12/12 Network Flow Study Group

- Homework harder, discussion is same level as exam, same for challenge problem
- Useful for **resource allocation** problems.
- Runtime complexity for FF: $O(|E| * |f * |)$
- Runtime complexity for Edmonds-Karp: $O(|V| * E|^2)$ (for example in resource allocation d_j is not strictly defined, use this)
- Pick the right one! Calculate the runtime if needed.

Node Demands (slides)

- Given G=(V, E), edges with costs and nodes with demands
- Find if there exists an a feasible flow f, such that flow in(v) - flow out(v) = d for all v

Algorithm:

1) If sum of demands != 0, there is not a feasible flow

- Why? Because for each edge, flow in = flow out
- So the sum of all the flows is 0. Otherwise, no feasible flow!

2) If sum of demands = 0, add super-source s^*, super-sink t^* to G. Add edges (s^*, v) for nodes with negative demand, with capacity $-d_v$. Add edges for nodes with positive demand, with capacity d_v.
3) Run Ford-Fulkerson on G, check if $|f^*| = \sum_{v \in V}(d_v) = 0$.

- If $|f^*| = \sum_{v \in V}(d_v)$, means all edges entering sink are at capacity.

Node Capacities

- flow in(v) = flow out(v) $\leq c_v$

Reduction:

- Split each node with incoming edges into two nodes ($A_{\text{in}}, A_{\text{out}}, B_{\text{in}}, B_{\text{out}}$)
- Add edge between split nodes with capacity as the node capacity
- This edge restricts flow into the node to a potential bottleneck which is its node capacity (so, cannot exceed node capacity)
- Pictures are not proofs!

Valid answer example (!!):

Consider $G' = \{s, t\} \cup \{v_{\text{in}}, v_{\text{out}}\} : v \in V \setminus \{s, t\}$

Let $E' = \emptyset$.

Add $(v_{\text{out}}, v_{\text{in}})$ with capacity $c_{u,v}$ for all $(u, v) \in E$

Add $(v_{\text{in}}, v_{\text{out}})$ with capacity c_v for all $v \in V$.

Compute FF on $G' = (V', E')$.

Resource Allocation

- There are n workers, m jobs
- Each worker i can work up to $c(i, j)$ hours on job j.
- Each job j requires d_j many hours of work
- Each worker can work a total of c_i hours
- Can all jobs get done?
- Before reduction: what is the resource we are allocating? **work hours**

- Nodes must model these constraints in the network

Reduction
- Start with source and sink
- Job is done by a worker to finish it
  - So, n worker nodes
  - and m job nodes
- Source->worker capacity: c_i
- Worker->job capacity: $c(i, j)$
- Job->sink capacity: d_j
- Find max flow in Ford-Fulkerson
- If $|f^*| = \sum_{j \in [m]} d_j$ then feasible, otherwise, no.

Resource Allocation (with more constraints)
- Each worker belongs to one of l different unions.

Algorithm

Unions-Hours constraint
- Each union requires that the total hours worked by its workers is $\leq U_k$.
- Need one middle layer added. Where?
- Between source and the worker nodes
- Because the constraint on unions depends on workers and total hours
- source->union capacity: U_k
- union->worker capacity: c_i

Job-Unions constraint
- All jobs say they don't want all their workers to be from the same union.
- How to deal with this?
- Can't have unions in between worker and job nodes, blocks information
- Let's remove the union layer and split this layer...
- Workers->unions and workers->jobs, so have a union-job layer.
- Worker->(union,job) capacity: need to tell what union they belong to, and the job they can go to
  - Capacity: $c(i, j)$
- (union,job)->job capacity: $d_j - 1$ (none of the jobs are all done by the same union, so −1 to prevent saturation)
- How do we deal with U_k? with the previous constraint?
  - We can't use the same strategy as before, so union don't match up
  - Have another layer (worker, union) after source
  - source->(worker,union) capacity: U_k
  - (worker,union)->worker capacity: c_i

Sports
- There is a sports league with n teams.
- You support team A.
- You know the points table (how many points each team has) and the remaining fixtures.
- Each game is worth 1 point (team can either win 1 point or lose and nothing)
- Can the team win the league?

Algorithm

- Add a layer for teams…
- Points flow into a team how? Team gains points by winning a game.
- Add a layer for games, before teams
- So, graph structure is s->games->teams->t
- source->games capacity: 1 (can only win 1 point per game)
- Edges from games to teams that played the game
- games->teams capacity: 1
- teams->sink capacity: maximum amt of points any team can get (so our max points −1)
- Identify all games involving A, assume A wins all of them.
- $P_A^* +$ # of games with A.
  - teams->sink capacity: $P_A^* - 1$
- Edge from source to teams with P_i
- If $|f^*| < \sum(p_i) + \#$ of games $=$ total $\#$ points in the league $- P_A^*$ then not feasible
- Min-cut between source and games

Graph:

$G = (V, E)$ with n denoting $|V|$ & m denoting $|E|$
Graph Representation: Adjacency List, Edge List; Adjacency Matrix
Graph Traversal: BFS & DFS $O(m + n)$
Key Terms: DAG; Topological Ordering; Complete Graph; Bipartite Graph

Greedy:

1. Algorithm Design: Come up with a heuristic (usually straightforward)
2. Runtime Analysis:

   Input Preprocessing (e.g., Sort $O(nlogn)$) + Other Manipulation (usually $O(n)$)

3. Algorithm Correctness:

   Greedy Stays Ahead:

   (1) Define time step t.

   (2) Define $G = \{g_1, g_2, g_3, \dots, g_m\}$.

   Define $O = \{o_1, o_2, o_3, \dots, o_n\}$.

   Note: Ordering of G and O should be decided (otherwise, it may not be able to achieve one-to-one correspondence between elements in G and those in O)

   (3) Show by induction: for each time step $k \leq m$, $f(g_1, \dots g_k) \geq f(o_1, \dots, o_k)$.

   (4) Conclusion: greedy is optimal since it stays ahead of any arbitrary optimal.

   Exchange Argument:

   (1) Define $S = \{s_1, s_2, \dots, s_m\}$.

   Define $S^* = \{s_1^*, s_2^*, \dots, s_n^*\}$.

   (2) State differences between S and S^*.

   (3) For each difference, show S^* can be transformed into S by reorder/replace/remove/add without worsening the value.

   (4) By keep transforming, S^* can be transformed into S without worsening the value.

   (5) The greedy solution produced is just as good as any optimal arbitrary solution, and hence is optimal itself.

Important Algorithms: Interval Scheduling; Minimize Max Lateness;
Dijkstra's Shortest Path; MST (Prim's & Kruskal's)

D&C:

1. Algorithm Design: Binary Search Type or Merge Sort Type
2. Runtime Analysis:

(1) Find a recurrence relation:

Do *not* forget base case(s)!

$T(n) = aT\left(\frac{n}{b}\right) + C(n) + D(n)$ where:

a: number of recursive calls
b: size of each subproblem
$C(n)$: runtime for *merge* step
$D(n)$: runtime for *divide* step

(2) Use Unrolling or Recurrence Tree to solve recurrences

3. Algorithm Correctness:

(1) Soundness: strong induction on sample size

Base Case(s): trivially correct most of the time

Inductive Steps: Assume recursive calls give you the correct value at step k, WTS at step $k+1$, correct value is being returned (most of the time, it is about proving merge process merges the value in each subproblem correctly)

(2) Completeness: Recursive calls make progress towards base case(s) (trivially correct most of the time)

Important Algorithms: Binary Search; Merge-Sort

DP:

1. Algorithm Design:

*(0). Find a recursive solution

(1). Using memoization:

(1.1) Define a solution matrix A with dimensions explicitly stated.

(1.2) Find Bellman Equation. Do *not* forget base case(s)!

(1.3) Populating the matrix A & state where is the solution at.

(2). Backtrack (Recover) the choices that leads to the solution if you are asked to

2. Runtime Analysis:

$P(n) + DP(n) + F(n)$ where:

$P(n)$: Preprocessing time (e.g., sorting which takes $O(nlogn)$)
$DP(n) = (\text{number of cells}) \times (\text{runtime for filling up 1 cell})$
$F(n)$: Runtime for returning the final answer

3. Algorithm Correctness:
   Strong induction over the order of cell population of A. Prove the Bellman equation is essentially correct.

Important Algorithms: WIS; LIS; Games; 0-1 Knapsack; Bellman-Ford's Shortest Path

NF:

1. Algorithm Design: Max Flow / Min-Cut
2. Runtime Analysis:
   (1) Create Graph $O(|V| + |E|)$
   (2) Run Ford-Fulkerson Algorithm: $O(|E| \cdot f^*)$ where f^* is the max flow

Important Algorithms: Bipartite Matching; Image Segmentation;
Extensions: Node Demand & Lower Bounds; Project Selection

Intractability:

NP-C problems: IS; VC; SC; SP; 3SAT; CSAT; Hamiltonian Cycle/Path; TSP; Clique; 3D-Matching; 3-Coloring; Subset Sum; 0-1 Knapsack; Bin-Packing
Important NP-C problems: IS; VC; 3SAT

Showing $X \in$ NP-complete:

1. Prove $X \in$ NP (i.e., Prove X is polynomial-time verifiable)
2. $Y \leq_p X$ where $Y \in$ NP-complete
   (a) Provide efficient reduction.
   (b) Prove $S_Y \rightarrow S_X$
   (c) Prove $S_X \rightarrow S_Y$

Randomization:

Randomized algorithms use randomization to make decisions, so they are non-deterministic (same input may end up with different answers).

Types of randomized algorithms:

(i) *Monte Carlo*: with probability p returns the correct answer and provide an approximation *guarantee in expectation*
(ii) *Las Vegas*: always returns the correct answer, or informs about failure; runtime is polynomial *in expectation*
(iii) *Atlantic City*: probabilistic runtime and correctness

Probability Theory:

For each $i \in \Omega$ with Ω denoting the sample space:

(i) $p(i) \in [0,1]$

(ii) $\sum_{i\in\Omega} p(i) = 1$

For an event ε:

(i) $\Pr[\varepsilon] = \sum_{i\epsilon\varepsilon} p(i)$
(ii) $\Pr[\bar{\varepsilon}] = 1 - \Pr[\varepsilon]$

For an event A and an event B:

(i) Conditional Probability: $\Pr[\mathrm{A}|\mathrm{B}] = \frac{\Pr[\mathrm{A}\cap\mathrm{B}]}{\Pr[\mathrm{B}]}$
(ii) Independence: $\Pr[\mathrm{A}|\mathrm{B}] = \Pr[\mathrm{A}]$; $\Pr[\mathrm{B}|\mathrm{A}] = \Pr[\mathrm{B}]$;
$\Pr[\mathrm{A}\cap\mathrm{B}] = \Pr[\mathrm{A}]\cdot\Pr[\mathrm{B}]$
(iii) Mutually Exclusive: $\Pr[\mathrm{A}\cap\mathrm{B}] = 0$; $\Pr[\mathrm{A}\cup\mathrm{B}] = \Pr[\mathrm{A}] + \Pr[B]$

For a random variable X, a random variable Y, constants a and b, $\mathrm{E}[X] =$

$\sum_{j=0}^{\infty} j\cdot\Pr[X=j]$; $\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y]$; $\mathrm{E}[aX+b] = a\mathrm{E}[X] + b$

For *independent* random variables X and Y, $\mathrm{E}[XY] = \mathrm{E}[X]\mathrm{E}[Y]$

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

Asymptotic Analysis

1. *Kleinberg, Jon. Algorithm Design (p. 67, q. 3, 4).* Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

   (a) $f_1(n) = n^{2.5}$
   $f_2(n) = \sqrt{2n}$
   $f_3(n) = n + 10$
   $f_4(n) = 10n$
   $f_5(n) = 100n$
   $f_6(n) = n^2 \log n$

   Solution:

   f_2, $[f_3, f_4, f_5]$, f_6, f_1

   or $\sqrt{2n}$, $[n + 10, 10n, 100n]$, $n^2 \log n$, $n^{2.5}$

   (b) $g_1(n) = 2^{\log n}$
   $g_2(n) = 2^n$
   $g_3(n) = n(\log n)$
   $g_4(n) = n^{4/3}$
   $g_5(n) = n^{\log n}$
   $g_6(n) = 2^{(2^n)}$
   $g_7(n) = 2^{(n^2)}$

   Solution:

   g_1, g_3, g_4, g_5, g_2, g_7, g_6

   or $2^{\log n}$, $n(\log n)$, $n^{4/3}$, $n^{\log n}$, 2^n, $2^{(n^2)}$, $2^{(2^n)}$

2. *Kleinberg, Jon. Algorithm Design (p. 68, q. 5).* Assume you have a positive, non-decreasing function f and a positive such that $f(n) \geq 1$, non-decreasing function g such that $g(n) \geq 2$ and $f(n)$ is $O(g(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

   (a) $f(n)^2$ is $O(g(n)^2)$

   Solution:

   Since $f(n) = O(g(n))$, $\exists c_0, n_0 > 0$ such that $\forall n \geq n_0$, $f(n) \leq c_0 \cdot g(n)$.

   Because f and g are positive, we can square both sides of the inequality: $f(n)^2 \leq {c_0}^2 \cdot g(n)^2$

   Let $n_1 = n_0$ and $c_1 = c_0^2$. So we have $\forall n \geq n_1$, $f(n)^2 \leq c_1 \cdot g(n)^2$. Therefore, $f(n)^2$ is $O(g(n)^2)$.

   (b) $2^{f(n)}$ is $O(2^{g(n)})$

   Solution:

   False. Counterexample: $f(n) = \log_2 n^2$, $g(n) = \log_2 n$.

   $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2\log_2 n}{\log_2 n} = 2$

   $\lim_{n\to\infty} \frac{2^{f(n)}}{2^{g(n)}} = \lim_{n\to\infty} \frac{2^{\log_2 n^2}}{2^{\log_2 n}} = \lim_{n\to\infty} \frac{n^2}{n} = \lim_{n\to\infty} n = \infty$

   (c) $\log_2 f(n)$ is $O(\log_2 g(n))$

   Solution:

   Since $f(n) = O(g(n))$, $\exists c_0, n_0 > 0$ such that $\forall n \geq n_0$, $f(n) \leq c_0 \cdot g(n)$.
   So $\forall n \geq n_0$, $\log_2 f(n) \leq \log_2 c_0 + \log_2 g(n)$. Note that $\log_2 c_0$ and n_0 are constants.
   Let $n_1 = n_0$ and c_1 any positive number bigger than $\frac{\log_2 c_0}{\log_2 g(n_0)} + 1$. So $\forall n \geq n_1$, $\frac{\log_2 c_0}{\log_2 g(n)} + 1 \leq c_1$, and thus

   $$\log_2 c_0 + \log_2 g(n) \leq c_1 \log_2 g(n).$$

   Now, $\forall n \geq n_1$,

   $$\log_2 f(n) \leq \log_2 c_0 + \log_2 g(n) \leq c_1 \log_2 g(n).$$

   Therefore, $\log_2 f(n)$ is $O(\log_2 g(n))$. Note that $f(n) \geq 1$ assumption is needed for $\log_2 f(n) \geq 0$.

   P.S. The statement is actually false if the functions are not non-decreasing. Counterexample: $f(n) = 2(1 + \frac{1}{n})$, $g(n) = (1 + \frac{1}{n})$.

3. *Kleinberg, Jon. Algorithm Design (p. 68, q. 6).* You're given an array A consisting of n integers. You'd like to output a two-dimensional n-by-n array B in which $B[i, j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$ — that is, the sum $A[i] + A[i + 1] + ... + A[j]$. (Whenever $i \geq j$, it doesn't matter what is output for $B[i, j]$.) Here's a simple algorithm to solve this problem.

```
for i = 1 to n
  for j = i + 1 to n
    add up array entries A[i] through A[j]
    store the result in B[i, j]
  endfor
endfor
```

(a) For some function f that you should choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).

Solution: $O(n^3)$

(b) For this same function f, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.)

Solution: Since directly adding up entries `A[i]` through `A[j]` takes $j-i+1 > j-i$ operations, the total number of operations performed is at least

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i+1}^{n}(j-i) &= \sum_{i=1}^{n}\sum_{k=1}^{n-i}k \\
&= \frac{1}{2}\sum_{i=1}^{n}(n-i)(n-i+1) \\
&> \frac{1}{2}\sum_{i=1}^{n}(n^2-2ni+i^2) \\
&= \frac{1}{2}\left(n^3 - 2n\sum_{i=1}^{n}(i) + \sum_{i=1}^{n}(i^2)\right) \\
&= \frac{1}{2}\left(n^3 - n^2(n+1) + \frac{1}{6}n(n+1)(2n+1)\right) \\
&= \frac{1}{2}\left(\frac{1}{6}n(n+1)(2n+1) - n^2\right) = \Omega(n^3).
\end{aligned}$$

(c) Although the algorithm provided is the most natural way to solve the problem, it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time $O(g(n))$, where $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

Solution:

Initialize $B[1,1] = A[1]$, then fill in the first row as follows:

```
for i = 2 to n
   B[1,i] = B[1,i-1] + A[i]
endfor
```

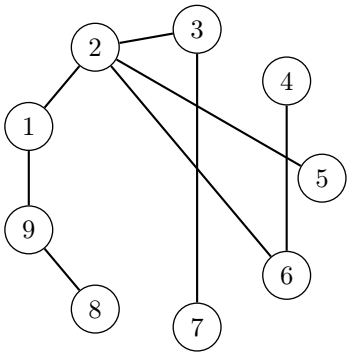
To fill in the rest of the array:

```
for row = 2 to n
   for col = row + 1 to n
      B[row,col] = B[row-1,col] - A[row-1]
   endfor
endfor
```

The first step runs in $O(n)$ time and the second step runs in $O(n^2)$ time, so the overall runtime for this algorithm is $O(n^2)$.

Graphs

4. Given the following graph, list a possible order of traversal of nodes by breadth-first search and by depth-first search. Consider node 1 to be the starting node.



Solution:

Breadth-First Search: in the groups, order doesn't matter
1, [2, 9], [3, 5, 6, 8] , [4, 7]

Depth-First Search: not an exhaustive list of solutions
1, 9, 8, 2, 3, 7, 6, 4, 5
1, 2, 3, 7, 6, 4, 5, 9, 8
1, 2, 6, 4, 5, 3, 7, 9, 8

5. *Kleinberg, Jon. Algorithm Design (p. 108, q. 5).* A binary tree is a rooted tree in which each node has at most two children. Show by induction that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

Solution:

In a binary tree with n nodes, let t_n be the number of nodes with two children and l_n be the number of leaves.

WTS: In a binary tree with n nodes, $t_n = l_n - 1$.

Base Case: A binary tree with 1 node has $l_1 = 1$ and $t_1 = 0$, so it holds that $t_1 = l_1 - 1$.

Inductive Hypothesis: In a binary tree with k nodes, $t_k = l_k - 1$.

Inductive Step: In a binary tree with $k+1$ nodes, consider an arbitrary leaf c. Since the tree has more than 1 node, c has a parent, p.

Consider the tree of k nodes that results from removing c:

1. If c was the only child of p, p is now a leaf in the place of c, and $l_{k+1} = l_k$. Additionally, the number of two-child nodes did not change, so $t_{k+1} = t_k$. Since by the inductive hypothesis, $t_k = l_k - 1$, we have $t_{k+1} = l_{k+1} - 1$.

2. If p had another child in addition to c, then removing c decrements the number of two-child nodes, so $t_{k+1} - 1 = t_k$. Additionally, p is not a leaf, so the number of leaves also decrements: $l_{k+1} - 1 = l_k$. Since by the inductive hypothesis, $t_k = l_k - 1$, we have $t_{k+1} - 1 = l_{k+1} - 1 - 1 \implies t_{k+1} = l_{k+1} - 1$.

6. *Kleinberg, Jon. Algorithm Design (p. 108, q. 7).* Some friends of yours work on wireless networks, and they're currently studying the properties of a network of n mobile devices. As the devices move around, they define a graph at any point in time as follows:

   > There is a node representing each of the n devices, and there is an edge between device i and device j if the physical locations of i and j are no more than 500 meters apart. (If so, we say that i and j are "in range" of each other.)

   They'd like it to be the case that the network of devices is connected at all times, and so they've constrained the motion of the devices to satisfy the following property: at all times, each device i is within 500 meters of at least $\frac{n}{2}$ of the other devices. (We'll assume n is an even number.) What they'd like to know is: Does this property by itself guarantee that the network will remain connected?

   Here's a concrete way to formulate the question as a claim about graphs:

   Claim: Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least $\frac{n}{2}$, then G is connected.

   Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

Solution:

Proof by Contradiction:

Suppose the graph is not connected, so there exists $x, y \in V$ such that there is no path between x and y.

Let X and Y be the set of nodes reachable from x and y, respectively. Note that $X \cap Y = \emptyset$.

Since nodes in G have degree at least $\frac{n}{2}$, we have that $|X| \geq \frac{n}{2} + 1$ and $|Y| \geq \frac{n}{2} + 1$.

Therefore, $|X \cup Y| = |X| + |Y| \geq 1 + \frac{n}{2} + 1 + \frac{n}{2} = n + 2$, which is a contradiction, since the graph G only has n nodes.

Coding Question: DFS

7. Implement depth-first search in either C, C++, C#, Java, Python, or Rust. Given an undirected graph with n nodes and m edges, your code should run in $O(n + m)$ time. Remember to submit a makefile along with your code, just as with the first coding question.

   Input: the first line contains an integer t, indicating the number of instances that follows. For each instance, the first line contains an integer n, indicating the number of nodes in the graph. Each of the following n lines contains several space-separated strings, where the first string s represents the name of a node, and the following strings represent the names of nodes that are adjacent to node s.

   The order of the nodes in the adjacency list is important, as it will be used as the tie-breaker. For example, consider an instance

```
4
xy v0 b
b xy
v0 xy a
a v0
```

   The tie break priority is `xy` → `v0` → `b` → `a`, so your search should start at `xy`, then choose `v0` over `b` as the second node to visit. Overall, your code should produce the following output:

```
xy v0 a b
```

   Input constraints:

   - $1 \leq t \leq 1000$
   - $1 \leq n \leq 100$
   - Strings only contain alphanumeric characters
   - Strings are guaranteed to be the names of the nodes in the graph.

   Output: for each instance, print the names of nodes visited in depth-first traversal of the graph, *with ties between nodes visiting the first node in input order*. Start your traversal with the first node in input order. The names of nodes should be space-separated, the output of each instance should be terminated by a newline, and the lines should have **no trailing spaces**.

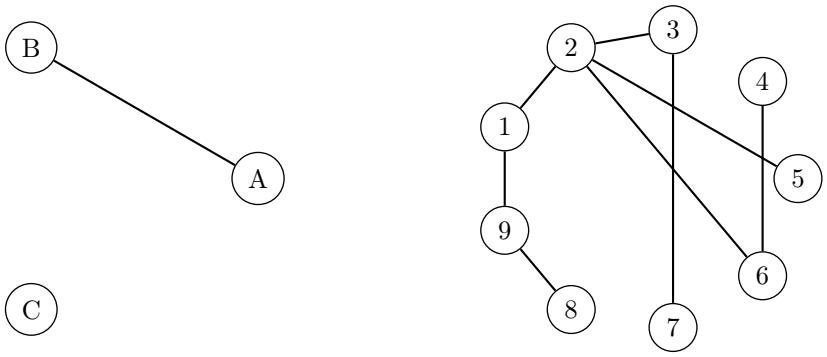
   Sample Input:

```
2
3
A B
B A
C
9
1 2 9
2 1 6 5 3
4 6
6 2 4
5 2
3 2 7
7 3
8 9
9 1 8
```

   Sample Output:

```
A B C
1 2 6 4 5 3 7 9 8
```

The sample input has two instances. The first instance corresponds to the graph below on the left. The second instance corresponds to the graph below on the right.



CS 577 # Assignment 3: Greedy Algorithms Fall 2024

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

Greedy Algorithms

1. In one or two sentences, describe what a greedy algorithm is. Your definition should be informal, something you could share with a non computer scientist.

Solution: A "greedy" algorithm only makes decisions which yield the highest immediate reward, but does not take into account how the decision will affect future states and their rewards. This is essentially treating every decision the algorithm makes as if that decision is the only one affecting the final result.

2. There are many different problems all described as "scheduling" problems. In the following questions, pay attention to the details of the problem setup, as they will change each time!

   (a) Let each job have a start time, an end time, and a value. We want to schedule as much value of non-conflicting jobs as possible. Use a counterexample to show that Earliest Finish First (the greedy algorithm we used for jobs with all equal value) does NOT work in this case.

Solution: Two jobs: The first job runs from time 0-2 and is worth 1. The second job runs from time 1-3 and is worth 2. Earliest Finish First selects the first job and scores 1, but the optimum is the second job and total value 2.

   (b) *Kleinberg, Jon. Algorithm Design (p. 191, q. 7)* Now let each job consist of two durations. A job i must be preprocessed for p_i time on a supercomputer, and then finished for f_i time on a standard PC. There are enough PCs available to run all jobs at the same time, but there is only one supercomputer (which can only run a single job at a time). The completion time of a schedule is defined as the earliest time when all jobs are done running on both the supercomputer and the PCs. Give a polynomial time algorithm that finds a schedule with the earliest completion time possible.

Solution: Longest Finish Time First

We sort all jobs by their finish times (ignoring preprocessing time) in $O(N \log N)$ time. Then we run jobs through the supercomputer in descending finish times. We assign each job to a standard PC for finishing as soon as it is done preprocessing.

(c) Prove the correctness and efficiency of your algorithm from part (b).

Solution: LFTF runs in $O(N \log N)$ time as described above as this is the time required to sort, and selecting jobs from the sorted list can be done in $O(N)$ time.

For correctness, suppose some optimal algorithm OPT chooses job x immediately before job y at some point, but $f_x \leq f_y$. (This is an inversion from LFTF.) We claim that exchanging jobs x and y results in an optimal schedule.

First, no other job's completion time is affected, because x, y has the same preprocessing duration as y, x. Because we preprocess y earlier as a result of the exchange, only job x may finish later. If we call the end of the combined preprocessing time p, then job x completes at $p + f_x$ after the exchange. But job y completed at $p + f_y$ before the exchange, and $f_x \leq f_y$. Thus, no job may increase the overall completion time and we may exchange all jobs into LFTF order while maintaining optimality.

3. *Kleinberg, Jon. Algorithm Design (p. 190, q. 5)*

(a) Consider a long straight road with houses scattered along it. We want to place cell phone towers along the road so that every house is within four miles of at least one tower. Give an efficient algorithm that achieves this goal using the minimum possible number of towers.

Solution: Proceed along the road from one end. Every time we reach a house not already in range of a tower, proceed four miles farther and plant a tower there.

(b) Prove the correctness of your algorithm.

Solution:

We write that our greedy algorithm places m towers in positions $g_1 < g_2 < \cdots = g_m$, and that an optimal solution places $n \leq m$ towers in positions $o_1 < o_2 < \cdots < o_n$. We then proceed with one proof by induction and one direct proof: the first showing that $o_i \leq g_i$ for all i from 1 to n, and the second showing that the greedy algorithm is optimal.

For the first proof, our induction is on the statement that $o_k \leq g_k$ for some value $k \leq n$. Our base case, $k = 1$, is simple: the greedy approach places the first tower as far along the road as possible (i.e. with as large a value as possible for g_1) while still being within range of the first house. To show the inductive hypothesis, we assume that $o_k \leq g_k$ for some $k < n$ and let x_h be the position of the first house too far along the road to be reached by greedy tower k (i.e. the smallest house position such that $x_h > g_k + 4$). Since $o_1 < o_2 < \cdots < o_k \leq g_k < x_h - 4$, the optimal approach must place another tower within range of x_h, so $o_{k+1} \leq x_h + 4$. The greedy approach, however, always places the next tower at position $g_{k+1} x_h + 4 \geq o_{k+1}$, so, if $o_k \leq g_k$ for some $k < n$, $o_{k+1} \leq g_{k+1}$. As such, by induction, we have that $o_k \leq g_k$ for all $k \leq n$.

To prove the optimality of the greedy algorithm, we first note that the greedy algorithm gives service to all houses with positions up to $g_k + 4$ with towers 1 through k (for any $k \leq m$): any house from positions $g_k - 4$ to $g_k + 4$ is satisfied by the tower at g_k, and no house further back than $g_k - 4$ can be given service by a later tower (as the g_i values are strictly increasing in i). We also observe that the optimal approach cannot give service to any houses after $o_n + 4$ with towers 1 through k, since all values for o_i are less than or equal to o_n. As such, all houses have positions less than or equal to $o_n + 4 \leq g_n + 4$, so all houses are given service by the first n towers placed by the greedy algorithm. As such, the greedy algorithm places only n towers, so $n = m$ and the greedy algorithm is optimal.

4. *Kleinberg, Jon. Algorithm Design (p. 197, q. 18)* Your friends are planning to drive north from Madison to the town of Superior, Wisconsin over winter break. They have drawn a directed graph with nodes representing potential stops and edges representing the roads between them.

   They have also found a weather forecasting site that can accurately predict how long it will take to traverse one of the edges on their graph, given the starting time t. This is important because some of the roads on their graph are affected strongly by the seasons and by extreme weather. It's guaranteed that it never takes negative time to traverse an edge, and that you can never arrive earlier by starting later.

   (a) Design an algorithm your friends can use to plot the quickest route. You may assume that they start at time $t = 0$, and that the predictions made by the weather forecasting site are accurate.

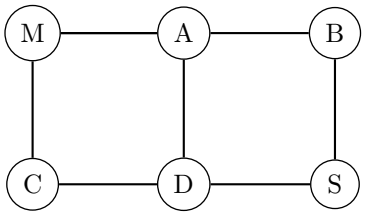
   Solution: Run Dijkstra's shortest paths algorithm, starting at Madison. Whenever we add node x to our shortest paths tree with shortest path $M \to x$ of length t, we query the forecasting site for the lengths of all edges outgoing from x at time t. Edge "lengths" change as a function of time, but since we can never reach the next node earlier by departing later, the key property behind Dijkstra's algorithm still holds: if an unvisited node x can be reached earlier than any other unvisited node, the shortest path through x does not go through any other unvisited nodes, since these can only be reached after we could reach x.

   To make recovering the path easier, each entry in the shortest path tree record includes both the shortest path distance to x and also the node y such that (y, x) was the last edge used.

(b) Demonstrate how your algorithm works using a small example with 6 nodes. Your demonstration should include any data structures you maintain during the execution of your algorithm and any queries you make to the weather forecasting site. For example, if your algorithm maintains a "current path" that grows from (M)adison to (S)uperior, you might show something like the following table:

Path	Total time
M	0
M,A	2
M,A,E	5
M,A,E,F	6
M,A,E	5
M,A,E,H	10
M,A,E,H,S	13

Solution:



Shortest paths:

Node	Distance	Predecessor	Priority queue (edges out of the shortest paths tree)
M	0		MA0:4,MC0:6
A	4	M	MC0:6,AD4:6,AB4:11
C	6	M	AD4:6,AB4:11
D	6	A	AB4:11,DS6:11
B	11	A	DS6:11
S	11	D	

Reconstructed path: M,A,D,S

Note: Priority queue holds queries of the form M to A at time t: t+edge time.

Coding Question

5. Implement the optimal algorithm for interval scheduling (for a definition of the problem, see the Greedy slides on Canvas) in either C, C++, C#, Java, or Python. Be efficient and implement it in $O(n \log n)$ time, where n is the number of jobs.

   The input will start with an positive integer, giving the number of instances that follow. For each instance, there will be a positive integer, giving the number of jobs. For each job, there will be a pair of positive integers i and j, where $i \leq j$, and i is the start time, and j is the end time.

   A sample input is the following:

   ```
   2
   1
   1 4
   3
   1 2
   3 4
   2 6
   ```

   The sample input has two instances. The first instance has one job to schedule with a start time of 1 and an end time of 4. The second instance has 3 jobs.

   For each instance, your program should output the number of intervals scheduled on a separate line. Each output line should be terminated by a newline. The correct output to the sample input would be:

   ```
   1
   2
   ```

Assignment 4 – More Greedy

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

More Greedy Algorithms

1. *Kleinberg, Jon. Algorithm Design (p. 189, q. 3).*

   You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

   Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Hint: Use the stay ahead method.

   Solution: By induction on the number of trucks, we will show that the described greedy strategy, FF, is always ahead of any other strategy. That is, FF has shipped at least as many boxes.

   Base case: 1 truck. With one truck, only what fits can be packed.

   Induction step: Assume the claim to be true for k trucks. Consider the case of $k+1$ trucks. By the induction hypothesis, we know that FF has shipped at least as many boxes as any strategy S in the first k trucks. For S to overtake FF with the $(k+1)$-th truck, S would have to pack, at the very least, all the items packed by FF in its $(k+1)$-th truck plus the next item j. Since FF did not pack j in the $(k+1)$-th truck, all those items cannot fit on a single truck.

2. *Kleinberg, Jon. Algorithm Design (p. 192, q. 8).* Suppose you are given a connected graph G with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Solution: Assume that G has at least 2 MSTs, T_1 and T_2, that differ. Let $e \in T_1 \setminus T_2 \cup T_2 \setminus T_1$ be the minimum cost edge not shared by T_1 and T_2. Without loss of generality, assume that e comes from T_1. We can create another graph $T_2 \cup e$ which now has a cycle C containing e. Let f be the most expensive edge in C.

If $f = e$, then T_1 is not MST as per Lemma 13 in lecture slides which is a contradiction.

If $f \neq e$, then, let $T_3 := T_2 \cup e \setminus f$. T_3 is a tree and, moreover, T_3 has must have a lower cost than T_2 since $c_e < c_f$ and they cannot be equal under the assumption of distinct edge weights. This is a contradiction as the overall cost of T_3 is strictly less than T_2.

3. *Kleinberg, Jon. Algorithm Design (p. 193, q. 10).* Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum-cost spanning tree T in G. Now assume that a new edge is added to G, connecting two nodes $v, w \in V$ with cost c.

   (a) Give an efficient ($O(|E|)$) algorithm to test if T remains the minimum-cost spanning tree with the new edge added to G (but not to the tree T). Please note any assumptions you make about what data structure is used to represent the tree T and the graph G, and prove that its runtime is $O(|E|)$.

Solution:

Data Structures: G will be represented by an adjacency list. For each adjacent node, the cost of the edge and a bit is associated. The associate bit being 1 means that edge is in the MST.

Algorithm: Beginning from v, do a DFS on T until w is found, storing the simple path from v to w path. Consider the cycle C formed by adding the new edge. If the cost of the new edge is not the maximum cost edge of C, then T is no longer an MST.

Run Time:

- DFS: $O(|V| + |E|) = O(|E|)$.
- Checking the max cost of the v to w path: $O(|E|)$.

   (b) Suppose T is no longer the minimum-cost spanning tree. Give a linear-time algorithm (time $O(|E|)$) to update the tree T to the new minimum-cost spanning tree. Prove that its runtime is $O(|E|)$.

Solution:

Algorithm: Beginning from v, do a DFS on T until w is found, storing the simple path from v to w path. Consider the cycle C formed by adding the new edge. Let f be the most expensive edge. Update the MST bit for f to 0, and update the MST bit for the new edge to 1.

Run Time:

- DFS: $O(|V| + |E|) = O(|E|)$.
- Finding the max cost edge of the v to w path: $O(|E|)$.

4. In class, we saw that an optimal greedy strategy for the paging problem was to reject the page the furthest in the future (FF). The paging problem is a classic online problem, meaning that algorithms do not have access to future requests. Consider the following online eviction strategies for the paging problem, and provide counter-examples that show that they are not optimal offline strategies.[1]

   (a) FWF is a strategy that, on a page fault, if the cache is full, it evicts all the pages.

   Solution: Request sequence: $\sigma = \langle a, b, c, a \rangle$
   Cache size: $k = 2$

   FWF: After the first two requests, the cache is full and FWF will evict the entire cache causing 2 more page faults. Overall 4 page faults.

   FF: After the first two requests, the cache is full and FF will evict b to bring in c with no page fault on the last request. Overall 3 page faults.

   (b) LRU is a strategy that, if the cache is full, evicts the least recently used page when there is a page fault.

   Solution: Request sequence: $\sigma = \langle a, b, c, a \rangle$
   Cache size: $k = 2$

   LRU: After the first two requests, the cache is full and LRU will evict a to bring in c. Then, evict b to bring in a. Overall 4 page faults.

   FF: After the first two requests, the cache is full and FF will evict b to bring in c with no page fault on the last request. Overall 3 page faults.

[1] An interesting note is that both of these strategies are k-competitive, meaning that they are equivalent under the standard theoretical measure of online algorithms. However, FWF really makes no sense in practice, whereas LRU is used in practice.

Coding Problem

5. For this question you will implement Furthest in the future paging in either C, C++, C#, Java, Python, or Rust. Your solution should be no worse than $O(nk)$ time, though try to aim for $O(n \log k)$ (where n is the number of page requests and k is the size of cache).

   The input will start with an positive integer, giving the number of instances that follow. For each instance, the first line will be a positive integer, giving the number of pages in the cache. The second line of the instance will be a positive integer giving the number of page requests. The third and final line of each instance will be space delimited positive integers which will be the request sequence.

   Input constraints:

   - $1 \leq k \leq 2000$
   - $1 \leq n \leq 100000$

   A sample input is the following:

```
3
2
7
1 2 3 2 3 1 2
4
12
12 3 33 14 12 20 12 3 14 33 12 20
3
20
1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

   The sample input has three instances. The first has a cache which holds 2 pages. It then has a request sequence of 7 pages. The second has a cache which holds 4 pages and a request sequence of 12 pages. The third has a cache which holds 3 pages and a request sequence of 20 pages.

   For each instance, your program should output the number of page faults achieved by furthest in the future paging assuming the cache is initially empty at the start of processing the page request sequence. One output should be given per line. The correct output for the sample input is

```
4
6
12
```

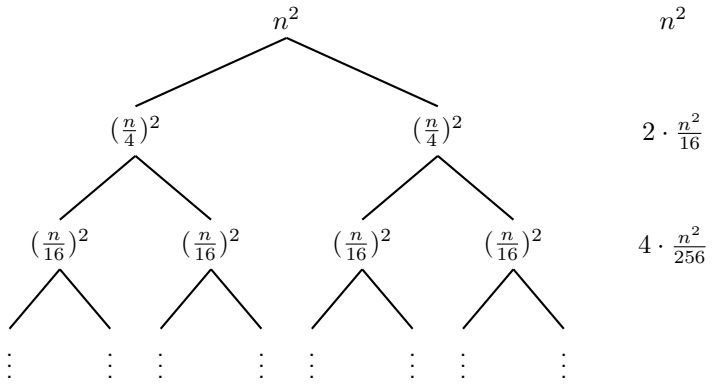
Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

Divide and Conquer

1. *Erickson, Jeff. Algorithms (p.49, q. 6).* Use recursion trees to solve each of the following recurrences.
   (a) $C(n) = 2C(\frac{n}{4}) + n^2$; $C(1) = 1$.

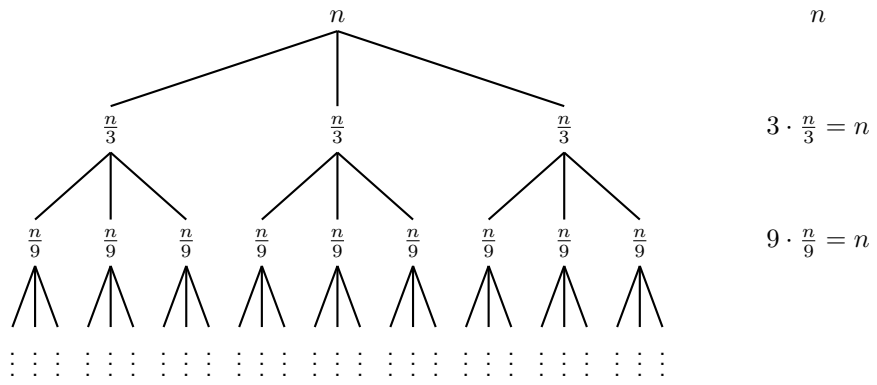
Solution:



Let d be the recursion tree depth. Notice that the amount of time consumed at depth i for any $i \in \mathbb{N}$ is equal to $2^i \cdot \frac{n^2}{16^i}$. Therefore, total runtime is $\sum_{i=0}^{d} 2^i \cdot \frac{n^2}{16^i} = n^2 \sum_{i=0}^{d} \left(\frac{2}{16}\right)^i \leq n^2 c$ for some constant c. So the runtime is $O(n^2)$.

   (b) $E(n) = 3E(\frac{n}{3}) + n$; $E(1) = 1$.

Solution:



The tree depth is $\log_3 n$ and each level sums to n, so the runtime is $O(n \log_3 n) \equiv O(n \log n)$.

2. *Kleinberg, Jon. Algorithm Design (p. 246, q. 1).* You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values—so there are $2n$ values total—and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the nth smallest value.

   However, the only way you can access these values is through queries to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k^{th} smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

   (a) Give an algorithm that finds the median value using at most $O(\log n)$ queries.

Solution: Label the databases A and B. Keep track of lower bounds in A_{lower} and B_{lower} (initialized to 1) and upper bounds in A_{upper} and B_{upper} (initialized to n). Let a and b be the results of querying A and B, respectively.

Base Case: If $A_{upper} \leq A_{lower}$ and $B_{upper} \leq B_{lower}$, the median is $min(a, b)$.

Recursive Case:

Query A on $k_1 = \frac{A_{lower}+A_{upper}}{2}$ and B on $k_2 = \frac{B_{lower}+B_{upper}}{2}$ (assume integer division).

If $a > b$, set $A_{upper} = k_1$ and $B_{lower} = k_2 + 1$ (don't change A_{lower} and B_{upper}) and recurse.

If $a < b$, set $A_{lower} = k_1 + 1$ and $B_{upper} = k_2$ (don't change A_{upper} and B_{lower}) and recurse.

   (b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence. Show your work and do not use the master theorem.

Solution: Observe that, in each recursive call, the range

$$A[A_{lower}, \ldots, A_{upper}] \cup B[B_{lower}, \ldots, B_{upper}]$$

of values left to search is cut in half, either to

$$A\left[A_{lower}, \ldots, \frac{A_{lower} + A_{upper}}{2}\right] \cup B\left[\frac{B_{lower} + B_{upper}}{2} + 1, \ldots, B_{upper}\right] \tag{1}$$

if $a > b$, or to

$$A\left[\frac{A_{lower} + A_{upper}}{2} + 1, \ldots, A_{upper}\right] \cup B\left[B_{lower}, \ldots, \frac{B_{lower} + B_{upper}}{2}\right]$$

if $a < b$. So the recurrence is $T(n) = T(n/2) + O(1)$; $T(1) = 1$. This yields a recurrence tree with $\log(n)$ layers and $O(1)$ work at each layer, hence the solution is $O(\log n)$.

(c) Prove correctness of your algorithm in part (a).

Solution: Let x be the desired nth smallest value, and let $R_A = A[A_{lower}, \ldots, A_{upper}]$ and $R_B = B[B_{lower}, \ldots, B_{upper}]$. We prove the following statement $(*)$ by (strong) induction on the size $A_{upper} - A_{lower} + 1 + B_{upper} - B_{lower} + 1$ of the remaining search range $R_A \cup R_B$.

$(*)$ If the (smaller) median of $R_A \cup R_B$ is x, then the algorithm returns x.

Initially, $A_{lower} = B_{lower} = 1$ and $A_{upper} = B_{upper} = n$, so $(*)$ asserts our algorithm is correct.

For the base case $A_{upper} - A_{lower} = 0$ and $B_{upper} - B_{lower} = 0$ (equivalently $A_{upper} = A_{lower}$ and $B_{upper} = B_{lower}$), $R_A \cup R_B$ contains just two values, the smaller of which, by assumption, is x. The algorithm returns the min of the two values, which is x, so $(*)$ holds.

Now assume $(*)$ holds for all ranges smaller than $R_A \cup R_B$. Observe that a and b are the medians of R_A and R_B, respectively.

Suppose $a > b$. Then $x \leq a$. Otherwise, if $x > a$, then x is larger than more than half the elements of R_A, and, since $x > a > b$, x is also larger than more than half the elements of R_B. This contradicts the assumption that x is the median of $R_A \cup R_B$. Similarly, $x \geq b$. In this case, the algorithm restricts to the range $R'_A \cup R'_B$ in (1). This removes $(A_{upper} + A_{lower})/2 - 1$ elements smaller than a, hence smaller than x, and $(B_{upper} + B_{lower})/2 - 1$ elements larger than b, hence larger than x. By symmetry of the A and B indices, $(A_{upper} + A_{lower})/2 - 1 = (B_{upper} + B_{lower})/2 - 1$, so, to create $R'_A \cup R'_B$, we have removed from $R_A \cup R_B$ an equal number of elements larger and smaller than x, which by assumption is the median of $R_A \cup R_B$. Hence x is also the median of $R'_A \cup R'_B$. By induction, $(*)$ also holds for the smaller range $R'_A \cup R'_B$, and x is the median of $R'_A \cup R'_B$, so by $(*)$ the algorithm returns x.

A symmetric proof applies for the case $a < b$.

3. *Kleinberg, Jon. Algorithm Design (p. 246, q. 2).* Recall the problem of finding the number of inversions. As in the text, we are given a sequence of n numbers $a_1, ..., a_n$, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

   We motivated the problem of counting inversions as a good measure of how different two orderings are. However, this measure is very sensitive. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$.

   (a) Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

Solution:

Return Values: The number of significant inversions and sorted version of input sequence.

Divide: Split the input sequence into two halves and recursively calculate the number of significant inversions in each half as well as the sorted version of the input sequence.

Merge: Since we have the result of inversions in the left and right halves, we just need to calculate inversions between halves. Let the left and right half be L and R respectively. Let i and j be initialized to the size of L and R respectively. Initialize result N to be the sum of the counts of inversions exclusively on the left and right halves. If $L[i] \leq 2R[j]$, then, if $j > 1$, set $j \leftarrow j - 1$, and, if $j = 1$, stop. If $L[i] > 2R[j]$, set $N \leftarrow N + j$. Then, if $i > 1$, set $i \leftarrow i - 1$, and, if $i = 1$, stop. Finally, use the mergesort merge step to generate the sorted sequence to return. Return N and the sorted sequence.

Base case: If the length of the input sequence is 1, just return 0 inversions and the input sequence.

(b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence. Show your work and do not use the master theorem.

Solution: In the Divide step, the algorithm makes two recursive calls, each on half of the array ($2T(n/2)$). The loop in the Merge step goes through the left and right half exactly once, then runs the mergesort merge procedure, each of which takes time linear in n.

$$T(n) = 2T(n/2) + O(n).$$

This is identical to the recurrence for mergesort, and has solution $O(n \log n)$.

(c) Prove correctness of your algorithm in part (a).

Solution: We prove the following statement $(*)$ by (strong) induction on the length k of the input sequence.

$(*)$ On input sequences of length k, the algorithm returns the correct number of inversions and the correctly sorted sequence.

For $k = n$, $(*)$ asserts that our algorithm is correct.

Base case: For $k = 1$, the algorithm correctly asseses that a sequence of length 1 has no inversions and is already sorted.

Induction hypothesis: $(*)$ holds for sequences of length less than k.

Induction step: The Divide step recurses on the first and second half of the sequence, each of which have length $< k$, so, by our induction hypothesis, the algorithm obtains a correct value N and the two half-sequences are properly sorted. It remains to show that the Merge step is correct. If $L[i] \leq 2R[j]$, then (i, j) do not form a significant inversion. If $L[i] > 2R[j]$ then (i, j) are a significant inversion, and furthermore, since the right half is sorted in increasing order, (i, ℓ) are a significant inversion for every $1 \leq \ell \leq j$. Hence the algorithm correctly adds j significant inversions. Since every previous j satisfied $L[i] \leq 2R[j]$, there are no more significant inversions involving i, so the algorithm correctly decrements i. Since the left half is sorted, we don't miss any significant inversions by maintaining the same j value while decrementing i.

Thus the Merge steps correctly computes the number of significant inversions. We know from lecture that the mergesort merge step is correct, so $(*)$ holds.

4. *Kleinberg, Jon. Algorithm Design (p. 246, q. 3).* Suppose you're consulting for a bank that's concerned about fraud detection. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud.

   It's difficult to read the account number off a bank card directly, but the bank has an "equivalence tester" that takes two bank cards and determines whether they correspond to the same account.

   Their question is the following: among the collection of n cards, is there a set of more than $\frac{n}{2}$ of them that all correspond to the same account? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester.

   (a) Give an algorithm to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

   Solution:

   Return values: Card that represents majority element in a group.

   Divide: Split the cards into two halves and recursively find the majority element of each half.

   Merge: Since the majority element in a subproblem must also be the majority element in at least one of its halves, consider the majority element (if any) returned by each half.

   - If both sides have the same majority element, return this as the majority element.
   - If neither side has a majority element, return "none".
   - Otherwise, let a and b be the majority elements of the left and right halves, respectively. Compare a and b with every element in both halves to obtain numbers n_a and n_b of elements matching a and b in the combined array. If $n_a > \frac{n}{2}$, return a; if $n_b > \frac{n}{2}$, return b, and if both $n_a, n_b \leq \frac{n}{2}$, return "none".

   Base case: Only one card to compare. No need to invoke equivalence tester, but this card is the majority element of its group (size 1).

   (b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence. Show your work and do not use the master theorem.

   Solution: In the Divide step, the algorithm makes two recursive calls, each on half of the array ($2T(n/2)$). The Merge step, in the worst case, compares every element in the array to both a and b, requiring $2n$ steps. So the recurrence is

   $$T(n) = 2T(n/2) + O(n).$$

   This is identical to the recurrence for mergesort, and has solution $O(n \log n)$.

(c) Prove correctness of your algorithm in part (a).

Solution: We prove the following statement $(*)$ by (strong) induction on the length k of the input sequence.

$(*)$ On input of k cards, the algorithm correctly determines whether a majority exists and, if so, returns a member of the majority.

For $k = n$, $(*)$ asserts that our algorithm is correct.

Base case: For $k = 1$, the algorithm correctly returns the only card as representing a majority.

Induction hypothesis: $(*)$ holds for sets of fewer than k cards.

Induction step: The Divide step recurses on the first and second half of the cards, each of which is a set of $< k$ cards, so, by our induction hypothesis, the recursive call correctly determines the majority. It remains to show that the Merge step is correct. Let a, b, n_a, n_b be defined as in the solution to part (a).

1. If $a = b$ and $n_a, n_b > (n/2)/2$ (both sides have the same majority element), then $n_a + n_b > n/2$, so indeed $a = b$ is a majority element.
2. If neither side has a majority, then for any x in the left half and y in the right half, $n_x + n_y \leq (n/2)/2 + (n/2)/2 = n/2$, so the combined list has no majority.
3. If the sides have different majority elements a and b, then by similar reasoning to case 2, no element other than a or b can form a majority, so the algorithm behaves correctly.

Thus the algorithm returns the correct majority element, so $(*)$ holds.

5. **Inversion Counting:**
Implement an optimal algorithm for inversion counting in either C, C++, C#, Java, Python, or Rust. Be efficient and implement it in $O(n \log n)$ time, where n is the number of elements in the list.

The input will start with a positive integer, k, giving the number of instances that follow. For each instance, there will be a positive integer, j, giving the number of elements in the list.

Input constraints:

- $0 \leq k \leq 1000$
- $0 \leq j \leq 100000$
- $0 \leq$ list elements $\leq 2^{31} - 1$

Note that the results of some of the test cases may not fit in a 32-bit integer.

A sample input is the following:

```
2
5
5 4 3 2 1
4
1 5 9 8
```

The sample input has two instances. The first instance has 5 elements and the second has 4. For each instance, your program should output the number of inversions on a separate line. Each output line should be terminated by a newline. The correct output to the sample input would be:

```
10
1
```

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

Divide and Conquer

1. *Kleinberg, Jon. Algorithm Design (p. 248, q. 5)* Hidden surface removal is a problem in computer graphics where you identify objects that are completely hidden behind other objects, so that your renderer can skip over them. This is a common graphical optimization.

   In a clean geometric version of the problem, you are given n non-vertical, infinitely long lines in a plane labeled $L_1 \dots L_n$. You may assume that no three lines ever meet at the same point. (See the figure for an example.) We call L_i "uppermost" at a given x coordinate x_0 if its y coordinate at x_0 is greater than that of all other lines. We call L_i "visible" if it is uppermost for at least one x coordinate.

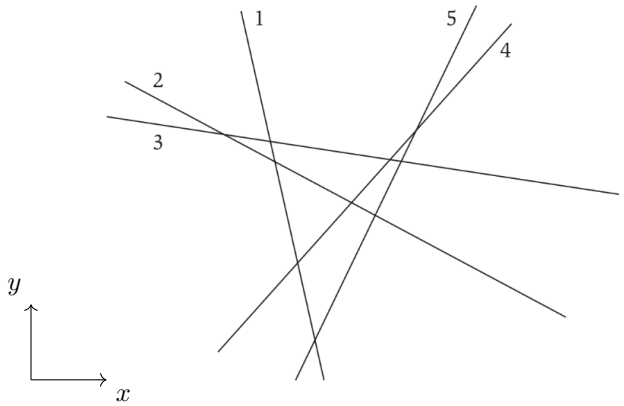


Figure 5.10 An instance of hidden surface removal with five lines (labeled *1–5* in the figure). All the lines except for *2* are visible.

   (a) Give an algorithm that takes n lines as input and in $O(n \log n)$ time returns all the ones that are visible.

Solution: Input: A set of lines $A = \{L_1, ..., L_n\}$.
Output: A set of visible lines of the form (L_i, a, b) where L_i is a line, a is the start of its visible interval, and b is the end of its visible interval. This set is sorted in increasing order with respect to the intervals of visibility of the contained lines.
Base Case: If $|A| = 1$, let $A = \{L_i\}$ and return $[(L_i, -\infty, \infty)]$
Divide: Arbitrarily split A into two sets L, R such that $|L| = \left\lceil \frac{|A|}{2} \right\rceil$, $|R| = \left\lfloor \frac{|A|}{2} \right\rfloor$. Then get the result L' and R' of calling the function on L and R respectively.
Merge: Consider the two sets L', R'. Create an empty set O and, similarly to merge sort, add elements from L' or R' in increasing order with respect to their intervals of visibility until there is an overlap in visibility between the current elements $(L_i, a, b) \in L'$ and $(L_j, c, d) \in R'$. Let (s, t) be the interval on which these two intervals of visibility overlap. Let x_0 be the x-coordinate of the intersection between L_i and L_j (if it exists), and, without loss of generality, assume L_i is above L_j to the left of x_0 (or always above if no intersection exists). If $x_0 > t$ or does not exist, set element $(L_j, c, d) = (L_j, t, d)$ if $t < d$, else skip this element. If $x_0 < s$ set $(L_i, a, b) = (L_i, a, s)$ if $a < s$, else skip this element. If x_0 is within (s, t), set $(L_i, a, b) = (L_i, a, x_0)$ and $(L_j, c, d) = (L_j, x_0, d)$. Once this terminates, return O.

(b) Write the recurrence relation for your algorithm.

Solution: $T(n) = 2T(\frac{n}{2}) + O(n)$

(c) Prove the correctness of your algorithm.

Solution: We will show that our algorithm is correct for all inputs of size $n \in \mathbb{N}$ by induction.
Base Case: If $n = 1$, our algorithm returns the only line with visibility range $(-\infty, \infty)$ which is correct.
Inductive Hypothesis: Assume that our algorithm is correct on all inputs of size j where $1 \leq j \leq k$ for some k.
Inductive Step: Consider an input of size $k + 1$. First, our algorithm divides this input into two sets of size $1 \leq \lceil \frac{k+1}{2} \rceil \leq k$ and $1 \leq \lfloor \frac{k+1}{2} \rfloor \leq k$ and recurses on these two sets giving us L' and R' which are in the format that we expect by our inductive hypothesis. Now, if there is no conflict in the intervals, adding them to O in increasing order similarly to merge sort will result in no issues. If there is a conflict, we skip an element or change the visibility intervals accordingly. We skip an element if the line is completely covered, so there is no need to include it in O. Otherwise, our interval update removes the conflict between the two lines by updating the visible intervals for each line with consideration for how the two lines affect each other. We know that adding in this way is valid because a line can only be visible over one contiguous interval, and, thus, if there are no conflicts in intervals between the two elements we consider, then there are no conflicts with any future element. We also add the elements in order, so the resultant list will be in order. Thus, once we have finished adding the elements to O, they will directly correspond to visible lines and the intervals on which they are visible in sorted order. So, our algorithm is correct on an input of size $k + 1$.

2. In class, we considered a divide and conquer algorithm for finding the closest pair of points in a plane. Recall that this algorithm runs in $O(n \log n)$ time. Let's consider two variations on this problem:

   (a) First consider the problem of searching for the closest pair of points in 3-dimensional space. Show how you could extend the single plane closest pairs algorithm to find closest pairs in 3D space. Your solution should still achieve $O(n \log n)$ run time.

   Solution: Our algorithm proceeds just as in the 2D case. We get a sorted list of the points by x, y, and now also z coordinate, and split the space in half using the x coordinate list, recursing on each half.

   Considering crossing pairs, note that by the same logic as in the 2D case there are only a constant number of points that may be "close enough" that we have to check if they are the closest pair, which is all we need to show that we can still do this in $O(n \log n)$ time.

   (b) Now consider the problem of searching for the closest pair of points on the surface of a sphere (distances measured by the shortest path across the surface). Explain how your algorithm from part a can be used to find the closest pair of points on the sphere as well.

   Solution: The two closest points as measured along the surface are also the two closest points as measured straight through the interior of the sphere, so we can apply the algorithm from part a directly to our set of points, and it will return the correct answer.

   (c) Finally, consider the problem of searching for the closest pair of points on the surface of a torus (the shape of a donut). A torus can be thought of taking a plane and "wrap" at the edges, so a point with y coordinate MAX is the same as the point with the same x coordinate and y coordinate MIN. Similarly, the left and right edges of the plane wrap around. Show how you could extend the single plane closest pairs algorithm to find closest pairs in this space.

   Solution: The difference between the "wrapped" plane and the standard closest pairs problem is that we don't immediately have the ability to sort points, and we must worry about crossing pairs on all sides.

   We'll mostly use the exact same 2D algorithm, but at the top level we need to do something to address the wrapping. At this level, either the closest pair is interior to the left half, interior to the right half, crosses the middle, crosses the top-bottom edge, crosses the left-right edge, or may cross the top-bottom edge and one of the two other boundaries. We can handle each crossing pairs problem in the same way we do for the standard 2D case without increasing the asymptotic time required.

3. *Erickson, Jeff. Algorithms (p. 58, q. 25 d and e)* Prove that the following algorithm computes $\gcd(x, y)$ the greatest common divisor of x and y, and show its worst-case running time.

```
BINARYGCD(x,y):
if x = y:
    return x
else if x and y are both even:
    return 2*BINARYGCD(x/2,y/2)
else if x is even:
    return BINARYGCD(x/2,y)
else if y is even:
    return BINARYGCD(x,y/2)
else if x > y:
    return BINARYGCD( (x-y)/2,y )
else
    return BINARYGCD( x, (y-x)/2 )
```

Solution: Assume for induction that BINARYGCD works for all $x \leq x_0$, $y \leq y_0$ (except x_0, y_0). We'll show that this implies BINARYGCD(x_0, y_0) as well. For a base case, note that BINARYGCD(1,1)=1 as it should.

We proceed for each case in the conditional and consider the mathematics. If $x = y$, then x is indeed the GCD. If both are even, then we can divide x, y by two and get the GCD divided by 2 as well, exactly as the algorithm does. If only x or y is even, then the GCD cannot itself be a factor of 2 and so we can divide out the 2 without changing the GCD–exactly as the algorithm does. Otherwise, since the GCD divides both x and y, it must divide $x - y$ as well. Note that both x and y must be odd, so $x - y$ must be even, so the algorithm can correctly divide by 2 as well. We've covered all possible combinations of x, y being even or odd, and in all cases correctly reduced BINARYGCD(x, y) to a recursive call on a strictly smaller case.

Assume that the running time of division and subtraction is $O(1)$. The worst-case running time of BINARYGCD is $O(log(x) + log(y)) = O(log(xy))$. In the worst-case, BINARYGCD will divide either x or y by 2 at each step. Alternatively, the solution can be written as $O(log(max(x, y))$

4. Use recursion trees or unrolling to solve each of the following recurrences. Make sure to show your work, and do NOT use the master theorem.
   (a) Asymptotically solve the following recurrence for $A(n)$ for $n \geq 1$.

$$A(n) = A(n/6) + 1 \qquad \text{with base case} \qquad A(1) = 1$$

Solution:
Recursion tree is a path descending from the root. The work in each layer is 1. The number of layers is $\log_6(n)$. Total work is then

$$A(n) = \sum_{k=1}^{\Theta(\log_6(n))} 1 = \Theta(\log n)$$

(b) Asymptotically solve the following recurrence for $B(n)$ for $n \geq 1$.

$$B(n) = B(n/6) + n \qquad \text{with base case} \qquad B(1) = 1$$

Solution:
Recursion tree is a path descending from the root. The work in each layer is one sixth that of the previous layer, where the root has work n. The number of layers is $\log_6(n)$. Total work is then

$$B(n) = \sum_{k=0}^{\Theta(\log_6(n))} \frac{n}{6^k} = n\frac{1 - \frac{1}{6^{\log_6(n)}}\frac{1}{6}}{1 - 1/6} = n\frac{1 - \frac{1}{6n}}{1 - 1/6} = n\frac{6n-1}{6n}\frac{6}{5} = \frac{6n-1}{5} \in \Theta(n)$$

(c) Asymptotically solve the following recurrence for $C(n)$ for $n \geq 0$.

$$C(n) = C(n/6) + C(3n/5) + n \qquad \text{with base case} \qquad C(0) = 0$$

Solution: The total work in each layer of the tree is $\frac{1}{6} + \frac{3}{5} = \frac{5}{30} + \frac{18}{30} = \frac{23}{30}$ times the previous layer, with n at the root. The value of $C(n)$ is then

$$C(n) = \sum_{k=0}^{\infty} n(\tfrac{23}{30})^k = \frac{n}{1 - \frac{23}{30}} = \frac{n}{7/30} = \frac{30n}{7} \in \Theta(n)$$

(d) Let $d > 3$ be some arbitrary constant. Then solve the following recurrence for $D(x)$ where $x \geq 0$.

$$D(x) = D\left(\tfrac{x}{d}\right) + D\left(\tfrac{(d-2)x}{d}\right) + x \qquad \text{with base case} \qquad D(0) = 0$$

Solution: The total work in each layer of the recursion tree is $(d-1)/d$ times the previous layer, with x at the root. The value of $D(x)$ is then

$$D(x) = x\sum_{k=0}^{\infty} (\tfrac{d-1}{d})^k = \frac{x}{1 - \frac{d-1}{d}} = \frac{x}{\frac{1}{d}} = dx$$

Coding Questions

5. **Line Intersections:**
Implement a solution in either C, C++, C#, Java, Python, or Rust to the following problem.

Suppose you are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Create a set of n line segments by connecting each point p_i to the corresponding point q_i. Your goal is to develop an algorithm to determine how many pairs of these line segments intersect. Your algorithm should take the $2n$ points as input, and return the number of intersections. Using divide-and-conquer, your code needs to run in $O(n \log n)$ time.
Hint: How does this problem relate to counting inversions in a list?

Input should be read in from stdin. The first line will be the number of instances. For each instance, the first line will contain the number of pairs of points (n). The next n lines each contain the location x of a point q_i on the top line. Followed by the final n lines of the instance each containing the location x of the corresponding point p_i on the bottom line. For the example shown in Fig 1, the input is properly formatted in the first test case below.

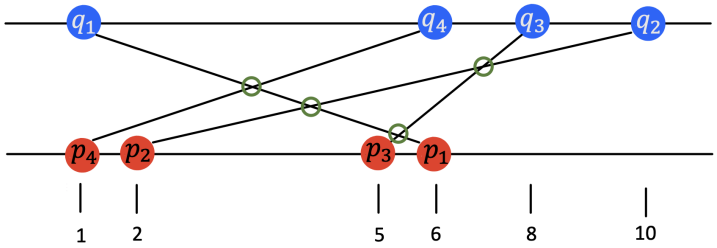


Figure 1: An example for the line intersection problem where the answer is 4

Constraints:

- $1 \leq n \leq 10^6$
- For each point, its location x is a positive integer such that $1 \leq x \leq 10^6$
- No two points are placed at the same location on the top line, and no two points are placed at the same location on the bottom line.
- Note that the results of some of the test cases may not fit in a 32-bit integer.

Sample Test Cases:

input:
2
4
1
10
8
6
6
2
5
1
5
9
21

1
5
18
2
4
6
10
1

expected output:
4
7

CS 577 # Assignment 7 – Dynamic Programming Fall 2024

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

Dynamic Programming

Do **NOT** write pseudocode when describing your dynamic programs. Rather give the Bellman Equation, describe the matrix, its axis and how to derive the desired solution from it.

1. *Kleinberg, Jon. Algorithm Design (p.313 q.2).*

   Suppose you are managing a consulting team and each week you have to choose one of two jobs for your team to undertake. The two jobs available to you each week are a low-stress job and a high-stress job.

   For week i, if you choose the low-stress job, you get paid ℓ_i dollars and, if you choose the high-stress job, you get paid h_i dollars. The difference with a high-stress job is that you can only schedule a high-stress job in week i if you have no job scheduled in week $i-1$.

   Given a sequence of n weeks, determine the schedule of maximum profit. The input is two sequences: $L := \langle \ell_1, \ell_2, \ldots, \ell_n \rangle$ and $H := \langle h_1, h_2, \ldots, h_n \rangle$ containing the (positive) value of the low and high jobs for each week. For Week 1, assume that you are able to schedule a high-stress job.

   (a) Show that the following algorithm does not correctly solve this problem.

Algorithm: JobSequence

```
Input  : The low (L) and high (H) stress jobs.
Output: The jobs to schedule for the n weeks
for Each week i do
    if h_{i+1} > ℓ_i + ℓ_{i+1} then
        Output "Week i: no job"
        Output "Week i+1: high-stress job"
        Continue with week i+2
    else
        Output "Week i: low-stress job"
        Continue with week i+1
    end
end
```

Solution:
Counter-example:
$L := \langle 1, 1, 1 \rangle$
$H := \langle 1, 10, 100 \rangle$

For this instance, the algorithm JobSequence produces a schedule of: $\langle -, H, L \rangle$ with a value of 11, whereas the optimal schedule is $\langle L, -, H \rangle$ for a value of 101.

(b) Give an efficient algorithm that takes in the sequences L and H and outputs the greatest possible profit.

Solution: This is solved with dynamic programming.

- A $(n+2)$-element array s, where $s[i]$ contains the greatest possible profit over the first i weeks and the indices run from -1 to n.
- Bellman Equation:
$$s[i] = \max\{s[i-1] + \ell_i, s[i-2] + h_i\} \ ,$$
where $s[-1] = s[0] = 0$.
- The value of an optimal schedule for n weeks is found at $s[n]$.
- Note that if we want to reconstruct the full schedule, we can do so by backtracing the decisions made at each max computation.

(c) Prove that your algorithm in part (c) is correct.

Solution: We prove that $s[n]$ is the profit of an optimal schedule (and that an optimal schedule can be reconstructed by backtracing) by strong induction over the weeks i.

Base case 1: $i = -1$ or 0. Nothing to schedule so optimal value is 0. The optimal schedule is the empty schedule.

Base case 2: $i = 1$. The possible schedules are either $(), (l_1)$, and (h_1). An optimal schedule is either (l_1) or (h_1), with the value of the optimal schedule being the maximum of h_1 and l_1. This agrees with the Bellman equation. Correctness of backtracing here is trivial; the choice of the job for week 1 determines the entire schedule.

Inductive Step: When scheduling week i, we can either schedule (1) a low stress job and combine it with the best schedule for the first $i-1$ weeks or (2) a high stress job combined with the best schedule for the first $i-2$ weeks. By the inductive hypothesis, we have that $s[i-1]$ and $s[i-2]$ are the values of optimal schedules for the first $i-1$ and $i-2$ weeks respectively. By correctness of $s[i-1]$ and $s[i-2]$, we have that the Bellman equation for $s[i]$ takes the max of the two possible options. Thus, the value of the optimal schedule for i weeks is found at $s[i]$.

Correctness of backtraced schedule follows from correctness of the backtraced schedules for $s[i-1]$ and $s[i-2]$. An optimal schedule will involve scheduling the low or high stress job, and copying an optimal solution for the first $i-1$ or $i-2$ weeks respectively. Since backtracing from $s[i-1]$ and $s[i-2]$ yields optimal schedules, the schedule constructed by backtracing from $s[i]$ is also optimal.

2. *Kleinberg, Jon. Algorithm Design (p.315 q.4).*

Suppose you're running a small consulting company. You have clients in New York and clients in San Francisco. Each month you can be physically located in either New York or San Francisco, and the overall operating costs depend on the demands of your clients in a given month.

Given a sequence of n months, determine the work schedule that minimizes the operating costs, knowing that moving between locations from month i to month $i+1$ incurs a fixed moving cost of M. The input consists of two sequences N and S consisting of the operating costs when based in New York and San Francisco, respectively. For month 1, you can start in either city without a moving cost.

(a) Give an example of an instance where it is optimal to move at least 3 times. Explain where and why the optimal must move.

Solution:
$M > 1$
$N := \langle 1, 3M, 1, 3M \rangle$
$S := \langle 3M, 1, 3M, 1 \rangle$
The optimal schedule is to start in NY and move between cities each month. The total cost of this schedule is $4 + 3M$. For any other schedule, you incur an operating cost of $3M$ for some month. If this other schedule involves no moves, the cost is $2 + 6M > 4 + 3M$. If the other schedule involves at least one move, then the cost is at least $3M + M + 3 = 4M + 3 > 4 + 3M$. In this case, a schedule with 3 moves is optimal.

(b) Show that the following algorithm does not correctly solve this problem.

Algorithm: WorkLocSeq

```
Input  : The NY (N) and SF (S) operating costs.
Output: The locations to work the n months
for Each month i do
    if N_i < S_i then
        Output "Month i: NY"
    else
        Output "Month i: SF"
    end
end
```

Solution:
Counter-example:
$NY := \langle 1, 2 \rangle$
$SF := \langle 2, 1 \rangle$
$M := 100$
The above algorithm will start in NY and then move to SF for month 2. The overall cost of this schedule is 102, whereas the optimal schedules are to stay in either NY or SF for both months, incurring an overall cost of 3.

(c) Give an efficient algorithm that takes in the sequences N and S and outputs the value of the optimal solution.

Solution: This is solved with dynamic programming.

- A $2 \times n$-element matrix s, where $s[\{1,2\}][i]$ contains the optimal value over the first i months and being in NY for month i if the first coordinate is 1 and SF if the first coordinate is 2. The second index runs from 1 to n.
- $s[1][1] = N_1$ and $s[2][1] = S_1$.
- Bellman Equations for $i = 2$ to n,
  - For NY in month i:

$$s[1][i] = N_i + \min\{s[1][i-1], s[2][i-1] + M\}$$

  - For SF in month i:

$$s[2][i] = S_i + \min\{s[1][i-1] + M, s[2][i-1]\}$$

- The value of an optimal schedule for the n months is given by $\min_{j \in \{1,2\}} s[j][n]$.

(d) Prove that your algorithm in part (c) is correct.

Solution: We prove that $\min_{j \in \{1,2\}} s[j][n]$ gives the lowest possible cost by induction over the months i.

Base case: $i = 1$. Are two possible schedules are to start in NY or to start in SF. The cost when starting in NY is N_1 and the cost of starting in SF is S_1, which correspond to the definition of $s[1][1]$ and $s[2][1]$. The lowest possible cost corresponds to the minimum of these two values.

Inductive Step: WLOG consider the situation that an optimal schedule is in NY for month i. When scheduling month i in NY, we will have either spent the previous month in NY or in SF. If we spent the previous month in NY, our schedule will include the minimum cost schedule for the first $i-1$ months that ends in NY. The cost of our schedule for the first i months will be the cost of month i in NY plus the cost of our optimal $i-1$-month schedule that ends in NY (given by $s[1][i-1]$ by the inductive hypothesis). Similarly, if we spent the previous month in SF, our schedule will include the minimum cost schedule for the first $i-1$ months that ends in SF. The cost of our schedule for the first i months will be the cost of month i in NY, plus the cost to move from SF to NY, plus the cost of our optimal $i-1$-month schedule that ends in SF (given by $s[2][i-1]$ by the inductive hypothesis). The Bellman equation takes the minimum of the optimal costs for our two options, producing the optimal overall value for $s[1][i]$. An analogous argument holds for SF and $s[2][i]$.

3. *Based on: Kleinberg, Jon. Algorithm Design (p.333, q.26).*

Consider the following inventory planning problem. You are running a company that sells trucks and have good predictions for how many trucks you sell per month. Let d_i denote the number of trucks you expect to sell in month i. If you have unsold trucks any given month, you can store up to s of them in inventory to sell the next month instead. Storage cost is described by a function $c(i, j)$, where i is the month and j is the number of trucks stored. Every month you can buy any number of trucks, and each order has an associated ordering fee $k(i, j)$, that is a function of month i and number of trucks ordered j. Trucks ordered in month i can both be used to fulfill demand in month i, or be stored for future months. You start out with no trucks in storage. The problem is to design an algorithm that decides how many trucks to order each month to satisfy all the demands $\{d_i\}$, and minimize the total cost.

To summarize, every month you pay a storage cost $c(i, j)$, where i is the month and j is the number of trucks stored. Additionally, for each order you place you pay an ordering fee $k(i, j)$, where i is the month and j is the number of trucks ordered.You have to satisfy the demand for trucks d_i each month by either ordering trucks or having trucks in storage. In any month you can store at most s trucks.

(a) Give a recursive algorithm that takes in s, c, k, and the sequence $\{d_i\}$, and outputs the minimum cost. (The algorithm does not need to be efficient.)

Solution: Let $m(i, j)$ be the minimum cost starting from the ith month, given that j trucks were stored in the previous month. Our recursion will look at all possible values j' of trucks to store for the next month, and choose whichever gives the minimum answer. If $j < d_i + j'$, we incur the ordering fee of $k(i, j' - j + d_i)$, while if $j = d_i + j'$, the ordering fee is 0. The case of $j > d_i + j'$ is not possible, as all unsold trucks must be stored.

$$m(i,j) = \min_{\max\{0, j-d_i\} \le j' \le s} \begin{cases} m(i+1, j') + c(i, j') & \text{if } j = d_i + j' \\ m(i+1, j') + c(i, j') + k(i, j' - j + d_i) & \text{if } j < d_i + j' \end{cases}$$

The full solution is given by $m(1, 0)$.

(b) Give an algorithm in time that is polynomial in n and s for the same problem.

Solution: This will be done with dynamic programming.

- We work backwards from the last month, at each step deciding how many trucks to keep in storage for the next month, and using the lowest costs already calculated for future months.
- A 2D matrix m containing $(n \times (s + 1))$-elements, where $m[i][j]$ contains the minimal cost accrued from months i through n given that we stored j trucks from month $i - 1$ to i, and the indices are 1 to n for i and 0 to s for j.
- Initialize $m[n][j] = k(n, d_n - j) + c(n, j)$ for all $j < d_n$, $m[n][j] = c(n, j)$ for $j = d_n$, and, in order to guarantee that in month n we do not have any extra trucks, $m[n][j] = \infty$ for all $j > d_n$.
- Bellman Equations for $i = n - 1$ to 1 and $j = 0$ to s,

$$m[i][j] = c(i, j) + \min_{j' :\ \max\{0, j-d_i\} \le j' \le s} (k(i, j' - j + d_i) + m[i + 1][j'])$$

- The minimum value for the n months is at $m[1][0]$, i.e., month 1 with no trucks stored from month 0 to 1.

The optimal cost is given by $m[1][0]$; the optimal schedule can be determined by backtracing.

(c) Prove that your algorithm in part (b) is correct.

Solution: We prove by reverse induction over the months i that $m[i][j]$ is the lowest possible cost accrued from months i to n given that j trucks are stored from month $i-1$ to month i. Correctness of the backtraced solution follows a similar argument.
Base case 1: $i = n$**.** The equation as defined above calculates the minimum cost for each $0 \leq j \leq s$.
Inductive Step: For each j for $m[i][j]$, by definition of the problem, the storage cost is $c(i, j)$. The minimizer portion of the Bellman equation considers all valid possibilities for the number of trucks j' to store for month $i+1$ given j. Notice that for a j larger than d_i, j' cannot be less than $j - d_i$. If j' is large enough to require an order, the cost is $k(i, j' - j + d_i)$. The last part considered in the minimizer is $m[i+1][j']$ which is the minimal value for the parameters $i+1$ and j' from the induction hypothesis.
Running time: The matrix consists of $(n \cdot (s+1))$ cells and, for each cell, we consider $O(s)$ cells from the previous month. Overall, we have a runtime of $O(n \cdot s^2)$.

4. Alice and Bob are playing another coin game. This time, there are three stacks of n coins: A, B, C. Starting with Alice, each player takes turns taking a coin from the top of a stack – they may choose any nonempty stack, but they must only take the top coin in that stack. The coins have different values. From bottom to top, the coins in stack A have values $a_1, \ldots, a_n$. Similarly, the coins in stack B have values $b_1, \ldots, b_n$, and the coins in stack C have values $c_1, \ldots, c_n$. Both players try to play optimally in order to maximize the total value of their coins.

   (a) Give an algorithm that takes the sequences $a_1, \ldots, a_n$, $b_1, \ldots, b_n$, $c_1, \ldots, c_n$, and outputs the maximum total value of coins that Alice can take. The runtime should be polynomial in n.

Solution: We will use dynamic programming.

We define two 3D DP arrays: AliceOpt$[x][y][z]$ represents the maximum value of remaining coins Alice can get if it is currently Alice's turn, and there are x, y, z coins left in piles A, B, C, respectively. We define BobOpt$[x][y][z]$ similarly, with the only difference being that it is Bob's turn. (In particular, we want BobOpt to still count the value of Alice's coins.)

Bellman Equations: We set the base cases AliceOpt$[0][0][0]$ = BobOpt$[0][0][0] = 0$.

$$\text{AliceOpt}[x][y][z] = \max \begin{cases} a_x + \text{BobOpt}[x-1][y][z] & \text{if } x > 0 \\ b_y + \text{BobOpt}[x][y-1][z] & \text{if } y > 0 \\ c_z + \text{BobOpt}[x][y][z-1] & \text{if } z > 0 \end{cases}$$

$$\text{BobOpt}[x][y][z] = \min \begin{cases} \text{AliceOpt}[x-1][y][z] & \text{if } x > 0 \\ \text{AliceOpt}[x][y-1][z] & \text{if } y > 0 \\ \text{AliceOpt}[x][y][z-1] & \text{if } z > 0 \end{cases}$$

The optimal value for the problem is given by AliceOpt$[n][n][n]$. The algorithm uses $\Theta(n^3)$ time and space, which is polynomial in n.

   (b) Prove the correctness of your algorithm in part (a).

Solution: We prove by strong induction on the triple x, y, z that AliceOpt$[x][y][z]$ correctly describes the maximum value of coins that Alice can gain, given it is Alice's turn, and x, y, z coins remain in piles A, B, C, respectively. We also prove the correctness of BobOpt.

Base Case: When there are 0 coins in each pile, Alice cannot gain any more value, regardless of whose turn it is. So AliceOpt$[0][0][0]$ and BobOpt$[0][0][0]$ are correct.

Inductive Step: We prove the correctness of AliceOpt$[x][y][z]$. Suppose it is Alice's turn and there are x, y, z coins in piles A, B, C. If Alice chooses to take a coin from pile A, she gains a_x value, and now it is Bob's turn and there are $x-1$, y, z coins in piles A, B, C. By the inductive hypothesis, she can gain a maximum of $a_x + \text{BobOpt}[x-1][y][z]$ value in total. We can use similar reasoning for the cases where she takes a coin from pile B or C. Since it is Alice's turn, the option that results in the maximum is chosen.

Now we prove the correctness of BobOpt$[x][y][z]$. If Bob takes a coin from pile A, Alice gains no value, and now it is Alice's turn and there are $x-1$, y, z coins in piles A, B, C. By our inductive hypothesis, Alice will gain AliceOpt$[x-1][y][z]$ value. We can once again use similar reasoning for the other piles. Since it is Bob's turn, the option that results in the minimum (for Alice's value) is chosen.

5. **Coding Question: WIS**

Implement the optimal algorithm for Weighted Interval Scheduling (for a definition of the problem, see the slides on Canvas) in either C, C++, C#, Java, Python, or Rust. Be efficient and implement it in $O(n^2)$ time, where n is the number of jobs. We saw this problem previously in HW3 Q2a, where we saw that there was no optimal greedy heuristic.

The input will start with an positive integer, giving the number of instances that follow. For each instance, there will be a positive integer, giving the number of jobs. For each job, there will be a trio of positive integers i, j and k, where $i < j$, and i is the start time, j is the end time, and k is the weight.

Input constraints:

- $1 \leq n \leq 2,000$
- $1 \leq i \leq 10,000,000$
- $1 \leq j \leq 10,000,000$
- $1 \leq k \leq 50,000,000$

A sample input is the following:

```
2
1
1 4 5
3
1 2 1
3 4 2
2 6 4
```

The sample input has two instances. The first instance has one job to schedule with a start time of 1, an end time of 4, and a weight of 5. The second instance has 3 jobs.

The objective of the problem is to determine a schedule of non-overlapping intervals with maximum weight and to return this maximum weight. For each instance, your program should output the total weight of the intervals scheduled on a separate line. Each output line should be terminated by exactly one newline. The correct output to the sample input would be:

```
5
5
```

or, written with more explicit whitespace,

```
"5\n5\n"
```

Notes:

- Endpoints are exclusive, so it is okay to include a job ending at time t and a job starting at time t in the same schedule.
- In the third set of tests, some outputs will cause overflow on 32-bit signed integers.

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

More Dynamic Programming

Do **NOT** write pseudocode when describing your dynamic programs. Rather give the Bellman Equation, describe the matrix, its axis and how to derive the desired solution from it.

1. *Kleinberg, Jon. Algorithm Design (p. 327, q. 16).*

   In a hierarchical organization, each person (except the ranking officer) reports to a unique superior officer. The reporting hierarchy can be described by a tree T, rooted at the ranking officer, in which each other node v has a parent node u equal to his or her superior officer. Conversely, we will call v a direct subordinate of u.

   Consider the following method of spreading news through the organization.

   - The ranking officer first calls each of her direct subordinates, one at a time.
   - As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates, one at a time.
   - The process continues this way until everyone has been notified.

   Note that each person in this process can only call *direct* subordinates on the phone.

   We can picture this process as being divided into rounds. In one round, each person who has already heard the news can call one of his or her direct subordinates on the phone. The number of rounds it takes for everyone to be notified depends on the sequence in which each person calls their direct subordinates.

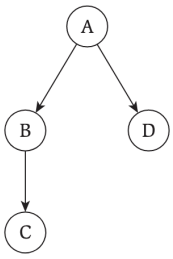


Figure 1: A hierarchy with four people. The fastest broadcast scheme is for A to call B in the first round. In the second round, A calls D and B calls C. If A were to call D first, then C could not learn the news until the third round.

Give an efficient algorithm that determines the minimum number of rounds needed for everyone to be notified, and outputs a sequence of phone calls that achieves this minimum number of rounds by answering the following:

(a) Give a recursive algorithm. (The algorithm does not need to be efficient)

Solution:

We will represent schedule as a list of lists S that contains the calls done at round i at $S[i]$.

Algorithm 1: CallSchedule

Input : An officer r with subordinates $[s_1, \ldots, s_n]$
Output: (Number of rounds required, schedule of call)
if $n = 0$ **then**
  return $(0, [])$;
end
$L \leftarrow []$;
foreach s_i **do**
  $L.append(\texttt{CallSchedule}(s_i))$
end
Sort L by the first entry;
$N \leftarrow \max(L[i][1] + i)$;
$S \leftarrow$ schedule: at round 1, s_1; at round 2, s_2 and round 1 of of s_1's schedule, ...;
return (N, S);

(b) Give an efficient dynamic programming algorithm.

Solution: We will solve this with dynamic programming.

Let $R(v)$ denote the minimum number of rounds needed to contact all nodes in the subtree rooted at v. Let $S_v[1 \ldots n]$ denote the list of direct subordinates i of v ordered by decreasing value of $R(i)$. Then, the Bellman equation can be given by:

$$R(v) = \max_{i=1,\ldots,n} \Big(i + R(S_v[i]) \Big) \tag{1}$$

Then, the solution to the problem is R(ranking officer).

To retrieve the schedule, we can think of S_v memoized as a tree. We can keep a stack of nodes such that its children are not exhausted. We initialize the stack with the top officer. At each round we pop off the all the nodes in the list. For all each of those nodes v, we add the next children c in S_v not yet considered to the schedule at that round. And we add c to the stack. And if there are still more children left for v, we add it back to the stack. We repeat until the stack is empty.

Let m denote the number of nodes in the tree. A naïve analysis of runtime can give $O(m^2 \log m)$, since there are m entries, and we sort each time taking $O(m \log m)$. A more careful analysis can yield $O(m \log m)$. Note that the work we actually do to compute each $R(v)$ is $O(n_v \log n_v)$ if n_v denotes the number of children of v. Since each node has a unique parent, the total work is $\sum_v O(n_v \log n_v)$ with $\sum_v n_v \approx m$. Note that for any $a, b > 0$, we have $a \log a + b \log b \leq a \log(a+b) + b \log(a+b) \leq (a+b) \log(a+b)$. So, the total work done is $O(m \log m)$ for computing R. The backtracing step takes linear time because the number of times each node gets added to the stack is equal to the number of children it has. In other words, each node adds its parent to the stack at most once, so it takes linear time with respect to m. Therefore in total, the runtime is $O(m \log m)$.

(c) Prove that the algorithm in part (b) is correct.

Solution:

The correctness follows from the correctness of the Bellman equation. We can prove the correctness by strong induction. Suppose we are considering the sequence of the calls for a node v. Given any sequence of calls by v, say $(u_1, ..., u_n)$, the minimal possible rounds needed for the sequence is at least each of $i + R(u_i)$, since u_i needs to make calls for its subordinates after it gets called by v at the ith round. Therefore, $R(v) = \min_{(u_1,...,u_n)} \max_{i=1,...,n} i + R(u_i)$.

Now, suppose $U = (u_1, ..., u_n)$ is not sorted by the decreasing R values. That means there are some indices $i < j$ such that $R(u_i) < R(u_j)$ and also $i + R(u_i) < j + R(u_j)$. Consider a new sequence $W = (w_1, ..., w_n)$ that is same as U except we swap the u_i and u_j. Then, $R(w_i) + i = R(u_j) + i < R(u_j) + j$ and $R(w_j) + j = R(u_i) + j < R(u_j) + j$. Therefore, removing an inversion in the sequence U does not increase the maximum value achieved. By the exchange argument, we can conclude that the minimal value is achieved when the sequence is sorted by the decreasing R value.

2. Consider the following problem: you are provided with a two dimensional matrix M (dimensions, say, $m \times n$). Each entry of the matrix is either a **1** or a **0**. You are tasked with finding the total number of square sub-matrices of M with all **1**s. Give an $O(mn)$ algorithm to arrive at this total count by answering the following:

   (a) Give a recursive algorithm. (The algorithm does not need to be efficient)

Solution:

Algorithm 2: SqSub

Input : $m \times n$ binary matrix M
Output: Number of square submatrix of M with all **1**
if $m = 0$ **then**
  return 0;
end
$N \leftarrow$ `SqSub(`$M[2 \dots m][1 \dots n]$`)`;
foreach $i \in [1 \dots n]$ **do**
  $k \leftarrow$ number of square submatrix with all **1** having $M[1, i]$ as its upper left corner;
  $N \leftarrow N + k$;
end
return N;

   (b) Give an efficient dynamic programming algorithm.

Solution: This is solved with dynamic programming.

Let $C(i, j)$ denote the side length of the biggest square submatrix with **1**s with bottom-right corner at $M[i][j]$. Note that this number $C(i, j)$ also counts the number of square submatrix with **1**s with bottom-right corner at $M[i][j]$.

$$C(i,j) = \begin{cases} M[i][j] & \text{if } i = 0 \text{ or } j = 0 \\ 0 & \text{if } M[i][j] = 0 \\ \min(C(i-1,j), C(i,j-1), C(i-1,j-1)) + 1 & \text{if } M[i][j] = 1 \end{cases}$$

We can solve this recurrence relation directly using a recursive algorithm. The solution is then $\sum_{i,j} C(i, j)$.
We are essentially filling up a table of size $m \times n$ with constant amount of work for each entry. Therefore, the runtime is $O(mn)$.

(c) Prove that the algorithm in part (b) is correct.

Solution:

The recurrence relation given above is correct because the largest square **1** matrix with $M[i][j]$ at the bottom-right corner is constrained by the largest such matrices at $M[i-1][j]$, $M[i][j-1]$, and $M[i-1][j-1]$. More formally, suppose length ℓ square is the largest **1** square matrix we can fit with bottom-right corner at $M[i][j]$. This square also includes three length $\ell - 1$ **1** squares with bottom-right corner at $M[i-1][j]$, $M[i][j-1]$, and $M[i-1][j-1]$. Therefore, $C(i,j) \leq C(i-1,j)+1$, $C(i,j) \leq C(i,j-1)+1$, and $C(i,j) \leq C(i-1,j-1)+1$, so $C(i,j) \leq \min(C(i-1,j), C(i,j-1), C(i-1,j-1))+1$. For the other direction, we can see one of the above three inequalities must be actually equality, since if not, we would be able to fit a length ℓ squares at each of those three corners, giving us a length $\ell+1$ square fitting at the corner $M[i][j]$. That is equivalent to taking the minimum of the three C values.

(d) Furthermore, how would you count the total number of square sub-matrices of M with all **0**s?

Solution:

We just need to convert M to M' that has opposite entries of M and use our algorithm on M'.

3. *Kleinberg, Jon. Algorithm Design (p. 329, q. 19).*

String x' is a *repetition* of x if it is a prefix of x^k (k copies of x concatenated together) for some integer k. So $x' = 10110110110$ is a repetition of $x = 101$. We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences x' and y', so that x' is a repetition of x and y' is a repetition of y. For example, if $x = 101$ and $y = 00$, then $s = 100010010$ is an interleaving of x and y, since characters $1, 2, 5, 8, 9$ form 10110—a repetition of x—and the remaining characters $3, 4, 6, 7$ form 0000—a repetition of y.

Give an efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y by answering the following:

(a) Give a recursive algorithm. (The algorithm does not need to be efficient)

Solution:

Assume we have long enough $X = x^k$ and $Y = y^k$ for some big enough k.

Algorithm 3: Interleave

Input : Strings s, X, and Y
Output: Whether s is an interleaving of X and Y or not
if $length(s) = 0$ **then**
  return True;
end
if $s[0] \neq X[0] \wedge s[0] \neq Y[0]$ **then**
  return False;
end
$b_X, b_Y \leftarrow$ False;
if $s[0] = X[0]$ **then**
  $b_X \leftarrow$ `Interleave(`$s[1\ldots], X[1\ldots], Y$`)`;
end
if $s[0] = Y[0]$ **then**
  $b_Y \leftarrow$ `Interleave(`$s[1\ldots], X, Y[1\ldots]$`)`;
end
return $b_X \vee b_Y$;

(b) Give an efficient dynamic programming algorithm.

Solution: This will be done with dynamic programming.

Let x^* and y^* be the infinite repetition of x and y respectively. Let $S(i, j)$ denote whether the substring $s_1 s_2 \ldots s_{i+j}$ is an interleaving of $x_1^* \ldots x_i^*$ and $y_1^* \ldots y_j^*$.

$$S(i,j) = \Big[S(i-1,j) \wedge \big(s_{i+j} == x_i^*\big)\Big] \vee \Big[S(i,j-1) \wedge \big(s_{i+j} == y_j^*\big)\Big] \tag{2}$$

with base case $S(0,0) = 1$, $S(i,0) = (s_i == x_i^*)$ and $S(0,j) = (s_j == y_j^*)$. The output will be true if there is some $i + j = \ell$ with $S(i,j)$ true. Note that we only need finite initial segment of x^* and y^*. Also, note that $x_i^* = x_{i \mod \ell}$ where ℓ is the length of x, so it can be computed in constant time. Therefore, it takes $O(n^2)$ time since i and j are in the range 0 to n where n is the length of s.

(c) Prove that the algorithm in part (b) is correct.

Solution:

The base cases are clearly true. Assume that $S(i, j-1)$ and $S(i-1, j)$ return the correct result. The string $s_1 \dots s_{i+j}$ can be interleaved with $x_1^* \dots x_i^*$ and $y_1^* \dots y_j^*$ if and only if the last character, s_{i+j} matches with one of x_i^* or y_j^*, and the substring $s_1 \dots s_{i+j-1}$ can be interleaved with the remaining of x^* and y^*. The recurrence relation expresses this statement. Also, we check for all possible repetitions of x and y.

4. *Kleinberg, Jon. Algorithm Design (p. 330, q. 22).*

To assess how "well-connected" two nodes in a directed graph are, one can not only look at the length of the shortest path between them, but can also count the number of shortest paths.

This turns out to be a problem that can be solved efficiently, subject to some restrictions on the edge costs. Suppose we are given a directed graph $G = (V, E)$, with costs on the edges; the costs may be positive or negative, but every cycle in the graph has strictly positive cost. We are also given two nodes $v, w \in V$.

Give an efficient algorithm that computes the number of shortest $v - w$ paths in G. (The algorithm should not list all the paths; just the number suffices.)

Solution: We will solve this with dynamic programming.

We keep track of two $2D$ matrices C and M such that $C[n][i]$ the cost of the shortest path from i to w of length exactly n, and $M[n][i]$ is the number of such paths. We initialize $C[1][i] = c_{iw}$ and $M[1][i] = 1$ if there is an edge $(i, w) \in E$ and $C[1][i] = \infty$ and $M[1][i] = 0$ otherwise.

We update the entries of M and C as follows: Let

$$C[n][i] = \min_{j \in V} \{c_{ij} + C[n-1][j]\}$$

and

$$M[n][i] = \sum_j M[n-1][j] \quad \text{for } j \in V \text{ with } c_{ij} + C[n-1][j] = C[n][i]$$

To find the solution, we first compute the cost of the shortest path from v to w, which is $c = \min_{1 \le n \le |V|-1} C[n][v]$. Then, for each j with $C[j][v] = c$, we return $m = \sum_j M[j][v]$.

We show correctness. Fix some n. Suppose c is the cost of the shortest path from i to w of length n. And suppose m is the number of such paths. Then, if $p_1, p_2, \dots, p_m$ are those paths, the cost of each one of them are c and they all have length n. We can partition them disjointly by the first vertex visited after i. Suppose without loss of generality that $p_1, \dots, p_k$ start with the edge (i, j). If $p'_1, \dots, p'_k$ denote the paths obtained by removing the first edge, then they must be the shortest path from j to w of length $n-1$. Then, by an inductive argument, $k = M[n-1][j]$ and the cost of each of the must be $C[n-1][j] = c - c_{ij}$. Also, since there is no negative cycle, all shortest paths must have length less than n, so the algorithm counts them all.

5. The following is an instance of the Knapsack Problem. Before implementing the algorithm in code, run through the algorithm by hand on this instance. To answer this question, generate the table, indicate the maximum value, and recreate the subset of items.

item	weight	value
1	4	5
2	3	3
3	1	12
4	2	4

Capacity: 6

Solution:

4	0	12	12	16	16	17	19
3	0	12	12	12	15	17	17
2	0	0	0	3	5	5	5
1	0	0	0	0	5	5	5
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Max value: 19

Items used: 2, 3, 4

6. **Coding Question: Knapsack**

   Implement the algorithm for the Knapsack Problem in either C, C++, C#, Java, Python, or Rust. Be efficient and implement it in $O(nW)$ time, where n is the number of items and W is the capacity.

   The input will start with an positive integer, giving the number of instances that follow. For each instance, there will two nonnegative integers, representing the number of items and the capacity, followed by a list describing the items. For each item, there will be two nonnegative integers, representing the weight and value, respectively.

   A sample input is the following:

```
2
1 3
4 100
3 4
1 2
3 3
2 4
```

   The sample input has two instances. The first instance has one item and a capacity of 3. The item has weight 4 and value 100. The second instance has three items and a capacity of 4.

   For each instance, your program should output the maximum possible value. The correct output to the sample input would be:

```
0
6
```

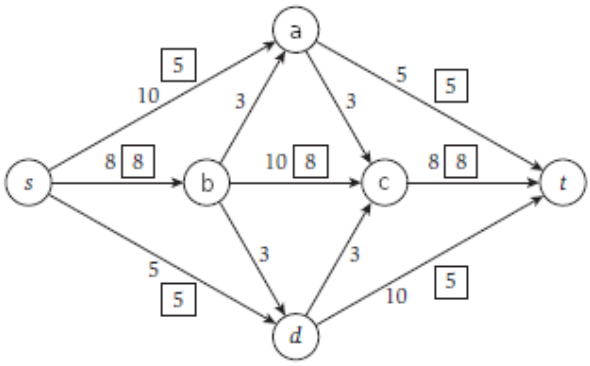
Assignment 9 – Network Flow

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

Network Flow

1. *Kleinberg, Jon. Algorithm Design (p. 415, q. 3a)* The figure below shows a flow network on which an $s-t$ flow has been computed. The capacity of each edge appears as a label next to the edge, and the flow is shown in boxes next to each edge. An edge with no box has no flow being sent down it.

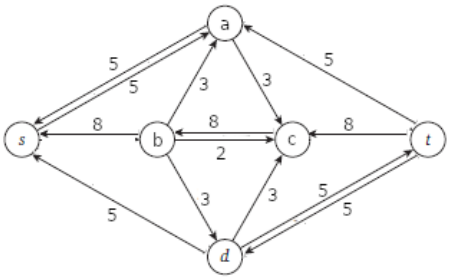


(a) What is the value of this flow?

Solution: 18

(b) Please draw the **residual graph** associated with this flow.

Solution:



(c) Is this a maximum $s-t$ flow in this graph? If not, describe an augmenting path that would increase the total flow.

Solution: No. We can add 3 flow along $s \to a \to c \to b \to d \to t$

2. *Kleinberg, Jon. Algorithm Design (p. 419, q. 10)* Suppose you are given a directed graph $G = (V, E)$. This graph has a positive integer capacity c_e on each edge, a source $s \in V$, a sink $t \in V$. You are also given a maximum $s - t$ flow through G: f. You know that this flow is *acyclic* (no cycles with positive flow all the way around the cycle), and every flow $f_e \in f$ has an integer value.

   Now suppose we pick an edge e^* and reduce its capacity by 1 unit. Show how to find a maximum flow in the resulting graph G^* in time $O(m + n)$, where $n = |V|$ and $m = |E|$.

Solution: First, generate the residual graph G_f in $O(m+n)$ time. Since all capacities are integers, if e^* has nonzero capacity in G_f, then reducing its capacity by 1 will still allow us to retain the maximum flow f.

Suppose e^* has no remaining capacity in G_f. It must have been part of some augmenting path $s \to t$. We can find such an augmenting path using BFS twice in G_f. Once to find a path from the source of e^* back to s, and once to find a path from t to the destination of e^*. (BFS runs in $O(m+n)$ time.) Reduce the flow through each edge by 1, increasing the capacities of the back edges to match. The result is a valid flow f^* in G^* with 1 less total flow.

Either f^* is a maximum flow in G^*, or we can find the maximum flow using one more augmenting path found in $O(m + n)$ time with BFS.

3. *Kleinberg, Jon. Algorithm Design (p. 420, q. 11)* Your friends have written a very fast piece of maximum-flow code based on repeatedly finding augmenting paths. However, after you've looked at a bit of output from it, you realize that it's not always finding a flow of *maximum* value. The bug turns out to be pretty easy to find; your friends hadn't really gotten into the whole backward-edge thing when writing the code, and so their implementation builds a variant of the residual graph that only *includes the forward edges*. In other words, it searches for s-t paths in a graph $\tilde{G}_f$ consisting only of edges e for which $f(e) < c_e$, and it terminates when there is no augmenting path consisting entirely of such edges. We'll call this the Forward-Edge-Only Algorithm. (Note that we do not try to prescribe how this algorithm chooses its forward-edge paths; it may choose them in any fashion it wants, provided that it terminates only when there are no forward-edge paths.)

   It's hard to convince your friends they need to reimplement the code. In addition to its blazing speed, they claim, in fact, that it never returns a flow whose value is less than a fixed fraction of optimal. Do you believe this? The crux of their claim can be made precise in the following statement.

   There is an absolute constant $b > 1$ (independent of the particular input flow network), so that on every instance of the Maximum-Flow Problem, the Forward-Edge-Only Algorithm is guaranteed to find a flow of value at least $1/b$ times the maximum-flow value (regardless of how it chooses its forward-edge paths). Decide whether you think this statement is true or false, and give a proof of either the statement or its negation.

   Solution: No. Imagine a graph G set up with a source node on the left, a sink node on the right, and two columns of nodes in the middle $a_1 \dots a_n$ and $b_1 \dots b_n$. There is an edge with capacity 1 from the source, to each node a_i, from each node a_i to its matching b_i, and from each b_i to the sink. Thus, there is a trivial flow of n available in this graph.

   Now imagine a graph G' which is identical to G except it also has an edge from each b_i to a_{i+1}. (Note: b_n does not get a new edge, nor does a_1.) The same flow from before is still available, so the maximum flow is at least n. Yet if my first augmenting path runs $s \to a_1 \to b_1 \to a_2 \to b_2 \dots a_n \to b_n t$, my friend's algorithm will halt immediately after completing this 1-flow path.

   No matter what n I might pick, there is a graph with $2n+2$ nodes where it is possible for my friend's algorithm to produce flow only $1/n$ times the maximum possible.

4. *Kleinberg, Jon. Algorithm Design (p. 418, q. 8)* Consider this problem faced by a hospital that is trying to evaluate whether its blood supply is sufficient:

   In a (simplified) model, the patients each have blood of one of four types: A, B, AB, or O. Blood type A has the A antigen, type B has the B antigen, AB has both, and O has neither. Patients with blood type A can receive either A or O blood. Likewise patients with type B can receive either B or O type blood. Patients with type O can only receive type O blood, and patients with type AB can receive any of the four types.

   (a) Let integers s_O, s_A, s_B, s_{AB} denote the hospital's current available blood supply, and let integers d_A, d_B, d_O, d_{AB} denote their projected demand for the coming week. Give a polynomial time algorithm to evaluate whether the blood supply is enough to cover the projected need.

   Solution: This is a bipartite matching problem.

   Create a graph G with a node for each type of blood supply, and a node for each type of blood demand. Draw an edge with unlimited capacity between each compatible supply-demand pair. (For example, s_A has outgoing edges to d_A and d_{AB}.) Then add a source node s with an edge to each supply node whose capacity is the amount of supply for that type of blood. Create a sink node t with an edge from each demand node whose capacity is the amount of demand for that type of blood.

   The supply is sufficient for the projected demand if and only if the maximum flow through G is equal to the sum of the demand$\rightarrow t$ edge capacities.

   To show that this algorithm takes polynomial time, observe that there are 10 nodes in this graph: s, t, the 4 supply nodes, and the 4 demand nodes. Thus, both $|V| = O(1)$ and $|E| = O(1)$. The Ford-Fulkerson max-flow method runs in $O(|E||f^*|)$ time (with $|E|$ being the number of edges in the graph and $|f^*|$ being the value of the max flow) in settings with integer capacities. Since the max-flow equals the min-cut, we know that the max-flow is $O(d_A + d_B + d_O + d_{AB})$ (a cut of the graph). Thus, our algorithm runs in polynomial time (linear) with respect to the sum of the demands (and the supplies by similar logic). Additionally, we could use a max-flow algorithm such as Edmonds-Karp or Orlin's whose runtimes only rely on the number of edges and vertices in the graph to get an $O(1)$ solution.

   (b) Network flow is one of the most powerful and versatile tools in the algorithms toolbox, but it can be difficult to explain to people who don't know algorithms. Consider the following instance. Show that the supply is **insufficient** in this case, and provide an explanation for this fact that would be understandable to a non-computer scientist. (For example: to a hospital administrator.) Your explanation should not involve the words *flow*, *cut*, or *graph*.

blood type	supply	demand
O	50	45
A	36	42
B	11	8
AB	8	3

   Solution: At least one patient of blood type O or A will be unable to receive blood according to this projection.

   Patients of these blood types cannot receive blood from type B or type AB suppliers. There are 86 units of blood in the combined O+A supply, and there is demand for 87 units between those two types.

   Extra: Properly applied, only one patient will go without: Supply patients of each blood type from the blood supplies of the matching type. This leaves only type A patients, and we can reroute the remaining 5 units of O type blood to 5 of the 6 remaining A type patients.

5. Implement the Ford-Fulkerson method for finding maximum flow in graphs with only integer edge capacities, in either C, C++, C#, Java, or Python. Be efficient and implement it in $O(mF)$ time, where m is the number of edges in the graph and F is the value of the maximum flow in the graph. We suggest using BFS or DFS to find augmenting paths. (You may be able to do better than this.)

   The input will start with a positive integer, giving the number of instances that follow. For each instance, there will be two positive integers, indicating the number of nodes $n = |V|$ in the graph and the number of edges $m = |E|$ in the graph. Following this, there will be $|E|$ additional lines describing the edges. Each edge line consists of a number indicating the source node, a number indicating the destination node, and a capacity $c(e)$. The nodes are not listed separately, but are numbered $\{1 \dots n\}$.

   Your program should compute the maximum flow value from node 1 to node n in each given graph.

   Constraints:

   - $2 \leq n \leq 100$
   - $1 \leq m \leq \frac{n(n-1)}{2}$. For $n = 100$, $m \leq 4,950$.
   - $0 \leq c(e) \leq 100$

   A sample input is the following:

```
2
3 2
2 3 4
1 2 5
6 9
1 2 9
1 3 4
2 4 1
2 5 6
3 4 4
3 5 5
4 6 8
5 6 5
5 6 3
```

   The sample input has two instances. For each instance, your program should output the maximum flow on a separate line. Each output line should be terminated by a newline. The correct output for the sample input would be:

```
4
11
```

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

More Network Flow

1. *Kleinberg, Jon. Algorithm Design (p.416 q.6).* Suppose you're a consultant for the Ergonomic Architecture Commission, and they come to you with the following problem.

   They're really concerned about designing houses that are "user-friendly", and they've been having a lot of trouble with the setup of light fixtures and switches in newly designed houses. Consider, for example, a one-floor house with n light fixtures and n locations for light switches mounted in the wall. You'd like to be able to wire up one switch to control each light fixture, in such a way that a person at the switch can see the light fixture being controlled.

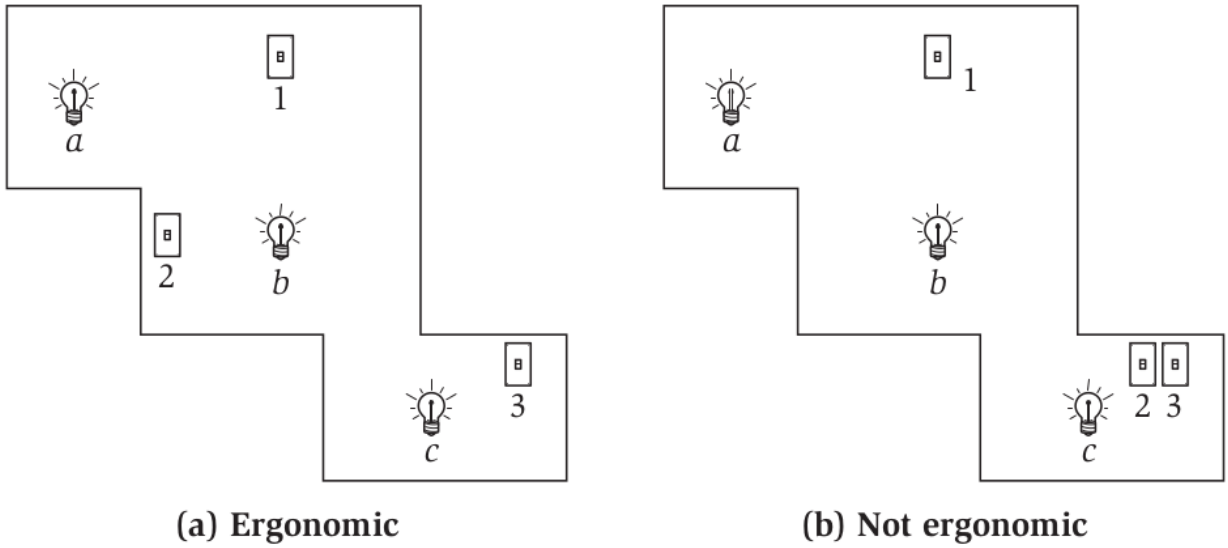


Figure 1: The floor plan in (a) is ergonomic, because we can wire switches to fixtures in such a way that each fixture is visible from the switch that controls it. (This can be done by wiring switch 1 to a, switch 2 to b, and switch 3 to c.) The floor plan in (b) is not ergonomic, because no such wiring is possible.

Sometimes this is possible and sometimes it isn't. Consider the two simple floor plans for houses in Figure 1. There are three light fixtures (labelled a, b, c) and three switches (labelled 1, 2, 3). It is possible to wire switches to fixtures in Figure 1(a) so that every switch has a line of sight to the fixture, but this is not possible in Figure 1(b).

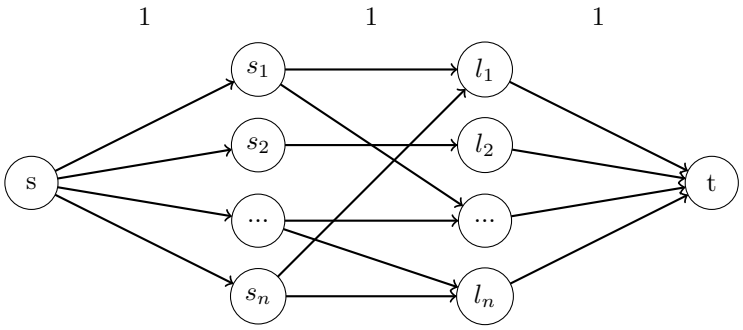
Let's call a floor plan, together with n light fixture locations and n switch locations, ergonomic if it's possible to wire one switch to each fixture so that every fixture is visible from the switch that controls it. A floor plan will be represented by a set of m horizontal or vertical line segments in the plane (the walls), where the i-th wall has endpoints $(x_i, y_i), (x_i', y_i')$. Each of the n switches and each of the n fixtures is given by its coordinates in the plane. A fixture is visible from a switch if the line segment joining them does not cross any of the walls.

Give an algorithm to decide if a given floor plan is ergonomic. The running time should be polynomial in m and n. You may assume that you have a subroutine with $O(1)$ running time that takes two line segments as input and decides whether or not they cross in the plane.

Solution:
Algorithm:

1. Build a bipartite graph with switches on one side and lights on the other.
   - For each switch i, construct a vertex s_i.
   - For each light j, construct a vertex l_j.
   - Add an edge (s_i, l_j) with capacity 1 if the line segment between the coordinates of switch i and light j does not intersect any of the m walls.
   - Add a source vertex s and edges (s, s_i) with capacity 1 for all switch i.
   - Add a target vertex t and edges (l_j, t) with capacity 1 for all lights j.
   - Overall this will take $O(n^2 m)$ time, i.e., check each n^2 pairs against each wall segment.
2. Run the bipartite matching max-flow algorithm: $O(n^3)$.
3. If we have a perfect matching (max-flow is n), then the floor plan is ergonomic.

Overall, this algorithm has a run-time of $O(n^2 * max\{m, n\})$.



2. *Kleinberg, Jon. Algorithm Design (p.426 q.20).*

Your friends are involved in a large-scale atmospheric science experiment. They need to get good measurements on a set S of n different conditions in the atmosphere (such as the ozone level at various places), and they have a set of m balloons that they plan to send up to make these measurements. Each balloon can make at most two measurements. Unfortunately, not all balloons are capable of measuring all conditions, so for each balloon $i = 1, \ldots, m$, they have a set S_i of conditions that balloon i can measure. Finally, to make the results more reliable, they plan to take each measurement from at least k different balloons. (Note that a single balloon should not measure the same condition twice.) They are having trouble figuring out which conditions to measure on which balloon.

Example. Suppose that $k = 2$, there are $n = 4$ conditions labelled c_1, c_2, c_3, c_4, and there are $m = 4$ balloons that can measure conditions, subject to the limitation that $S_1 = S_2 = c_1, c_2, c_3$, and $S_3 = S_4 = c_1, c_3, c_4$. Then one possible way to make sure that each condition is measured at least $k = 2$ times is to have

- balloon 1 measure conditions c_1, c_2,
- balloon 2 measure conditions c_2, c_3,
- balloon 3 measure conditions c_3, c_4, and
- balloon 4 measure conditions c_1, c_4.

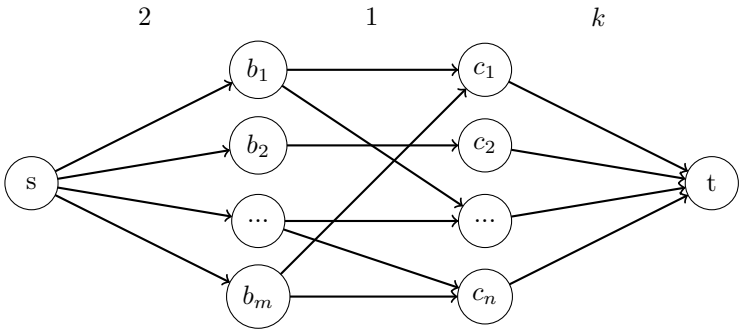
(a) Give a polynomial-time algorithm that takes the input to an instance of this problem (the n conditions, the sets S_i for each of the m balloons, and the parameter k) and decides whether there is a way to measure each condition by k different balloons, while each balloon only measures at most two conditions.

Solution:

- b_1 to b_m are the balloons.
- c_1 to c_n are the conditions to measure.

Solution 1:

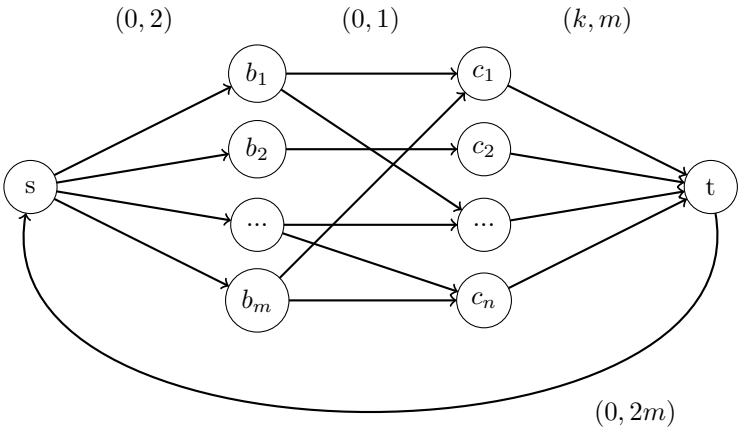
- An edge from b_i to c_j if b_i can measure c_j with capacity 1.
- A node s with edges to each balloon with capacity 2.
- A node t with edges from each condition with capacity k.
- All conditions can be measured if the max-flow is $|S| * k$ (number of conditions * number of balloons that must measure each condition)



Solution:

Solution 2:

- An edge from b_i to c_j if b_i can measure c_j with capacity $(0, 1)$.
- A node s with edges to each balloon with capacity $(0, 2)$.
- A node t with edges from each condition with capacity (k, m).
- Edge (t, s) with capacity $(0, 2m)$.
- The demand of every node is 0.
- All conditions can be measured if all the flow constraints are met



(b) You show your friends a solution computed by your algorithm from (a), and to your surprise they reply, "This won't do at all—one of the conditions is only being measured by balloons from a single subcontractor." You hadn't heard anything about subcontractors before; it turns out there's an extra wrinkle they forgot to mention...

Each of the balloons is produced by one of three different subcontractors involved in the experiment. A requirement of the experiment is that there be no condition for which all k measurements come from balloons produced by a single subcontractor.

Explain how to modify your polynomial-time algorithm for part (a) into a new algorithm that decides whether there exists a solution satisfying all the conditions from (a), plus the new requirement about subcontractors.

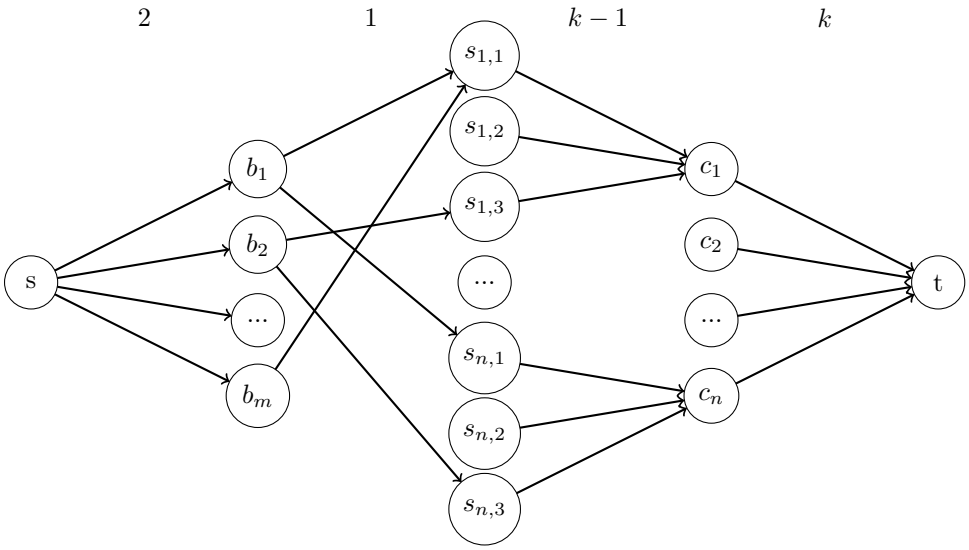
Solution:

- Remove the edges between balloons and conditions.

Solution 1:

- For each condition, add a node for each of the subcontractors with a node from the contractor to the condition with capacity $k-1$. This will add $3n$ nodes.
- Add edges from b_i to its subcontractor with capacity 1 associated with each condition that b_i can measure.
- Again, all conditions can be measured if the max-flow is $|S| * k$

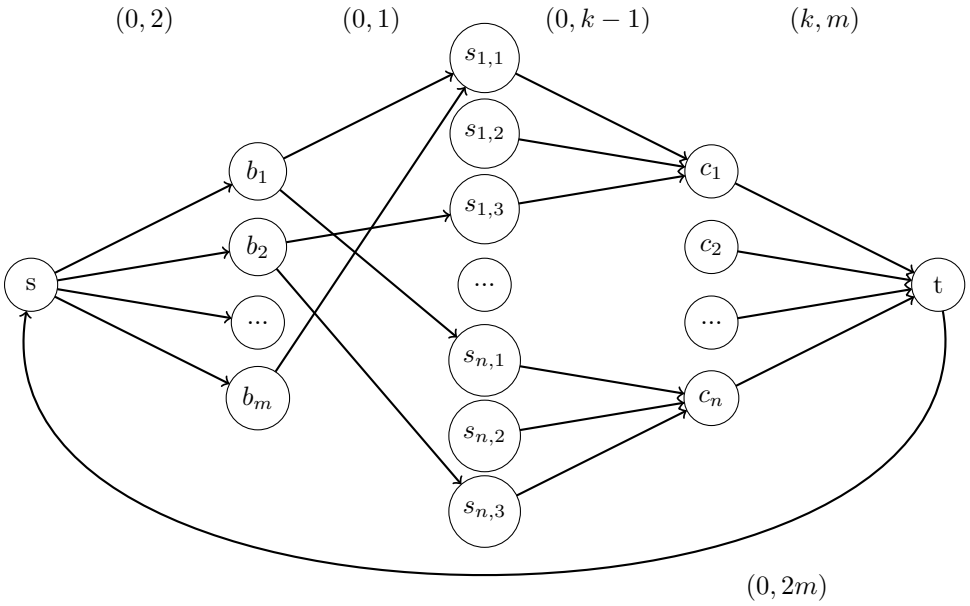
With the edge from subcontractor to condition with max capacity of $k-1$, no condition will be exclusively measured by balloons from just that subcontractor.



Solution:

Solution 2:

- For each condition, add a node for each of the subcontractors with a node from the contractor to the condition with capacity $(0, k-1)$. This will add $3n$ nodes.
- Add edges from b_i to its subcontractor with capacity $(0, 1)$ associated with each condition that b_i can measure.
- All conditions can be measured if all the flow constraints are met



3. *Kleinberg, Jon. Algorithm Design (p.442, q.41).*

Suppose you're managing a collection of k processors and must schedule a sequence of m jobs over n time steps.

The jobs have the following characteristics. Each job j has an arrival time a_j when it is first available for processing, a length ℓ_j which indicates how much processing time it needs, and a deadline d_j by which it must be finished. (We'll assume $0 < \ell_j \leq d_j - a_j$.) Each job can be run on any of the processors, but only on one at a time; it can also be preempted and resumed from where it left off (possibly after a delay) on another processor.

Moreover, the collection of processors is not entirely static either: You have an overall pool of k possible processors; but for each processor i, there is an interval of time $[t_i, t'_i]$ during which it is available; it is unavailable at all other times.

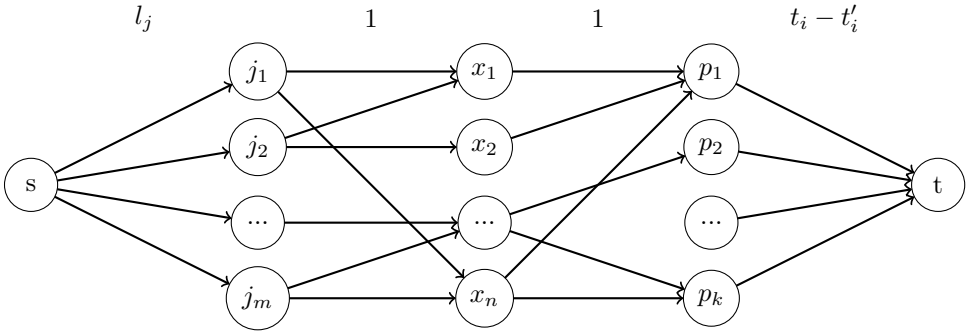
Given all this data about job requirements and processor availability, you'd like to decide whether the jobs can all be completed or not. Give a polynomial-time (in k, m, and n) algorithm that either produces a schedule completing all jobs by their deadlines or reports (correctly) that no such schedule exists. You may assume that all the parameters associated with the problem are integers.

Example. Suppose we have two jobs J_1 and J_2. J_1 arrives at time 0, is due at time 4, and has length 3. J_2 arrives at time 1, is due at time 3, and has length 2. We also have two processors P_1 and P_2. P_1 is available between times 0 and 4; P_2 is available between times 2 and 3. In this case, there is a schedule that gets both jobs done.

- At time 0, we start job J_1 on processor P_1.
- At time 1, we preempt J_1 to start J_2 on P_1.
- At time 2, we resume J_1 on P_2. (J_2 continues processing on P_1.)
- At time 3, J_2 completes by its deadline. P_2 ceases to be available, so we move J_1 back to P_1 to finish its remaining one unit of processing there.
- At time 4, J_1 completes its processing on P_1. Notice that there is no solution that does not involve preemption and moving of jobs.

Solution:

- Each job will have a node with an edge from source node, s, with capacity ℓ_j.
- Each time step x will have a node with edge of capacity 1 from job j if j is available at time x.
- Each processor p will have a node with an edge of capacity 1 from time x if p is available at time x.
- A sink node t with an edge from p with capacity $t_i - t'_i$.
- A schedule exists if max-flow is $\sum_{j=1}^{m} l_j$



4. *Kleinberg, Jon. Algorithm Design (p.444, q.45).*

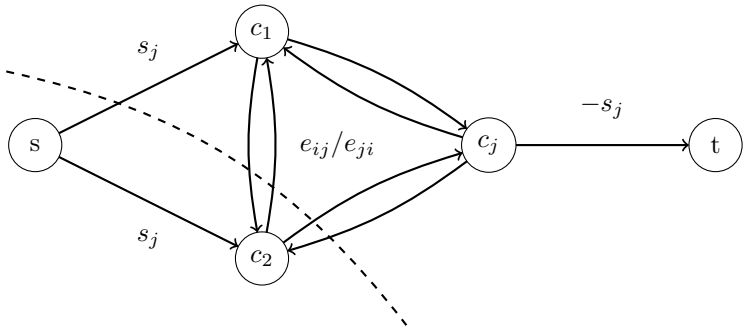
Consider the following definition. We are given a set of n countries that are engaged in trade with one another. For each country i, we have the value s_i of its budget surplus; this number may be positive or negative, with a negative number indicating a deficit. For each pair of countries i, j, we have the total value e_{ij} of all exports from i to j; this number is always nonnegative. We say that a subset S of the countries is *free-standing* if the sum of the budget surpluses of the countries in S, minus the total value of all exports from countries in S to countries not in S, is nonnegative. Give a polynomial-time algorithm that takes this data for a set of n countries and decides whether it contains a nonempty free-standing subset.

Solution:

- Let $S^+ = \sum_{s_j > 0} s_j$.
- Then for a subset of countries A the following *free-standing* sum, $f(A)$, can be calculated

$$f(A) = \sum_{j \in A} s_j - \sum_{i \in A, j \notin A} e_{ij} = S^+ + \sum_{j \in A : s_j < 0} s_j - \sum_{j \notin A : s_j > 0} s_j - \sum_{i \in A, j \notin A} e_{ij}$$

- Flow network:
  - A node for each country j.
  - For each pair of countries i, j, there is an edge (i, j) with capacity e_{ij} and an edge (j, i) with capacity e_{ji}.
  - A node s with an edge with capacity s_j to each country j with $s_j > 0$.
  - A node t with an edge with capacity $-s_j$ from each country j with $s_j < 0$.
  - Consider a cut (A, B):
    - An edge from $i \in A$ to t contributes $-s_j$.
    - A edge from s to $j \in B$ contributes s_j.
    - An edge from $i \in A$ to $j \in B$ contributes e_{ij}.
    - Hence, $c(A, B) = -\sum_{j \in A : s_j < 0} s_j + \sum_{j \notin A : s_j > 0} s_j + \sum_{i \in A, j \notin A} e_{ij}$ which corresponds to all but the constant in the definition of $f(A)$.
- Therefore, $f(A) = S^+ - c(A, B)$ and minimizing the cut (A, B) gives the maximum $f(A)$.
- If $f(A)$ for the min-cut is > 0, then the set $A \setminus \{s\}$ is free-standing.



Coding Question: Matching

5. Implement an algorithm to determine the maximum matching in a bipartite graph and if that matching is perfect (all nodes are matched) in either C, C++, C#, Java, Python, or Rust. Be efficient and use your max-flow implementation from the previous week.

   The input will start with an positive integer, giving the number of instances that follow. For each instance, there will be 3 positive integers m, n, and q. Numbers m and n are the number of nodes in node set A and node set B. Number q is the number of edges in the bipartite graph. For each edge, there will be 2 more positive integers i, and j representing an edge between node $1 \leq i \leq m$ in A and node $1 \leq j \leq n$ in B.

   A sample input is the following:

```
3
2 2 4
1 1
1 2
2 1
2 2
2 3 4
2 3
2 1
1 2
2 2
5 5 10
1 1
1 3
2 1
2 2
2 3
2 4
3 4
4 4
5 4
5 5
```

   The sample input has 3 instances.

   For each instance, your program should output the size of the maximum matching, followed by a space, followed by an N if the matching is not perfect and a Y if the matching is perfect. Each output line should be terminated by a newline. The correct output to the sample input would be:

```
2 Y
2 N
4 N
```

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

Intractibility

1. *Kleinberg, Jon. Algorithm Design (p. 506, q. 4).* A system has a set of n processes and a set of m resources. Each process specifies a set of resources that it requests to use. Each resource might be requested by many processes; but it can only be used by a single process. If a process is allocated all the resources it requests, then it is active; otherwise it is blocked.

   Thus we phrase the Resource Reservation Problem as follows: Given a set of processes and resources, the set of requested resources for each process, and a number k, is it possible to allocate resources to processes so that at least k processes will be active?

   For the following problems, either give a polynomial-time algorithm or prove the problem is NP-complete.

   (a) The general Resource Reservation Problem defined above.

Solution:

1. *The general Resource Reservation Problem (RRP) is in NP.*
   Given a set S of k processes to be activated, we will verify in polynomial time that there is no overlap in the resources required by the processes in S. For each resource, count how many processes in S have requested that resource. This can be done in time $O(km)$. If all counts are ≤ 1, the solution S is valid. If not, S is invalid. Checking the counts takes $O(m)$. Since a proposed solution S can be confirmed or rejected in polynomial time, this problem is in NP.
2. *The general RRP is NP-Hard.* (*Independent Set* $\leq_p$ *RRP*)
   An arbitrary instance of Independent Set is defined on a graph G and an integer k. Construct an instance of the RRP as follows:
   - Create a process for each node of G.
   - Create a resource for each edge of G.
   - For each node v of G, make the process corresponding to v require exactly the resources corresponding to the edges of G incident to v.

   This transformation can be done in polynomial time, since we can form the sets of processes and resources directly from the sets of vertices and edges in G, and determining which processes require which resources is a matter of finding edges adjacent to nodes, which is very easy with an adjacency list representation (but still polynomial with any "normal" graph representation). In this way, we transform an arbitrary instance of IS into an instance of RRP.
   We now prove correctness of the reduction (the Independent Set instance is true $\iff$ the RRP instance is true):
   ($\Rightarrow$) If we have an independent set of k nodes in G, the selected nodes have no edges in common. This means that no processes have any required resources in common by the way we transformed the instance above. So a if the independent set instance accepts, the general RRP instance accepts as well.
   ($\Leftarrow$) If we have k processes that have no overlap in required resources, then the k nodes that correspond to those processes must have no edges in common. So a yes instance for the general resource reservation is a yes instance for independent set.
   Thus, we have a polynomial time mapping from independent set to the general resource reservation problem, in which we have a yes instance for independent set if and only if we have a yes instance for the general resource reservation problem. Since Independent Set is NP-Hard, the general resource reservation problem is NP-Hard.

(b) The special case of the problem when $k = 2$.

Solution:

Consider the following polynomial-time algorithm which solves the special case of $k = 2$:

- For each pair of processes, check all m resources to see if they have any requirements in common. ($O(mn^2)$)
- If there is a pair of processes which have no requirements in common, the answer is yes.
- If no such pair exists, the answer is no.

(c) The special case of the problem when there are two types of resources–say, people and equipment– and each process requests at most one resource of each type (In other words, each process requests at most one person and at most one piece of equipment.)

Solution:

- Let S denote a maximal set of processes which only require one resource, and in which every process is disjoint in resource requirements. Activate the processes in S and discard all other processes which request a resource required by a process in S. (polynomial in number of processes)
- Create sets of nodes T_1 and T_2 representing the resources of the first and second type, respectively. For each resource r requested by a process in S, remove the node for r. (polynomial in number of resoures)
- For each process that requires $t_1 \in T_1$ and $t_2 \in T_2$, where neither t_1 nor t_2 have been removed, create an edge from t_1 to t_2. (polynomial in number of processes)
- Now this can be solved as a bipartite matching problem (which can be done in polynomial time), where the desired number of matches is $k - |S|$.

(d) The special case of the problem when each resource is requested by at most two processes.

Solution:

The reduction outlined in (a) corresponds to this special case, since the resource/edges of independent set are adjacent to exactly two process/nodes. Thus, this special case is still NP-Complete, by that reduction.

2. *Kleinberg, Jon. Algorithm Design (p. 506, q. 7).* The 3-Dimensional Matching Problem is an NP-complete problem defined as follows:

   Given disjoint sets X, Y, and Z, each of size n, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, does there exist a set of n triples in T such that each element of $X \cup Y \cup Z$ is contained in exactly one of these triples?

   Since 3-Dimensional Matching is NP-complete, it is natural to expect that the 4-Dimensional Problem is at least as hard.

   Let us define 4-Dimensional Matching as follows. Given sets W, X, Y, and Z, each of size n, and a collection C of ordered 4-tuples of the form $(w, x, y, z) \in W \times X \times Y \times Z$, do there exist n 4-tuples from C such that each element of $W \cup X \cup Y \cup Z$ appears in exactly one of these 4-tuples?

   Prove that 4-Dimensional Matching is NP-complete. Hint: use a reduction from 3-Dimensional Matching.

Solution:

1. *4-Dimensional Matching is in NP.*
   Given a set $S \subset C$ of n 4-tuples, we verify that S is a valid solution to the 4-D Matching problem. For each element $w \in W$, loop through the first entries of the 4-tuples in S, and ensure that exactly one of these entries is w. Then, for each $x \in X$, loop through the second entries of the 4-tuples in S, and ensure that exactly one of these entries is x. Perform a similar procedure for Y and Z. These loops run it time $4n^2 = O(n^2)$, which is polynomial in n. So 4-Dimensional Matching is in NP.
2. *4-Dimensional Matching is NP-Hard.* (*3-D Matching* $\leq_p$ *4-D Matching*)
   An arbitrary instance of 3-Dimensional matching is defined on sets X, Y, and Z, each of size n, along with a set $T \subseteq X \times Y \times Z$ of ordered triples. We can transform this into an instance of 4D matching by padding the tuples. One possibility for this is to create a distinct set of n-elements called W, and then creating $T' \subseteq W \times X \times Y \times Z$ by pairing each of the ordered triplets in T with *every* element of W. This is a polynomial-time transformation: checking if the elements in W are disjoint from $X \cup Y \cup Z$ takes $O(n^2)$ time, and creating the 4-tuples takes $O(n|T|)$ time.

   ($\Rightarrow$) Suppose there is a set of n triples that satisfy the instance of 3-Dimensional Matching. We can pair each of these n triples with a distinct $w \in W$, and as $|W| = n$, this would be a bijective mapping. Thus the transformed 4-tuples will also satisfy the transformed 4-Dimensional Matching instance.

   ($\Leftarrow$) If there is a set of n 4-tuples that satisfy the instance of 4-Dimensional Matching, the corresponding original triples will also satisfy the 3-Dimensional Matching instance, similar to as described above.

   Thus, 4-Dimensional Matching is NP-Hard.

3. *Kleinberg, Jon. Algorithm Design (p. 507, q. 6).* Consider an instance of the Satisfiability Problem, specified by clauses $C_1, ..., C_m$ over a set of Boolean variables $x_1, ..., x_n$. We say that the instance is monotone if each term in each clause consists of a nonnegated variable; that is, each term is equal to x_i, for some i, rather than $\overline{x_i}$. Monotone instances of Satisfiability are very easy to solve: They are always satisfiable, by setting each variable equal to 1.

   For example, suppose we have the three clauses

   $$(x_1 \vee x_2), (x_1 \vee x_3), (x_2 \vee x_3).$$

   This is monotone, and the assignment that sets all three variables to 1 satisfies all the clauses. But we can observe that this is not the only satisfying assignment; we could also have set x_1 and x_2 to 1, and x_3 to 0. Indeed, for any monotone instance, it is natural to ask how few variables we need to set to 1 in order to satisfy it.

   Given a monotone instance of Satisfiability, together with a number k, the problem of *Monotone Satisfiability with Few True Variables* asks: Is there a satisfying assignment for the instance in which at most k variables are set to 1? Prove this problem is NP-complete.

Solution:

1. *Monotone Satisfiability with Few True Variables (MSFTV) in in NP.*
   Given an assignment to the variables $x_1, \ldots, x_n$, we will verify in polynomial time that this assignment is a solution to MSFTV. First, verify in $O(n)$ time that at most k variables are set to true. Then substitute the assignments to the variables into Φ and ensure that every clause of Φ evaluates to true. This also takes $O(n)$ time, so our verification procedure runs in polynomial time. Hence MSFTV is in NP.
2. *MSFTV is NP-Hard.* (*Vertex Cover* $\leq_p$ *MSFTV*)
   An arbitrary instance of vertex cover is defined on a graph $G = (V, E)$ and a number k. For every $v \in V$, create a variable, x_v. For every $\{u, v\} \in E$, create a clause $(x_u \vee x_v)$. Create the formula Φ as the conjunction of all such clauses. This transformation is clearly polynomial, as we just need to process each vertex and edge in G. Additionally, this will always produce a monotone formula. In this way, we transform an arbitrary instance of vertex cover into an instance of MSFTV.

   ($\Rightarrow$) Suppose there is a vertex cover for G of size at most k. For each vertex v in the vertex cover, setting the corresponding variable x_v to true in Φ will result in every clause evaluating to true, as, by construction of Φ and the definition of vertex cover, each clause will have at least one of its two variables set to true. Thus Φ is satisfiable.

   ($\Leftarrow$) Suppose there is a satisfying assignment for Φ in which at most k variables are set to 1. Since Φ is monotone, Φ is satisfiable if and only if at least one variable in every clause is set to 1. If we take the vertices that correspond to the variables that are set to 1, we have at most k vertices and, by construction of Φ, each edge of G has at least one of its endpoints included, so these vertices form a vertex cover.

   Thus, Monotone Satisfiability with Few True Variables is NP-Hard.

4. *Kleinberg, Jon. Algorithm Design (p. 509, q. 10).* Your friends at WebExodus have recently been doing some consulting work for companies that maintain large, publicly accessible Web sites and they've come across the following Strategic Advertising Problem.

   A company comes to them with the map of a Web site, which we'll model as a directed graph $G = (V, E)$. The company also provides a set of t trails typically followed by users of the site; we'll model these trails as directed paths $P_1, P_2, ..., P_t$ in the graph G (i.e., each P_i is a path in G).

   The company wants WebExodus to answer the following question for them: Given G, the paths $\{P_i\}$, and a number k, is it possible to place advertisements on at most k of the nodes in G, so that each path P_i includes at least one node containing an advertisement? We'll call this the Strategic Advertising Problem, with input $G, \{P_i : i = 1, ..., t\}$, and k. Your friends figure that a good algorithm for this will make them all rich; unfortunately, things are never quite this simple.

   (a) Prove that Strategic Advertising is NP-Complete.

Solution:

1. *Strategic Advertising is in NP.*
   Consider an instance of the Strategic Advertising Problem on G, $\{P_i : i = 1, ..., t\}$, and k. If we are given k (or fewer) nodes on which to place advertisements, we can check the validity of this solution by checking all t paths for the presence of any of the k nodes ($O(kt|V|)$), rejecting if any path misses all selected nodes and accepting otherwise. So the Strategic Advertising Problem is in NP.
2. *Strategic Advertising is NP-Hard.* (*Vertex Cover* $\leq_p$ *Strategic Advertising*)
   An arbitrary instance of vertex cover is defined on a graph $G = (V, E)$ and a number k. Since G is undirected and Strategic Advertising is defined on a directed graph, arbitrarily direct all edges ($O(|E|)$) to form G'. For each new directed edge e_i, define a path P_i (which consists of the single edge), also $O(|E|)$. In this way, we can take an arbitrary instance of Vertex Cover and turn it into an instance of Strategic Advertising.

   ($\Rightarrow$) If G has a vertex cover of at most k vertices, then every path in G' (each of which is simply a directed version of an edge in G) includes as least of of those vertices, and thus there is a strategic advertisement.

   ($\Leftarrow$) If there is a strategic advertisement for G' of at most k vertices, then every path contains at least one of those vertices (by definition of the problem). If every path in G' contains at least one of the selected vertices, then every edge in G is indicent on one of the (at most) k vertices, and thus there is a vertex cover of size at most k.

   Thus, Strategic Advertising is NP-Hard.

(b) Your friends at WebExodus forge ahead and write a pretty fast algorithm $\mathcal{S}$ that produces yes/no answers to arbitrary instances of the Strategic Advertising Problem. You may assume that the algorithm $\mathcal{S}$ is always correct.

Using the algorithm $\mathcal{S}$ as a black box, design an algorithm that takes input $G, \{P_i : i = 1, ..., t\}$, and k as in part (a), and does one of the following two things:

- Outputs a set of at most k nodes in G so that each path P_i includes at least one of these nodes.
- Outputs (correctly) that no such set of at most k nodes exists.

Your algorithm should use at most polynomial number of steps, together with at most polynomial number of calls to the algorithm $\mathcal{S}$.

Solution:

1. Query $\mathcal{S}$ on the original instance of the problem.
   - If $\mathcal{S}$ returns no, output that no such set of at most k nodes exists.
   - If $\mathcal{S}$ returns yes, continue.
2. Create a graph $G' = (V', E')$ that is a copy of G, as well as a copy P' of the paths and a tracker k' which starts equal to k. Finally, we need a set A of vertices for advertising, initially the empty set.
3. Arbitrarily select a vertex $v \in V'$.
   - For each occurrence of v in E':
     - If $\{u, v\}$ is a directed edge and no edge $\{v, w\}$ exists, remove $\{u, v\}$ from E'.
     - If $\{v, w\}$ is a directed edge and no edge $\{u, v\}$ exists, remove $\{v, w\}$ from E'.
     - If $\{u, v\}$ and $\{v, w\}$ are both directed edges, form a new edge $\{u, w\}$ and remove $\{u, v\}$ and $\{v, w\}$ from E'.
   - The above steps also need to be taken on all paths P'_i that contain v, altogether this can be done in $O(t|E'|^2)$.
   - Remove v from V'.
4. Query $\mathcal{S}$ on G', P', and k'.
   - If $\mathcal{S}$ returns no:
     - Add v to A.
     - Remove all paths from P' that contained v.
     - Decrement k'.
     - If k' is 0 or $|A| = k$, output the contents of A.
     - If $k' > 0$, return to step 3.
   - If $\mathcal{S}$ returns yes:
     - Return to step 3.

Assignment 12 – More Intractability

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

1. *Kleinberg, Jon. Algorithm Design (p. 512, q. 14)* We've seen the Interval Scheduling Problem several times now, in different variations. Here we'll consider a computationally much harder version we'll call *Multiple Interval Scheduling*. As before, you have a processor that is available to run jobs over some period of time.

   People submit jobs to run on the processor. The processor can only work on one job at any single point in time. Jobs in this model, however, are more complicated than we've seen before. Each job requires a *set* of intervals of time during which it needs to use the processor. For example, a single job could require the processor from 10am to 11am and again from 2pm to 3pm. If you accept the job, it ties up your processor during those two hours, but you could still accept jobs that need time between 11am and 2pm.

   You are given a set of n jobs, each specified by a set of time intervals. For a given number k, is it possible to accept at least k of the jobs so that no two accepted jobs overlap in time?

   Show that Multiple Interval Scheduling is NP-Complete.

Solution:

1. *Multiple Interval Scheduling is in NP.*

   Given a set of jobs $J_1, \ldots, J_k$, for each job J_i, iterate through the intervals in the job and flag the processor as occupied during each. If you try to flag the processor during an interval when it is already flagged, then the solution is invalid. If you iterate through all k jobs without trying to flag the same time twice, then the certificate is valid. This process takes at most time equal to the number of available time intervals, since at worst we flag each interval once.

2. *MIS is NP-Hard. (Independent Set $\leq_p$ MIS)*

   An arbitrary instance of Independent Set is defined by a graph G.

   - For each edge e_j in G, create a time interval t_j.
   - For each node v_i in G, create a job J_i which uses the intervals t_j which correspond to edges adjacent to v_i.

   ($\Rightarrow$) If there is a size k independent set, then we can select k nodes from G such that no edge is adjacent to more than one node in our set. Let J_i be in our candidate set of non-conflicting jobs if and only if v_i is in this independent set. By construction, each time interval t_j corresponds to an edge in G, so only one of its adjacent nodes in G could have been picked, and only the jobs corresponding to those nodes require t_j. Since we only picked one of those, our candidate set of jobs is non-conflicting.

   ($\Leftarrow$) If there is a size k set of non-conflicting jobs, then by the same logic as above the set of nodes $\{v_i\}$ corresponding to the jobs $\{J_i\}$ constitute an independent set in G.

   We have a polynomial time mapping from Independent Set to Multiple Interval Scheduling in which we have a yes instance of IS if and only if we have a yes instance of MIS. Since Independent Set is NP-complete, Multiple Interval Scheduling is NP-Hard.

2. *Kleinberg, Jon. Algorithm Design (p. 519, q. 28)* Consider this version of the Independent Set Problem. You are given an undirected graph G and an integer k. We will call a set of nodes I "strongly independent" if, for any two nodes $v, u \in I$, the edge (v, u) is not present in G, and neither is there a path of two edges from u to v, that is, there is no node w such that both (v, w) and (u, w) are present in G. The Strongly Independent Set problem is to decide whether G has a strongly independent set of size at least k.

   Show that the Strongly Independent Set Problem is NP-Complete.

Solution:

1. *Strongly Independent Set is in NP.*

   Given a set of nodes I, we can check whether any nodes $v, u \in I$ are too close together in polynomial time. For each $u \in I$, we can use BFS to see if any of the other nodes are distance 1 or 2 edges away. This requires $O(nm)$ time in total.

2. *SIS is NP-Hard. (Independent Set $\leq_p$ SIS)*

   An arbitrary instance of Independent Set is defined by a graph G. We will construct a graph G' as an instance of Strongly Independent Set.

   - For each node v_i in G, add v_i to G'.
   - For each edge $e_j = (u, v)$ in G, add a node w_j to G' and add edges (u, w_j) and (w_j, v).
   - Finally add an edge between each pair of added w nodes.

   ($\Rightarrow$) If there is a size k independent set, then the same set of nodes is a size k strongly independent set in G'. Because we have subdivided each edge in G into two edges in G', if two nodes were not adjacent in G (and both had at least one adjacent edge), the distance between them is now 3 ($u - w_i - w_j - v$).

   ($\Leftarrow$) If there is a size k strongly independent set in G', that doesn't use any of the added w nodes, then it follows by construction that they constitute an independent set in G. It remains to show that no nodes w can be part of a strongly independent set. Without loss of generality, consider a particular such node w^*. Because w^* is adjacent to all other w nodes, it can only be part of a strongly independent set if no other w node is, and additionally no other node adjacent to a w node is. In other words, w^* can only be part of a strongly independent set if it is the only node in the set (excluding nodes with no incident edges at all). Note that if there is a strongly independent set which includes w^*, there must exist a second one that includes no w nodes by replacing w^* with any non-w node that has at least one incident edge.

   We have a polynomial time mapping from Independent Set to Strongly Independent Set in which we have a yes instance of IS if and only if we have a yes instance of SIS. Since Independent Set is NP-complete, Strongly Independent Set is NP-Hard.

3. *Kleinberg, Jon. Algorithm Design (p. 527, q. 39)* The *Directed Disjoint Paths Problem* is defined as follows: We are given a directed graph G and k pairs of nodes $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$. The problem is to decide whether there exist node-disjoint paths $P_1, \ldots, P_k$ so that P_i goes from s_i to t_i.

   Show that Directed Disjoint Paths is NP-Complete.

Solution:

1. *Directed Disjoint Paths is in NP.*

   Given a set of paths $P_1, \ldots, P_k$, for each path P_i, iterate through the path and flag each node. If you encounter a node that is already flagged, then the solution is invalid. If you iterate through all k paths without trying to flag the same node twice, then the certificate is valid. This process takes at most time $O(|V|)$, since at worst we flag each node in G once.

2. *DDP is NP-Hard. (3-SAT $\leq_p$ DDP)*

   Suppose we are given a logical formula ϕ with variables $x_1, \ldots x_m$ and clauses $c_1, \ldots c_k$. We construct a graph G in the following way. For each variable x_i, we make two paths:

   - $s_i^v \to T_{i,1} \to T_{i,2} \to \ldots \to T_{i,k} \to t_i^v$, call it T path.
   - $s_i^v \to F_{i,1} \to F_{i,2} \to \ldots \to F_{i,k} \to t_i^v$, call it F path.

   For each clause c_j, we make three paths for each literal occuring inside c_j:

   - If $x_i \in c_j$, then $s_j^c \to T_{i,j} \to t_j^c$
   - If $\overline{x_i} \in c_j$, then $s_j^c \to F_{i,j} \to t_j^c$

   This creates an instance of DDP with the graph G and $m + k$ pair of nodes (s_i^v, t_i^v) and (s_j^c, t_j^c) for $1 \leq i \leq m$ and $1 \leq j \leq k$. ($\Rightarrow$) Suppose ϕ is satisfiable. If x_i is assigned true, then we choose the F path between (s_i^v, t_i^v). If x_i is assigned false, then we choose the T path between (s_i^v, t_i^v). For each clause c_j, there is a literal $\ell_i \in \{x_i, \overline{x_i}\}$ that is assigned to true since ϕ is satisfiable. If $x_i \in c_j$ is assigned to true, then we can choose $s_j^c \to T_{i,j} \to t_j^c$. If $\overline{x_i} \in c_j$ is assigned to true, then we can choose $s_j^c \to F_{i,j} \to t_j^c$. Note that all paths are disjoint. ($\Leftarrow$). Suppose there are disjoint paths between all (s, t) pairs. We interpret the path $s_j^c \to T_{i,j} \to t_j^c$ as the clause c_j satisfied by assigning true to x_i. Similarly, the path $s_j^c \to F_{i,j} \to t_j^c$ as the clause c_j satisfied by assigning false to x_i. Note that there is no conflict in the interpretation, since if there were two distinct clauses c_j and $c_{j'}$ that chose $T_{i,j}$ and $F_{i,j'}$, there cannot be a path between s_i^v and t_i^v.

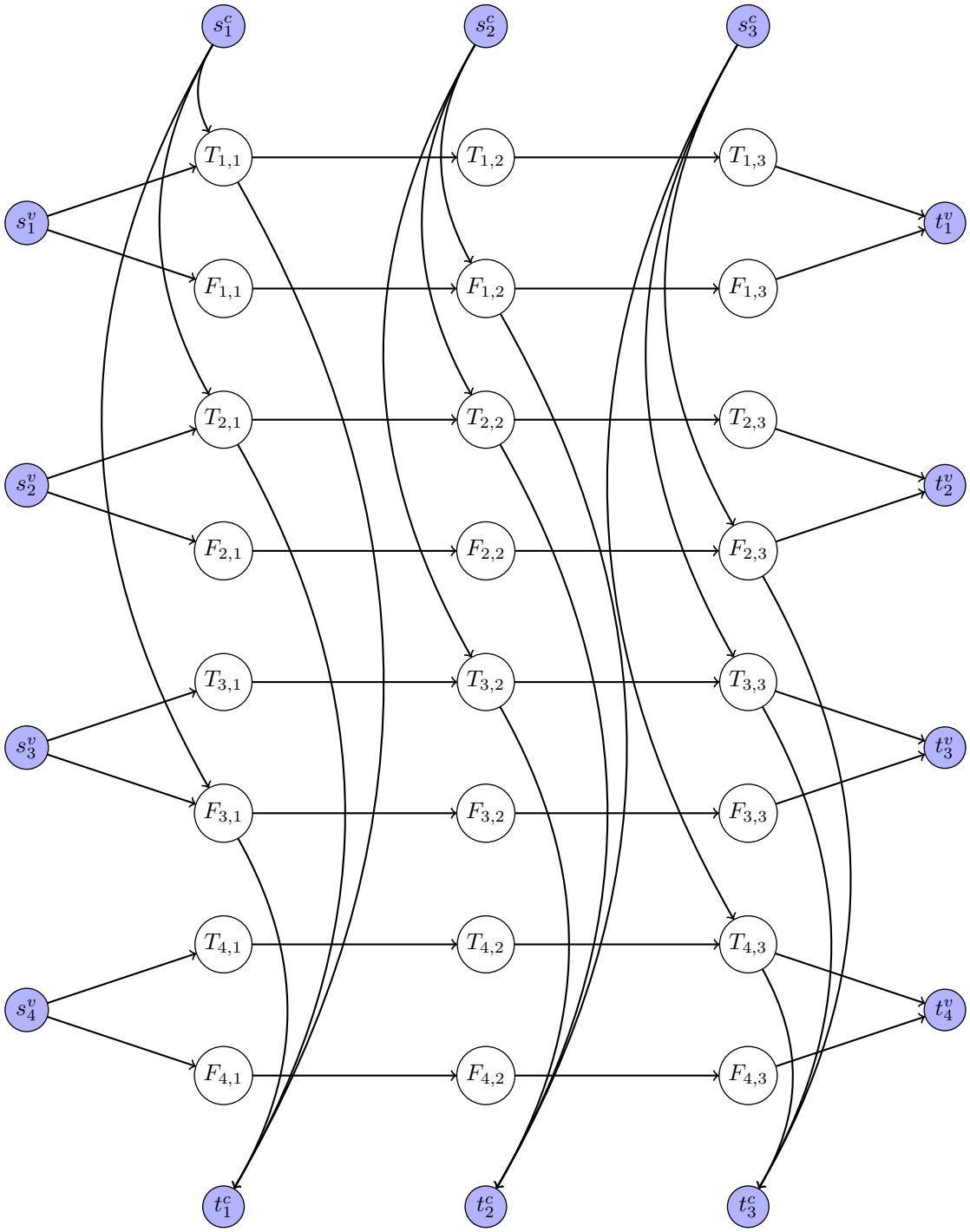
   We have a polynomial time mapping from 3-SAT to DDP in which we have a yes instance of 3-SAT if and only if we have a yes instance of DDP. Since 3-SAT is NP-complete, DDP is NP-Hard.

An example of the reduction is given in the next page.

Solution: Suppose we are given the following 3-SAT formula ϕ.

$$\phi(x_1, x_2, x_3, x_4) = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee x_3 \vee x_4)$$

We will construct the following graph as an input to DDP.



Solution: Alternate reduction: (proof omitted)

We will reduce Independent Set to DDP. Let $(G = (V, E), k)$ be an instance of independent set. Create k pairs $(s_1, t_1), \ldots (s_k, t_k)$. For each vertex $v \in V$, create nodes a_v and b_v, and edges (s_i, a_v) and (b_v, t_j) for all $i, j = 1, \ldots, k$. For each edge e, create a node x_e. Finally, for each vertex v with incident edge set $I(v) \subset E$, choose some order $e_1, \ldots, e_{|I(v)|}$ of $I(v)$ and create directed edges

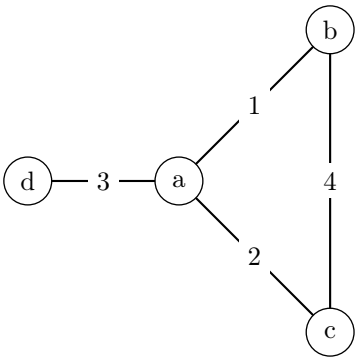
$$(a_v, x_{e_1}), (x_{e_1}, x_{e_2}), (x_{e_2}, x_{e_3}), \ldots, (x_{e_{|I(v)|-1}}, x_{e_{|I(v)|}}), (x_{e_{|I(v)|}}, b_v).$$

The idea is that choosing a path $(s_i, a_v, x_{e_1}, x_{e_2}, \ldots, x_{e_{|I(v)|-1}}, x_{e_{|I(v)|}}, b_v, t_i)$ corresponds to adding vertex v to the independent set; we cannot also add any vertex u adjacent to v because then the paths will intersect at the node x_e corresponding to the edge e between u and v.

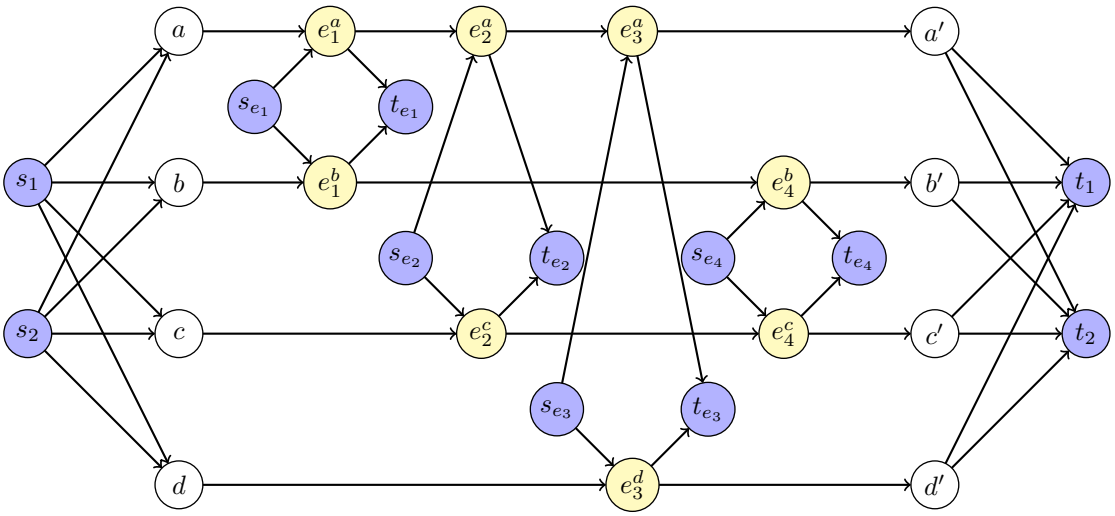
However, this reduction isn't quite correct because there's no guarantee that a path containing a_v will continue on to b_v – it could take the wrong edge at some x_e node. To fix this, replace each node x_{uv} for edge uv with two nodes x^u_{uv} and x^v_{uv}, where the edges along the u path go in and out of x^u_{uv} and the edges along the v path go in and out of x^v_{uv}. Then add a new source/target pair (s_{uv}, t_{uv}) and edges $(s_{uv}, x^u_{uv}), (s_{uv}, x^v_{uv})$ and $(x^u_{uv}, t_{uv}), (x^v_{uv}, t_{uv})$ (still forcing the vertex paths to use at most one of the nodes x^u_{uv} and x^v_{uv}).

An example of the reduction is given below.

Suppose we are given the following graph G with $k = 2$ as an input to Independent Set:



We will construct the following graph as an input to DDP.



4. *Kleinberg, Jon. Algorithm Design (p. 508, q. 9)* The *Path Selection Problem* may look initially similar to the *Directed Disjoint Paths Problem*, but pay attention to the differences between them! Consider the following situation: You are managing a communications network, modeled by a directed graph G. There are c users who are interested in making use of this network. User i issues a "request" to reserve a specific path P_i in G on which to transmit data.

   You are interested in accepting as many path requests as possible, but if you accept both P_i and P_j, the two paths cannot share any nodes.

   Thus, the Path Selection Problem asks, given a graph G and a set of requested paths $P_1, \ldots, P_c$ (each of which must be a path in G), and given a number k, is it possible to select at least k of the paths such that no two paths selected share any nodes?

   Show that Path Selection is also NP-Complete.

Solution:

1. *Path Selection is in NP.*

   Given a set of paths $P_1, \ldots, P_k$, for each path P_i, iterate through the path and flag each node. If you encounter a node that is already flagged, then the solution is invalid. If you iterate through all k paths without trying to flag the same node twice, then the certificate is valid. This process takes at most time $O(|V|)$, since at worst we flag each node in G once.

2. *PS is NP-Hard. (Independent Set $\leq_p$ PS)*

   An arbitrary instance of Independent Set is defined by a graph G. We will create a graph G' on which to run Path Selection.

   - For each edge in G, create a node in G'.
   - For each node v_i in G, create a path P_i that visits each node in G' which corresponds to an edge in G that is incident to v_i. In other words: let $e_{j_1}, e_{j_2}, \ldots, e_{j_{h(i)}}$ be the edges incident to a node v_i in G, then we create a path P_i with edges $e_{j_1} \to e_{j_2} \to \ldots e_{j_{h(i)}}$ (order doesn't matter). (see the figure on next page for an illustration).

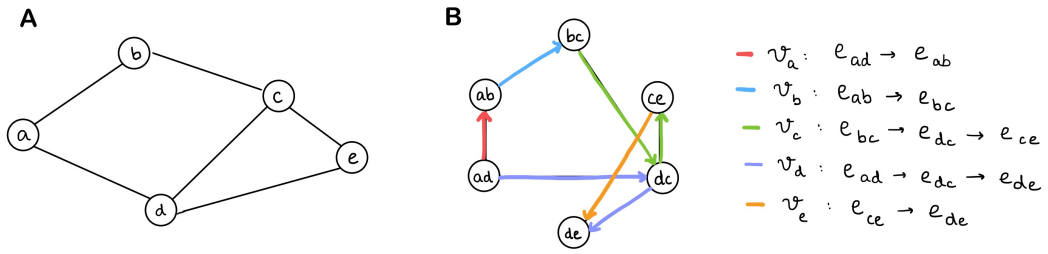
   G has an independent set of size k if and only if there are k paths that are node-disjoint in G'.

   ($\Rightarrow$) If there is a size k independent set, then we can select k nodes from G such that no edge is adjacent to more than one node in our set. Let P_i be in our candidate set of disjoint paths if and only if v_i is in this independent set. By construction, each node e_j in G' corresponds to an edge in G, so only one of its adjacent nodes in G could have been picked, and only the paths corresponding to those nodes include e_j in G'. Since we only picked one of those, our candidate set of paths is indeed disjoint.

   ($\Leftarrow$) If there is a size k set of node-disjoint paths, then by the same logic as above the set of nodes $\{v_i\}$ corresponding to the disjoint paths $\{P_i\}$ constitute an independent set in G.

We have a polynomial time mapping from Independent Set to Path Selection in which we have a yes instance of IS if and only if we have a yes instance of PS. Since Independent Set is NP-complete, Path Selection is NP-Hard.

Solution:



Path Selection Problem. **A** Example directed graph $G = (V, E)$. Observe that the maximum independent set for G has size 2 ($\{a, e\}, \{b, d\}, \{b, e\}$, or $\{a, c\}$). **B** We build a graph G'. For each edge in G, we create a node in G'. Then for every vertex v_i of G with adjacent edges $e_{j_1}, e_{j_2}, \ldots, e_{j_h(i)}$, we create a path P_i with edges $e_{j_1} \to e_{j_2} \to \ldots e_{j_h(i)}$ picked in some arbitrary order. Observe that G' has a maximum of 2 node-disjoint paths: (P_a:red, P_e:orange), (P_b:blue, P_d:purple), (P_b:blue, P_e:orange), and (P_a:red, P_c:green).

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

Randomization

1. *Kleinberg, Jon. Algorithm Design (p. 782, q. 1).*

   3-Coloring is a yes/no question, but we can phrase it as an optimization problem as follows.

   Suppose we are given a graph $G = (V, E)$, and we want to color each node with one of three colors, even if we aren't necessarily able to give different colors to every pair of adjacent nodes. Rather, we say that an edge (u, v) is *satisfied* if the colors assigned to u and v are different. Consider a 3-coloring that maximizes the number of satisfied edges, and let c^* denote this number. Give a polynomial-time algorithm that produces a 3-coloring that satisfies at least $\frac{2}{3}c^*$ edges. If you want, your algorithm can be randomized; in this case, the expected number of edges it satisfies should be at least $\frac{2}{3}c^*$.

Solution:

Assign each vertex a uniformly random color. Then for each edge (u, v), this edge is only not satisfied if the two vertex colors of u and v are the same, which has a $1/3$ probability of occurring. Thus the probability that any particular edge is satisfied is $2/3$, so in expectation $(2/3)|E|$ edges are satisfied. Since $c^* \leq |E|$, this satisfies at least $(2/3)c^*$ edges in expectation.

2. *Kleinberg, Jon. Algorithm Design (p. 787, q. 7).*

In lecture, we designed an approximation algorithm to within a factor of 7/8 for the MAX 3-SAT Problem, where we assumed that each clause has terms associated with three different variables. In this problem, we will consider the analogous MAX SAT Problem: Given a set of clauses $C_1, \cdots, C_k$ over a set of variables $X = \{x_1, \cdots, x_n\}$, find a truth assignment satisfying as many of the clauses as possible. Each clause has at least one term in it, and all the variables in a single clause are distinct, but otherwise we do not make any assumptions on the length of the clauses: There may be clauses that have a lot of variables, and others may have just a single variable.

(a) First consider the randomized approximation algorithm we used for MAX 3-SAT, setting each variable independently to true or false with probability 1/2 each. Show that in the MAX SAT, the expected number of clauses satisfied by this random assignment is at least $k/2$, that is, at least half of the clauses are satisfied in expectation.

Solution:

For a MAX SAT instance of n variables $x_1, \cdots, x_n$ and k clauses $C_1, \cdots, C_k$, we set each variable independently to true or false with probability 1/2. For a clause with m variables, there is only one assignment under which it is not satisfied (all terms are false), so the probability that it is satisfied is $1 - \left(\frac{1}{2}\right)^m$. Since $m \geq 1$, $1 - \left(\frac{1}{2}\right)^m \geq 1 - \frac{1}{2} = \frac{1}{2}$, so each clause has at least a 1/2 probability of being satisfied. Thus the expected number of clauses satisfied is at least $k/2$ by linearity of expectation.

(b) Give an example to show that there are MAX SAT instances such that no assignment satisfies more than half of the clauses.

Solution:

For any $n \in \mathbb{N}$, we can create an instance with n variables and $2n$ clauses, where for each variable x_i, we add both the clause $\{x_i\}$ and the clause $\{\overline{x_i}\}$. Thus the final formula looks like

$$x_1 \wedge \overline{x_1} \wedge x_2 \wedge \overline{x_2} \wedge \cdots \wedge x_n \wedge \overline{x_n}$$

Any assignment to this formula satisfies n clauses, exactly half of all the clauses.

(c) If we have a clause that consists only of a single term (e.g., a clause consisting just of x_1, or just of $\overline{x_2}$), then there is only a single way to satisfy it: We need to set the corresponding variable in the appropriate way. If we have two clauses such that one consists of just the term x_i, and the other consists of just the negated term $\overline{x_i}$, then this is a pretty direct contradiction. Assume that our instance has no such pair of "conflicting clauses"; that is, for no variable x_i do we have both a clause $C = \{x_i\}$ and a clause $C' = \{\overline{x_i}\}$. Modify the randomized procedure above to improve the approximation factor from 1/2 to at least 0.6. That is, change the algorithm so that the expected number of clauses satisfied by the process is at least $0.6k$.

Solution:

Simpler solution:

One thing which may help students arrive at this answer on their own is to consider that we can choose a probability with which a literal can be assigned a value; that is to say, we do not have to assign literal values uniformly at random. For example, we could choose to always set a literal to true, or do so 90%, 50%, or 0% of the time.

To elaborate on this point, it may be advantageous to use a different strategy depending on how many singleton clauses there are. For example, if there are overwhelmingly many singleton clauses, then it makes sense to satisfy them with a high probability, as this will yield a high expected number of satisfied clauses. If there are few or no singleton clauses, a random assignment may yield at least $1 - (0.5 * 0.5) = 0.75$ satisfied clauses in expectation. More generally, consider a variable x which represents the proportion of singleton clauses out of all clauses:

- If $x \geq 0.6$, it is enough to satisfy the singleton clauses to achieve an approximation factor of x, which is ≥ 0.6.
- If $x \leq 0.6$, then a UAR assignment to literals should yield

$$E[X] = (x)(1 - 0.5) + (1 - x)(1 - (0.5)^2) = (0.5x) + (0.75 - 0.75x) = 0.75 - 0.25x$$

satisfied literals in expectation. Since $x \leq 0.6$,

$$E[X] = 0.75 - 0.25x \geq 0.75 - 0.25(0.6) = 0.75 - 0.15 = 0.6$$

More complicated but equally valid solution:

If x_i appears in a singleton clause, then instead of assigning x_i randomly, we bias the assignment so that it has probability $p \geq 1/2$ of satisfying the singleton clause. Then, for each clause, there are two cases:

1. The clause is a singleton. In this case, the probability of satisfying the clause is p.
2. The clause contains $m \geq 2$ variables. In this case, the probability of satisfying the clause is at least $1 - p^m$. This is because the only way to not satisfy the clause is if all m literals are false, which, because $p \geq 1/2$, has probability at most p^m (in the worst case, each literal also appears negated as a singleton clause). We loosen the lower bound to be $1 - p^2$, since $p \leq 1$, $p^m \leq p^2$, thus $1 - p^m \geq 1 - p^2$.

Now, we want to choose $p \in (1/2, 1]$ to maximize $\min(p, 1 - p^2)$ in order to maximize the expected number of clauses satisfied. By setting $p = 1 - p^2$ we get $p = (\sqrt{5} - 1)/2 \approx 0.618$. Thus in expectation, $0.618k > 0.6k$ clauses are satisfied.

(d) Give a randomized polynomial-time algorithm for the general MAX SAT Problem, so that the expected number of clauses satisfied by the algorithm is at least a 0.6 fraction of the maximum possible. (Note that, by the example in part (a), there are instances where one cannot satisfy more than $k/2$ clauses; the point here is that we'd still like an efficient algorithm that, in expectation, can satisfy a 0.6 fraction of the maximum that can be satisfied by an optimal assignment.)

Solution:

We can use the same algorithm as in part (c), but now we need to check for conflicting clauses. We can do this in polynomial time by searching for contradictory pair of clauses $\{x_i\}$ and $\{\overline{x_i}\}$. For each such pair, we remove one of the clauses from the formula. If we end up removing m clauses (and $k - m$ clauses remaining), note that the maximum number of clauses that can be satisfied is at most $k - m$ since we can only satisfy one of the clauses in each pair. Calling the algorithm from part (c) on the reduced formula gives an approximation we want.

3. *Kleinberg, Jon. Algorithm Design (p. 789, q. 10).*

Consider a very simple online auction system that works as follows. There are n *bidding agents*; agent i has a bid b_i, which is a positive natural number. We will assume that all bids b_i are distinct from one another. The bidding agents appear in an order chosen uniformly at random, each proposes its bid b_i in turn, and at all times the system maintains a variable b^* equal to the highest bid seen so far. (Initially b^* is set to 0.) What is the expected number of times that b^* is updated when this process is executed, as a function of the parameters in the problem?

Solution:

Bid i only updates b^* if $b_i > b_j$ for all $j < i$. Since the ordering is chosen uniformly at random, given the set of the first i bids (the bids without the order), all orders of these i bids are equally likely, and thus the probability that bid i updates b^* is $\frac{(i-1)!}{i!} = \frac{1}{i}$ (the denominator is the number of permutations for the i bids, and the numerator is the number of permutations with the last number being the largest). Then by linearity of expectations, we sum over i from 1 to n to obtain that the expected number of updates is $\sum_{i=1}^{n} \frac{1}{i}$ (the nth Harmonic number).

4. Recall that in an undirected and unweighted graph $G = (V, E)$, a cut is a partition of the vertices $(S, V \backslash S)$ (where $S \subseteq V$). The size of a cut is the number of edges which cross the cut (the number of edges (u, v) such that $u \in S$ and $v \in V \backslash S$). In the MAXCUT problem, we try to find the cut which has the largest value. (The decision version of MAXCUT is NP-complete, but we will not prove that here.) Give a randomized algorithm to find a cut which, in expectation, has value at least $1/2$ of the maximum value cut.

Solution:

Assign each vertex to a side of the cut independently and uniformly at random. Then the probability that a given edge (u, v) is cut is $1/2$, so in expectation $(1/2)|E|$ edges are cut. The optimal cut can't have value larger than $|E|$, so this is at least $(1/2)$ times the optimal cut.

5. Implement an algorithm which, given a MAX 3-SAT instance, produces an assignment which satisfies at least 7/8 of the clauses, in either C, C++, C#, Java, Python, or Rust.

   The input will start with a positive integer n giving the number of variables, then a positive integer m giving the number of clauses, and then m lines describing each clause. The description of the clause will have three integers x y z, where $|x|$ encodes the variable number appearing in the first literal in the clause, the sign of x will be negative if and only if the literal is negated, and likewise for y and z to describe the two remaining literals in the clause. For example, $3\ -1\ -4$ corresponds to the clause $x_3 \vee \overline{x_1} \vee \overline{x_4}$. A sample input is the following:

```
10
5
-1 -2 -5
6 9 4
-9 -7 -8
2 -7 10
-1 3 -6
```

   Your program should output an assignment which satisfies at least $\lfloor \frac{7}{8} m \rfloor$ clauses. Return n numbers in a line, using a ± 1 encoding for each variable (the ith number should be 1 if x_i is assigned TRUE, and -1 otherwise). The maximum possible number of satisfied clauses is 5, so your assignment should satisfy at least $\lfloor \frac{7}{8} \times 5 \rfloor = 4$ clauses. One possible correct output to the sample input would be:

```
-1 1 1 1 1 1 -1 1 1 1
```

CS577: Introduction to Algorithms (Fall 2024)

Discussion Review Solutions

Problems

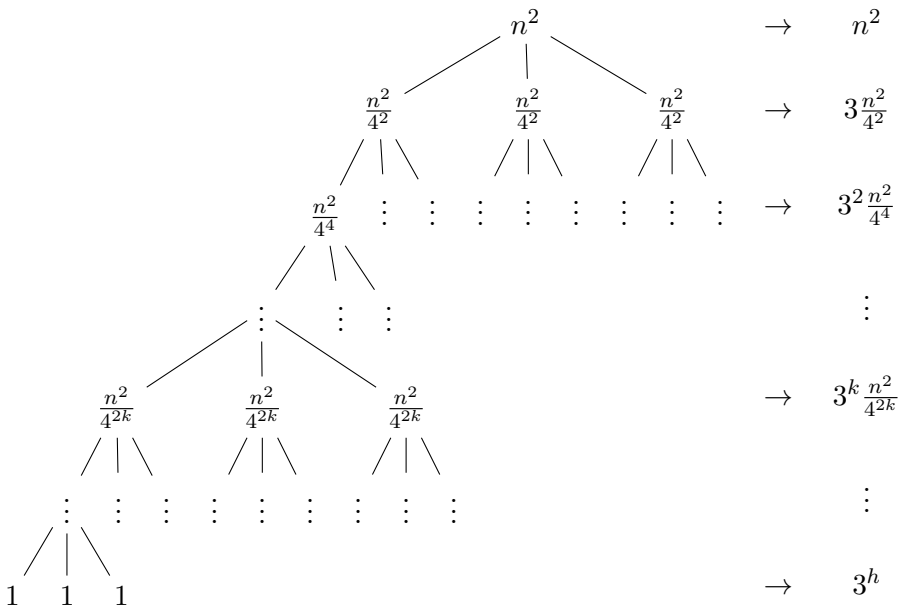
1. Solve the following recurrence:

   - $T(n) = 3T(n/4) + n^2$ and $T(1) = 1$. Use a recursion tree.

     Claim 1

     $$T(n) = 3^{\log_4 n} + n^2 \sum_{k=0}^{\log_4 n - 1} (\frac{3}{16})^k$$

     Proof:



     Thinking of $T(n)$ as representing the running time of a recursive algorithm on an input of size n, we will view n^2 as the amount of work done by the algorithm outside of recursive calls and $3T(n/4)$ as the work done recursively. In particular, the $3T(n/4)$ represents that we recursively solve the problem 3 times on instances of size $n/4$. Now, to construct a recursion tree, we want to summarize the total amount of work done on inputs of each size that will be called recursively. To start, we will always do n^2 work on an input of size n and this is the root of our tree. Then, this root has three children, each of size

$n/4$. For a specific child of the root, $(n/4)^2 = \frac{n^2}{16}$ work will be done outside of its further recursive calls. Since there are 3 total children, we have $\frac{3}{16}n^2$ work done on all instances of size $n/4$ (ignoring further recursive calls). Next, each of these children will spawn 3 additional children of their own, each of size $n/4/4 = n/16$. Again, each such child will do $(n/16)^2$ work ignoring further recursion. The total number of children on this second level (each a recursive instance of size $n/16$) is 9 since we have 3 children for each of the 3 children on the previous level. Hence, we do $\frac{9}{16^2}n^2$ work in total on this level of the tree. Now, we can see a pattern forming. We do $(\frac{3}{16})^k n^2$ work total on level k. Also, since each child spawns 3 further children, the total number of nodes on level k will be 3^k.

If we let h denote the total height of the tree, then we do $\sum_{k=0}^{h-1}(\frac{3}{16})^k$ total work for all the levels in the tree other than the deepest level. For the bottom level consisting of the leaves, the total amount done at each node is different. Instead of the input size squared as we have been assuming the work to be, it will now be the value of the base case, which is $T(1) = 1$. We know we must've hit a base case because the recursion stopped, which only happens when we hit a base case. The total amount of work done in the leaves is then 1 times the total number of leaves, which is 3^h for a complete ternary tree. Thus, we just need to calculate h and we are done. Since the input size is divided by 4 every time we recurse, we know that after k recursive steps the input size is now $n/4^k$. The recursion stops when we hit our base case input of 1 and so the height of the tree is exactly the point at which $n/4^h = 1$. Using log and exponent rules, we have the solution is $h = \log_4 n$. Plugging in this value of h gives us our final closed form solution for T.

$$\begin{aligned} T(n) &= 1 \cdot 3^h + n^2 \sum_{k=0}^{h-1} (\frac{3}{16})^k \\ &= 3^{\log_4 n} + n^2 \sum_{k=0}^{\log_4 n - 1} (\frac{3}{16})^k \end{aligned}$$

■

We can now simplify $T(n)$ as follows.

$$\begin{aligned} T(n) &= 3^{\log_4 n} + n^2 \sum_{k=0}^{\log_4 n - 1} (\frac{3}{16})^k \\ &< n^{\log_4 3} + \frac{16}{13} n^2 = O(n^2) \end{aligned}$$

2. Prove the following statements using induction:

   (a) For all $n \in \mathbb{N}$,

$$1^3 + 2^3 + \cdots + n^3 = \frac{1}{4}n^2(n+1)^2$$

Proof: When $n = 1$, the RHS is $\frac{1}{4}1^2(2)^2 = \frac{4}{4} = 1$. The inductive hypothesis that, for some particular $k \in \mathbb{N}$, $1^3 + 2^3 + \cdots + k^3 = \frac{1}{4}k^2(k+1)^2$. We want to show this holds for $k+1$.

$$\begin{aligned} 1^3 + 2^3 + \cdots + k^3 + (k+1)^3 &= \frac{1}{4}k^2(k+1)^2 + (k+1)^3 && \text{by IH} \\ &= \frac{1}{4}\left(k^2(k+1)^2 + (4k+4)(k+1)^2\right) \\ &= \frac{1}{4}(k+1)^2(k^2+4k+4) \\ &= \frac{1}{4}(k+1)^2(k+2)^2 \end{aligned}$$

■

   (b) For all $n \in \mathbb{N}$,

$$\frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \cdots + \frac{1}{n\cdot(n+1)} < 1$$

(Hint: This is not possible if your inductive hypothesis is $1/(1\cdot 2)+\cdots+1/(k\cdot(k+1)) < 1$. Instead, try to prove an equality of the form $1/(1\cdot 2)+\cdots+1/(n\cdot(n+1)) = f(n)$, where $f(n) < 1$ for all n. Try small values of n to find what $f(n)$ should be.)

Proof: Following the hint, use $f(n) = \frac{n}{n+1}$. Clearly $\frac{n}{n+1} < 1$ for any $n \in \mathbb{N}$. Thus we will attempt to show that $1/(1\cdot 2) + \cdots + 1/(n\cdot(n+1)) = n/(n+1)$ by induction. For the $n = 1$ case, the LHS is $1/2$ and the RHS is $\frac{1}{1+1} = 1/2$. Next, supposing that for some $k \in \mathbb{N}$, $1/(1\cdot 2) + \cdots + 1/(k\cdot(k+1)) = k/(k+1)$.

$$\begin{aligned} \frac{1}{1\cdot 2} + \cdots + \frac{1}{k(k+1)} + \frac{1}{(k+1)(k+2)} &= \frac{k}{k+1} + \frac{1}{(k+1)(k+2)} && \text{by IH} \\ &= \frac{k(k+2)+1}{(k+1)(k+2)} \\ &= \frac{k^2+2k+1}{(k+1)(k+2)} \\ &= \frac{(k+1)^2}{(k+1)(k+2)} \\ &= \frac{k+1}{k+2} \end{aligned}$$

■

(c) All trees with $|V| \geq 2$ have at least two leaves

Proof:

Base case:
A tree with $|V| = 2$ has 2 nodes with one edge between them, hence has 2 leaves.

Inductive Case:
Suppose all trees with $2 \leq |V| \leq k$ have at least two leaves. Now, consider a tree with $|V| = k + 1$.

We should be able to remove one edge to disconnect the graph. Otherwise, the original tree would have had a cycle which contradicts the definition of a tree. As a result of removing an edge, we now have two trees, each of which must have $|V| \leq k$.

For each tree, consider what happens when we reintroduce the egde and restore the original tree:

- If $|V| = 1$, reintroducing the edge will necessarily result in a leaf node.
- Otherwise, by our inductive hypothesis, the tree must have at least 2 leaves. Reintroducing the edge may turn at most one leaf node into an internal node, leaving at least one leaf node.

Thus, at least two leaf nodes must exist in the original tree.

■

3. (a)

Algorithm 1 `BINARY-SEARCH(A, x, left, right)`

Input: `A`: a sorted list of numbers of length n, `x`: a number, `left`: an integer ≥ 1, `right`: an integer $\leq n$.

Output: If `x` $\in$ `A`$[$`left`, `right`$]$ return `True`. Otherwise return `False`.

```
if left ≤ right then
    mid ← (left + right)/2
    if A[mid] < x then
        return BINARY-SEARCH(A, x, mid+1, right)
    else if A[mid] > x then
        return BINARY-SEARCH(A, x, left, mid-1)
    else
        return True
    end if
else
    return False
end if
```

Proving the correctness of the binary search algorithm :

- Proving by strong induction on the $n = right - left + 1$, the length of the array A. The algorithm behaves according to its specification defined in **Input** and **Output**.
- **Base case**: $n = 0$. This means `right` must be less than `left`. Algorithm returns false, which is correct because x can't be in empty section of array.
- **Induction Hypothesis**: Assume that for A of size less than k where $k >= 1$, the algorithm returns true if x is in A, otherwise it returns false.
- **Strong Induction**: Show that for A of size k, the algorithm returns true if x is in A, otherwise it returns false.
  - Case 1: $A[mid] < x$. Since A is sorted, x must be in $A[mid + 1, ..., high]$ if it is in A. $A[mid + 1, ..., high]$ has less than k elements. Note that $mid + 1 \geq 1$ since $1 \leq left \leq right$, so the input to the recursive call is valid. By our induction hypothesis, the algorithm returns the correct result.
  - Case 2: $A[mid] > x$. Since A is sorted, x must be in $A[low, ..., mid - 1]$ if it is in A. $A[low, ..., mid - 1]$ has less than k elements. Note that $mid - 1 \leq n$ since $left \leq right \leq n$, so the input to the recursive call is valid. By our induction hypothesis, the algorithm returns the correct result.
  - Case 3: $A[mid] = x$. Algorithm returns true, which is correct.

(b)

Algorithm 2 `FIND-MAX(A)`

Input: `A`: a list of numbers
Output: *ans* is the largest number of `A`.

```
ans ← A[1]
n ← len(A)
for j ← 2, ..., n do
    ans ← max(ans, A[j])
end for
return ans
```

Proving the correctness of the find max algorithm:

- **Loop Invariant**: At the start of the iteration with index j of the loop, the variable *ans* should contain the maximum of the numbers from the subarray $A[1 \dots j-1]$.
- **Base Case**: At the start of the first loop, we have $j = 2$. Therefore the loop invariant states: At the start of the iteration with index j of the loop, the variable *ans* should contain the maximum of the numbers from the subarray $A[1 \dots 1]$, which is $A[1]$. This is what *ans* has been set to.
- **Inductive Step**: Assume that the loop invariant holds at the start of iteration j. Then it must be that *ans* contains the maximum of numbers in subarray $A[1 \dots j-1]$. There are two cases:
  - $A[j] > ans$. From the loop invariant we get that $A[j]$ is larger than the maximum of the numbers in $A[1 \dots j-1]$. Thus, $A[j]$ is the maximum of $A[1 \dots j]$. In this case, the algorithm sets *ans* to $A[j]$, thus in this case the loop invariant holds again at the beginning of the next loop.
  - $A[j] \leq ans$. That is, the maximum in $A[1 \dots j-1]$ is at least as large as $A[j]$, thus the maximum of $A[1 \dots j-1]$ is the same as the maximum of $A[1 \dots j]$. The algorithm also doesn't change *ans*, thus in this case the loop invariant holds again at the beginning of the next loop.
- **Termination**: When the for-loop terminates $j = n + 1$. Now the loop invariant gives: The variable *ans* contains the maximum of all numbers in subarray $A[1 \dots n]$, which is A. This is exactly the value that the algorithm should output, and which it then outputs. Therefore the algorithm is correct.

4. (Bonus) Solve the following recurrence:

- $T(n) = T(n-1) + T(n-2) + 5$, $T(1) = 1$, and $T(0) = 0$. Use unrolling.

  Claim 2

  $$T(n) = F_n + 5\sum_{i=1}^{n-1} F_i$$

  Proof: First, recall the Fibonacci sequence is defined recursively by $F_n = F_{n-1} + F_{n-2}$ whenever $n \geq 2$. The base cases for the sequence are $F_0 = 0$ and $F_1 = 1$. We note that the recurrence defining $T(n)$ is similar to F_n and so we might expect the Fibonacci sequence to appear in the closed form solution for $T(n)$. In fact, the very similar recurrence $R(n) = R(n-1) + R(n-2) + 1$, $R(1) = 1$, and $R(0) = 0$, has closed form solution $R(n) = \sum_{k=1}^{n} F_k$, so we should find a similar solution for $T(n)$. Now, let's unroll the recurrence for $T(n)$ by continually plugging in the recursive definition for $T(n)$ into itself. The first time we apply the definition, we have $T(n) = T(n-1)+T(n-2)+5$. Now, to continue unrolling, we must apply the definition again to one of the T terms on the right side of the equation. In particular, let's expand the one with the larger argument, $T(n-1)$. Using the definition of T with $n-1$ being substituted for n, we have $T(n-1) = T(n-2) + T(n-3) + 5$. Hence,

  $$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 5 \\ &= (T(n-2) + T(n-3) + 5) + T(n-2) + 5 \\ &= 2T(n-2) + T(n-3) + 2 \cdot 5 \end{aligned}$$

  Again, we will apply the definition of T to $T(n-2)$ to get:

  $$\begin{aligned} T(n) &= 2T(n-2) + T(n-3) + 2 \cdot 5 \\ &= 2(T(n-3) + T(n-4) + 5) + T(n-3) + 2 \cdot 5 \\ &= 3T(n-3) + 2T(n-4) + 4 \cdot 5 \end{aligned}$$

  A pattern may not be clear yet, so let's apply the definition one more time.

  $$\begin{aligned} T(n) &= 3T(n-3) + 2T(n-4) + 4 \cdot 5 \\ &= 3(T(n-4) + T(n-5) + 5) + 2T(n-4) + 4 \cdot 5 \\ &= 5T(n-4) + 3T(n-5) + 7 \cdot 5 \end{aligned}$$

  Now, we can notice that every time we have done this process, the number multiplying the recursive term with larger argument is a fibonacci number F_k, and the number multiplying the other recursive term is the previous fibonacci number F_{k-1}. Also, the multiple of 5 is a sum of fibonacci numbers $\sum_{i=1}^{k} F_i$. In particular, after the kth time we have repeated the unrolling process, we have an equation of the form

  $$T(n) = F_{k+1}T(n-k) + F_kT(n-k-1) + 5\sum_{i=1}^{k} F_i$$

We can stop unrolling when get to a point where we know the value of the function T for its given arguments. In particular, this will be when the arguments to T are the base case values. Both $n-k$ and $n-k-1$ will be base case inputs when $k = n-1$ so that $n-k = 1$ and $n-k-1 = 0$. Then, we have

$$\begin{aligned} T(n) &= F_{k+1}T(n-k) + F_k T(n-k-1) + 5\sum_{i=1}^{k} F_i \\ &= F_n T(1) + F_{n-1}T(0) + 5\sum_{i=1}^{n-1} F_i \\ &= F_n + 5\sum_{i=1}^{n-1} F_i \end{aligned}$$

Where the last equality uses the fact that $T(1) = 1$ and $T(0) = 0$. Having derived the closed form formula for $T(n)$, it is a straightforward exercise for the reader to formally prove the formula is correct by induction. ■

A simpler but inexact solution. If you use the recursion tree approach for this recurrence, you will observe that at level i of the tree, we have at most 2^{i-1} subproblems, with the total work done being equal to 5 times 2^{i-1}. The tree has at most n levels. Adding these numbers up, we get $T(n) \leq 5 \cdot 2^n$.

CS577: Introduction to Algorithms (Fall 2024)

Discussion 2 Solutions

Recap

Asymptotic Bounds

Bound	Informal Notion	Formal Definition	Limit Test
$f(n) \in O(g(n))$	'$f(n) \leq g(n)$'	$O(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid 0 \leq f(n) \leq cg(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq d$ for $d \in \mathbb{R}_{>0}$
$f(n) \in \Omega(g(n))$	'$f(n) \geq g(n)$'	$\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid 0 \leq cg(n) \leq f(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$ for $c \in \mathbb{R}_{>0}$
$f(n) \in \Theta(g(n))$	'$f(n) = g(n)$'	$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$	$c \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq d$ for $c, d \in \mathbb{R}_{>0}$
$f(n) \in o(g(n))$	'$f(n) \ll g(n)$'	$o(g(n)) = \{f(n) : \forall c > 0 \exists n_0 > 0 \mid 0 \leq f(n) < cg(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) \in \omega(g(n)$	'$f(n) \gg g(n)$'	$\omega(g(n)) = \{f(n) : \forall c > 0 \exists n_0 > 0 \mid 0 \leq cg(n) < f(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Problems

1. Arrange the following functions in ascending order of asymptotic growth rate. Assume the base of the logarithm is 2.
   - $\sqrt{5^{\log n}}$
   - $\log(5^n)$
   - 5^n
   - $\sqrt{n}$
   - $n^{\log n}$
   - 3^{n+10}
   - $\log n$
   - $(\log n)^n$
   - $5^{\sqrt{\log n}}$

Solution:

The correct answer (using $\precsim$ to denote the relation "asymptotically smaller than or equal to" – that is $f(n) \precsim g(n) \iff f(n) = O(g(n))$) is:

$$\log n \precsim 5^{\sqrt{\log n}} \precsim \sqrt{n} \precsim \log(5^n) \precsim \sqrt{5^{\log n}} \precsim n^{\log n} \precsim 3^{n+10} \precsim 5^n \precsim (\log n)^n$$

To find this ranking, we start by giving a ranking of the relatively simpler functions:

$$\log n \precsim \sqrt{n} \precsim \log(5^n) = n \log 5 \precsim 3^{n+10} \precsim 5^n \precsim (\log n)^n$$

Now we are left to find the positions of $n^{\log n}$, $\sqrt{5^{\log n}}$, and $5^{\sqrt{\log n}}$. We will use the following fact: If $\log(f(n)) = o(\log(g(n))$, then $f(n) \precsim g(n)$ (as $\log(n)$ is a monotonic increasing function).

1. $n^{\log n}$

   Let's compare the asymptotic growth of $n^{\log n}$ to c^n for a fixed (positive) constant c. Taking the log of both functions gives

   $$\log(c^n) = n \log c$$
   $$\log(n^{\log n}) = (\log n)(\log n) = (\log n)^2$$

   and since $(\log n)^2 = o(n \log c)$, we conclude $n^{\log n} \precsim c^n$. Thus $n^{\log n} \precsim 3^{n+10}$. For a lower bound, clearly $n \log 5 \precsim n^{\log n}$.

2. $\sqrt{5^{\log n}}$

   $$\sqrt{5^{\log n}} = (5^{\log n})^{\frac{1}{2}} = 5^{\frac{1}{2}\log n} = (2^{\log 5})^{\frac{1}{2}\log n} = 2^{\log(n^{\frac{1}{2}\log 5})} = n^{\frac{1}{2}\log 5} \precsim n^{\log n}.$$

   On the other hand, since $\log 5$ is between 2 and 3 (5 is between 2^2 and 2^3), $\frac{1}{2}\log 5 > 1$ and thus $n \log 5 \precsim n^{\frac{1}{2}\log 5} = \sqrt{5^{\log n}}$.

3. $5^{\sqrt{\log n}}$

   First we compare to $\sqrt{n}$. We do this by taking logarithms (base 5) of both functions:

   $$\log_5 \sqrt{n} = \frac{1}{2}\log_5 n$$
   $$\log_5 5^{\sqrt{\log n}} = \sqrt{\log n}$$

   and since $\sqrt{\log n} = o(\frac{1}{2}\log_5 n)$, we conclude $5^{\sqrt{\log n}} \precsim \sqrt{n}$. Now we compare to $\log n$. Again we take the logarithm base 5 of $\log n$, and since $\log_5 \log n = o(\sqrt{\log n})$ (if this is not clear, try substituting $n = 2^m$) we conclude that $\log n \precsim 5^{\sqrt{\log n}}$.

2. *Kleinberg, Jon. Algorithm Design (p. 110, q. 9).* There's a natural intuition that two nodes that are far apart in a communication network — separated by many hops — have a more tenuous connection than two nodes that are close together. There are a number of algorithmic results that are based to some extent on different ways of making this notion precise. Here's one that involves the susceptibility of paths to the deletion of nodes.

   Suppose that an n-node (connected) undirected graph $G = (V, E)$ contains two nodes s and t such that the distance between s and t is strictly greater than $\frac{n}{2}$.

   (a) Show that there must exist some node v, not equal to either s or t, such that deleting v from G destroys all $s-t$ paths. (In other words, the graph obtained from G by deleting v contains no path from s to t.)

   Solution:

   Let $d(x, y)$ be the distance between nodes x and y.

   Since $d(s, t) > n/2$, there is at least one node with distance $1, \ldots, \lfloor n/2 \rfloor$ from s, respectively. We want to prove the following claim:

   Claim: There exists $k \in \{1, \ldots, \lfloor n/2 \rfloor\}$ such that there is only 1 node with distance k from s.

   Proof by Contradiction: Suppose that for each k, there are at least two nodes with distance k from s. Then the graph would have at least $2 \cdot \lfloor n/2 \rfloor \geq 2 \cdot (n-1)/2 = n-1$ nodes not including s and t, which is a contradiction because the graph has only n nodes.

   Therefore, there must be some distance k such that there is only one node v with distance k from s, and thus if we remove v, there cannot be any path from s to t.

   (b) Give an algorithm with running time $O(m + n)$ to find such a node v.

   Solution:

   Starting at s, run breadth-first search on the graph.

   During the exploration of a layer, we count the number of nodes in this layer. If there is exactly 1 node in this layer, return such node. Otherwise, explore the next layer and repeat this process.

   As breadth-first search takes $O(m + n)$ time, this algorithm takes $O(m + n)$ time.

CS577: Introduction to Algorithms (Fall 2024)

Discussion Greedy Solutions

Recap

In lecture, we saw two main techniques for showing that a greedy algorithm, GREEDY, is optimal: Stays Ahead and Exchange Argument.

Stays Ahead

A *Stays Ahead* argument requires that you define a notion of time steps over which you can show by induction that GREEDY is better according to some measure $g(i)$ than any other algorithm, or an optimal algorithm for all time steps i. The opimality of GREEDY can follow immediately, or may require a proof using the property shown in the induction.

The canonical problem presented in lecture was the *Interval Scheduling Problem*, and the optimal GREEDY was the *finish first* heuristic.

Steps for a Stays Ahead argument:

1. Establish a notion of time steps. These times steps should be well-defined for any algorithm.

   Ex: For the Interval Scheduling Problem, the time steps were the index in a set of jobs S produced by an algorithm when the jobs are sorted by finish time from smallest to largest. Note that this is well-defined for any algorithm.

2. Using induction over the time steps, show that, for some measure $g(i)$, GREEDY is the same or better than any other algorithm for all time steps i.

   Ex: For the Interval Scheduling Problem, $g(i)$ was the property that the finish time of the ith job of GREEDY was less than or equal to the ith job scheduled in any other schedule.

3. Using the property shown in the induction to claim the optimality of GREEDY.

   Ex: For the Interval Scheduling Problem, if GREEDY had scheduled k items, then $g(k)$ told us that the finish time of the kth job of GREEDY was less than or equal to the kth job scheduled in any other schedule, implying that GREEDY could schedule any $k+1$ job that any other algorithm would schedule, a contradiction to GREEDY scheduling k items.

Exchange

An *exchange* argument starts from an solution S^* that may be defined as an optimal solution, or the solution of any algorithm. The solution is transformed by some sort of *exchange* to a new solution S_1 that is closer the to the solution S produced by the GREEDY heuristic and no worse than S^*. Then, by induction, we can repeat this exchange producing a sequence of equivalent solutions until we arrive at S. That is, $S^* \equiv S_1 \equiv S_2 \equiv \cdots \equiv S$ for the function we are trying to optimize for the given problem.

The canonical problem presented in lecture was a job scheduling problem of *minimizing max lateness*, and the optimal GREEDY was the *earliest deadline first* (EDF) heuristic.

Steps for a Exchange argument:

1. Consider solutions produced by GREEDY, and determine characteristic properties of those solutions that may not be shared by an arbitrary optimal solution.

   Ex: For minimizing the max lateness, the solutions produced by the EDF had no idle time and jobs were scheduled by increasing deadlines.

2. Define the exchanges need to transform any solution into a solution closer to the solution produced by GREEDY, and show that such an exchange produces a solution that is no worse.

   Ex: For minimizing the max lateness, we showed that (1) idle time could be removed from any schedule without increasing the max lateness, and (2) that in schedule (without idle time) where the jobs are not ordered by increasing deadlines, there are at least two sequential jobs where there deadlines are out of order (an inversion). These two jobs can be swapped (removing an inversion, and sorting them by their deadlines) without increasing the lateness.

3. By induction, transform any solution into a solution that is equivalent to the GREEDY solution.

   Ex: For minimizing the max lateness, the GREEDY schedule S has no idle time and the jobs are scheduling by increasing deadline. The schedule S' produced by starting from any schedule S^*, removing idle time and repeatedly doing exchanges until no inversions remain also has no idle time and the jobs are scheduling by increasing deadline. The schedule S and S' may differ on the order of jobs with the same deadlines, but this does not change the max lateness.

Problems

1. In the year 2048 (when teleportation is already invented), you have opened the first delivery company (*FastAlgo Express*) that uses a teleportation machine to deliver the packages. Suppose on a certain day, n customers give you packages to deliver. Each delivery i should be made within t_i days and the customer pays you a fixed amount of p_i dollars for doing it on time (if you don't do it on time, you get paid 0 dollars). On-time delivery means that if package i is due within $t_i = k$ days, we should deliver it on one of the days $1, 2, \ldots, k$, to be on time. Unfortunately, your teleportation machine is able to do at most **one** delivery to **one** destination in one day (you cannot deliver packages to multiple destinations within 1 day). Your goal is to figure out which deliveries to make and when.

   Input: A set of n deliveries with due dates $t_i \in \mathbb{N}, t_i \geq 1$ and payments $p_i > 0$ for each delivery $i \in \{1, \cdots, n\}$.

   Output: A delivery order such that your company's profit (the sum of p_i for the deliveries made on time) is maximized.

   Example: You have 4 deliveries with due dates: $t_1 = 2$, $t_2 = 3$, $t_3 = 1$, $t_4 = 2$. You cannot deliver all of them on time, but you can make the third delivery on day 1, then the first on day 2 and the second on day 3.

   (a) We call a subset S of the deliveries *feasible* if there is a way to order the deliveries in S so that each one is made on time. Prove that S is feasible if and only if for all days t that $t \leq n$, the number of deliveries in S due within t days is no more than t.

   (Hint: Consider sorting the deliveries in increasing order of due date.)

Solution:

We need to show that the following statements are equivalent for any subset S of deliveries:
(i) S is feasible and
(ii) for all integers $t \leq n$, the number of deliveries in S due within t days is at most t.

- We will show by contrapositive that $(i) \Rightarrow (ii)$. In other words, we will show that if for some $t \leq n$, there are more than t deliveries in S due on or before day t, then S cannot be feasible. Consider any such t. Given that we can make at most one delivery per day, it is impossible to make more than t deliveries in t days. This means that there exists at least one delivery in S we cannot make on time and thus S is not feasible.
- Now we show that $(ii) \Rightarrow (i)$ by induction on the number of deliveries in S, say k. Specifically, given (ii) we will give a schedule to make all deliveries in S before their deadline. For the base case $k = 0$, the argument is obviously correct. Suppose now there are $k + 1$ deliveries in S. Then there must exist a delivery $d \in S$ with deadline $t \geq k + 1$, otherwise the number of deliveries in S due within k days is $k + 1$, which contradicts (ii). Now consider the set of deliveries $S \setminus \{d\}$. Note that this set satisfies (ii) for the same reason that S satisfies (ii): for all integers $t \leq n$, the number of deliveries in S due within t days is at most t. Therefore $S \setminus \{d\}$ is feasible by the induction hypothesis. Thus we can first do all deliveries in S except d within k days and then finish delivery d in the $(k + 1)$-th day. This proves that S is feasible.

(b) Describe and analyze an efficient[1] algorithm to determine the order in which the deliveries should be made to maximize the profit.

(Hint: Build up a feasible set of deliveries by considering them in a particular order and adding each delivery to your schedule if the set of selected deliveries continues to remain feasible. Use part (a) to check for feasibility. What order should you consider deliveries in to maximize the profit? Use an "exchange" argument to prove the correctness of your algorithm.)

Solution:

We present a greedy algorithm that returns a feasible set of deliveries S that maximizes the profit. By part (a), completing the deliveries in S in order of their due date gives us a schedule for deliveries in S that meets all deadlines. The algorithm builds a feasible set S by including deliveries in decreasing order of points while maintaining the invariant that the set S is feasible.

Algorithm 1: Pseudocode of greedy

```
Input: number of deliveries n; each delivery's deadline t_i and payment amounts p_i.
Sort the deliveries by decreasing order of payment amount;
Let L represent this sorted list;
Initialize set S = ∅;
foreach d ∈ L do
    if S ∪ {d} is feasible then
        add d to S;
return S.
```

[1] You should aim for $O(n^2)$ or better

To test the "If" condition, we maintain an array A that keeps track of number of deliveries in S due on or before day t for all t. That is, $A[t]$ keeps track of number of deliveries in S due on or before day t. When a delivery with due date f is added to S, add 1 to $A[t]$ for all $t \geq f$.

Proof of Correctness:

By construction the set S that the algorithm outputs is feasible. We will show that S also attains maximum possible profit or in other words that S is optimal, via an exchange argument.

Let the deliveries in S listed in decreasing order of payment amount be $\{d_1, d_2 \dots d_k\}$. Suppose that there is an optimal solution M that is different from S and that j is the smallest index such that d_j is not in M. That is, assume there is an optimal solution $M = \{d_1, d_2, \dots d_{j-1}, g_j, \dots g_{k'}\}$, where $g_j, \dots, g_{k'}$ are different from d_j. Note that since M is feasible, so is its subset $\{d_1, \dots, d_{j-1}, g_j\}$. Now we observe that g_j cannot have larger value than d_j, otherwise our algorithm would have considered it before d_j and added it to our solution. Thus g_j has value less than d_j and in fact all the other g's have smaller payment amount than d_j.

We will now carry out an exchange argument. Specifically, we will exchange d_j with some other delivery g in M such that the resulting set $M \cup \{d\} \setminus \{g\}$ is both feasible and has more profit than M. We first consider the set $M' = M \cup \{d_j\}$. This set may not be feasible, however since M was feasible, it must hold that for any day t, the number of deliveries in M' due on or before day t is $\leq t + 1$. Let t_0 be the smallest t for which the number of deliveries due on or before day t is exactly $t + 1$. Note that if we delete a delivery from M' due on or before t_0, then the resultant set is feasible. The key observation here is that there must exist a delivery g, of value smaller than d_j, that is due before t_0. This is because, if the only deliveries due on or before t_0 in M' are the d_i's, then the number of deliveries due on or before t_0 must be no more than t_0 (because $\{d_1, \dots d_j\}$ form a feasible set). Therefore deleting the delivery g from M' results in a feasible set of total value greater than M. This is a contradiction, and thus S is an optimal set.

The argument in the prior case shows that we can start with an optimal set M that contains the $j-1$ deliveries $d_1, d_2, \dots d_{j-1}$, but not d_j, and transform it into a feasible set of value equal to or greater than M. Repeating this gives us an optimal solution T that contains S. Since T must be feasible, and adding any delivery to S gives a set that is not feasible, we have $T = S$. This completes the proof.

Running time analysis:

Sorting the list with respect to payment amount takes $O(n \log n)$ time. To check whether the set S is feasible at any given stage, we keep track of an array of length n in which the i-th entry stores the number of deliveries in S due on or before day i. Updating this array every time a delivery is added to S and checking whether S remains feasible requires $O(n)$ time. Since we need to do this n times, the entire algorithm requires $O(n^2)$ time. Finally sorting S with respect to due date requires $O(n \log n)$ time. Therefore the algorithm requires $O(n^2)$ time.

2. When there is more than one shortest path from a node s to another node t, it is often convenient to choose a shortest path with the fewest edges; call this a *best path* from s to t. Suppose we are given a directed graph G with positive edge weights and two nodes s and t in G. Describe and analyze an efficient algorithm to compute a best path in G from s to t.

Solution:
The algorithm for this problem is a modification of Dijkstra's algorithm. For each vertex v, besides keeping track of the tentative distance $d'(v)$ from the source s, we also keep track of the tentative number of edges $h(v)$ in the best path from the source.

As in Dijkstra's algorithm we use a priority queue, the only difference being that the key of each vertex v is now a pair of values $(d'(v), h(v))$. When comparing keys, if either $d'(v) < d'(u)$, or $d'(v) = d'(u)$ and $h(v) < h(u)$, then $(d'(v), h(v)) < (d'(u), h(u))$.

The details of the algorithm are provided below:

Algorithm 2: Pseudocode of modified Dijkstra

```
   Input: starting vertex s
 1 Initialize d'(s) := 0, h(s) := 0;
 2 For each vertex v ≠ s, set d'(v) := ∞, h(v) := ∞;
 3 For each vertex v, set prev(v) := Null;
 4 Initialize priority queue Q by inserting each vertex v with key (d'(v), h(v));
 5 while Q is not empty do
 6     u := Get(Q);
 7     d(u) := d'(u);
 8     for each edge (u, v) do
 9         if (d'(v) > d'(u) + ℓ_(u→v)) or (d'(v) = d'(u) + ℓ_(u→v) and h(v) > h(u) + 1) then
10             prev(v) := u;
11             Update key of v in Q as follows:
12             d'(v) := d'(u) + ℓ_(u→v);
13             h(v) := h(u) + 1;
```

The proof of correctness and running time analysis are identical to that of Dijkstra's algorithm, and we leave these details as an exercise.

3. Let $G = (V, E)$ be an undirected weighted graph with distinct weights w_e for every edge $e \in E$. Let $T(V, E)$ denote the edge set of the unique MST of (V, E). Let $W(T) = \sum_{e \in T} w_e$ be the weight of the edge set T.

   Let $e \in E$. Show that the following two statements are equivalent:

   1. $W(T(V, E \setminus \{e\})) > W(T(V, E))$
   2. For every cycle C in G with $e \in C$, there exists $e' \in C$ such that $w_{e'} > w_e$

Solution:
We will show each direction separately:

- 1. $\Rightarrow$ 2.

  To show this, we will show that the contrapositive holds, that is we will show that if there exists a cycle C such that e is the maximum weight edge in that cycle then $W(T(V, E \setminus \{e\})) \leq W(T(V, E))$.

  We can actually show something stronger: if e is the maximum weight edge in some cycle, then it can not be in a MST. So, $T(V, E) = T(V, E \setminus \{e\})$.

  Suppose for a contradiction that a MST T contains e. If we removed e, we would create a *cut* with two distinct sets of nodes S and $V \setminus S$. We would then use another edge e' to cross the cut, and specifically we would choose one from C, since we know $w_{e'} < w_e$ for all edges e' in C. This would give us a new spanning tree T' with $W(T') < W(T)$, which is a contradiction since we supposed T was an MST.

- 1. $\Leftarrow$ 2.

  Suppose for contradiction that $W(T(V, E \setminus \{e\})) \leq W(T(V, E))$. That means there is a MST T of G such that $e \notin T$. However, $T + \{e\}$ must contain a cycle C, and by assumption, there must be an edge $e' \in C$ such that $w_{e'} > w_e$. Then, $T' = T + \{e\} - \{e'\}$ is a tree with smaller weight than T, which contradicts the assumption that T is a MST.

CS577: Introduction to Algorithms (Fall 2024)

Discussion Divide and Conquer

Recap - Divide and Conquer

Algorithms that fit the Divide and Conquer paradigm, like MERGESORT are most often presented as recursive algorithms since the paradigm is naturally recursive. It is important to remember that recursion has no more computational power than an iteration, and that a recursive algorithm can be described iteratively, and vice-versa. For example, on the GeeksForGeeks website, you can find:

- a recursive implementation of MERGESORT (https://www.geeksforgeeks.org/merge-sort/), and
- an iterative implementation of MERGESORT (https://www.geeksforgeeks.org/iterative-merge-sort/).

Divide and Conquer Paradigm

1. *Divide:* Split the problem into smaller sub-problems.
2. *Conquer: (Recurse)* Solve the smaller sub-problems (usually through recursion).
3. *Combine: (Merge)* Combine the solutions to the sub-problems into a solution for the original problem.

Often a useful technique to improve the efficiency of the solution.

Divide and Conquer Correctness

Assuming a recursive algorithm:

Soundness: Typically strong induction on input size.

1. Show that the base case(s) is correct.
2. Assume that the recursive calls return the correct value.
3. Given the assumption above, show that the value return will be correct.

Completeness: Show that the recursive calls make progress towards the base case.

Divide and Conquer Runtime Analysis

Assuming recursive algorithm:

From Algorithm Definition to Recurrence:

The runtime for a recursive Divide and Conquer algorithm with input of size n is:

- $T(n) = \sum_i T(n_i) + f(n)$, where n_i is the size of the input to the ith recursive call, and $f(n)$ is the cost of the work done outside of the recurrences for a call of input size n.
- Don't forget the base cases!

Example:

Algorithm: DCALG

Input : A
Output: $f(A)$
if $|A| = 1$ **then return** $g(A)$
$A_1 :=$ DCALG(Front-half of A)
$A_2 :=$ DCALG(Last 1/4 of A)
$A_3 :=$ DCALG(First 1/4 of A)
return $h(A_1, A_2, A_3)$

If $g(A)$ is constant time, and $h(A_1, A_2, A_3)$ is linear time:

$$T(n) \leq T(\frac{n}{2}) + 2T(\frac{n}{4}) + cn;\ T(1) = c$$

If $g(A)$ is linear time, and $h(A_1, A_2, A_3)$ is constant time:

$$T(n) \leq T(\frac{n}{2}) + 2T(\frac{n}{4}) + c;\ T(1) = cn$$

If $g(A)$ is quadratic time, and $h(A_1, A_2, A_3)$ is quadratic time:

$$T(n) \leq T(\frac{n}{2}) + 2T(\frac{n}{4}) + cn^2;\ T(1) = cn^2$$

Solving Recurrences: For the main techniques, see Week 1 discussion.

Divide and Conquer Problems

1. You are given 2^s arrays of sorted integers. Each array has n elements. You want to combine these arrays into a single sorted array of $2^s n$ elements.

   (a) One way of doing this is to repeatedly apply the MERGE subroutine from MERGESORT. That is, first merge the 1st and 2nd array, then merge the resulting array with the 3rd array, and so on. Asymptotically, how many times will this algorithm perform a comparison of two integers? Briefly justify your answer. (*Hint*: Merging two sorted arrays of size m and n requires $O(m+n)$ pairwise comparisons of integers.)

   Solution: During each pairwise merge, one of the arrays is size n and the other is size $O(2^s n)$. So each pairwise merge takes $O(2^s n)$ comparisons. We complete $O(2^s)$ pairwise merges for a total of $O((2^s)^2 n) = O(4^s n)$ total comparisons.

   (b) Devise a divide and conquer algorithm that uses $O(2^s sn)$ comparisons of integer pairs to combine the arrays into one sorted array.

   Solution: We can apply pairwise merging in a tree fashion using the MERGE step from MERGESORT which takes two sorted arrays of size m and n and returns a combined sorted array in time $O(m+n)$. First, we pairwise merge arrays of length n until we have $2^s/2 = 2^{s-1}$ arrays of length $2n$. Then we pairwise merge these arrays until we have 2^{s-2} arrays of length $4n$. We continue this process until we have 1 array of length $2^s n$. We return this array. We can define this recursively as follows:

   Algorithm: COMBINE($A_1, A_2, \ldots A_{2^s}$): returns a single sorted array that combines the 2^s input arrays, each of size n

   ```
   if 2^s = 1 then
       return A_1.
   else
       for i = 1 ... 2^{s-1} do
           B_i = MERGE(A_i, A_{2^{s-1}+i})
       return Combine(B_1, B_2, ... B_{2^{s-1}})
   ```

   (c) Use induction to prove that your algorithm correctly combines the 2^s arrays into one sorted array.

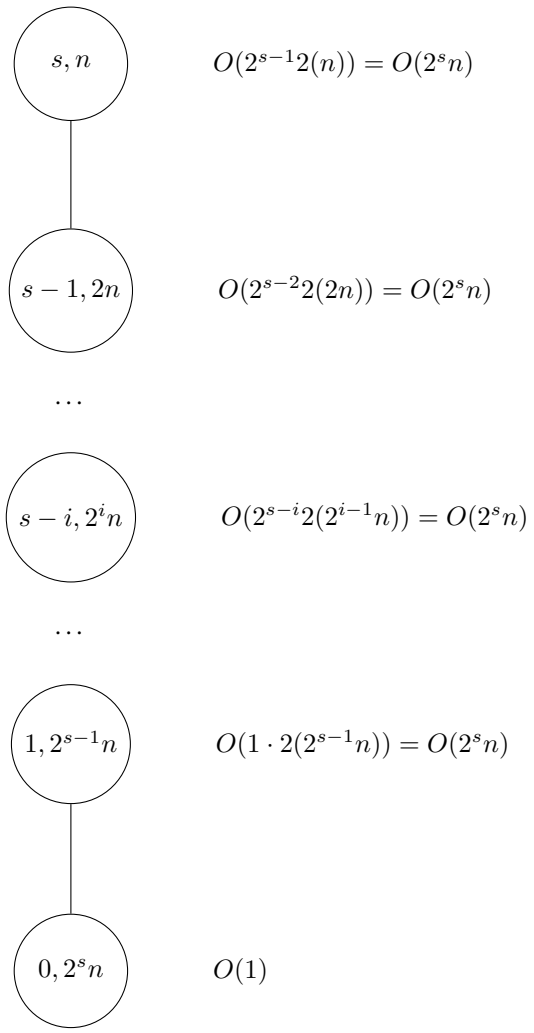
   Solution: We will use induction to prove that COMBINE correctly combines 2^s arrays of size n into one sorted array.

   We will induct over s.

   - Base Case: $s = 0$ In this case there is only one array that is already sorted. COMBINE would simply return this sorted array, which is correct.
   - Inductive Hypothesis: COMBINE correctly combines 2^k sorted arrays into one sorted array.
   - Inductive Step: $s = k+1$ COMBINE first merges 2^{k+1} arrays of size n. By the correctness of MERGE, each of these initial merges results in a combined sorted array of size $2n$. At this point we have $2^{k+1-1} = 2^k$ sorted arrays of size $2n$. We can apply the inductive hypothesis on this recursive call since we have 2^k arrays. We know that these 2^k arrays can be correctly combined, meaning that our original 2^{k+1} arrays are correctly combined.

(d) Write a recurrence as a function of n and s for the number of times your algorithm performs a comparison of two integers. State the solution of your recurrence in asymptotic notation. No explanation is required.

Solution: Let $T(s, n)$ denote the number of comparisons made by COMBINE given 2^s arrays of size n. At the first level of recursion, COMBINE makes $O(2n)$ comparisons for 2^{s-1} pairwise merges, resulting in $O(2^{s-1}2n) = O(2^s n)$ local comparisons. We then make a recursive call to COMBINE, inputting 2^{s-1} arrays of size $2n$. This results in the following recurrence: $T(s, n) = T(s-1, 2n) + O(2^s n)$. The recurrence tree is as follows, where the nodes represent recursive calls and the equations to the right of the nodes represent the amount of local work done at that level of recursion. Note that the last level does no comparisons because it is the base case.



Note that there are $s-1$ levels of the recursion tree excluding the base case. At each level, $O(2^s n)$ local comparisons are completed. This means that in total, COMBINE performs $O(2^s sn)$ comparisons.

2. You are given an $n \times n$ matrix A with unknown values. Your goal is to find an entry that is a local maximum with the smallest numberof queries made to individual entries. An entry $A_{i,j}$ is a local maximum if it is not smaller than any of its neighbors (top, bottom,left or right if they exist) i.e. $A_{i,j} = \max\{A_{i,j}, A_{i-1,j}, A_{i,j-1}, A_{i+1,j}, A_{i,j+1}\}$. You only need to finda local maximum, not the global maximum. Note that there may be multiple local maxima.

   Try out your strategy at the following link (solve this version in 90 or fewer queries): `https://tzamos.com/pf.html?N=30&M=30`.

   (a) Design an algorithm to find a local maximum in $O(n)$ number of queries. *(Hint: Start by thinking about a strategy for a 1D array, and then extend it to the 2D setting.)*

Solution: To find a local maximum of an array (or a line), a simple solution is to compare the middle element with its neighbors. If the middle element is not smaller than any of its neighbors, then we return the middle element. Since the middle element is larger than the neighbors, it is a local maxima and the algorithm ends. If the middle element is smaller than its left neighbor, then there exists at least one local maxima in left half. Thus, we recurse on the left half of the array. Else, there is at least one local maxima on the right half. So we recurse on the right half of the array. The subproblem analyzes half of the initial array, so the recursive term is $T(\frac{n}{2})$, and the initial step checking if the middle element is the local maxima take constant, O(1), time. Therefore, the runtime is $T(n) = T(\frac{n}{2}) + O(1) = O(\log(n))$.

A first attempt at an algorithm for this problem is to simply start at any entry and repeatedly move to larger-valued neighbor entries until we reach a local maxima. This algorithm does work. It terminates because the values of the nodes that we visit are strictly increasing, so we cannot visit the same node twice. Furthermore, we stop when we can no longer find a larger-valued neighbor, meaning that termination occurs when we've reached a local maxima. However, this algorithm is not $O(n)$.

Another attempt is to simply recurse on a portion of the grid, since any subgrid must have at least one local maxima (for instance, the subgrid's global maxima). This doesn't quite work either, since it's possible that a local maxima of a subgrid won't be a local maxima of the whole grid. This can occur if the local maximum of the subgrid appears along its perimeter, since it's then possible for that entry to have neighbors in the grid Diwhich don't exist in the subgrid.

We can draw inspiration from the above two ideas to construct a working $O(n)$ algorithm. We can use divide-and-conquer to examine a increasing sequence of grid values in gradually smaller portions of the grid while avoiding problems that arise from subgrid perimeter values. Our grid local maximum algorithm, $GLM(G)$, takes as input an $n \times n$ grid G and returns a local maximum of G.

$GLM(G)$:

(a) Manually find the maximum value in G which is either on the perimeter or in the middle row or middle column. (If n is even, search through both middle rows and both middle columns.)

(b) Check the neighbors of this maximum value node to determine whether it is a local maximum in the grid. If it is, then we are done. Otherwise, move to step (iii).

(c) Since the node found in step (i) is not a local maximum, it must have a neighbor of greater value. Find such a neighbor

(d) The middle row and column chosen in step (i) divide the grid into four "quadrants", and the neighbor found in step (iii) lies in one of these quadrants. Recurse on the quadrant containing

this neighbor, including adjacent nodes from the perimeter, middle row, and middle column of the whole grid. Return the result of this recursive call.

In the grid below, we search through the middle row, column, and perimeter and find that the maximum value among those cells is 72.

14	40	5	28	20	3	30
33	24	10	46	4	26	16
47	12	19	31	35	81	44
6	39	11	27	42	72	37
32	36	13	25	18	49	15
41	22	48	45	9	29	21
23	7	43	8	34	17	38

If 72 were a local maximum, we would be done. In this case, the cell with value 72 has a larger neighbor— namely, 81—so we recurse on the quadrant of the grid containing 81, including the gray cells from the original grid.

28	20	3	30
46	4	26	16
31	35	81	44
27	42	72	37

(b) Prove the correctness of your algorithm.

Solution: One way to prove the correctness of this algorithm is induction. For n ≤ 3, every grid node is along the perimeter or in the middle row or column. The algorithm therefore checks every grid node and finds the global maximum. The global maximum of a grid is necessarily a local maximum.

Now assume inductively that the algorithm works for any grid of dimension less than n. We must prove that the algorithm also works for a grid of dimension n, where $n \geq 4$. Let G be our grid, and let G' be the subgrid in the first recursive call. Since G' has dimension less than n, we know by our hypothesis that the algorithm correctly finds a local maximum of G' . If this local maximum is also a local maximum in G, then we are done.

We can only run into trouble if the local maximum we find in G' is not a local maximum of G. We will now prove that this cannot happen. Let u be the local maximum found in G' , and suppose by way of contradiction that u is not a local maximum of G. This means that u must have a greater-valued neighbor in G which is not in G' , i.e., u must be along the perimeter of G' . Every perimeter node of G' was one of the perimeter or middle row/column nodes in G, so u was one of the nodes checked in step (i) when the algorithm was run on G. Note that u could not have been the maximum value node found in step (i); if it had been, then we would have moved to one of its larger-valued neighbors and recursed on that quadrant of G. Since u has no larger-value neighbors in G' , our recursive call would have been performed on another quadrant instead of G' . Therefore, when performing step (i) with G, we found another node v with value larger than u.

We now proceed by making two observations. First, u must be present in the subgrid for each recursive call. Otherwise, it will be eliminated from consideration somewhere along the way and cannot be returned at the end of the algorithm. Second, the values of the maxima found in step (i) will not decrease as we descend through the recursive calls. The maximum node found in step (i) lies along the perimeter of the subgrid on which we recurse, so when we perform step (i) in the recursive call, we will either find that that node is still the maximum or else we will find a node with an even larger value. In

either case, the value of the step (i) maximum does not decrease.

These observations lead us to a contradiction once the recursive calls of the algorithm reach one of our base cases. From the first observation, we know that u is present in this final subgrid. From the second observation, we know that the maximum node value in this grid is at most the same as the value of v, which itself is larger than the value of u. Thus, u does not have the maximum value in this final subgrid. The algorithm will return the global maximum of this subgrid, which will not be u. But this means that the algorithm does not return u, and so we obtain a contradiction. This completes the proof of correctness.

(c) Analyze the time complexity of your algorithm by finding recurrence for the number of queries and solving it. You don't need to show your work for this part.

Solution: The number of nodes along the grid perimeter is $4n - 4$, and the number of additional nodes along the middle row(s) and column(s) is at most $4n - 12$. (When n is odd, the number of middle row/column nodes is instead about $2n - 5$.) This means that the number of values we must check in step (i) is at most $8n - 16$. The maximum value node we find in this step has at most 4 neighbors, and we must check at most all of them. We conclude that the number of nodes checked in steps (i) through (iii) is at most $(8n - 16) + 4$ which is bounded by $8n$.

Roughly speaking, the dimension of the grid is halved with each recursive call. Using the above reasoning repeatedly and tracing through the algorithm's recursive calls, we compute that the total work for the algorithm is roughly $8[n + (n/2) + (n/4) + ...]$. This is bounded above by $16n$ and is therefore $O(n)$. This leads us to guess that $T(n) \leq cn$ for some constant c.

If n is odd, the dimension of the subgrid is $\frac{n+1}{2}$; else, if n is even, then the dimension of the subgrid is exactly $\frac{n}{2}$. In either case, the subgrid in the recursive call has dimension at most $\frac{n+1}{2}$. This means that a recurrence for this algorithm's runtime is

$$T(n) \leq T(\frac{n+1}{2}) + 8n$$

Substituting our runtime guess into the recurrence, we can find a constant c for which $T(n) \leq cn$:

$$T(n) \leq T(\frac{n+1}{2}) + 8n$$

$$\leq c\,(\frac{n+1}{2}) + 8n$$

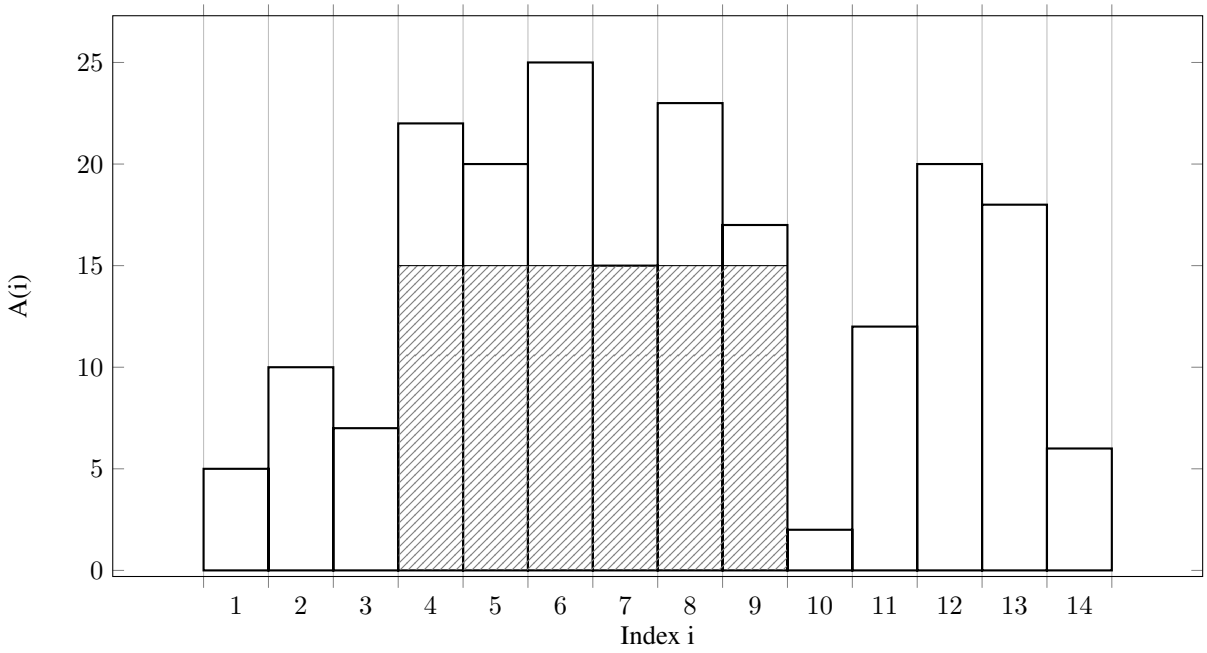
$$\leq (\frac{c}{2} + 8)n + \frac{c}{2}$$

To obtain $T(n) \leq cn$, we therefore want

$$(\frac{c}{2} + 8)n + \frac{c}{2} \leq cn$$

There are many values of c which satisfy this inequality. We only need to show that one such c exists, and we can afford to be lazy in our selection. For example, $c = 30$ satisfies the inequality for any $n \geq 4$. This is sufficient, since the recurrence only applies for $n \geq 4$ (the values $n \leq 3$ are our base cases). It is easy to verify that $T(n) \leq 30n$ for $n = 1, 2, 3$ too, and so we conclude that $T(n) = O(n)$.

3. You are given an array $A[1, \ldots, n]$ of positive integers. Let $m(i, j) = \min_{i \le k \le j} A[k]$. Design an $O(n \log n)$ algorithm that finds the largest value of $m(i, j) \cdot (j - i + 1)$, where i and j are integers such that $1 \le i \le j \le n$. Intuitively, you are trying to maximize the area of a rectangle that fits inside the histogram given by A, namely the rectangle with base $[i, j]$ on the x-axis and height $m(i, j)$. See visualization below:



Solution: We are looking for the maximum area of an axis-parallel rectangle that fits inside the histogram defined by a given array $A[1, \ldots, n]$. For short, we will refer to such rectangles simply as "rectangles inside $A[1, \ldots, n]$". We solve this problem via divide and conquer. **The base case** is when $n = 1$, in which case the answer is $A[1]$. In all other cases, we recurse as follows:

1. Split the array into two halves: the left subarray $L = A[1, \ldots, p]$ and the right subarray $R = A[p + 1, \ldots, n]$ where $p = \lfloor \frac{n}{2} \rfloor$.
2. Recursively solve the problem for L and for R. Call those solutions a_L and a_R, respectively.
3. Find the maximum area a_c of rectangles inside $A[1, \ldots, n]$ that cross the boundary between L and R.
4. Output the maximum of a_L, a_R, and a_c.

For the base case when $n = 1$, the single rectangle $A[1]$ is the largest that fits under A. For the recursive cases, note that the rectangles considered in steps 2 and 3 cover all the possibilities. The critical step is to execute step 3 efficiently. If we manage to do that in time $O(n)$, the resulting algorithm runs in time $O(n \log n)$. as it follows the same recursion pattern as Merge Sort.

In order to execute step 3, we use a two-pointer approach that is reminiscent of the one we used in class for merging two sorted arrays. We use a pointer ℓ to a position in L and a pointer r to a position in R. We also keep track of a variable a for the maximum area of a rectangle inside $A[\ell, \ldots, r]$ that crosses the boundary between L and R. Finally, we maintain the following invariant:

- No rectangle inside $A[1, \ldots, n]$ that crosses the boundary between L and R that falls inside $A[\ell, \ldots, r]$ can have height more than $m = \min_{\ell \leq k \leq r} A[k]$.

We initialize ℓ to the right end of L, and move it towards the left. Similarly, we initialize r to the left end of R, and move it towards the right. The invariant holds trivially for the initial settings, and $a = 2m$.

Until one of L or R is exhausted, in each step we look at $A[\ell - 1]$ and $A[r + 1]$.

- If at least one of those values is no smaller than m, we move the corresponding pointer (always breaking ties arbitrarily). The value of m does not change.
- If both values are less than m, then there are no rectangles that have height m or more and either start before ℓ or end after r. The largest height that rectangles crossing the boundary between L and R while not being contained in $A[\ell, \ldots, r]$ can have is $\max(A[\ell - 1], A[r + 1])$. We move the pointer of the part that realizes this maximum, updating m and maintaining the invariant.

In both cases, by the invariant, the only rectangle we need to consider for updating a is the one with basis the new $[\ell, r]$ and height the new m.

If one of L and R is exhausted, we move the other pointer and update the variables consistent with the two cases above. Note each iteration involves a constant amount of work, and moves one of the pointers further towards the end of the array. It follows that our algorithm for step 3 runs in time $O(n)$ as desired.

4. You have a backpack that can hold up to k pounds and can be filled with pixie dust or dragon scales. The value of i pounds of pixie dust is determined by the sequence a_i, while the value of j pounds of dragon scales is determined by the sequence b_j. Given a weight limit k and sequences $a_0 \dots a_n$ and $b_0 \dots b_n$, your goal is to calculate the maximum value you can obtain by filling the backpack with some combination of pixie dust and dragon scales.

   Note that in some cases the rate of increase in an item's value decreases the more you have of it. Such a sequence s_i is called *concave* and satisfies that $s_i - s_{i-1}$ is a decreasing function of i.

   Part 1: For the first part of this question we will assume that both sequences a and b are concave.

   (a) Prove that for any k, the sequence $y_i = a_i + b_{k-i}$ for $0 \leq i \leq k$ is concave.

   Solution: To show that $y_i = a_i + b_{k-i}$ is concave, we first note the definition of concave which is $s_i - s_{i-1} \geq s_{i+1} - s_i$. From this definition let us consider $y_i - y_{i-1} = a_i - a_{i-1} + b_{k-i} - b_{k-i+1}$. Since a is concave, we then have that $a_i - a_{i-1} \geq a_{i+1} - a_i$. Additionally, since b is concave, we have that $b_{k-i+1} - b_{k-i} \leq b_{k-i} - b_{k-i-1}$ which implies via negation that $-b_{k-i+1} + b_{k-i} \geq -b_{k-i} + b_{k-i-1}$. Adding these two inequalities, we get that $a_{i+1} - a_i + b_{k-i} - b_{k-i+1} \geq a_{i+1} - a_i + b_{k-i-1} - b_{k-i}$ which is just the expansion of $y_i - y_{i-1} \geq y_{i+1} - y_i$. Hence, y is concave.

   (b) The maximum total value of the items in your backpack defines a sequence v_k as a function of the capacity k, where $v_k = \max_{i \in [0,k]} a_i + b_{k-i}$. Provide a divide-and-conquer algorithm based solution to compute v_k which has a running time of $O(\log k)$ along with a brief (2-3 lines) proof of correctness.

   Solution: We can consider the sequence of numbers $y_i, i = 1, 2, 3, \cdots n$. Our goal is then to find the i such that it maximizes y_i. Since y is concave with respect to i, it has one local maximum which is also a global maximum, and we can find this efficiently as follows. Let us consider the question of whether the maximum value is in the first half of numbers, or the second half. We can determine this with three queries. We query the center value, and its two neighbors. From this we have the following cases. Consider the left value being largest. This would imply that the maximum must lie in the first half, since going right only decreases the value of y. This logic also applies to the right most value. Now, if the center value is the largest, we have the maximum value, since going right or left only decreases the value of y. Upon eliminating half of the array, we can then recurse on the remaining half. This algorithm is essentially the same as problem 2(a) (which itself is very similar to binary search), and the correctness here follows in much the same way.

   Part 2: In the second part we will consider the scenario where only sequence *b* is concave and *a* is not.

   Show that in such a situation computing v_k for a fixed value of k takes $\Omega(k)$ time in the worst-case. Argue this by noting that the corresponding function $y_i = a_i + b_{k-i}$ can result in an arbitrary unsorted sequence and that finding the maximum element in an unsorted list of length k takes $\Omega(k)$ time as one needs to look at all the elements.

   Solution: We consider a scenario where b is just a sequence of 0s (which is trivially concave); that is, $b_{k-i} = 0$ for all choices of i. Let a be an unsorted sequence. Any algorithm that finds v_k inherently finds the maximum value in a, and, since finding the maximum value in an unsorted list takes time $\Omega(k)$, any algorithm that finds v_k must take at least $\Omega(k)$ time.

CS577: Introduction to Algorithms (Fall 2024)

Discussion Dynamic Programming Solutions

Recap – Dynamic Programming

Problems that can be effectively solved by dynamic programming are:

- Problems that can be broken up into at most a polynomial number of subproblems that can be aggregated into a final solution efficiently.
- Typically a recursive solution that has a polynomial number of unique recursive calls which allows for memoization.

Dynamic Programming Paradigm

Step 1 - Recursive Solution: Develop a recursive solution for the problem. For many dynamic program solution, the recursive solution will be inefficient with an exponential number of recursive calls, but the number of unique calls is polynomial.

E.g.: nth Fibonacci number: $f(n) = f(n-1) + f(n-2); f(2) = 1; f(1) = 1;$. An exponential number of recursive calls, but only n unique calls.

Step 2 - Memoize: Means "write it down and remember". Dynamic programs are defined recursively, but use a table/array/matrix to record the values of the recursive calls. Calculate once, record, and lookup on future calls.

E.g.: For the nth Fibonacci number, we can use a 1-D table to record the values from 1 to n. That would take the complexity down to a linear number of calculations from an exponential number of calls.

Important Notes:

- The memoization table for a dynamic program can any number of dimensions. We will see mostly 1-D and 2-D solutions, but there is nothing preventing 3-D, 4-D, or more.
- Determining the dimensions of memoization table: This will usually correspond to the explicit parameters of your recursion solution. In other words, the parameters that characterize the unique recursive calls, i.e., the degrees of freedom.
- There is always a context for the values stored in the table: You should always be able to describe the semantics of the value in a cell of the memoization table.
- For some problems, the table of values may represent a data structure like a tree.

Presenting Your Dynamic Program Solution

For dynamic programs, we do not want to be pseudocode. Rather, we want you to describe the details of solution within the context of the program.

Definitions: Describe all the definitions and preprocessing needed for the dynamic programming solution.

E.g. Weighted Interval Scheduling (WIS):

- Preprocessing: Sort the input by finish time.
- Definitions: For a given job at index j, let $i_j < j$ be the largest index such that $f_i \leq s_j$.

Matrix/table description: Describe the dimensions of the matrix, the contents of a cell, and any cells that can be initialized.

E.g. WIS: A 1D array M, where $M[j]$ is the maximum value of a compatible schedule for the first j items in the sorted input. Initialize $M[1] = v_1$.

Bellman Equation: The recursive definition for populating a cell in the table.

E.g WIS: $M[j] = \max\{M[j-1], M[i_j] + v_j\}$.

How to populate: Describe the order in which the cells of the matrix will be populated.

E.g. WIS: Populate from 2 to n.

Solution: Describe where to find the solution in the matrix, or how to calculate the solution.

E.g. WIS: The maximum value of a compatible schedule for the n jobs is found at $M[n]$.

Dynamic Program Correctness

Soundness: (Strong) induction over the order of the population of cells of the solution matrix:

1. Show that the base case(s) is correct.
2. Assume that the recursive calls return the correct value.
3. Given the assumption above, show that the Bellman equation will calculate the correct value.

Completeness: Usually immediate from the definition of the dynamic program.

Dynamic Program Runtime Analysis

- Preprocessing time, and
- Time to populate the matrix: Number of cells $\times$ Time to calculate Bellman equation, and
- Time to calculate the final solution.

E.g. WIS: $O(n \log n)$

- Preprocessing: $O(n \log n)$
- Populate the matrix: $O(n) \times O(\log n) = O(n \log n)$
- Final solution: $O(1)$

Problems

1. For a sequence of n numbers $a_1, \ldots, a_n$, let $o_{1,2}, o_{2,3}, \ldots, o_{i,i+1}, \ldots, o_{n-1,n}$, be a sequence of $n-1$ operators where $o_{i,i+1}$ is the operator placed between a_i and a_{i+1}.

   Consider the problem where you are given the number sequence $a_1, \ldots, a_n$, but the choices of these $n-1$ operators and positions of parentheses are both undetermined. Moreover, you can set each of the $n-1$ operators to be either $+$, $-$, or $\times$. Your goal is to find the **maximum** possible value the numerical expression

   $$[a_1]o_{1,2}[a_2]o_{2,3}\ldots[a_{n-1}]o_{n-1,n}[a_n]$$

   can evaluate to when you fill in $+$, $-$, or $\times$ for these $n-1$ operators and insert pairs of parentheses. Note that you should not change the ordering of numbers in $a_1 ... a_n$ but you may give precedence to some operations with appropriate parenthesizing. For example, the following number sequence of

   $$[-3] \; ? \; [-4] \; ? \; [-5]$$

   can achieve a maximum value of 35 by filling in $o_{1,2}$ to $o_{2,3}$ and adding parentheses in the following way:

   $$\big([-3] + [-4]\big) \times [-5]$$

   (a) Clearly state the subproblems you will use to solve this problem.

   (b) Give a dynamic programming algorithm to solve this problem. That is, describe your algorithm by including a clear statement of your recurrence, any necessary base case(s), and your final output. You do not need to include any pseudocode.

   Solution:

   For $j \geq i$, define $\mathsf{MaxValue}(i, j)$ as the maximum value possible achieved using the subsequence of numbers $a_i \ldots a_j$ and operators $o_{i,i+1} \ldots o_{j-1,j}$. In the case that $i = j$, there is no operator and only the number a_i. Similarly, define $\mathsf{MinValue}(i, j)$ as the minimum value possible. Recall that for dynamic programming we will be computing these values iteratively (not recursively) by using a memoization matrix. So we will initialize an $n \times n$ $\mathsf{MaxValue}$ table and an $n \times n$ $\mathsf{MinValue}$ table, where $\mathsf{MaxValue}(i, j)$ corresponds to the cell at the i-th row and j-th column in the $\mathsf{MaxValue}$ table.

   First, we establish the base cases. When $i = j$, since there is no operator and no parentheses to place, clearly $\mathsf{MaxValue}(i, i) = a_i$ and $\mathsf{MinValue}(i, i) = a_i$. (We can also initialize the cases when $i > j$ but this would be unnecessary since we will never reach these cases per our algorithm.)

   We now present the following recursive relationship (or Bellman equation) for iteratively computing $\mathsf{MaxValue}$ and $\mathsf{MinValue}$. For $j > i$,

   $$\mathsf{MaxValue}(i, j) = \max_{i \leqslant k < j} \begin{cases} \mathsf{MaxValue}(i, k) + \mathsf{MaxValue}(k+1, j) \\ \mathsf{MaxValue}(i, k) - \mathsf{MinValue}(k+1, j) \\ \mathsf{MaxValue}(i, k) \times \mathsf{MaxValue}(k+1, j) \\ \mathsf{MaxValue}(i, k) \times \mathsf{MinValue}(k+1, j) \\ \mathsf{MinValue}(i, k) \times \mathsf{MaxValue}(k+1, j) \\ \mathsf{MinValue}(i, k) \times \mathsf{MinValue}(k+1, j) \end{cases} \tag{0.1}$$

$$\mathsf{MinValue}(i,j) = \min_{i \leqslant k < j} \begin{cases} \mathsf{MinValue}(i,k) + \mathsf{MinValue}(k+1,j) \\ \mathsf{MinValue}(i,k) - \mathsf{MaxValue}(k+1,j) \\ \mathsf{MaxValue}(i,k) \times \mathsf{MaxValue}(k+1,j) \\ \mathsf{MaxValue}(i,k) \times \mathsf{MinValue}(k+1,j) \\ \mathsf{MinValue}(i,k) \times \mathsf{MaxValue}(k+1,j) \\ \mathsf{MinValue}(i,k) \times \mathsf{MinValue}(k+1,j) \end{cases} \tag{0.2}$$

Thus, each recursive relationship requires considering 6 cases for each k. Computing the values for these tables should be done in a diagonal fashion, where we iterate from $|j-i| = 1$ to $n-1$. After computing the table of MaxValue and MinValue, the algorithm should return the answer at $\mathsf{MaxValue}(1,n)$.

(c) Prove that your algorithm is correct.

Solution:

We shall approach the proof of correctness using induction in a similar spirit of using induction in divide and conquer. Note that our recursive relationship uses smaller subproblems to compute the max and min values where "smaller" here refers to the smaller size intervals. Thus, we will induct on the size of the interval $j - i$ to prove that $\mathsf{MaxValue}(i,j)$ is the maximum value possible for the subsequence $a_i, \ldots, a_j$. We will also simultaneously prove that $\mathsf{MinValue}(i,j)$ is the minimum, as it is necessary to couple MaxValue and MinValue in the same induction because they are recursively dependent on each other.

1. **Base case**: $j - i = 0$
   When $i = j$, the maximum should simply be a_i as no operations are involved. The base cases we defined in our algorithm, $\mathsf{MaxValue}(i,i) = a_i$, reflect this observation, so $\mathsf{MaxValue}(i,i)$ is the maximum value possible. The same logic applies for $\mathsf{MinValue}(i,i)$.
2. **Inductive Hypothesis**: $j - i \leq k$
   Assume that all sub-problems $\mathsf{MaxValue}(i,j)$, where $j - i \leq k$, are indeed the maximum value possible for the subsequence $a_i, \ldots, a_j$. Assume the analogous for $\mathsf{MinValue}(i,j)$.
3. **Inductive Step**: $j - i = k + 1$
   The intuition to find $\mathsf{MaxValue}(i,j)$ or the maximum value possible achieved with numbers a_i through a_j is that we can try inserting parentheses at all possible positions and experimenting with $+, -$,and $\times$ to concatenate parenthesized sub group of numbers. In other words, with memoization, we are trying to concatenate

   $$\begin{cases} \mathsf{MaxValue}(i,i) \text{ and } \mathsf{MaxValue}(i+1,j) \text{ with } +, -, \text{ and } \times \\ \mathsf{MaxValue}(i,i+1) \text{ and } \mathsf{MaxValue}(i+2,j) \text{ with } +, -, \text{ and } \times \\ \ldots \\ \mathsf{MaxValue}(i,j-1) \text{ and } \mathsf{MaxValue}(j,j) \text{ with } +, -, \text{ and } \times \end{cases}$$

   However, since the multiplication of two negative numbers can yield a positive number, we should also consider the case where we are concatenating the minimum value of two sub-expressions with $+, -$,and $\times$, and also the case where we are concatenating the min value of one sub-expression and the max value the other. The main reason why we only

consider the operations between maximum and minimum values (and no value in between) is because of the following fact.

Claim 1 *Consider $a_1 \in [l_1, r_1]$ and $a_2 \in [l_2, r_2]$ and applying a binary operation in that order. The maximum sum possible is $r_1 + r_2$ and the maximum difference is $r_1 - l_2$. The maximum product is $\max\{l_1 l_2, l_1 r_2, r_1 l_2, r_1 r_2\}$.*
Proof: It is clear that the maximum sum should be adding the maximum of a_1 and a_2. Similarly, the maximum difference is straightforward. The maximum product cannot be anything other than the product of the extreme values of the intervals. This is because, if the maximum product were $a_1 a_2$ where $a_1 \neq l_1, r_1$, then you can always increase or decrease a_1 so that the product increases. ■

Due to the claim above, for a fixed middle-index k, we only need to consider 6 cases in total (1 for $+$, 1 for $-$, and 4 for $\times$), so we do not have to perform all the concatenations we did above, such as $\mathsf{MaxValue}(i, k) + \mathsf{MinValue}(k, j)$ and etc.

Thus, $\mathsf{MaxValue}(i, j)$, where $j - i = k + 1$, is correctly computed by considering all the cases as in part (a), since all sub-problem calculations of interval size $\leq k$ are correct by the inductive hypothesis. The proof for the correctness of $\mathsf{MinValue}(i, j)$ follows similarly.

(d) Analyze the run-time of your algorithm.

Solution: We can analyze the time complexity for the dynamic program as follows. The runtime is equal to the number of unique sub-problems $\times$ the time complexity per sub-problem, which is $O(n^2 \cdot n) = O(n^3)$.

2. After running the teleportation delivery company *Algo Express* for many years, you get fired for being "not consistently candid" with the board. You leave the company to start a new venture (*DPAlgo Express*) that can process very big delivery orders. In particular, each order now takes several days for the teleportation machine to complete.

   Suppose on a certain day, n customers give you packages to deliver. Each delivery i should be made within d_i days, takes t_i days to deliver, and the customer pays you p_i dollars for doing it on time (if you don't do it on time, you get paid 0 dollars). On-time delivery means that if package i is due within $d_i = k$ days, the delivery should be completed on or before day k to be on time (that is, it should start on or before day $k - t_i$). As before, your teleportation machine can only make one delivery at a time.

   Input: A set of n deliveries with due dates $d_i \in \mathbb{N}, d_i \geq 1$, number of days needed for delivery $t_i \in \mathbb{N}, t_i \geq 1$ and payments $p_i > 0$ for each delivery $i \in \{1, \cdots, n\}$.

   Describe and analyze an efficient algorithm to determine which deliveries to make and in what order so as to maximize your profit. (Note: unlike the previous version of the problem, deliveries may now take more than one day). Your algorithm should have a pseudo-polynomial running time – running time polynomial in n and T, where T is the latest deadline among all deliveries.

   Solution:
   We will give a dynamic program which constructs the optimal schedule for deliveries in a manner similar to the knapsack algorithm. That is, we build up solutions to sub problems restricted to the first i items, then increment i. For the knapsack problem, the order in which we considered items didn't matter. Here, however, the order in which the deliveries are made matters, so we must carefully consider the order in which we choose to do these deliveries.

   Consider any schedule of k deliveries $\sigma = (a_1, a_2, ..., a_k)$, where a_1 is the the first delivery being made and a_k is the last. For some delivery i, let d_i be the deadline and t_i be the amount of days required to deliver it. We use an exchange argument to show that the deliveries in σ can be made in increasing order of deadlines: suppose, for some index i, that $d_i > d_{i+1}$. Then we can swap the order of delivery i and delivery $i+1$. This is because delivering package $i+1$ before package i has no effect on $i+1$, as $(i+1)'s$ delivery just starts earlier. Also, if the earlier schedule was feasible then $t_i + t_{i+1} \leq d_{i+1}$ but since $d_i > d_{i+1}$ even for i the new delivery schedule is still feasible. In particular, an optimal schedule can be so arranged that the packages with earlier deadline are delivered before packages with later deadlines. In the rest of the solution, we assume the packages are numbered in order of increasing deadlines. That is, $d_1 \leq d_2 \leq ... \leq d_n$.

   We now try to come up with a recurrence for the problem. Lets consider some package i and some start day s. Suppose we want to find a delivery schedule from package i onwards, and with a starting date of s, such that we maximize our profit. We have two choices for package i, we can either deliver it or leave it. Suppose we decide to deliver this package, we then gain a profit of p_i but we have spent t_i days to deliver it. In this case, when we are considering package $i+1$, we can only start deliveries at day $s + t_i$. If we decide to not deliver package i, then we have not gained any profit, but we can start delivering package $i+1$ and onwards from day s. The above can be used to come up with a recurrence equation containing i and s, where i represents the package under consideration and s represents how many days have passed since the first day the next delivery can start. The value of i goes from 1 to n and the value of s goes from 0 to T (where T is the max

deadline of any package). We address one final issue in our recurrence relation: Depending on the deliveries made before i we might be at a start date s, from which there is no way to deliver package i before its deadline. In such a case, there is no profit in delivering this package and we will simply choose not to deliver it. This leads to the following recurrence below.

$$\mathsf{OPT}[i,s] = \max \begin{cases} \mathsf{OPT}[i+1,s] \\ \mathsf{OPT}[i+1,s+t_i] + p_i, \text{only if } s + t_i \leq d_i \end{cases} \tag{0.3}$$

Note that n is the number of packages and T is the latest deadline among all packages. Now, suppose we have a 2d-array OPT of size $(n+1) \times (T+1)$. We can see from the dependencies of the above equations that our base cases are when $i = n+1$, and we set $\mathsf{OPT}_1[n+1,s] = 0 \ \forall s$ such that $0 \leq s \leq T$. Therefore, considering these dependencies, we can compute the other values in this 2d-array by proceeding through each row that is adjacent to the previously computed row (starting with the base case bottom row). Our solution is then in $\mathsf{OPT}_1[1,0]$, the maximum value can be obtained by considering packages 1 to n and starting on day 0.

The run time of our algorithm is then equal to the time taken to fill in the array. Since the size of the array is $(n+1) \times (T+1)$ the runtime is $O(nT)$.

3. Describe and analyze an efficient algorithm to determine the length of the longest common subsequence across 3 strings. The algorithm should be polynomial in input sizes. A subsequence is a sequence that appears in the same relative order but is not necessarily contiguous. For example, "abc", "abg", "bdf", "aeg", '"acefg", .. etc are subsequences of "abcdefg". The input will be three distinct strings $s1, s2, s3$ with lengths $n1, n2, n3$ respectively. The output will be a single number for the length of the subsequence. For example, the longest common subsequence of the 3 strings "**GGCA**CC**ACG**", "AC**GGC**GG**A**T**ACG**", and "TA**GG**T**C**T**AAC**T**G**" is "GGCAACG".

Solution:
We will construct a dynamic programming algorithm to build the longest common sub-sequence by processing each of the 3 input strings - character by character. Let's denote the i^{th} character of the strings as $s1_i, s2_i, s3_i$ respectively. Note also that this dynamic programming problem will have a 3D solution matrix (and recurrence) because we are processing three variable inputs. We start processing the characters of each string and comparing them to determine whether they will make it to the longest common sub-sequence, increasing the resultant length when they do. When we reach the i^{th}, j^{th}, k^{th} characters of $s1, s2, s3$ respectively, there are a few possibilities. However, $s1_i$ is guaranteed to be in any common sub-sequence if and only if for some j and some k, $s1_i = s2_j$ and $s1_i = s3_k$ and the sub-sequence preceding these characters are also common to all 3 strings. In this case, the length increases by 1. If not, we explore the other possible combinations and the sub-sequences they might lead to. These originate from excluding 1 of the three characters.

Let us define $\mathsf{OPT}(i, j, k)$ as the longest common sub-sequence up till $s1_i, s2_j$ *and* $s3_k$ characters. Then the recurrence relation is as follows.

$$\mathsf{OPT}(i,j,k) = \begin{cases} \mathsf{OPT}(i-1, j-1, k-1) + 1 & s1_i = s2_j = s3_k \\ max \begin{cases} \mathsf{OPT}(i-1, j, k) \\ \mathsf{OPT}(i, j-1, k) \\ \mathsf{OPT}(i, j, k-1) \end{cases} & otherwise \end{cases} \tag{0.4}$$

Now, suppose we have a 3-dimensional array OPT of size $(n_1 + 1) \times (n_2 + 1) \times (n_3 + 1)$. We can see from the dependencies of the above equations that our base cases are when $i = 0$, $j = 0$ or $k = 0$, so we set $\mathsf{OPT}_1[i, j, 0] = 0\ \forall i, j$, $\mathsf{OPT}_1[i, 0, k] = 0\ \forall i, k$, and $\mathsf{OPT}_1[0, j, k] = 0\ \forall j, k$. There are multiple ways to populate the rest of OPT. Let $\{(i, j, k) | 1 \leq i, j, k \leq n\}$ be the set of indices that remain to be filled out. Sort this set in increasing order with respect to one of the three indices, break ties with respect to one of the other indices, and break any remaining ties with respect to the last index. Populating in this order will satisfy the dependency constraints. Our solution is then in $\mathsf{OPT}_1[n_1, n_2, n_3]$, the longest common sub-sequence considering all n_1, n_2, n_3 characters of each s_1, s_2, s_3 string.

The runtime complexity of this algorithm is just the time required to populate dp and space complexity is the space required to store dp. Both $O(n1 * n2 * n3)$

4. In class, we developed an $O(nW)$ time algorithm for the Knapsack problem, where n is the number of items and W is the capacity of the knapsack. Recall that in this problem we are given n items $\{1, \ldots, n\}$ with values v_i and weights w_i. Our goal is to fill a knapsack with items whose weights add up to no more than W and whose total value is maximized. Each item can only be picked once. In other words, find a subset $S \subseteq [n]$ such that $\sum_{i \in S} w_i \leq W$ maximizing $\sum_{i \in S} v_i$.

   Suppose that the capacity W is very large (say, exponential in n), so that the running time $O(nW)$ is prohibitively large. Suppose further that each item has a small integral value. Design a different dynamic program for this problem that runs in $O(nV)$ time (that is, a running time independent of W), where $V = \sum_i v_i$ is the sum of values of all the items. You may assume that integer operations take $O(1)$ time.

   (**Hint**: you will need to modify the recursive definition we used to construct the $O(nW)$ time algorithm.)

Solution: We modify the recursive equation of knapsack problem. The original equation $\mathsf{OPT}(i, w)$ was defined as the maximum value we could obtain using a subset of items $\{1, 2, ..., i\}$ and with a total remaining weight of w. The equation is as follows

$$\mathsf{OPT}(i, w) = \max \begin{cases} v_i + \mathsf{OPT}(i-1, w - w_i) & \text{if } w \geq w_i \\ \mathsf{OPT}(i-1, w) \end{cases} \tag{0.5}$$

We modify this equation by changing the role of w and v. $\mathsf{OPT}_2(i, v)$ represents the minimum possible weight subset of items $\{1, 2, ..., i\}$ that achieves value at least v.

$$\mathsf{OPT}_2(i, v) = \min \begin{cases} w_i + \mathsf{OPT}(i-1, \max(0, v - v_i)) \\ \mathsf{OPT}(i-1, v) \end{cases} \tag{0.6}$$

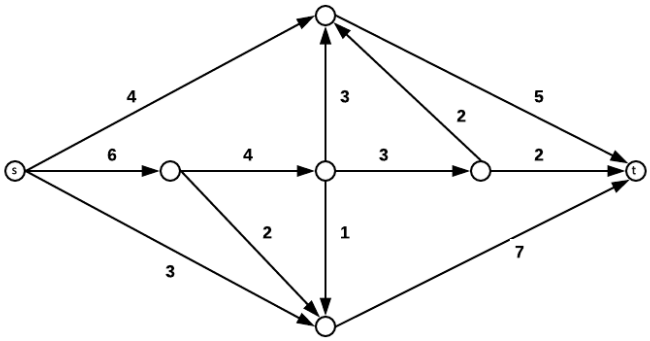
It is easy to make this into an iterative algorithm based on the recursive relationship. Note that each entry is based only on entries with smaller i and v values. Let $\mathsf{OPT}_2(0, v) = \infty$ as our base cases. Construct a 2D array from top to bottom (increasing value of i) and left to right (increasing value of v).

We will have n rows, since there are n items. We'll have V columns representing the minimum possible weight of a subset attaining values $1, 2, \ldots, V$. Clearly, we don't need more than V columns since its not possible to achieve a value higher than the sum of values of all the items, which is V.

The final solution is the index of the greatest column in the bottom row with weight (entry) at most W.

CS 577: Introduction to Algorithms Discussion Solutions for Week 9 & 10

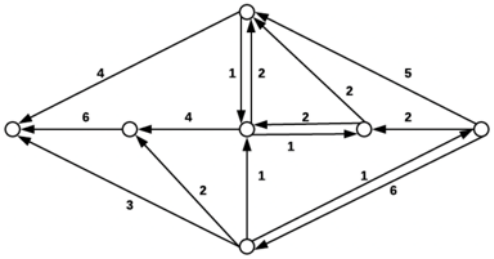
Problem 1



(a) For the network G above determine the max s-t flow, f^*, the residual network G_{f^*}, and a minimum s-t cut.

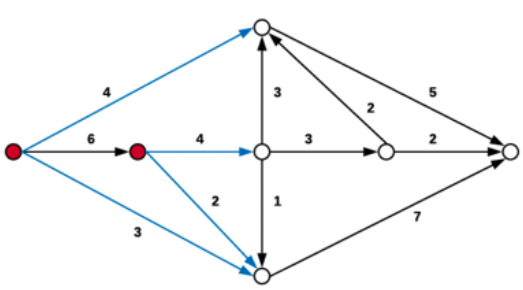
Solution:

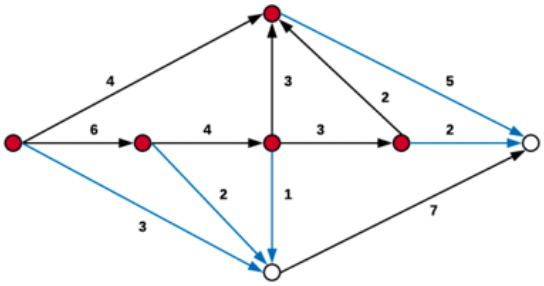
The following are a maximum flow and its residual graph.

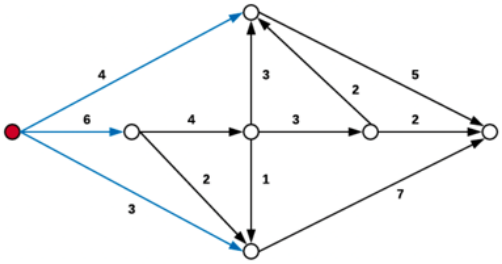


There are many maximum flows and corresponding residual graphs. As long as there are 13 units flowing to the sink and out of the source, each vertex is outputting the same as it receives, and no edge is using more capacity than it can hold, it is a valid maximum flow. For the corresponding residual graph, each directed edge that has remaining capacity should have a value of the remaining capacity, otherwise there should be a directed edge in the reversed direction with the value of the flow.

Below are the three minimum cuts. The red vertices are in S, and the black ones are in T. The blue edges indicate the edges in the cut going from the s side to the t side.



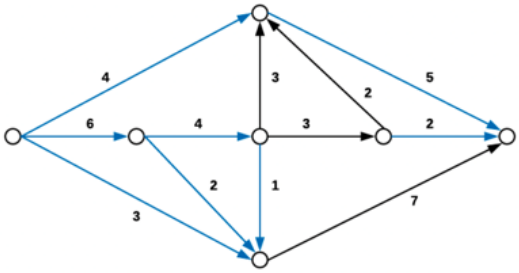


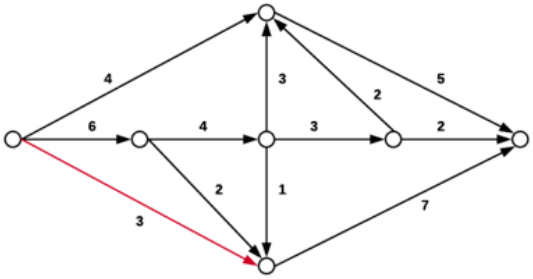


(b) An edge in a flow network is called *upper-binding* if increasing its capacity by one unit increases the maximum flow in the network. Similarly, an edge in a flow network is called *lower-binding* if reducing its capacity by one unit decreases the maximum flow in the network. Identify all of the *upper-binding* and all of the *lower-binding* edges in the above flow network.

Solution:

The lower-binding edges are colored blue, the upper-binding edge is colored red.





(c) Describe and analyze an algorithm for finding **all** of the upper-binding edges in a flow network G when given a maximum flow f^* in G. Your algorithm should run in time $O(n + m)$, where n is the number of nodes and m is the number of edges.

Solution:

We assume all the edge capacities are integral and nonzero. First, we observe that if an edge e' is not used to its capacity, i.e. $f^*(e') < c(e')$, then it cannot be upper-binding. What happens when $f^*(e) = c(e)$ for an edge $e = (u, v)$? It means that, in the residual graph, there is an edge going from v to u with residual capacity $c(e)$, and there is no edge going from u to v (in other words the residual capacity is 0). What happens in the residual graph if the edge capacity of e is increased by 1? There will be an edge going from u to v with residual capacity 1.

From *Ford-Fulkerson Algorithm* we know that since f^* is a maximum flow in the original graph, there is no path from s to t in G_{f^*}. However, after the edge capacity increases by 1, the edge from u to v is available, which

could potentially create a new augmenting path from s to t. If there is such a path going through (u, v), then we know there is 1 more unit of flow going from s to t, i.e. the flow f^* is not maximum anymore. This is, e is upper-binding. If there is no such path, then we know that with the increased capacity of e, f^* is still the maximum flow. Because if there is a flow with higher value, we must be able to find an augmenting path from s to t.

Therefore, in order to determine whether (u, v) is upper-binding, we need to determine whether there is an $s - t$ path in $G_{f^*} \cup \{(u, v)\}$, in other words whether there is an $s - u$ path as well as a $v - t$ path in G_{f^*}.

Alternate characterization: A different way of thinking about lower binding and upper binding edges is that the former is the set of all edges that belong to **some** minimum s-t cut. The latter is the set of all edges that belong to **every** minimum s-t cut.

Algorithm: With the help of the above analysis, we give the algorithm as follows:

1. Construct G_{f^*}. Use BFS/DFS to find all the vertices reachable from s in G_{f^*}. Denote the set by S.
2. Let G'_{f^*} denote the graph G_{f^*} with all edge directions reversed.[1] Use BFS/DFS to find all the vertices reachable from t in G'_{f^*}. This is the set of all the vertices that have a path going to t in G_{f^*}. Denote the set by T.
3. Find all the pairs in $S \times T$ which are edges of G and the capacity of those edges are fully used. This step can be implemented by scanning through the list of all edges (u, v) in G and checking whether $u \in S$ and $v \in T$. These are all the upper-binding edges in G.

Correctness: We already know from the above analysis that $e = (u, v)$ is upper-binding if and only if there is an augmenting path from s to t in G_{f^*} after we add the edge (u, v). Our algorithm finds exactly these edges. Note that without any modification, there is no path from s to t in G_{f^*}. Adding (u, v) creates a path if and only if u is reachable from s and t is reachable from v. Step 1 finds all the vertices S which are reachable from s; Step 2 finds all the vertices T which have paths to t; Step 3 finds all the edges (u, v) satisfying $u \in S$ and $v \in T$. These are exactly the edges whose increased capacity would facilitate a new augmenting path from s to t.

Running time: Step 1 and step 2 takes $O(m + n)$, and step 3 takes $O(m)$. The total running time is linear.

Problem 2

Given a flow network (G, c, s, t), where $G = (V, E)$ is a directed acyclic graph, c_e is the non-negative integer capacity on edges $e \in E$, s is the source node, and t is the sink node. Two flows f^1 and f^2 are considered *distinct* if there exists one edge $e \in E$ such that $f^1_e \neq f^2_e$. Let $f^\star$ be a maximum flow of the network (G, c, s, t).

(a) Show that G has multiple distinct maximum flows if and only if its final residual network, $G_{f^\star}$, has a directed cycle with positive residual capacity.

Solution:

We divide the proof into two directions.
Claim 1. *If the final residual network has a directed cycle with positive residual capacity, then there are multiple distinct maximum flows.*

Proof. Let C be a cycle in $G_{f^\star}$ with positive residual capacity. Since C has positive residual capacity, we can, starting from $f^\star$, push positive flow around C, to get a different flow f' which is still a feasible flow in G. Moreover, since pushing flow in a cycle has no effect on the net flow leaving s, f' has the same value as $f^\star$. Since f' is distinct from $f^\star$ and f' is a maximum flow, it follows that $f^\star$ is not unique. □

Claim 2. *If there are multiple distinct maximum flows, then the final residual network has a directed cycle with positive residual capacity.*

[1] That is, if G_{f^*} contains a directed edge from u to v, then G'_{f^*} contains a directed edge from v to u.

Proof. Let f' be a maximum flow distinct from $f^\star$. Since they are both maximum flows,

$$Val\left(f'\right) = Val\left(f^\star\right).$$

Define the following function, $\Delta : E \to \mathbb{R}$ by,

$$\Delta\left(e\right) = f'_e - f^\star_e, e \in E.$$

Since $f^\star$ and f' both satisfy flow conservation, it follows that Δ does as well by noting that for each vertex $v \in V$ including s and t, the difference between the flow into v and the flow out of v is

$$\begin{aligned}\sum_{e \in E \text{ entering } v} \Delta\left(e\right) - \sum_{e \in E \text{ leaving } v} \Delta\left(e\right) &= \sum_{e \in E \text{ entering } v} \left(f'_e - f^\star_e\right) - \sum_{e \in E \text{ leaving } v} \left(f'_e - f^\star_e\right) \\ &= \left(\sum_{e \in E \text{ entering } v} f'_e - \sum_{e \in E \text{ leaving } v} f'_e\right) - \left(\sum_{e \in E \text{ entering } v} f^\star_e - \sum_{e \in E \text{ leaving } v} f^\star_e\right) \\ &= 0.\end{aligned}$$

The last equality holds due to the conservation law for $u \notin \{s, t\}$, and due to the fact that $Val\left(f'\right) = Val\left(f^\star\right)$ for $u \in \{s, t\}$.

We can now construct a cycle with positive residual capacity in $G_{f^\star}$. First note that since $f^\star \neq f'$, there is some edge $e_1 \in E$ with $\Delta\left(e_1\right) \neq 0$. If $\Delta(e_1) > 0$, then let w_0 and w_1 be the tail and head vertex of e_1, respectively (so $e_1 = (w_0, w_1)$), and vice-versa if $\Delta(e_1) < 0$ (so $e_1 = (w_1, w_0)$). In the first case, e_1 carries positive flow into w_1, and in the second case, e_1 carries negative flow out of w_1. In either case, w_1 has a flow surplus (the difference between the flow into w_1 and the flow out of w_1 is positive), so to satisfy flow conservation of Δ, there must either be an edge $e_2 = (w_1, w_2)$ with $\Delta(e_2) > 0$, or an edge $e_2 = (w_2, w_1)$ with $\Delta(e_2) < 0$. Now the same reasoning applies to w_2, then w_3, and so on: if we have found w_i, the conservation law for Δ implies that there is another edge $e_{i+1} \in E$ either

1. leaving w_i with $\Delta\left(e_{i+1}\right) > 0$ (so $e_{i+1} = (w_i, w_{i+1})$), or,
2. entering w_i with $\Delta\left(e_{i+1}\right) < 0$ (so $e_{i+1} = (w_{i+1}, w_i)$).

In addition, note that for any edge $e = (u, v) \in E$,

- if $\Delta\left(e\right) = f'_e - f^\star_e > 0$, then since $c_e \geq f'_e$, we have $c_e > f^\star_e$, so the edge (u, v) has positive residual capacity relative to f^*, or
- if $\Delta\left(e\right) = f'_e - f^\star_e < 0$, then we have $f^\star_e > 0$, so the edge (v, u) has positive residual capacity relative to f^*.

Therefore, for every i, there is an edge from w_i to w_{i+1} in $G_{f^\star}$ with positive residual capacity. Since an infinite sequence of vertices must eventually repeat a vertex, there must be a cycle in $G_{f^\star}$ with positive residual capacity. □

(b) Describe a polynomial time algorithm to determine whether the maximum flow is unique.

Solution:

The previous part provides the key characterization of a flow network with a unique maximum flow. This leads to the following algorithm.

The runtime of line 1 is the same as an efficient maximum flow minimum cut algorithm (e.g. fewest edges augmenting path, which is polynomial in $|V|$ and $|E|$. The runtime of line 2 is $O\left(|E|\right)$. The runtime of line 3 is the same as DFS which is $O\left(|V| + |E|\right)$. The runtime of line $4, 5, 6$ is $O\left(1\right)$. Therefore, the algorithm is polynomial in $|V|$ and $|E|$.

Algorithm 1:

Input: A flow network (G, c, s, t) with $G = (V, E)$ acyclic.

Find a maximum flow f^* and the final residual network G_{f^*};
Remove all zero-capacity edges from G_{f^*};
Use DFS to check if there is a directed cycle in G_{f^*};
if *a cycle exists* **then**
    return FALSE (maximum flow is not unique);
return TRUE (maximum flow is unique).

Problem 3

A group of n people are carpooling to work together for d days, but they want to make sure the carpool arrangement is fair and doesn't overload any single person with too much driving. This is complicated by the fact that the subset of people in the car varies from day to day due to different work schedules.

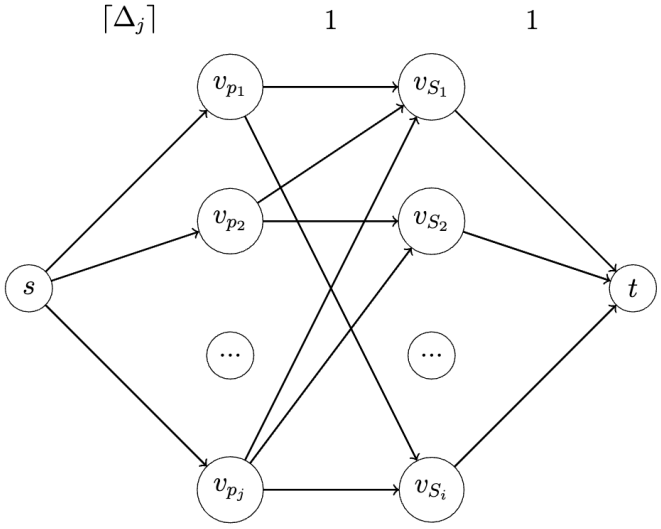
Let $S = \{p_1, ..., p_n\}$ denote the set of people. On the i-th day a nonempty subset $S_i \subseteq S$ of the people go to work, and we say that each person in S_i incurs a **driving obligation** of $\frac{1}{|S_i|}$ that day. Over d days, a person p_j incurs a **total driving obligation** $\Delta_j = \sum_{i:j \in S_i} \frac{1}{|S_i|}$.

A **driving schedule** is the assignment of drivers to days. Note that a driving schedule may be partial, i.e. there could be days where no driver is assigned. We say that a driving schedule is "fair" if each driver p_j drives at most $\lceil \Delta_j \rceil$ times over d days.

(a) Design and analyze an efficient algorithm for computing a fair schedule that maximizes the number of days to which a driver is assigned. *Hint: Reduce the problem to network flow. What can you say about the value of the maximum flow in your constructed network?*

Solution:

We reduce the construction of an optimal fair schedule to an instance of MAX-FLOW. We construct the flow network as follow. Let s and t be the source and the sink. For each person p_j, we add a node v_{p_j} and an edge (s, v_{p_j}) of capacity $\lceil \Delta_j \rceil$ to enforce the constraint that person p_j may drive no more than $\lceil \Delta_j \rceil$ times. For each day i, we add a node v_{S_i} and an edge (v_{S_i}, t) with capacity 1 to enforce the constraint that there is at most one driver on any given day. Finally, for each person p_j that goes to work on day i, i.e., $p_j \in S_i$, we add an edge (v_{p_j}, v_{S_i}) of capacity 1 to indicate that person p_j can drive once on day i.



We now show that there is a one-to-one correspondence between integral flow in the network and fair schedules.
Claim. *There exists a fair schedule with a driver on k days if and only if there is an* integral *flow of value k.*

Proof. Suppose there is fair schedule that has a driver on k days. Then one can obtain an integral flow of value k as follows: If person p_j drives on day i, send 1 unit of flow along the path s, v_{p_j}, v_{S_i}, t. We do so for all days with a driver. Since we started with a fair schedule with a driver on k days, all capacity constraints in the network are satisfied, and we send k units of flow from s to t. By the construction of the flow, it is integral.

Now for the other direction, let us suppose that we have an integral flow of value k. We obtain a fair driving schedule with a driver on k days as follows: If an edge (v_{p_j}, v_{S_i}) carries a unit of flow, we have person p_j drive on day i. By the capacity constraints on edges leaving the source and edges entering the sink, each person p_j drives at most $\lceil \Delta_j \rceil$ times and each day has at most one driver. This tells us that the schedule we constructed is fair. Now, since the flow has value k, there are k days i such that the edge (v_{S_i}, t) carries a unit of flow. Since the flow into an vertex is equal to the flow out of a vertex, this means that there are k days i for which there exists a person p_j such that (v_{p_j}, v_{S_i}) carries a unit of flow. This tells us that there are k days that have a driver in the schedule we constructed. □

By the claim, to find a fair schedule that maximizes the number of days to which a driver is assigned, we only need to find an integral max-flow in the network. One can use the Ford-Fulkerson algorithm to do so. Given an integral max-flow, we construct a fair schedule as we did in the proof of the claim.

Running Time The running time of our algorithm is given by the time required to construct the network plus the time to find the max-flow in the network plus the time to retrieve a fair schedule from the max-flow. The size of the network we construct is $O(nd)$. As a result, the time required to construct the network is $O(nd)$, and one can retrieve a fair schedule given the max-flow in $O(nd)$ time. The running time of Ford-Fulkerson is $O(|E||f^*|)$, where $|E|$ is the number of edges in the graph and $|f^*|$ is the value of the maximum flow. Since there are d edges entering the sink each of capacity 1, $|f^*|$ is at most d. Therefore, finding the max-flow takes $O(nd^2)$ time (the number of edges in the graph is $O(nd)$) . It follows that our algorithm for finding the optimal fair schedule runs in $O(nd^2)$ time.

(b) Prove that for any choice of sets $S_1, ..., S_d$, there exists a fair schedule that assigns a driver to *each* of the d days.

Solution:

By the claim, to show that there is a fair schedule that has a driver on each day, it is sufficient to show that there exists an integral flow of value d in the network. To show that there exists an integral flow of value d, it is

sufficient to show that there exists a flow of value d. We construct such a flow as follows: For each person p_j and for each day i on which p_j goes to work, send $\frac{1}{|S_i|}$ units of flow along the path s, v_{p_j}, v_{S_i}, t. The amount of flow through an edge (s, v_{p_j}) is exactly $\sum_{i:j \in S_i} \frac{1}{|S_i|} = \Delta_j \leq \lceil \Delta_j \rceil$. The flow along an edge (v_{p_j}, v_{S_i}) is $\frac{1}{|S_i|}$. And the amount of flow along an edge (v_{S_i}, t) is $\sum_{j \in S_i} \frac{1}{|S_i|} = 1$. Therefore, each edge (v_{S_i}, t) carries one unit of flow and the total flow into the sink is d. This shows that there is a flow of value d. It follows that there is a fair schedule with a driver on all days.

Problem 4

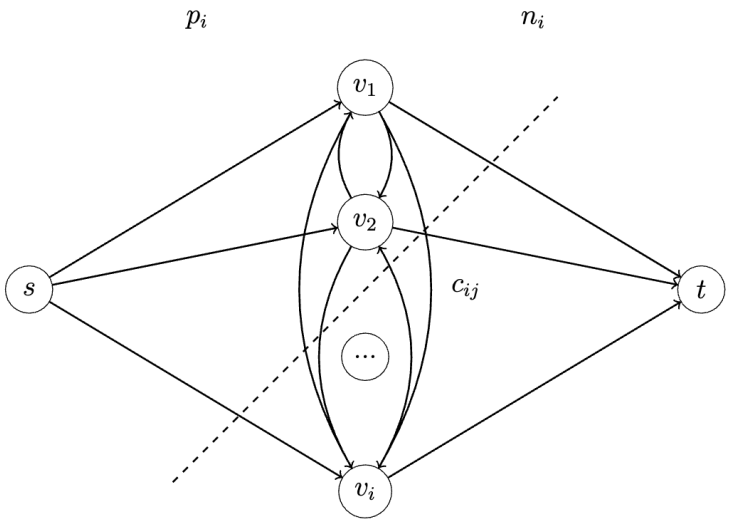
Two companies Pineapple and Nanosoft make competing versions of m different software products. Pineapple charges p_i for product i and Nanosoft charges n_i for its version of i. A consumer wants to buy one version of each product. While the customer prefers cheaper versions, she also prefers to buy most software from the same company. In particular, products i and j, if bought from different companies, impose an incompatibility cost of $c(i, j)$ on the customer. The customer's total cost from buying software is the prices paid to the two companies, plus a sum over all incompatible pairs of the respective incompatibility costs. The goal is to determine, for each software product, from which company the customer should buy that product so as to minimize her total cost.

Design and analyze an efficient algorithm for this problem.

Solution:

The problem is a selection problem with a minimization objective. Therefore, we convert this to a problem of finding the MIN-CUT. We set up our graph in the following way:

- Each product is represented as a node in the graph and is connected to the source (s) with an edge having capacity equal to its respective p cost (Pineapple's version) and is also connected to the sink (t) with an edge having capacity equal to its respective n cost (Nanosoft's version)
- Each product node is connected to every other product node with an edge having a capacity equal to the incompatibility cost between those two products.



In this setup finding the min s-t cut (S, T) corresponds to finding the minimum cost of purchasing the products. For each product, we buy Pineapple's version if the product node is in T and Nanosoft's version if the product node is in S.

We also make some basic observations:

- We purchase one version of each product: If we don't purchase any version of a product i, then s and t are still connected via $s \underset{p_i}{\longrightarrow} i \underset{n_i}{\longrightarrow} t$. Since we use an s-t cut, this is not possible.
- For any two products with different versions we account for the incompatibility cost: Let us say we purchase product i from Pineapple and product j from Nanosoft and we don't account for the incompatibility cost. Then s and t are still connected via $s \underset{p_j}{\longrightarrow} j \underset{c_i}{\longrightarrow} i \underset{n_i}{\longrightarrow} t$. Again, this is not possible due to the s-t cut.

We now argue that there exists a s-t cut (S, T) with a capacity C, if and only if there is a way to purchase products such that the total cost is C. We prove the forward direction first. Assume we buy all the products in S from Pineapple and all the products in T from Nanosoft. Then, the cost of buying these products is $\sum_{i \in S} p_i + \sum_{j \in T} n_j + \sum c(i, j)$, this is exactly equal to the capacity of the (S, T) cut. Now in the opposite direction, let there exist a s-t cut (S, T) with capacity C, then for a product i we buy i from Pineapple if $i \in S$, else we buy from Nanosoft. Similar to the above sum, this will amount to a cost of C. The claim combined with the two observations we made, prove that our algorithm is correct.

CS 577: Intro. to Algorithms **Discussion Intractability Solutions**

Problem 1

In the SQUAREDSUM problem, you are given a set of integers U, and numbers k and B. You want to know whether it is possible to partition the members of U into k sets $S_1, \cdots, S_k$ so that the squared sums of the sets add up to at most B:

$$\sum_{i=1}^{k} \left(\sum_{s \in S_i} s \right)^2 \leq B$$

In this exercise, we are going to show that SQUAREDSUM is NP-hard by exploiting the fact that polynomial-time reductions are transitive.

Let PARTITION be the following problem: you are given a set of integers U as input, and your objective is to decide whether there is a way to partition the members of U into two sets S and $U \setminus S$ such that

$$\sum_{s \in S} s = \sum_{s \in U \setminus S} s.$$

(a) Describe a polynomial-time mapping reduction from SUBSETSUM to PARTITION.
(b) Prove that the mapping reduction from part (a) is correct.
(c) Describe a polynomial-time mapping reduction from PARTITION to SQUAREDSUM.
(d) Prove that the mapping reduction from part (c) is correct.

Solution - parts (a) and (b)

First, we present the reduction from SUBSETSUM to PARTITION. Recall that the decision version of the SUBSETSUM problem is defined as follows: given a collection U of positive integers and a target value t, determine whether there is a subset $S \subseteq U$ such that the sum of the integers in S is exactly t, i.e., determine whether there is a subset S s.t. $\sum_{s \in S} s = t$.

The reduction is as follows: let U, t be an instance of SUBSETSUM and let W be sum of elements in U. Note that if this is a true instance, then we can partition the elements of U into the sets S and $U \setminus S$ such that

$$\sum_{s \in S} = t, \text{ and } \sum_{s \in U \setminus S} = W - t.$$

Since in the PARTITION problem we want the partitions to have the same sum, it makes sense to add the element $2t - W$ to S, obtaining U'. In that case, both sets S and $U' \setminus S$ would have sum equal to t. Indeed, it turns out that this idea is sufficient for the reduction, because one of the sets does not contain element $2t - W$. We present more details in the following discussion.

The instance of PARTITION we construct from U, t is

$$U' = U \cup \{2t - W\}.$$

After constructing U', we call the black-box for PARTITION with input U' and accept if and only if PARTITION accepts.

To argue correctness, we need to show that there is a subset of U that sums to t if and only if there is a subset S' of U' such that

$$\sum_{s \in S'} s = \sum_{s \in U' \setminus S'} s = 1/2 \cdot \sum_{u \in U'} u = (W + 2t - W)/2 = t.$$

Suppose there is a subset S of U that sums to t, then the same subset S serves as a solution for PARTITION, as it has total sum exactly half the sum of all elements in U'.

For the other direction, suppose that there is a subset S' of U' such that $\sum_{s \in S'} s = \sum_{s \in U' \setminus S'} = t$. One of the sets S' or $U' \setminus S'$ must not contain the element $2t - W$ that we added to U to obtain U'. In that case, the same set serves as a solution for SUBSETSUM, as it only contains elements from U and adds up to t.

Solution - parts (c) and (d)

Recall the definition of PARTITION: given a collection U' of positive integers that sum to W, determine whether there is a subset S such that

$$\sum_{s \in S} s = \sum_{s \in U \setminus S} s = 1/2 \cdot \sum_{u \in U} u = W/2.$$

Note that even though the SQUAREDSUM problem allows for other values of k, it makes sense to try coming up with a reduction from PARTITION to SQUAREDSUM using $k = 2$, because then the objective for both problems becomes more similar. With this in mind, consider maintaining the input U we get from PARTITION. If we have a true instance of PARTITION, then there is a set $S \subseteq U$ such that

$$\sum_{s \in S} s = \sum_{s \in U \setminus S} s = W/2.$$

If we square both sums, each is then equal to $(W/2)^2 = W^2/4$ and their sum is equal to $W^2/2$. Now, recall that if we have two numbers P and Q that add up to W, we minimize $P^2 + Q^2$ exactly when $P = Q = W/2$. This property follows from the following inequality:

$$2(P^2 + Q^2) \geq (P + Q)^2,$$

with equality if and only if $P = Q$.

Note that this is exactly what we need, so we could just set the target to $W^2/2$, and then both problems would be checking for the exact same condition. We now formalize this reduction: given an instance U of PARTITION, we construct the instance $(U, k = 2, B = W^2/2)$ of SQUAREDSUM. Then, we call the black-box for SQUAREDSUM with input $(U, 2, W^2/2)$ and return yes if and only if it returns yes.

By the inequality above, S_1, S_2 is a solution for this instance of SQUAREDSUM if and only if $\sum_{s \in S_1} s = \sum_{s \in S_2} s = W/2$. The latter holds if and only if S_1 is a solution to the initial instance of the PARTITION. We have thus given a reduction from PARTITION to SQUAREDSUM.

Problem 2

Consider the following problem, called BOXDEPTH: Given a set of n axis-aligned rectangles in the plane and a positive integer k, does there exist k rectangles that contain a common point?

(a) Describe a reduction from BOXDEPTH to the CLIQUE problem.
(b) Briefly describe and analyze a polynomial-time algorithm for BOXDEPTH.
Hint: $O(n^3)$ time should be easy, but $O(n \log n)$ time is possible.
(c) Why don't these two results imply that P = NP ?

Solution

(a) Given an instance of BOXDEPTH, made up of rectangles $\{1, \ldots, n\}$ and a number k, here is an algorithm that decides whether there is a subset of size k such that the rectangles in that subset share a common point.

Let G be the graph with one vertex v_i for each rectangle i, and with the edge (v_i, v_j) iff rectangles i and j have a common point.

We call CLIQUE(G, k) and return the same result.

We leave it as an exercise to prove that a set of rectangles has box depth $\geq k$ if and only if G contains a clique on $\geq k$ vertices, therefore that the algorithm is correct.

(b) We give an $O(n^3)$ time algorithm for BOXDEPTH. The key observation is the following: the intersection of two or more rectangles is also a rectangle. Furthermore, each corner of this rectangle is either an intersection point (of the sides of two or more rectangles) or the corner of one of the original rectangles. In particular, any set of overlapping rectangles must all contain a common point that is either an intersection point or the corner of some rectangle. Let the set of such points be P.

To find the largest value K for which there are K overlapping rectangles that share a common point, we only need to find for each point p in P, the number of rectangles containing p and output the maximum over all points. We return 'yes' if $K \geq k$.

The number of points in P is at most $O(n^2)$. This is because the number of intersection points is at most $4n^2$ (every pair of rectangles intersect at at most 4 points) and the number of corners is at most $4n$. For each point in P, checking whether a given rectangle contains the point takes $O(1)$ time, and as a result finding the number of rectangles that contain the point takes at most $O(n)$ time. Therefore, the algorithm described above runs in $O(n^3)$ time.

(c) The reduction in part (a) and the algorithm in part (b) do not together imply $\mathrm{P} = \mathrm{NP}$ because the $\mathrm{P} = \mathrm{NP}$ question can be equivalently stated as whether SAT, or any problem that SAT reduces to, in particular CLIQUE, has a polynomial-time algorithm. What we showed in part (a) and part (b) says nothing about this question. What we did show in part (a) is that if CLIQUE has a polynomial-time algorithm, i.e., if $\mathrm{P} = \mathrm{NP}$, then so does BOXDEPTH. We then showed in part (b) that "if $\mathrm{P} = \mathrm{NP}$" is an unnecessary assumption; there is, unconditionally, a polynomial-time algorithm for BOXDEPTH.

Problem 3

The world is in the midst of a pandemic and a vaccine has finally been developed, however, the vaccine is very difficult to produce in large quantities. Since it is impossible to vaccinate everyone at the moment, the CDC is trying to determine who should receive the vaccine first. They have an idea that the vaccines should be spread through the population in a way that helps everyone by universally increasing resistance. We define the idea of a person p being "resistant" to the virus if at least one of the following holds.

- Person p has been vaccinated.
- Person p interacts with someone who has been vaccinated.

Thus, we phrase the *Vaccine Distribution Problem* as follows. Given a group of people, a record of the people with whom each person regularly interacts,and a number of available vaccines k, is there a way to distribute the vaccines so that every person is resistant?

(a) Describe the graph that encodes an instance of this problem.
(b) Despite great effort to develop an efficient algorithm to decide vaccine distribution, no one has succeeded, and time is of the essence. Unfortunately for the CDC, it appears that their efforts may be fruitless. Prove that *Vaccine Distribution* is NP-Complete.

Solution

(a)
- Represent people as vertices (people-nodes)
- If two people interact, connect their vertices by an edge (interaction-edges)

(b) 1. Vaccine Distribution is in NP.

An instance of *Vaccine Distribution* can be defined on a graph $G = (V, E)$ of people-nodes and interaction-edges, and a number of vaccines, k. If we have a subset of people-nodes of size $\leq k$, we can verify that everyone is either included in the subset or interacts with someone in the subset in time $O(k|V| \cdot |E|)$. So Vaccine Distribution is in NP.

2. *Vaccine Distribution is NP-Hard. (Vertex Cover $\leq_p$ Vaccine Distribution)*

An arbitrary instance of *Vertex Cover* is defined on a graph $G = (V, E)$ and a number k. Create a graph $G' = (P, I)$ of people-nodes and interaction-edges, initially just a copy of G. For each $\{u, v\} \in E$, create an additional people-node p_{uv} and add p_{uv} to P, as well as $\{u, p_{uv}\}$ and $\{p_{uv}, v\}$ to I.

Since *Vertex Cover* is only concerned with edges, isolated vertices are not included in a vertex cover. However, they need to be included here. Thus we will make the additional change that we will form a set of isolated people-nodes (no connecting edges) IP, and the number of vaccines for this instance is $k' = k+|IP|$. In the corresponding solutions, the vertex cover of G plus the nodes in IP give the solution for *Vaccine Distribution* of G' and the *Vaccine Distribution* solution for G' with the nodes in IP removed gives the vertex cover for G.

This adds $|E|$ new people-nodes and $2|E|$ new interaction-edges, as well as an additional set consideration of at most $|V|$, so this is a polynomial transformation.

($\Rightarrow$) Suppose we have a subset C of V with $|C| \leq k$ that forms a vertex cover. We want to show that $C + IP$ forms a solution for *Vaccine Distribution* of size $\leq k'$. If there are any isolated people-nodes, they are resistant by the inclusion of IP in the solution. For all other nodes, consider u and v such that $\{u, v\} \in E$. Since C is a vertex cover, at least u or v is in c. WLOG, suppose $u \in C$. Then it is the case that p_{uv} and v both interact with a vaccinated person, and are also resistant. Thus every node in G' is resistant.

($\Leftarrow$) Suppose we have a subset R of G' with $|R| \leq k'$ such that R forms a *Vaccine Distribution* solution. To get a vertex cover for G of size k, if any people-nodes in R are the p_{uv} vertices, it must be the case that p_{uv} is connected to u and v by the construction of G'. So we can replace p_{uv} by u or v and it will still be the case that u, v, and p_{uv} are resistant. With this modification, the vaccine distribution is still valid, since all the nodes that were resistant before will continue to be resistant. Additionally, remove any isolated people-nodes from R. This reduces the set from $\leq k'$ to $\leq k$. Now, R modified in the prescribed way is a vertex cover of size $\leq k$ for G, since for every $\{u, v\} \in E$, at least one of u or v is in R.

CS577: Introduction to Algorithms

Discussion 1: Discrete Review

Solving Recurrences

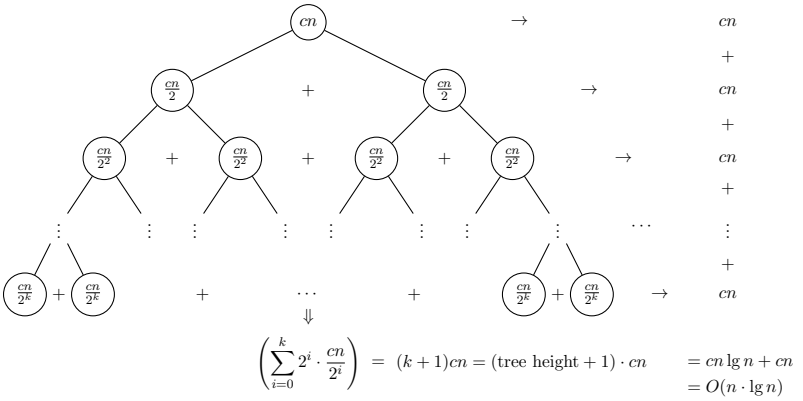
$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Unrolling / unwinding:

$$\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
&\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\
&\leq 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn \\
&\vdots \\
&\leq 2^k T\left(\frac{n}{2^k}\right) + kcn \\
&= nT(1) + cn\log(n), \text{ using } (\textbf{??}) \\
&= cn + cn\log n \\
&= O(n\log(n))
\end{aligned}$$

$$\begin{aligned}
1 &= \frac{n}{2^k} \\
\iff 2^k &= n \\
\iff k &= \log_2(n) \qquad (1)
\end{aligned}$$

Recursion Tree:



$$\left(\sum_{i=0}^{k} 2^i \cdot \frac{cn}{2^i}\right) = (k+1)cn = (\text{tree height} + 1) \cdot cn \qquad = cn\lg n + cn = O(n \cdot \lg n)$$

Induction

We want to show that a predicate $P(n)$ holds for $n = \{0, 1, 2, \ldots\}$.

1. Base Case(s): The starting point of the induction. Without a base case, there is not induction.
2. Inductive Hypothesis: We assume that $P(k)$ is true for some k. For *strong* induction, we assume that $P(k)$ is true for all $j \leq k$.
3. Inductive Step: We show that $P(k+1)$ holds, using the (2) inductive hypothesis.

So,

$$\text{Base Case(s)} \rightarrow P(0) \rightarrow P(1) \rightarrow P(2) \rightarrow P(3) \rightarrow P(4) \rightarrow \ldots$$

Program Correctness

A program/algorithm is correct if it is:

- **Sound/partial correctness/correct** (any returned value is true), and
- **Complete/termination** (returns a value for all valid inputs).

Proving correctness requires at least 2 proofs: One for soundness and one for completeness.

Iterative Code:

1. Identify and prove the loop invariants using induction.
2. Using the invariants, prove the soundness.
3. Prove the completeness: Show that the algorithm terminates for all input.

Recursive Code:

1. Prove the soundness via induction:
   (a) Show that the recursive base case is correct.
   (b) Assuming that the recursive calls return the correct value (ind hyp), show that the code returns the correct value (induction step).
2. Prove the completeness: Show that recursive calls make progress towards a base case.

CS577: Introduction to Algorithms

Discussion 2: Asymptotic Analysis and Graphs

Asymptotic Bounds

Bound	Informal Notion	Formal Definition	Limit Test
$f(n) \in O(g(n))$	'$f(n) \leq g(n)$'	$O(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid 0 \leq f(n) \leq cg(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq d$ for $d \in \mathbb{R}_{>0}$
$f(n) \in \Omega(g(n))$	'$f(n) \geq g(n)$'	$\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid 0 \leq cg(n) \leq f(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$ for $c \in \mathbb{R}_{>0}$
$f(n) \in \Theta(g(n))$	'$f(n) = g(n)$'	$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$	$c \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq d$ for $c, d \in \mathbb{R}_{>0}$
$f(n) \in o(g(n))$	'$f(n) \ll g(n)$'	$o(g(n)) = \{f(n) : \forall c > 0 \exists n_0 > 0 \mid 0 \leq f(n) < cg(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) \in \omega(g(n)$	'$f(n) \gg g(n)$'	$\omega(g(n)) = \{f(n) : \forall c > 0 \exists n_0 > 0 \mid 0 \leq cg(n) < f(n)\ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

CS577: Introduction to Algorithms

Discussion 3 and 4: Greedy

In lecture, we saw two main techniques for showing that a greedy algorithm, GREEDY, is optimal: Stays Ahead and Exchange Argument.

Stays Ahead

A *Stays Ahead* argument requires that you define a notion of time steps over which you can show by induction that GREEDY is better according to some measure $g(i)$ than any other algorithm, or an optimal algorithm for all time steps i. The opimality of GREEDY can follow immediately, or may require a proof using the property shown in the induction.

The canonical problem presented in lecture was the *Interval Scheduling Problem*, and the optimal GREEDY was the *finish first* heuristic.

Steps for a Stays Ahead argument:

1. Establish a notion of time steps. These times steps should be well-defined for any algorithm.

   Ex: For the Interval Scheduling Problem, the time steps were the index in a set of jobs S produced by an algorithm when the jobs are sorted by finish time from smallest to largest. Note that this is well-defined for any algorithm.

2. Using induction over the time steps, show that, for some measure $g(i)$, GREEDY is the same or better than any other algorithm for all time steps i.

   Ex: For the Interval Scheduling Problem, $g(i)$ was the property that the finish time of the ith job of GREEDY was less than or equal to the ith job scheduled in any other schedule.

3. Using the property shown in the induction to claim the optimality of GREEDY.

   Ex: For the Interval Scheduling Problem, if GREEDY had scheduled k items, then $g(k)$ told us that the finish time of the kth job of GREEDY was less than or equal to the kth job scheduled in any other schedule, implying that GREEDY could schedule any $k+1$ job that any other algorithm would schedule, a contradiction to GREEDY scheduling k items.

Exchange

An *exchange* argument starts from an solution S^* that may be defined as an optimal solution, or the solution of any algorithm. The solution is transformed by some sort of *exchange* to a new solution S_1 that is closer the to the solution S produced by the GREEDY heuristic and no worse than S^*. Then, by induction, we can repeat this exchange producing a sequence of equivalent solutions

until we arrive at S. That is, $S^* \equiv S_1 \equiv S_2 \equiv \cdots \equiv S$ for the function we are trying to optimize for the given problem.

The canonical problem presented in lecture was a job scheduling problem of *minimizing max lateness*, and the optimal GREEDY was the *earliest deadline first* (EDF) heuristic.

Steps for a Exchange argument:

1. Consider solutions produced by GREEDY, and determine characteristic properties of those solutions that may not be shared by an arbitrary optimal solution.

   Ex: For minimizing the max lateness, the solutions produced by the EDF had no idle time and jobs were scheduled by increasing deadlines.

2. Define the exchanges need to transform any solution into a solution closer to the solution produced by GREEDY, and show that such an exchange produces a solution that is no worse.

   Ex: For minimizing the max lateness, we showed that (1) idle time could be removed from any schedule without increasing the max lateness, and (2) that in schedule (without idle time) where the jobs are not ordered by increasing deadlines, there are at least two sequential jobs where there deadlines are out of order (an inversion). These two jobs can be swapped (removing an inversion, and sorting them by their deadlines) without increasing the lateness.

3. By induction, transform any solution into a solution that is equivalent to the GREEDY solution.

   Ex: For minimizing the max lateness, the GREEDY schedule S has no idle time and the jobs are scheduling by increasing deadline. The schedule S' produced by starting from any schedule S^*, removing idle time and repeatedly doing exchanges until no inversions remain also has no idle time and the jobs are scheduling by increasing deadline. The schedule S and S' may differ on the order of jobs with the same deadlines, but this does not change the max lateness.

CS577: Introduction to Algorithms

Discussion 5 and 6: Divide and Conquer

Algorithms that fit the Divide and Conquer paradigm, like MergeSort are most often presented as recursive algorithms since the paradigm is naturally recursive. It is important to remember that recursion has no more computational power than an iteration, and that a recursive algorithm can be described iteratively, and vice-versa. For example, on the GeeksForGeeks website, you can find:

- a recursive implementation of MergeSort (`https://www.geeksforgeeks.org/merge-sort/`), and
- an iterative implementation of MergeSort (`https://www.geeksforgeeks.org/iterative-merge-sort/`).

Divide and Conquer Paradigm

1. *Divide:* Split the problem into smaller sub-problems.
2. *Conquer: (Recurse)* Solve the smaller sub-problems (usually through recursion).
3. *Combine: (Merge)* Combine the solutions to the sub-problems into a solution for the original problem.

Often a useful technique to improve the efficiency of the solution.

Divide and Conquer Correctness

Assuming a recursive algorithm:

Soundness: Typically strong induction on input size.

1. Show that the base case(s) is correct.
2. Assume that the recursive calls return the correct value.
3. Given the assumption above, show that the value return will be correct.

Completeness: Show that the recursive calls make progress towards the base case.

Divide and Conquer Runtime Analysis

Assuming recursive algorithm:

From Algorithm Definition to Recurrence:

The runtime for a recursive Divide and Conquer algorithm with input of size n is:

- $T(n) = \sum_i T(n_i) + f(n)$, where n_i is the size of the input to the ith recursive call, and $f(n)$ is the cost of the work done outside of the recurrences for a call of input size n.
- Don't forget the base cases!

Example:

Algorithm 1 DCAlg

Input : A
Output: $f(A)$
1 **if** $|A| = 1$ **then return** $g(A)$
2 A_1 := DCAlg(Front-half of A) A_2 := DCAlg(Last 1/4 of A) A_3 := DCAlg(First 1/4 of A) **return** $h(A_1, A_2, A_3)$

If $g(A)$ is constant time, and $h(A_1, A_2, A_3)$ is linear time:

$$T(n) \leq T(\frac{n}{2}) + 2T(\frac{n}{4}) + cn;\ T(1) = c$$

If $g(A)$ is linear time, and $h(A_1, A_2, A_3)$ is constant time:

$$T(n) \leq T(\frac{n}{2}) + 2T(\frac{n}{4}) + c;\ T(1) = cn$$

If $g(A)$ is quadratic time, and $h(A_1, A_2, A_3)$ is quadratic time:

$$T(n) \leq T(\frac{n}{2}) + 2T(\frac{n}{4}) + cn^2;\ T(1) = cn^2$$

Solving Recurrences: For the main techniques, see Week 1 discussion.

CS577: Introduction to Algorithms

Discussion 7 and 8: Dynamic Programming

Problems that can be effectively solved by dynamic programming are:

- Problems that can be broken up into at most a polynomial number of subproblems that can be aggregated into a final solution efficiently.
- Typically a recursive solution that has a polynomial number of unique recursive calls which allows for memoization.

Dynamic Programming Paradigm

Step 1 - Recursive Solution: Develop a recursive solution for the problem. For many dynamic program solution, the recursive solution will be inefficient with an exponential number of recursive calls, but the number of unique calls is polynomial.

E.g.: nth Fibonacci number: $f(n) = f(n-1) + f(n-2); f(2) = 1; f(1) = 1;$. An exponential number of recursive calls, but only n unique calls.

Step 2 - Memoize: Means "write it down and remember". Dynamic programs are defined recursively, but use a table/array/matrix to record the values of the recursive calls. Calculate once, record, and lookup on future calls.

E.g.: For the nth Fibonacci number, we can use a 1-D table to record the values from 1 to n. That would take the complexity down to a linear number of calculations from an exponential number of calls.

Important Notes:

- The memoization table for a dynamic program can any number of dimensions. We will see mostly 1-D and 2-D solutions, but there is nothing preventing 3-D, 4-D, or more.
- Determining the dimensions of memoization table: This will usually correspond to the explicit parameters of your recursion solution. In other words, the parameters that characterize the unique recursive calls, i.e., the degrees of freedom.
- There is always a context for the values stored in the table: You should always be able to describe the semantics of the value in a cell of the memoization table.
- For some problems, the table of values may represent a data structure like a tree.

Presenting Your Dynamic Program Solution

For dynamic programs, we do not want to be pseudocode. Rather, we want you to describe the details of solution within the context of the program.

Definitions: Describe all the definitions and preprocessing needed for the dynamic programming solution.

E.g. Weighted Interval Scheduling (WIS):

- Preprocessing: Sort the input by finish time.
- Definitions: For a given job at index j, let $i_j < j$ be the largest index such that $f_i \leq s_j$.

Matrix/table description: Describe the dimensions of the matrix, the contents of a cell, and any cells that can be initialized.

E.g. WIS: A 1D array M, where $M[j]$ is the maximum value of a compatible schedule for the first j items in the sorted input. Initialize $M[1] = v_1$.

Bellman Equation: The recursive definition for populating a cell in the table.

E.g WIS: $M[j] = \max\{M[j-1], M[i_j] + v_j\}$.

How to populate: Describe the order in which the cells of the matrix will be populated.

E.g. WIS: Populate from 2 to n.

Solution: Describe where to find the solution in the matrix, or how to calculate the solution.

E.g. WIS: The maximum value of a compatible schedule for the n jobs is found at $M[n]$.

Dynamic Program Correctness

Soundness: (Strong) induction over the order of the population of cells of the solution matrix:

1. Show that the base case(s) is correct.
2. Assume that the recursive calls return the correct value.
3. Given the assumption above, show that the Bellman equation will calculate the correct value.

Completeness: Usually immediate from the definition of the dynamic program.

Dynamic Program Runtime Analysis

- Preprocessing time, and
- Time to populate the matrix: Number of cells $\times$ Time to calculate Bellman equation, and
- Time to calculate the final solution.

E.g. WIS: $O(n \log n)$

- Preprocessing: $O(n \log n)$
- Populate the matrix: $O(n) \times O(\log n) = O(n \log n)$
- Final solution: $O(1)$

CS577: Introduction to Algorithms

Discussion 9 and 10: Network Flow

Flow Network

- A directed graph $G = (V, E)$, where $|V| = n$ and $|E| = m$.
  - Each edge $e \in E$ has a *capacity* $c_e \geq 0$.
  - A *source* node $s \in V$, and a *sink* node $t \in V$.
  - All other nodes $(V \setminus \{s, t\})$ are internal nodes.
  - Let $C = \sum_{e \text{ out of } s} c_e$.
- *Flow function*: $f : E \to R^+$, where $f(e)$ is the flow across edge e.
  - Valid flow function conditions:
    - i *Capacity*: For each $e \in E, 0 \leq f(e) \leq c_e$.
    - ii *Conservation*: For each $v \in V \setminus \{s, t\}$, $\sum_{e \text{ into } v} f(e) = f^{\text{in}}(v) = f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$.
- The flow starts at s and exits at t.
- Flow value $v(f) = f^{\text{out}}(s) = f^{\text{in}}(t)$.

An $s - t$ Cut

- A Cut: Partition of V into sets (A, B) with $s \in A$ and $t \in B$.
- Cut capacity: $c(A, B) = \sum_{e \text{ out of } A} c_e$.

Max-Flow = Min-Cut

- *Max-Flow*: Given a flow network G, the max-flow is the flow function f that maximizes $v(f)$.
- *Min-Cut*: Given a flow network G, the min-cut is the $s-t$ cut (A, B) that minimizes $c(A, B)$.
- For a flow network G, let f^* be the maximum flow function, and let (A^*, B^*) be the minimum cut, then $v(f^*) = c(A^*, B^*)$.

Max-Flow / Min-Cut Algorithms

Residual Graph: Given a flow network G and a flow f on G, we define the residual graph G_f:

- Same nodes as G.
- For edge (u, v) in E:
  - Add edge (u, v) with capacity $c_e - f(e)$.
  - Add edge (v, u) with capacity $f(e)$.

Ford-Fulkerson Method: $O(mC)$

- Initialize $f(e) = 0$ for all edges.
- While G_f contains an augmenting path P:
  - Update flow f by $\textsc{bottleneck}(P, G_f)$ along P.

Other algorithms:

- Scaled Ford-Fulkerson: $O(m^2 \log C)$.
- Fewest Edges Augmenting Path [BFS] (Edmonds-Karp): $O(m^2 n)$.
- Dinitz 1970: $O\left(\min\left\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\right\} m\right)$.
- Preflow-Push 1974/1986: $O(n^3)$.
- Best: Orlin 2013: $O(mn)$

Network Flow Reductions

Many problems can be *reduced*[1] efficiently to a network flow whose solution (max-flow/min-cut) and the resulting flow function solves the original problem.

Reduction to max-flow problems:

- How can the problem be encoded in a graph?
- Source/sink: Are they naturally in the graph encoding, or do additional nodes and edges have to be added?
- For each edge: What is the direction? Is it bi-directional? What is the capacity?

[1] A reduction is a transformation of any instance of one problem $\mathcal{P}$ into an instance of another problem $\mathcal{Q}$, where the solution to the instance of $\mathcal{Q}$ gives a solution to the original instance of $\mathcal{P}$.

Solutions via max-flow reductions:

- Drawing the flow network can be helpful, but a solution must describe all the details of the reduction to a network flow graph.
- Bullet points are a great way to enumerate all the steps of the reduction.
- An efficient solutions requires:
  - An efficient reduction to a network flow graph (the size of the graph cannot have more than a polynomial number of nodes and edges as a function of the original problems input size).
  - Taking the network flow solution and deriving the solutions to the original problem must also be efficient.
- The running time of a network flow solution is determined by:
  - the time to construct the network flow graph for a given an instance of the original problem,
  - the time to solve a the max-flow using one of the known max-flow algorithms, and
  - the time to determine the solution to the original problem based on the max-flow solution.

Network Flow Extensions

The network flow extensions of *node demands* and *capacity lower bounds* can make for more natural reductions. Note that it does not add anymore computational power to the model. That is, a problem with a reduction to a network flow graph with demands and lower bounds has a reduction to a standard network flow problem.

Flow network with node demands:

- Each node has a demand d_v:
  - if $d_v < 0$: a source that demands $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.
  - if $d_v = 0$: internal node ($f^{\text{in}}(v) - f^{\text{out}}(v) = 0$).
  - if $d_v > 0$: a sink that demands $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.
- S is the set of sources ($d_v < 0$).
- T is the set of sinks ($d_v > 0$).
- Flow conditions:
  - i Capacity: For each $e \in E$, $0 \leq f(e) \leq c_e$.
  - ii Conservation: For each $v \in V$, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.

- Goal is *feasibility*: Does there exist a flow that satisfies the conditions?
  - If there is a feasible flow, then $\sum_{v \in V} d_v = 0$.

(1) Reduction to max-flow from flow network with node demands:

- Super source s^*: Edges from s^* to all $v \in S$ with $d_V < 0$ with capacity $-d_v$.
- Super sink t^*: Edges from all $v \in T$ with $d_V > 0$ with capacity d_v to t^*.
- Maximum flow of $D = \sum_{v:d_v>0 \in V} d_v = \sum_{v:d_v<0 \in V} -d_v$ in G' shows feasibility.

Flow network with node demands and capacity lower bounds:

- Node with demands as described above.
- For each edge e, define a lower bound ℓ_e, where $0 \leq \ell_e \leq c_e$.
  - i Capacity: For each $e \in E$, $\ell_e \leq f(e) \leq c_e$.
  - ii Conservation: For each $v \in V$, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.
- Goal is *feasibility*: Does there exist a flow that satisfies the conditions?

(2) Reduction to flow network with node demands from flow network with node demands and capacity lower bounds:

- Consider an f_0 that sets all edge flows to ℓ_e: $L_v = f_0^{\text{in}}(v) - f_0^{\text{out}}(v)$.
  - if $L_v = d_v$: Condition is satisfied.
  - if $L_v \neq d_v$: Imbalance.
- ¡2-¿ For G':
  - Each edge e, $c'_e = c_e - \ell_e$ and $\ell_e = 0$.
  - Each node v, $d'_v = d_v - L_v$.

Then, using the (1) reduction, you can obtain a max-flow network that you can solve to check for feasibility.

CS577: Introduction to Algorithms

Discussion 11 and 12: Intractability

Polynomial-Time Reductions

$Y \leq_p X$

- Consider any instance of problem Y.
- Assume we have a black-box solver for problem X.
- Efficiently transform an instance of problem Y into a polynomial number of instances of X that we solve via black-box solver for problem X, and aggregate the solutions efficiently to solve Y.

Corollaries from polynomial time reductions:

- Suppose $Y \leq_p X$. If X is solvable in polynomial time, then Y can be solved in polynomial time.
- Suppose $Y \leq_p X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.
- Suppose $Z \leq_p Y$ and $Y \leq_p X$. Then, $Z \leq_p X$.

Intractability Definitions

Decision Problems

- Binary output: yes / no answer.
- Our complexity definitions assume decision problem versions of the problems.
- No less powerful: we can go between decision and optimization version of problems.

Easy vs Hard Problems

- *Easy Problems*: problems that can be solved by efficient algorithms.
- *Hard Problems*: problems for which we do not know how to solve efficiently.

Input Formalization

- Let s be a binary string that encodes the input.
- $|s|$ is the length of s, i.e., the # of bits in s.

Complexity class: P

- Polynomial run-time: Algorithm A has a *polynomial run-time* if run-time is $O(\mathsf{poly}(|s|))$ in the worst-case, where $\mathsf{poly}(\cdot)$ is a polynomial function.
- P is the set of all problems for which there exists an algorithm A that solves the problem with polynomial run-time.

Efficient Certification: Certifier $B(s, t)$ for a problem P:

- s is an input instance of P.
- t is a certificate; a proof that s is a yes-instance.
- Efficient: For every s, we have $s \in P$ iff there exists a t, $|t| \leq \mathsf{poly}(|s|)$, for which $B(s, t)$ returns yes.

Complexity class: NP

- Set of all problems for which there exists an efficient certifier $B(s, t)$.
- I.e., the set of all problems for which it is efficient to verify a potential solution.
- **N**on-deterministic, **P**olynomial time: can be solved in polynomial time by testing every certificate (t) simultaneously (non-deterministic).

Complexity class: NP**-Hard:** Problem X is NP-Hard if:

- For all $Y \in \mathsf{NP}$, $Y \leq_p X$.
- NP-Hard problem may or may not be in NP.

Complexity class: NP**-Complete:** Problem X is NP-Complete if:

- For all $Y \in \mathsf{NP}$, $Y \leq_p X$.
- X is in NP.

Showing that Problem X is NP-Complete: Cook Karp Reduction

Step 1: Prove that $X \in \mathsf{NP}$.

(a) Define a certificate (t) for X.

(b) Define an efficient certifier (algorithm) $B(s,t)$ for X and t as defined in (a).

Step 2: Choose a problem $Y \in \mathsf{NP}$-Complete.

- Y must be a problem that is known to be NP-Complete.
- It will be used to show that $Y \leq_p X$ in step 3:
  - Since $Y \in \mathsf{NP}$-Complete, then all NP problems $\leq_p Y$.
  - Therefore, showing $Y \leq_p X \implies$ all NP problems $\leq_p X \implies X \in \mathsf{NP}$-hard.

Step 3: $Y \leq_p X$ ($X \in \mathsf{NP}$-hard).

- Karp Reduction: For an arbitrary instance s_Y of Y, show how to construct, in polynomial time, an instance s_X of X such that s_y is a yes iff s_x is a yes.
  Steps:
  (a) Provide efficient reduction.
  (b) Prove $\Rightarrow$: if s_Y is a yes, s_X is a yes.
  (c) Prove $\Leftarrow$: if s_X is a yes, then s_Y had to have been a yes.

CS577: Introduction to Algorithms

Discussion 13: Randomization

Randomized Algorithms

- Algorithm flips a coin to make some decisions.
- Non-Deterministic: simultaneously considers multiple algorithms weighted by the probability distribution.

Types of Randomized Algorithms:

- *Monte Carlo*: With probability p returns the correct answer:
  - Run multiple times to boost the probability of correct answer.
  - Provide an approximation guarantee in expectation.
- *Las Vegas*: Has a run-time that is polynomial in expectation. It will always returns the correct solution, or informs about failure.
- *Atlantic City*: Probabilistic run-time and correctness.

Probability Definitions

Probability Space:

- *Sample space* Ω of all possible outcomes.
  - Can be infinite, but we will focus on finite.
  - Ex: 6-sided fair die (D6): $\Omega = \{1, 2, 3, 4, 5, 6\}$.
- *Probability mass*: each $i \in \Omega$ has a nonnegative probability mass: $1 \geq p(i) \geq 0$.
- Total probability mass is 1: $\sum_{i \in \Omega} p(i) = 1$.

Probability Event:

- An event ε is a set of outcomes of Ω.
- $\Pr[\varepsilon] = \sum_{i \in \varepsilon} p(i)$.
- Note: $\Pr[\overline{\varepsilon}] = 1 - \Pr[\varepsilon]$

Conditional Probability: Probability of ε given $\mathcal{F}$.

$$\Pr[\varepsilon|\mathcal{F}] = \frac{\Pr[\varepsilon \cap \mathcal{F}]}{\Pr[\mathcal{F}]}$$

Independent Events:

- Events ε and $\mathcal{F}$ are independent if $\Pr[\varepsilon|\mathcal{F}] = \Pr[\varepsilon]$ and $\Pr[\mathcal{F}|\varepsilon] = \Pr[\mathcal{F}]$.
- This implies $\Pr[\varepsilon \cap \mathcal{F}] = \Pr[\varepsilon] \cdot \Pr[\mathcal{F}]$.
- Generalization: Say $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$ are independent.

$$\Pr\left[\bigcap_{i=1}^{n} \varepsilon_i\right] = \prod_{i=1}^{n} \Pr[\varepsilon_i]$$

Union Bound:

$$\Pr\left[\bigcup_{i=1}^{n} \varepsilon_i\right] \leq \sum_{i=1}^{n} \Pr[\varepsilon_i],$$

where equality *only* if events are mutually exclusive.

Random Variables:

- Technical: Given a probability space, a random variable X is a function from the sample space to the natural (finite – real if infinite) numbers, such that, for number j, $X^{-1}(j)$ is the set of all sample points taking the value j is an event.
- Informally: A random variable X takes on a value that depends on a random process.
- Ex: $\Pr[X = 1] = 1/6$, where X is a toss of a 6-sided die.

Expectation Definitions

Expected Value:

- "Weighted average value"
- $\mathbb{E}[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j]$
- Ex: $\mathbb{E}[X] = \sum_{i=1}^{6} \frac{1}{6} \cdot i = 3.5$, where X is a toss of a 6-sided fair die.

Expectation Properties: Let X and Y be random variables, and a be a constant.

- Linearity of expectation:
  - $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$
  - $\mathbb{E}[aX] = a\,\mathbb{E}[X]$
- If X and Y are independent, $\mathbb{E}[XY] = \mathbb{E}[X]\,\mathbb{E}[Y]$.

Guarantee in Expectation: An algorithm that returns a solution that has a r approximation ratio in expectation:

$$\forall I, \mathbb{E}[\textsc{alg}(I)] \leq r \cdot \textsc{opt}(I) + \eta$$

Vertex Cover (VC)

Find the minimum number of nodes in a graph that touch all edges. Equivalent to IS.

Set Cover (SC)

Find the minimum number of subsets that includes all elements from a universe. Reducible from VC.

Independent Set (IS)

Find the largest subset of nodes such that no two nodes are neighbors. Equivalent to VC.

Set Packing (SP)

Find the largest group of subsets without overlaps. Reducible from IS.

3-D Matching (3DM)

Given a set of tuples $T \subseteq X \times Y \times Z$, find the largest subset of T without overlap.

3 Satisfiability (3SAT)

Clauses of 3 literals, each clause has an "AND" operator between them. Each literal in each clause has an "OR" between them. Find literals that make the entire statement true. Reducible from IS.

Not-all-equal 4 Satisfiability (NAE-4SAT)

Clauses of 4 literals with the same restrictions as 3-sat. All four variable CANNOT have the same truth value. Reducible from 3SAT.

Circuit Satisfiability (CSAT)

Given a boolean circuit, can we find an input such that the output is true. Reducible from 3SAT.

Hamiltonian Cycle

Is there a cycle that hits every node exactly once and returns to start?

Traveling Salesperson Problem

Find shortest cycle (weighted graph) that hits all nodes exactly once and then loops to start.

Graph Coloring

Use as minimum number of colors for vertices of a graph where adjacent nodes have different color.

Subset sum problem

Find a subset that adds to a specific amount

Knapsack

Find a subset that maximizes a value while staying under a capacity

Proving a problem is NP-complete

1) Find a **certificate** that can verify a solution in polynomial time. 2) Find another **NP-complete problem** you can reduce it from (YES in one problem **if and only if** YES in another problem).

Important Algorithm

- **Graph traversal** BFS vs DFS, both are $O(|V| + |E|)$
- **Minimum spanning tree** Prim's (add vertex based on a PQ on edges until MST is formed, $O(|E| \log |V|)$) vs Kruskal's (sort edges then union groups of vertexes, $O(|E| \log |V|)$ with union find for cycle detection).
- **Shortest path** Dijkstra (edges weights are non-negative, $O((|V| + |E|) \log |V|)$ with priority queue) vs Bellman-Ford (creates a $|V| \times |E|$ DP array, allows negative weights, $O(|V||E|)$ complexity)
- **Max flow** Ford-Fulkerson ($O(|E|f)$ complexity when f is the largest flow capacity). Other algorithms can reach true polynomial time.
- **Network flow conversion**:
  - **Edge demands:** if edge $u \to v$ has a demand of x, we can undo the demand by adding two edges $s \to v$ and $u \to t$, both of capacity x. The idea is that we remove x units of flow from u and restore it to v.
  - **Node capacity:** a node v with a capacity c and lower bound (demand) ℓ can be transformed into two vanilla nodes v_1,v_2 where $v_1 \to v_2$ with the same demand and capacity.
- **Master Theorem** *The recurrence*

$$T(n) = aT(n/b) + cn^k$$
$$T(1) = c,$$

where $a, b, c,$ and k *are all constants, solves to:*

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

UW Madison CS577 Algorithms Final Exam Review

Created by Martin Diges

Advice for Studying for the Exam

- **Try the practice exam on Canvas!**
- Come to the review session (Dec 15 Humanities 3650 @ 14:00-17:00) - we go over the solutions to the practice exam.
  - the review session will also be livestreamed and recorded
- To review homework problems quickly, spend 5 minutes on each one coming up with a rough draft of what your solution would look like. Then, check whether your thought process was correct.
- The exam is open notes/book. If you feel it is necessary, bring an example of every algorithm paradigm and related work that you need. We will review those paradigms today.

Advice for Taking the Exam

- If a question looks difficult, pass and come back to it later. Pick questions on a topic where you're strongest and try them first.
- Ration your time. The exam is 2 hours and there are 5 questions, graded out of 4. Try to keep a 30-minute pace. This will force you to at least attempt other questions, which may yield you some points.
- Suggested problems to be comfortable at the end of each study group slides (see Canvas).

Graphs

Graph Terminology

Graph - a set of nodes, V, and edges connecting nodes, E.

Connected graph - all pairs of nodes are connected (path can be found between them)

Complete graph - all nodes directly connected to each other by an edge

Acyclic graph - for any 2 nodes u,v there is at most 1 path that connects them

Directed Acyclic Graph (DAG) - edges are now directed

Topological Ordering - a DAG that represents precedence relations

Tree - a connected, acyclic graph. It may have a root. $|V| = |E| + 1$ for $|V| > 0$

Minimum Spanning Tree (MST) - Collection of edges from the original graph with lowest sum of weights that results in a connected graph

Bipartite Graph - Graph where V can be split into disjoint sets A, B, where $A + B = V$, such that the only edges present are those that cross from one set to the other

Subgraph - for some graph G, a subset $(V', E') \subseteq G$

Strongly Connected Component - for every pair of nodes u,v , there is a path $u \rightarrow v$ and a path $v \rightarrow u$

Graph Representation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\|V\|^2)$	$O(1)$	$O(\|V\|)$
Adjacency list	$O(\|V\| \cdot \min(\|E\|, \|V\|))$	$O(\min(\|V\|, \|E\|))$	$O(1)$
Edge list	$O(\|E\| + \|V\|)$	$O(\|E\|)$	$O(\|E\|)$
Incidence matrix	$O(\|V\|\|E\|)$	$O(\|E\|)$	$O(\|V\|\|E\|)$

Might be useful in a pinch but you can definitely come up with something on the spot.

- slide 8

Graph Algorithms
- Exploration

Breadth-First Search (BFS)

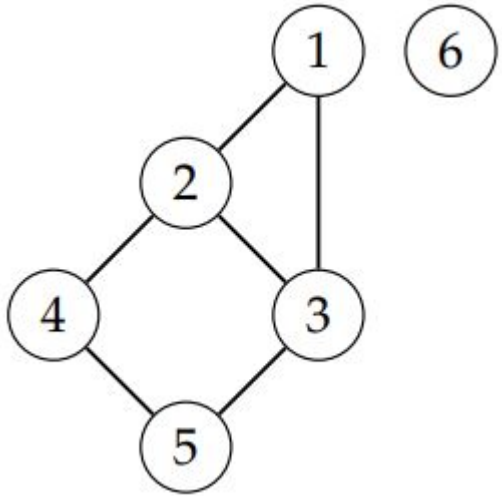
Choose a starting node s and add it to both a queue and a set of visited nodes

While the queue is not empty:

Pop a node from the front of the queue. Add all of its adjacent nodes which are not in the visited set to the visited set and to the back of the queue.

O(|V| + |E|)

For edges with weight 1, finds the shortest path from s to a node.



Choosing s=1, we have the following queue:
[1, 2, 3, 4, 5]

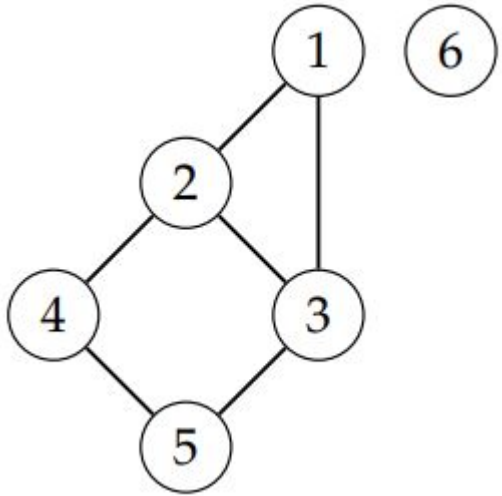
- slide 11

Depth-First Search (DFS)

Choose a starting node s and add it to both a stack and a set of visited nodes

While the stack is not empty:

Pop a node from the top of the stack. Add an adjacent node which is not in the visited set to the visited set and to the top of the stack. If there are none, continue to next loop iteration.



O(|V| + |E|)

Choosing s=1 and breaking ties by alphabetical order, we have the following stack:
[1, 2, 3, 5, 4]

- slide 12

Graph Algorithms
- MST

Prim's, Kruskal's Minimum Spanning Tree (MST) algorithms

- Find an edge set $F \subseteq E$ with minimum cost that keeps the graph connected. That is, F should minimize $\sum_{e \in F} c_e$.

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Reverse-Delete (Kruskal's 1956) Algorithm

- Sort edges by cost from highest to lowest.
- Remove edges unless graph would become disconnected.

- slide 26

Graph Algorithms

- Shortest Path

Dijkstra’s Shortest Path (Dijkstra’s)

Choose a node s and add it to a visited set S

While there are unvisited nodes reachable in one edge from S:

Choose the node u in S and v in V \ S with the shortest (edge(u,v) + dist(s)). Add v to S and record which node reached it: u.

Runtime:

O(mn) ← O(n) for each node, O(m) to find each min

O(m*log(n)) ← Use a priority queue to keep track of nodes nearest to s

To retrieve the shortest path from s to v, work back from v and choose the node that reached it. Repeat this process until you have reached s.

- Greedy slides 29

Greedy

How do we prove the optimality of a greedy algorithm?

The most common arguments employed in proofs for the optimality of greedy algorithms:

- “**Stays ahead**” argument, where we show that our algorithm maintains some property after each greedy decision which allows our algorithm to “stay ahead” (more generally, not fall behind) other algorithms. We show that G being strictly worse than O contradicts our property, hence G is at least as good as O.
- “**Exchange**” argument, where we show that we could modify the decisions made by other algorithms, one by one, to match those of our greedy algorithm while the solution either stays as good or even improves with respect to the other algorithm’s original solution. Thus, we show G must be at least as good as O.

Remember! We want to show our greedy algorithm G produces output that is at least as good (>= or <= depending on context) than another algorithm O.

Stays-Ahead Argument example: Interval Scheduling

Objective: Produce a compatible schedule of jobs, S, that has a maximum cardinality (the most jobs possible)

Heuristic: Earliest deadline first

Argument: **We establish a timestep t** such that two algorithms, our greedy algorithm G and another algorithm O make **choose their t-th earliest job at timestep t**.

At t=1, G will choose the earliest job. Hence, G <= O

For t=k, assume G's k-th job has a finish time g_k <= o_k. G will thus have access to further jobs with at least as early finish times as O, so G's k+1-th job will also satisfy g_{k+1} <= o_{k+1}.

(1). Thus, for all t, G's t-th job will have a finish time <= the finish time of O's t-th job.

(2). Suppose G is strictly worse than other algorithms O ⇒ O schedules more jobs (m+1) than G (m) ⇒ **G must not have been able to schedule job m+1**.

By the property we have shown (1), we know that the last job in G (m) must have ended at least as early as the penultimate job in O (m) **⇒ G should have been able to schedule the last job in O (m+1). Thus, (2) contradicts (1)!**

Thus, G is at least as good as other algorithms O.

Exchange Argument example: Minimizing Max Lateness

Objective: schedule jobs with deadlines such that maximum lateness is minimized

Heuristic: earliest deadline first (EDF)

Argument: Take the output of our algorithm G that schedules jobs in increasing order of deadline and the output of another algorithm O which makes at least one different decision.

O behaves differently from G at some point ⇒ O must have at least 2 adjacent jobs i, j which have an inversion in terms of deadline.

If we take 2 such jobs i,j and switch their order, we do not increase max lateness.

- Swapping i and j means that l'_j (lateness in S') is less than that in S^*.
- Lateness of i may increase, but:
  $l'_i = f'_i - d_i = f^*_j - d_i \leq f^*_j - d_j = l^*_j$.

We can continue to do this O → O'... until O' === G (O' is ordered according to G's heuristic). Thus, lateness(G) <= lateness(O) ⇒ our greedy algorithm is no worse than any other algorithm and therefore optimal.

Divide and Conquer

Breakdown of Divide & Conquer Problems

	BinarySearch-like	MergeSort-like
Divide	✅	✅
Conquer	✅	✅
Merge		✅

1. Divide: Split the input into smaller chunks
2. Conquer: Choose what to do with each chunk (probably involves a recursive call on one or more chunks).
3. Merge: If multiple chunks had associated recursive calls, merge the results of those calls.

Most D&C algorithms can be thought of in terms of being either similar to BinarySearch or being similar to MergeSort.

Examples of BinarySearch-like: Median Of Medians (MOM), QuickSelect, 2 DBs homework problem

Examples of MergeSort-like: Line Visibility hw problem, Max area under histogram discussion problem

Proving Correctness

Typically accomplished by using strong induction on the relevant input size ⇒ Assume recursive calls return the correct answer.

The divisive nature of the algorithms means the input size of a recursive call is likely to be small enough (say, k/2) to be covered by strong induction, allowing us to prove correctness for the k+1 th case.

After assuming your recursive calls will return correct values, show how you use return values to obtain a correct answer (Merge step).

Solving Recurrences: Unwinding

- Slide 13

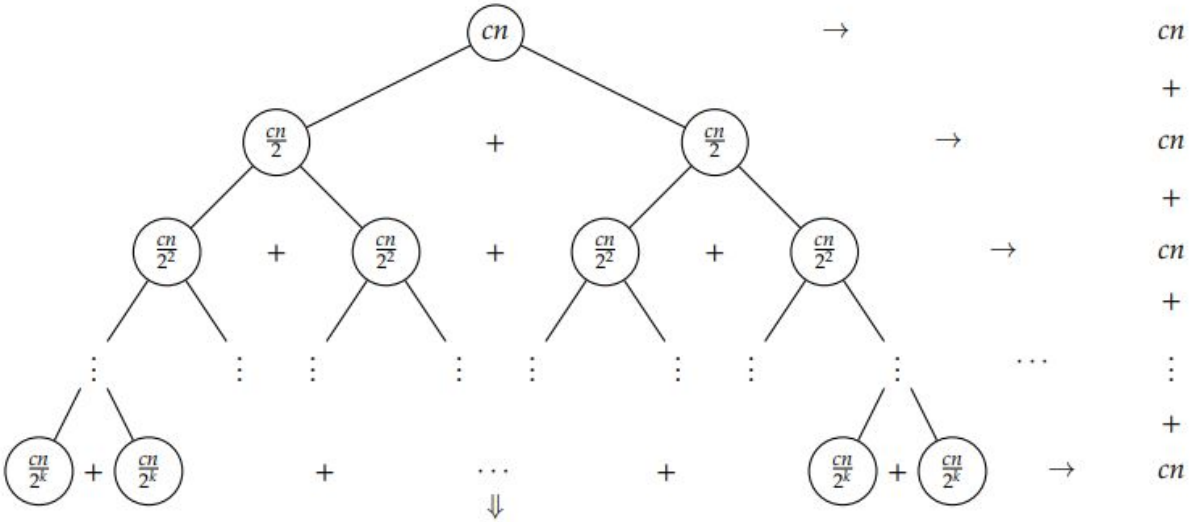
$$
\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
&\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\
&\leq 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn \\
&\vdots \\
&\leq 2^k T\left(\frac{n}{2^k}\right) + kcn \\
&= nT(1) + cn\log(n) \\
&= cn + cn\log n \\
&= O(n\log(n))
\end{aligned}
$$

$$
\begin{aligned}
1 &= \frac{n}{2^k} \\
\iff 2^k &= n \\
\iff k &= \log_2(n)
\end{aligned}
$$

Solving Recurrences: Recursion Tree

- Slide 14

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



$$\left(\sum_{i=0}^{k} 2^i \cdot \frac{cn}{2^i}\right) = (k+1)cn = (\text{tree height} + 1) \cdot cn = cn \lg n + cn = O(n \cdot \lg n)$$

Sample Proof: MaxSubarray (slide 54)

Algorithm: MaxSubarray

Input : Array A of n ints.
Output: Max subarray in A.
if $|A| = 1$ **then return** $A[1]$
$A_1 :=$ MaxSubarray(Front-half of A)
$A_2 :=$ MaxSubarray(Back-half of A)
$M :=$MidMaxSubarray(A)
return *Array with max sum of* $\{A_1, A_2, M\}$

Algorithm: MidMaxSubarray

Input : Array A of n ints.
Output: Max subarray that crosses midpoint A.
$m :=$ mid-point of A
$L :=$ max subarray in $A[i, m-1]$ for $i = m-1 \to 1$
$R :=$ max subarray in $A[m, j]$ for $j = m \to n$
return $L \cup R$ `// subarray formed by combining` L `and` R.

Analysis

- Correctness: By induction, A_1 and A_2 are max for subarray and M is max mid-crossing array.
- Complexity: Same recurrence as MergeSort.

Dynamic Programming

What should you know about Dynamic Programming?

For this course, you should know how to:

- **Formulate a problem as a recursive algorithm** with a polynomial/pseudo-polynomial number of unique input values.
  - Recursive logic usually describes some range of choices (e.g., a **dichotomy**)
  - Algorithm usually translates directly into a **Bellman equation**.
  - Describe any preprocessing needed before your algorithm runs.
- **Describe the data structure** you use to “memoize” - cache results of - recursive calls (often **n-dimensional table**)
  - What values it contains to begin with ← base cases
  - In what order entries should be filled in ← described by Bellman equation
- **Explain the in-context meaning of a single entry** within your data structure
- Detail **how to recover the desired solution** from your data structure
- Prove that your algorithm is correct
  - **strong induction** on population order of recursive calls ← derived from Bellman equation
- Provide and show the runtime of your algorithm

DP Problems: Weighted Interval Scheduling

Algorithm: WeightIntDP

Sort σ by finish time
$m[0] := 0$
for $j = 1$ *to* n **do**
  Find index i
  $m[j] = \max(m[j-1], m[i] + v_j)$
end

DP Solutions

- DP algorithms are formulaic.
- We understand how loops work.
- NO Pseudocode.

We want:

- Definitions required for algorithm to work
- Description of matrix
- Bellman Equation
- Location of solution, order to populate the matrix

DP Problems: Weighted Interval Scheduling

Definitions required for algorithm to work

- σ sorted by finish time, ascending order.
- For a given job at index j, $i_j < j$ is the largest index such that $f_i \leq s_j$.

Description of matrix

- 1D array M, where $M[j]$ is the maximum value of a compatible schedule for the first j items in sorted σ. Initialize $M[1] = v_1$.

Bellman Equation

- $M[j] = \max\{M[j-1], M[i_j] + v_j\}$

Solution, order to populate

- The maximum value of a compatible schedule for the n jobs is found at $M[n]$. Populate from 2 to n.

DP Problems: Knapsack

2D Approach

- 2D Matrix v:
  - i: Item indices from 0 to n.
  - w: Max weight from 0 to W.
  - $v[i, w]$ is the subset of the first i items of maximum sum $\leq w$.
- Indicator: $x_{i,w} := 0$ if $w_i > w$ and 1 otherwise.
- Bellman Equation:

$$v[i, w] = \max(v[i-1, w], x_{i,w} \cdot (v[i-1, w-w_i] + w_i))$$

- $v[0, w] := 0$ for all w and $v[i, 0] := 0$ for all i
- Solution value: $v[n, W]$.

Network Flow

Must know

- Max-flow = Min-cut
  - For a max-flow with capacity Q, there must exist a min-cut with capacity Q
- Obtain a max-flow manually by running an algorithm like Ford-Fulkerson: while there exists a path $s \rightarrow t$ with bottleneck > 0, send flow = bottleneck along the path and add equivalent flow to residual edges.
- Obtain the max flow from a residual graph: add residual edges coming out of t
- Obtain a min-cut from the residual graph: run BFS from t, reachable nodes go into B. Remaining nodes go into A.
  - Edges which have one end in A and the other in B also describe a min-cut.
- Know the runtime of at least one Max-Flow algorithm, such as Orlin's O(mn)

Questions to ask yourself when making a NF reduction

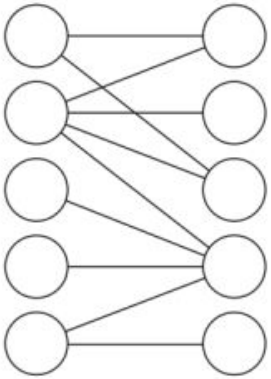
What is the objective of this problem? (Max/Min/Feasibility)

Is there a natural choice of source and sink?

Which nodes should we create in G?

What are our problem’s constraints? How do we encode them into G?

Common Network Flow Setups: Bipartite Matching



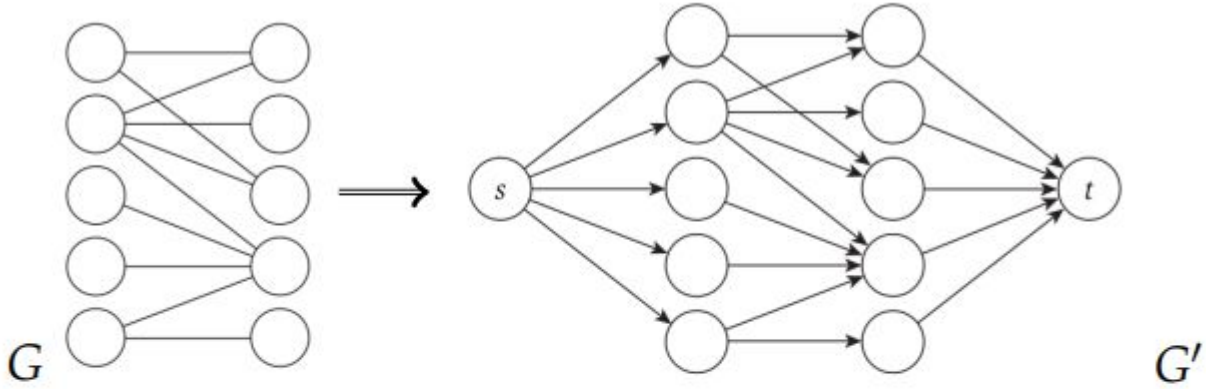
Definition

- Bipartite Graph $G = (V = X \cup Y, E)$.
- All edges go between X and Y.
- Matching: $M \subseteq E$ s.t. a node appears in only one edge.
- Goal: Find largest matching (cardinality).

Reduction to Max-Flow Problem

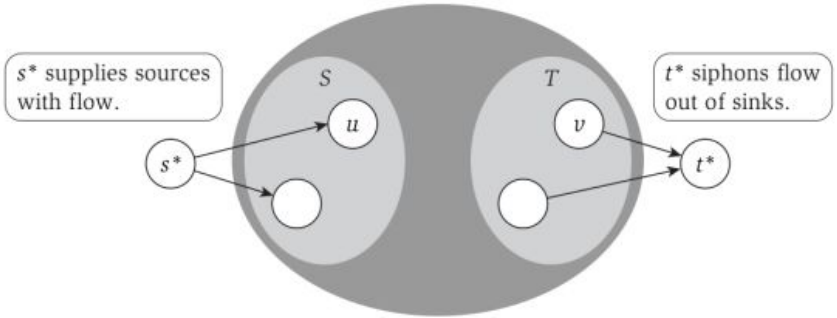
- Goal: Create a flow network based on the the original problem.
- The solution to the flow network must correspond to the original problem.
- The reduction should be efficient.

Common Network Flow Setups: Bipartite Matching



- Add source connected to all X.
- Add sink connected to all Y.
- Original edges go from X to Y.
- Capacity of all edges is 1.

Common Network Flow Setups: Node demands



Reduction from G (demands) to G' (no demands)

- Super source s^*: Edges from s^* to all $v \in S$ with $d_V < 0$ with capacity $-d_v$.
- Super sink t^*: Edges from all $v \in T$ with $d_V > 0$ with capacity d_v to t^*.
- Maximum flow of $D = \sum_{v:d_v>0 \in V} d_v = \sum_{v:d_v<0 \in V} -d_v$ in G' shows feasibility.

Common Network Flow Setups: Survey Design

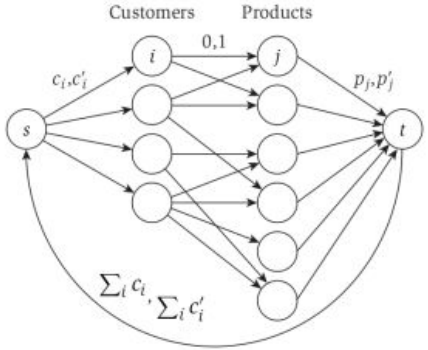
Problem

- Study of consumer preferences.
- A company, with k products, has a database of n customer purchase histories.
- Goal: Define a product specific survey.

Survey Rules

- Each customer receives a survey based on their purchases.
- Customer i will be asked about at least c_i and at most c_i' products.
- To be useful, each product must appear in at least p_i and at most p_i' surveys.

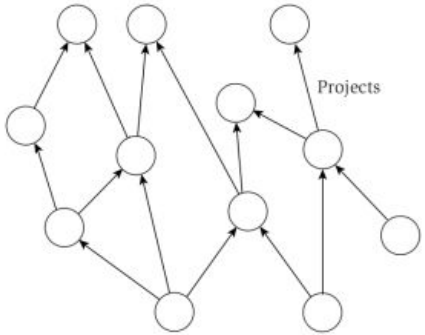
Common Network Flow Setups: Survey Design



Reduction

- Bipartite Graph: Customers to products with min of 0 and max of 1.
- Add s with edges to customer i with min of c_i and max of c_i'.
- Add t with edges from product j with min p_j and max of p_j'.
- Edge (t, s) with min $\sum_i c_i$ and max $\sum_i c_i'$.
- All nodes have a demand of 0.

Common Network Flow Setups: Project Selection

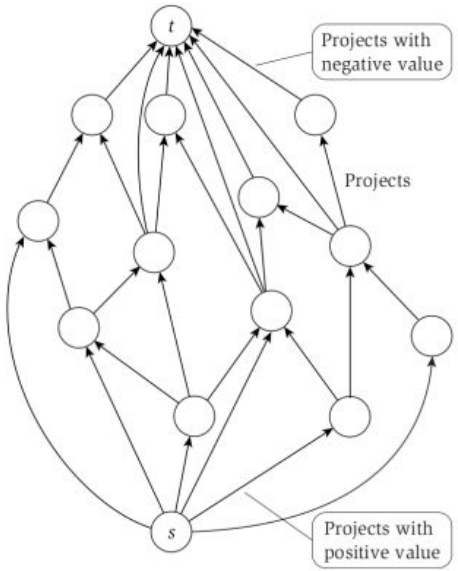


Use Min-Cut to solve this problem.

Problem

- Set of projects: P.
- Each $i \in P$: profit p_i (which can be negative).
- Directed graph G encoding precedence constraints.
- Feasible set of projects A: $\text{PROFIT}(A) = \sum_{i \in A} p_i$.
- Goal: Find A^* that maximizes profit.

Common Network Flow Setups: Project Selection



Reduction

- Use Min-Cut
- Add s with edge to every project i with $p_i > 0$ and capacity p_i.
- Add t with edge from every project i with $p_i < 0$ and capacity $-p_i$.
- Max-flow is $\leq C = \sum_{i \in P: p_i > 0} p_i$.
- For edges of G, capacity is ∞ (or $C + 1$).

Intractibility

Problem Complexity Classification

We commonly classify problems into these categories of computational complexity:

- **P**: Any instance of these problems can be **solved in Polynomial time**.
- **NP**: Any **candidate solution** for these problems can be **checked/verified for correctness in Polynomial time**.
  - This implies that if at least we could somehow verify every candidate solution (even those which are incorrect) at the same time - “Non-deterministic” - and each verification takes Polynomial time, it would take Polynomial time to solve the problem.
- **NP-Hard**: **at least as hard as** the hardest problems in **NP**. Any problem in NP can be reduced to a problem in NP-Hard.
- **NP-Complete**: both **NP** and **NP-Hard** at the same time

Reductions and problem hardness

Reductions can also be useful when talking about problem hardness. In particular, we can use reductions to show that a particular problem is at least as “hard” as another.

Consider problems A and B. Suppose we know B is very difficult.

If we reduce A to B (A <=r B), does this tell us anything about the hardness of problem A?

- No.
- Analogy: If we run a program that adds 2 numbers together on a quantum computer, do we know whether the program is computationally complex? All that our reduction shows is that our program can be run on a quantum computer. We could have run our problem on a far less powerful computer just fine.

If we reduce B to A (B <=r A), does this tell us anything about the hardness of problem A?

- Yes! It tells us A is at least as “hard” as B.
- If we are able to get a solution for B efficiently by using a solver for A, then A’s solver must be at least as powerful as a solver for B. Since the solver for A is capable of solving something at least as hard as B, A must be at least as hard as B.

Proving a problem is NP-Complete

To prove a problem B is in the class NP-Complete → NP & NP-Hard

- Prove problem B is in NP → Describe a polynomial solution checker
- Prove problem B is in NP-Hard → Describe a polynomial time reduction from an existing NP-Complete problem A to the problem B
  - Prove that this reduction is a bijection. i.e., show that an original problem instance and its reduced instance will return the same result. Prove:
  - A instance “Yes” ⇒ B reduction instance “Yes” AND
  - B reduction instance “Yes” ⇒ A instance “Yes”

Relevant NP-Hard Problems and descriptions

Packing Problems

- **Independent Set**: for graph G and natural number k, find set of vertices V of size >= k where no two vertices are connected.
- **Set Packing**: given universe of elements U, a family S of subsets of U, and a natural number k, is there a collection of at least k sets where the sets are element-disjoint?

Covering Problems

- **Vertex Cover**: find a set of vertices in a graph such that there is at least one vertex for each edge in the graph.
- **Set Cover**: given a universe U, a family S of subsets of U, and a natural number k, is there a collection of sets which covers U and is of size at most k?

Relevant NP-Hard Problems and descriptions

Partitioning Problems

- **3D Matching**: given 3 disjoint sets X, Y, Z, all of size n, and a collection of tuples each containing (x_i, y_j, z_k), is there a subset of these tuples which covers all the elements in the 3 sets without repeating elements?
- **Graph Colouring**: Is there a way to color a graph with at most k colors such that no 2 adjacent vertices have the same color?
- **Clique**: Is there a fully connected subgraph in G of size at least k?

Numerical Problems

- **Subset Sum**: Given a set of numbers, is there a subset which adds to a specific target t?
- **Knapsack**: Can a value of at least V be achieved without exceeding the weight W by placing items into the knapsack?

Relevant NP-Hard Problems and descriptions

Constraint Satisfaction Problems

- **3SAT**: Given a set of literals X_1 … X_n and a collection of clauses c_1 … c_m, where each clause is an OR of 3 literals. Is there an assignment of the literals such that c_1 AND c_2 … AND c_m = True?

Sequencing Problems

- **Traveling Salesperson Problem (TSP)**: Given a list of cities and distances between cities, what is the shortest path which visits each city exactly once and returns to the origin city?
- **Hamiltonian Cycle**: Is there a cycle through the graph which visits each node exactly once?
- **Hamiltonian Path**: Is there a path between two vertices in the graph which visits each vertex exactly once?

Randomization

Know

- The properties of expectation, such as linearity of expectation, i.e.
  - E[X_1 + .. . + X_n] = E[X_1] + … + E[X_n]
  - E[a*X_1] = a * E[X_1]
  - E[X + Y] = E[X] + E[Y]
  - E[X * Y] = E[X] * E[Y] ONLY IF X and Y are **independent**
- Know the expectation for a unary (i.e. 0 or 1) random variable:
  - E[X_i] = 0 * P(X_i = 0) + 1 * P(X_i = 1) = P(X_i = 1)
- Be aware of the harmonic series being bounded by log(n) and remember that it doesn't have to be the exact harmonic series, can be different up to constant factors.

$$\sum_{i=1}^{n} \frac{1}{i} = H_n = O(\log n)$$