contradiction: MST has eMIN in it
    - Assume eMIN is not in MST:
    - This means that we still have a spanning tree
    - Because graph is connected, for a given two vertices v and w, there MUST be a path
    - Given MST has no eMIN, we will add an eMIN edge (thus creating a cycle)
    - Pick arbitrary edge connecting both sets of bipartitions that is not eMIN, we will receive a tree back
    - The total weight will be <= to the original contradiction MST
        - This is because eMIN is less than every edge in terms of weight
    - Therefore you did not start with an MST, a contradiction

Pick eMIN from systematic partitions, after n – 1 iterations, we will have an MST

<u>Binary Search: O(logn)</u>

<u>Merge Sort: O(logn)</u>

Lecture 11: Closest pair of points (5.4), Divide and Conquer, Dynamic Programming

Given a collection of points in two dimensions, what are the two closest points?

- Assume Euclidian distance as means to track distance
- Brute Force: O(n^2) → Check distance of all n points to all n-1 other points
- Idea: Merge Sort, divide points into two sides and recursively calculate smallest distance
    - In any recursive case, the shortest distance is either on the left, the right, or in between
    - Still O(n^2)
- Idea: For each P, there are 8 points to consider

Dynamic Programming

- Sub problems overlap
- Solve each sub problem optimally
- Combine optimal solutions to find final optimal solution

Interval Scheduling Problem (1-D Dynamic Programming)

- Given weighted schedules, find maximal weight non-overlapping schedule
- Idea: Controlled Exhaustive Search

2-D Dynamic Programming Problem:

- Get as much value in sack as possible
- Each item has a size and value
- Create table of solutions, find the most optimal path to the bottom right

Polynomial in nature

<u>Lecture 12: Assorted DP Problems</u>

Controlled Exhaustive Search:

- An element is either in the solution or not
- Normally a loop which analyzes the in solution or not in solution cases

Maximum Subsequence / Sequence Alignment / Longest Common Subsequence

- If you have two sequences, L and R, of sizes m and n, what is the maximum common subsequence between L and R?
    - Note: Longest Subsequence is not unique
    - L = A, B, C
    - R = A, D, B
- Subsequence: Subset of given sequence, not necessarily contiguous, order is retained
    - For L: empty, a, b, c, ab, bc, ac are subsequences
- Alternatives: Least Common Subsequence, SA?

Find Size First → Find Subsequence Order Next

Approach:

- Look at first i elements of L, first j elements of R
- OPT(I, j) refers to longest subsequence
- Assume I know OPT(i, j) for any value less than i and j
    - Look at L(i), R(j):
    - Case: L(i) == R(j)
        - If same, match them up
        - OPT(i, j) = OPT(i – 1, j - 1) + 1
        - Proof:
            - Assume Matching Optimal Solution does not include (i, j)
            - If we include (i, j) into this solution, the solution becomes more optimal (longer)
            - Thus, a contradiction
        - Note: No crossing is allowed in an optimal solution

            - If we don't match
    - Case: L(i) != R(j)
        - In final optimal solution:
            - Case: L(i) has match
                - OPT(i, j) = OPT(i – 1, j)
            - Case: R(j) has match
                - OPT(i, j) = OPT(i, j - 1)
            - Case: Neither L(i) and R(j) have matches
                - OPT(i, j) = OPT(i – 1, j – 1)
                - Technically covered by previous two cases, so we can ignore this case
            - NOTE: L(i) and R(i) cannot both have matches, as this would result in a crossing, thus a contradiction

Algo:

- Create a table with axis L, R
- Compute (i, j) for every element of table
- Compute in <u>row-major order</u>
- Look at L(i – 1), R(j – 1)
    - Case: L(i – 1) == R(j – 1)
        - Solution += 1
    - Case: L(i – 1) != R(j – 1)
        - Look at either (L(i), R(j – 1)) or (L(i – 1), R(j))
    - Select max(diagonal + 1, left, up)
        - Diagonal + 1 only if they are the same, otherwise, just diagonal
        - If multiple are the same, arbitrarily select one
- Diagonals define matches, searching is either left or up

Time Complexity: O(m * n)

- Inputs: m elements for L, n elements for R, thus m + n
- Polynomial

Space Complexity: O(2m) or O(2n)

- Only 2 rows of memory are needed for the whole algorithm

RNA Sequence

- A, U, G, C
    - A can match with U
    - G can match with C
    - Duplicates cannot match, no other matches are accepted
- Find maximum subsequence of matches
    - Distance must be at least of 4 (there must be 4 characters in between) → No sharp corners
    - No Crossing is allowed (A future match matches with a previous match)

Approach:

- OPT(i, j) = max(for a given t(OPT(t + 1, j - 1) + OPT(1, t – 1) + 1)
    - T are defined by distance, character matching
- For given X, consider all possible matches with Y before it
    - Case: X gets matched
        - X must be matched with a Y before it
        - Consider X to be matched with everything before it (Exhaustive Search)
    - Case: X does not get matched
        - Consider intervals containing Y to the right of X and to the right of X

Algo:

- OPT(I, j) = 0 I >= j – 4
    - Roughly O(n^2) intervals, iterate through O(n) t regions

<u>Lecture 13: Network Flow</u>

Network flow (7.1, 7.2, 7.3) [No more max weight with negatives]

- Maximum Flow (Algorithm, Time Complexity, Proof)
- Application of Maximum Flow problems (Indications of Max Flow Problems)

Network

- Idea: We want to maximize the amount of things we send from start to end (denoted as a flow)
- Weighted Directed Graph with properties
- Two vertices are distinguished as start and end
- Each edge has a weight, considered a capacity
    - RULE: For every edge, flow of edge i <= capacity of edge i
    - RULE: capacities are integers >= 1 (0 is equivalent to removing the edge)
    - Edge is considered <u>saturated</u> if it cannot hold any more flow/is already maximized
- Conservation of Flow: For every vertex, the incoming **<u>flow</u>** from other vertices into current vertex must be the same as the outgoing vertex **<u>flow</u>** from current vertex
    - Basically the Sum of **<u>flows</u>** entering = Sum of flows exiting for any given vertex i
    - **<u>Applies to all vertices except start and end</u>**
- A correct flow/legal flow is a network where all flows follow Conservation of Flow
- Max Flow: Network that is maximized across all flows (there is no possible way to increase flows at any point in the graph)

Problem: Select an edge to replace to better maximize flow

- Select "bottleneck" edges

Problem: Given a network, create a flow. Afterwards, create a maximum flow.

```
S ------→a ------→\
 \       ⬇️         END
  -----→B ---→---/
```

Idea: Greedy (Dijkstra is overkill)

- Problem: Find a path from s to t
    - Approach: Select a path
        - START - a - END → 1 + 1 → 2
    - Edge is considered <u>saturated</u> if it cannot hold any more flow/is already maximized
    - Ignore saturated edges, continue finding paths
    - Once start and end are separated (all edges connecting them are saturated/unavailable), stop and output the current network's flow
        - Total flow is sum of all flows exiting START node

Examples (using Greedy):

- Greedy for Shortest Path does not work
- Greedy grabbing maximum flow does not give optimal solution

Idea: Modified Greedy (where you can change your mind) [Augmented Network]

- Approach: For a given flow, create a backward edge with **<u>capacity</u> <u>of size flow</u>**
    - **<u>CAPACITY OF BACKWARD EDGE IS FLOW OF INCOMING VERTEX</u>**
    - Capacity of back flows are same as flows corresponding with entering node
    - If I want to change my mind, I replace current flow with the new subtraction of the backward edges
    - Iterate through augmented network with ability to use backwards edges as regular edges
    - Find new paths until START and END are disconnected (all edges in between are saturated)
    - Final network is optimal

Algorithm (to find MAX FLOW):

- Find any path from START to END
    - For each edge of path, apply 1 flow
- Create local augmented graph
- Find another path (using backward edges) from START in END in augmented network, create new augmented network

- Continue making augmented networks (ie Modify the previous one) until no more paths from S to T exist

Idea: Make Cuts (partitions where START and END are in different partitions)

- Capacity of cut: Sum of capacities of all edges going from **left to right** given a partition
- Min Cut: Cut with minimum capacity over all possible cuts (partitions)
- Given a cut, assuming the cut cuts edges which are used in the MAX PATH:
    - Regardless of path, the flows **MUST** go through the edges being cut
    - Flow is bounded by the capacity of cut
        - Max Flow <= Sum of capacities being cut
        - Cuts define an upper bound on flow
- If we cut an "impossible graph," the min cut will be 0 as there is no path from START to END
- Max Cut Algorithm and Min Cut Algorithm are same

Proof (Augmented graph outputted is optimal): (Fordan, Folkerson)

- Statements:
    - f is a max flow in G
    - The augmented/residual network Nf contains no augmenting path
        - Augmented network: Network with backward edges
        - Augmenting path: Path from S to T in augmented network
    - Value of flow is equal to capacity of some cut in network N
        - |f| = C(S,T) for some (S, T) cut in G
        - C(S,T) is Capacity of cut (S,T)
- Show all 3 statements are equivalent
    - If I have a maxflow (Theorem #1), then there no augmenting path (Theorem #2)
    - If there is no augmenting path (Theorem #2), the value of flow is equal to capacity of some cut (Theorem #3)
    - If the value of flow is equal to capacity of some cut (Theorem #3), then I have a maxflow (Theorem #1)
- Algorithm stops when there is no more augmenting path, meaning I should have a maxflow

- 1 → 2
    - Contradiction: If there exists a better augmented path, then we do not have a maxflow (We would be able to increase flow by 1) (aka If we have a maxflow we cannot increase it)
- 2 → 3
    - To Prove: A cut's capacity is equal to flow
    - Given a network with some saturated edges, remove/cut all saturated edges
        - Case: S and T are disconnected
            - Capacity of cut is equal to flow of cut
            - As we only remove saturated edges (which maximizes flow), this means that the flow is equal to capacity
        - Case: S and T are still connected
            - Impossible, as this means network has no augmenting path
- 3 → 1
    - Given a graph, flow is equal to some capacity
    - We know that flow is upper bounded by capacity
        - |f| <= capacity(S,T)
    - Whenever flow is equal to capacity, this means that we have a MAXFLOW

Lecture 14: Min Cut

Network flow (7.1, 7.2, 7.3, 7.5) *If time -- Shortest path with negative edge (6.8)*

- Min Cut
- Design Problems (Network Flow)

Pseudo Polynomial

- Pseudo Polynomial implies that the Time Complexity accounts for variables which are not a part of the problem input

Why do we want 1-to-1 matching for the example problem?

Capacity of outgoing edges for S is 1 due to 1-to-1 matching?

Exam Question:

- **Given an algorithm, provide a counter example if not possible.**

Given a network, how do you find a flow?

- Send one unit of flow into path, remove saturated paths

Min Cut:

- Max Flow Algorithm, reprove 2→3

Given a Max flow, can you construct an instance of a Max Flow?

You have a n cell phones in an area, you also have m cell towers. At any given instance in time, assign a cell phone to a tower.

If you have too many cell phones, not all can be assigned to towers.

Assumptions:

- Each phone can be connected to a tower within a certain radius
  - The radius is constant, known prior to us
- Each tower has a capacity

Idea: Network Flow

- Assignment of phone to tower is a unit of flow
    - Edges are added if and only if cell phone is within the radius of the tower
- Tower capacity is the upper bound of flow for a given tower
- Apply Ford-Fulkerson (Requires START and END nodes in graph)
    - Connect all towers to END node, weight of each directed edge as capacity of tower
    - Connect START to all phones
- Because we use Ford-Fulkerson: Flow out of S will be maximized
    - Number of matches is maximized
- <u>Limit connections of phones such that each phone can only (if I have arbitrary capacity, the infinite capacity implies that a singular phone can connect to multiple towers (as the problem only maximizes the connections))</u>
    - <u>**Splitting**</u>
        - Break single cell phone into multiple cell phones, where each outgoing flow's capacity is 1
        - Sets upper capacity on node (as the algorithm only accounts for edge capacity)
    - Modify Edge
        - Modify incoming edge such that capacity is 1

MAX FLOW = Max # of matches

- PBC, if we have MAX FLOW + 1, there must be MAX FLOW + 1 outgoing edges from START, which is not true, a contradiction

Every unit of flow corresponds with a match

<u>**Ford-Fulkerson ONLY works on integer flows**</u>

Runtime of Ford-Fulkerson:

- Finding a path from S → T
    - Graph is connected, E = n already as E and n are in same order, **O(|f| * E)** to find singular path
        - |f| = max flow
    - Pseudo polynomial depending on what max flow is

Bipartite Matching

- Two sets, A and B
- Edges only connect between vertices of A and vertices of B
- Bipartite graphs can not have odd cycles
    - Thus, bipartite graphs are two colorable
- Degree of vertices are arbitrary
- Assume connected graph (although it doesn't have to be)

Problem: Find maximum matches between A and B

- Idea: Take arbitrary instance of problem, solve sub problem to prove entire problem (Network Design problem)
    - Aka correspond maxflow to matches
- Add START and END nodes, connect A set to START, connect B set to END
- Add directions flowing from START to END
    - BLUE + RED problem (Note: there is no capacity on RED edges)
- Place capacity of one for edges between START and A vertices
    - Matching is 1 to 1, ensures 1 to one due to conservation of flow
- Place capacity of one for edges between END and B vertices
    - Matching is 1 to 1, ensures 1 to one due to conservation of flow

Algorithmic Transformation:

If Y is polynomial time, transformable to X is true

- Take input to Y, make some changes (reduce the input), take a problem and solve it, and output X's output as Y, then X can be used to solve Y
- Y: Are there two numbers that are equal in a list (Polynomial time)

    - X is sorting problem
        - Compare output of sorting numbers, check adjacent numbers

Given a set of cities, a travelling salesman wants to travel to each city in a minimal amount of time. (Factorial runtime is trivial, do it in polynomial)

Satisfiability Problem: Is there an assignment of 1s and 0s to inputted n variables (where some are complemented) that makes F true?

- For a given section, one element MUST be true

Prove mathematically that a problem is as difficult as NP-complete/NP-hard problems

- NP → Non-deterministic polynomial time Turing machine

<u>Lecture 15: NP Complete</u>

Problem Transformation

- Problem Y can be solved using Problem X
    - Y ≤p X

Suppose Y is polynomial transformable to X.

- If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time
    - This establishes a lower bound of a problem (via Problem Transformation)
- Proof by contradiction: If Y can be solved in polynomial time, then X can be solved in polynomial time, however Y cannot be solved in polynomial time, thus a contradiction

Come up with a bunch of problems (as problem Y), use proof techniques that we have a brand new problem which is as difficult (transformable) to problem Y

Travelling Salesman problem, Satisfiability Problem → No Solution (in Polynomial)

Idea: Max Clique ≤p max-independent set

Given a nondirected, non-weighted graph with vertices. Find the maximum clique. A clique is a set of vertices which are pairwise (directly) connected

- A-B-C is clique of size 3
- Maximum

Independent set → Set of vertices that are pairwise **not** connected

- A-B-D does not become an independent set

Show MC ≤p MIS
Proof:
Start with an arbitrary instance of maximum clique

- Proof cannot rely on example, although example can be used for visual argument
- From graph G, construct a graph G complement
    - Complement is constructed as vertices of complement is original vertices of graph G. Wherever we have an edge in graph G, we will NOT have an edge in G complement
    - G + G complement = no duplicate edges, and all possible edges (every pair of vertices)
    - MIS of G complement is 3
    - Every pair of vertices not connected in G complement **must** be connected in original G
        - Therefore, wherever there is no pair of vertices in G complement, we will have it in G

Garey-Johnson NP complete problems

Vertex Cover

- Given a graph G, is minimum number of vertices that are adjacent to all edges
- B, c are not adjacent → Does not touch all edges
- B, c, D, adjacent → Covers all edges

- A → Vertex Cover

Set Cover

- A={a, b, c}
- B={a, c, d, e}
- C={c, d }
- Given a collection of sets, where each set has elements, a set cover is a bunch of sets that touches/covers all possible elements in the group
- Minimum set cover is A, B

Idea: Vertex-Cover ≤p Set Cover

- There is no relation between these problems
- REMEMBER WE ARE SOLVING FOR LOWER BOUND TRANSFORMATION IN THIS CASE, where LHS is NP-COMPLETE
- Start from an arbitrary instance of vertex cover
    - Come up with sets, where each set is all edges

Two Steps

- Establish transformation
- Show solution of second problem is solution of first problem
- Prove all of this occurs in polynomial time

Final Questions

- DO we know any nlogn algorithm for Travelling Salesman? (1)
- Given 2 problems, where one is NP-Coomplete, transofmr it to another problem

<u>Lecture 16: Assorted Final Problems</u>

#4

Use dynamic programming to find a subsequence of a given sequence such that the subsequence sum is maximized and the subsequence elements are in increasing order.
The subsequence is not necessarily continuous.

Example: Given a sequence (0, 8, 4, 12, 2, 9) The best subsequence is (8, 12) with max sum being 20 = 8 + 12

DP ➔ Find dimension to break problem into subproblems
OPT(i) ➔ Optimal solution of first i elements

Algo:
Set all OPT(i) = max(0, arr[i])
For all i: ➔ O(n)

- For all j=0, …, i: ➔ O(n)
    - If arr[i] > arr[j]
        - OPT[i] = max(OPT[i],
            - If OPT[i] > OPT[j]: OPT[j] + arr[i]) ➔ O(1)
- Return max(OPT) ➔ O(n)

Time Complexity ➔ O(n^2)

Proof of Correction:
Induction Proof
Base Case: OPT(1) is correct (There is only one subsequence that ends at first val, you can include it or not include it)
Inductive Hypothesis: Assume we have the OPT solutions up to i
Induction:

- Exhaustively search for all j to append i to and take max

From the class we know that max-clique problem is NP-complete. Consider the class of graphs where degree of all vertices is the same. Is max-clique also NP-complete on this class of graphs? Prove your answer.

Idea to Agree: Show polynomial transformation from max-clique ➔ max-clique on all vertices
Idea to Disagree: Solve max-clique on all vertices in polynomial time

- <u>Demonstrating current problem is a subset of NP-Complete is not enough to proof</u>
  - Counter Argument → Base case can be calculated trivially

Idea: Some NP-Complete Problem/General Max-Clique ≤p Max-Clique Problem on K-Regular Graph

- K-Degree Graph is graph where all nodes have degree k

Approach:
Given a two partitions of graph G: A, B, where A has nodes of degree1 and B has nodes of degree2 where d1 < d2:

- Connect all nodes in A with helper nodes/vertices to increase the degree of all nodes of A to nodes of degree B
- Create d2 copies of original graph, any clique in original graph will be in 1 of the d2 copies in the graph. All helper nodes will connect to another helper nodes from another iteration of the original graph.

How to extract max-clique of original G?

- Any max-clique in graph H (The messy ugly one) is in the graph G as the helper nodes can't create cliques, <u>any non-trivial clique cannot be created because edges cannot connect between graphs directly (without the use of helper nodes)</u> aka, the helper nodes does not increase the max clique of H, max(G) == max(H)
- Since we don't delete edges, any clique in G would also be in H
- Analysis of Time Complexity of Transformation:
  - Worse Case:

        - Create n^2 edges
            - Add O(n) nodes
            - Create O(n) copies of G
        - Create O(n) Transformations
    - O(n^2 * n) ➔ O(N^3)
        - Transformation is polynomial, thus the problem is NP-Complete for k-regular graphs

An edge cover in a graph is a subset of edges such that the union of edge
endpoints corresponds to the entire vertex set of the graph.

Design an algorithm for finding a minimum edge cover in a bipartite graph. Describe the algorithm, prove its correctness and analyze its time complexity.

All nodes have degree 1

Approach: Network Flow with 1 to 1 matching

- Similarities of bipartite Matching and Minimizing edges of edge cover
    - Each edge breaks into 2 vertices

Algorithm:

- Separate graphs into 2 bipartite sets
- Convert into Network flow
    - Set edge capacity of all edges exiting Start and Entering END to 1
    - Only connect nodes using optimal edges which only connect between 2 nodes
- Arbitrarily connect any unmatched leftover nodes (greedy approach, maintains optimalness)

Time Complexity:

- O(V * E)
    - Number of flow can be v vertices, crossing E edges, same runtime as maxflow

Proof:

- Maintaining optimal edges