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, increasing 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) $\log_2 f(n)$ 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 $c_1 = \frac{\log_2 c_0}{\log_2 g(n_0)} + 1$. 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 $O(\log_2 g(n))$.

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

   (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) $f(n)^2$ 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 $O(g(n)^2)$.

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 asympto tically tight bound of $\Theta(f(n))$ on the running time.)

Solution:

We have a triple-nested loop, in which the outer loop is $\Omega(n)$, the middle loop is $\Omega(n)$, and the inner loop is $\Omega(n)$. The other work within the loops is constant, so this algorithm is $\Omega(n^3)$.

(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
```

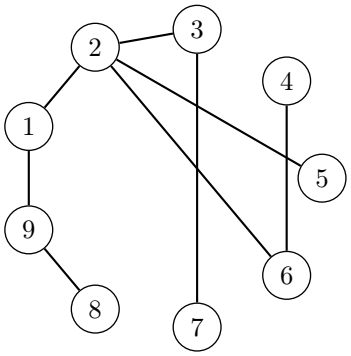
To fill in the rest of the array:

```
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 loop runs in $O(n)$ time and the second loop 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

7. Implement depth-first search in either C, C++, C#, Java, or Python. 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 week 1's 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. You can assume that the nodes are listed line-by-line in lexicographic order (0-9, then A-Z, then a-z), and the adjacent nodes of a node are listed in lexicographic order. For example, consider two consecutive lines of an instance:

   ```
   0, F
   B, C, a
   ```

   Note that 0 < B and C < a.

   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, and each line should be terminated by a newline.

   Sample:

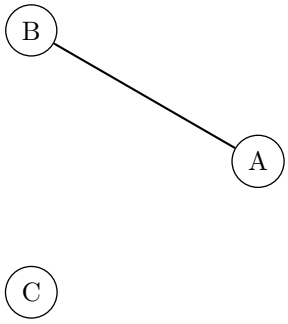
   Input:

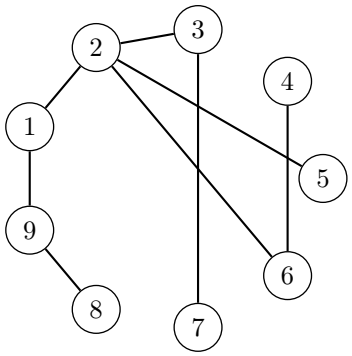
   ```
   2
   3
   A B
   B A
   C
   9
   1 2 9
   2 1 6 5 3
   4 6
   6 2 4
   5 2
   3 2 7
   7 3
   8 9
   9 1 8
   ```

   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.


B
A
C


3
2
4
1
5
9
6
8
7

CS 577 # Assignment 3: Greedy Algorithms Fall 2022

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 (c).

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 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: Note: We assume that the road is straight.

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'll use a "greedy stays ahead" approach. First, we show by induction that the ith tower the greedy algorithm places is at least as far along the road as the ith tower any optimal algorithm places. Let g_i be the position of the ith tower placed by the greedy algorithm, and o_j be the position of the jth tower placed by the optimal algorithm. We have $g_1 < g_2 < \ldots < g_m$ and $o_1 < o_2 < \ldots < o_n$, where the greedy algorithm places m towers and the optimal algorithm places n. We would like to show $g_i \geq o_i$ for all $1 \leq i \leq n$.

For the base case, we have $g_1 \geq o_1$ because both algorithms must cover the first house, and the greedy algorithm places the first tower as far along the road as possible while still covering the first house.

Now assume $g_{k-1} \geq o_{k-1}$ for some $k \leq n-1$. Suppose the greedy algorithm puts tower k 4 miles down the road from house h at position x_h, so $g_k = x_h + 4$. Since the greedy algorithm only would have done this if h was not already in range of the tower greedy places at g_{k-1}, we have $x_h - g_{k-1} > 4$. Then $x_h - o_{k-1} \geq x_h - g_{k-1} > 4$, so h is also not in range of the tower at o_{k-1}. Thus the optimal algorithm must also place tower k to cover house h, so $o_k \leq x_h + 4 = g_k$. This completes the proof by induction that $g_i \geq o_i$ for all $1 \leq i \leq n$.

Now use another induction to show that our algorithm always uses the minimum number of towers required to cover houses $1 \ldots k$ for any given k. (This is our inductive hypothesis.)

Base case: Any approach must use at least one tower to cover one house, and our algorithm uses exactly one.

Inductive step: Assume the hypothesis holds on $1 \ldots k$, so the greedy and optimal solutions both use i towers to cover houses $1 \ldots k$. Let house $k+1$ be at position x_{k+1}. Either the greedy algorithm uses an extra tower to cover house $k+1$ or it doesn't. If it doesn't, then the greedy solution on $1 \ldots k$ is still optimal on $1 \ldots k+1$. If it does use an extra tower, then the ith tower it placed didn't cover house $k+1$, so we have $x_{k+1} - o_i \geq x_{k+1} - g_i > 4$, so the optimal algorithm must also use an extra tower to cover house $k+1$.

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.

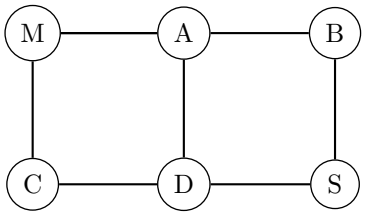
Solution: Run Dijkstra's shortest paths algorithm, starting at Madison. Whenever we add node x to our shortest paths tree with shortest path $M \to x$ of length t, we query the forecasting site for the lengths of all edges outgoing from x at time t. Edge "lengths" change as a function of time, but since 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 $i < j$, 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: $O(|E|)$.
- Changing the MST bits: $4 \times O(|E|)$

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.

5. Coding problem

For this question you will implement Furthest in the future paging in either C, C++, C#, Java, or Python.

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.

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 15 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
```

CS 577 **Assignment 5 – Divide and Conquer** Spring 2023

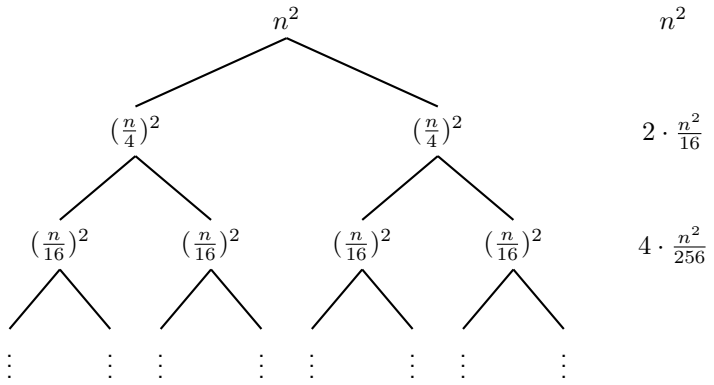
Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

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) $C(n) = 2C(n/4) + n^2$; $C(1) = 1$.

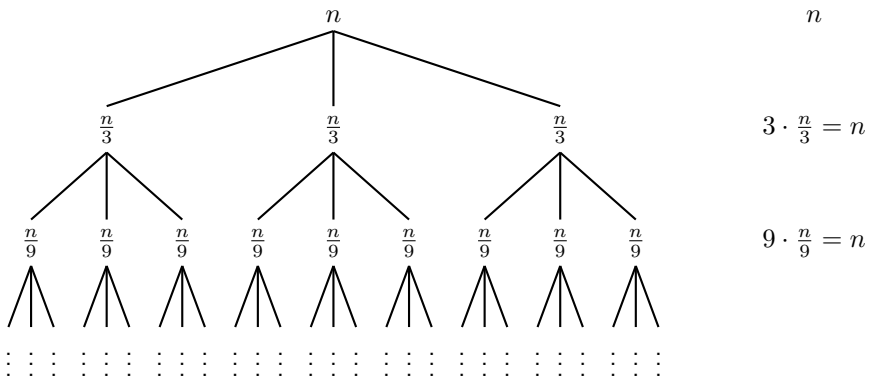
Solution:



Let d be the tree depth. The 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) $E(n) = 3E(n/3) + n$; $E(1) = 1$.

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 kth 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 Cases: If $A_{upper} - A_{lower} = 0$ and $B_{upper} - B_{lower} = 0$, the median is $min(a, b)$. If $A_{upper} - A_{lower} = 1$ and $B_{upper} - B_{lower} = 1$, the median is the 2nd smallest of the remaining 4 values.

Recursive Case:

Let $k_A = \frac{A_{lower}+A_{upper}}{2}$ and $k_B = \frac{B_{lower}+B_{upper}}{2}$. Query A on $\lfloor k_A \rfloor$ and B on $\lfloor k_B \rfloor$.

If $a > b$, set $A_{upper} = \lceil k_A \rceil$ and $B_{lower} = \lfloor k_B \rfloor$ (don't change A_{lower} and B_{upper}) and recurse.

If $a < b$, set $A_{lower} = \lfloor k_A \rfloor$ and $B_{upper} = \lceil k_B \rceil$ (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.

Solution: 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 (up to floors and ceilings, which don't affect the asymptotic runtime), either to

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

if $a > b$, or to

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

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. $(*)$ holds for the bases cases. 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 $\lceil (A_{upper} - A_{lower})/2 \rceil$ elements smaller than a, hence smaller than x, and $\lceil (B_{upper} - B_{lower})/2 \rceil$ elements larger than b, hence larger than x. By symmetry of the A and B indices, $\lceil (A_{upper} - A_{lower})/2 \rceil = \lceil (B_{upper} - B_{lower})/2 \rceil$, so, to create $R'_A \cup R'_B$, we have removed from $R_A \cup R_B$ an equal number of elements larger than 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.

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).* 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.

   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. Implement the 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 ranking.

   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 elements in the ranking. 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
```

CS 577 # Assignment 6 – Divide and Conquer 2 Spring 2023

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.

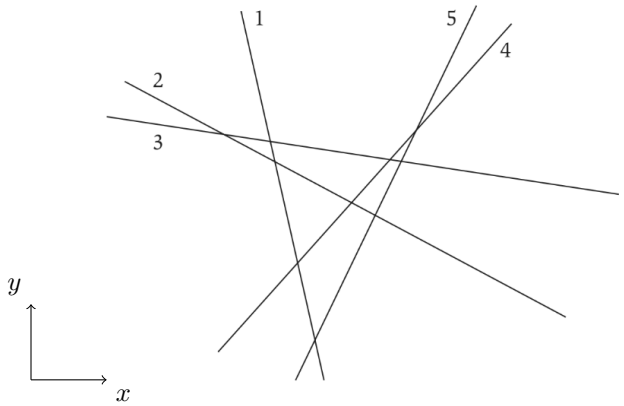


Figure 5.10 An instance of hidden surface removal with five lines (labeled *1–5* in the figure). All the lines except for *2* are visible.

   (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: Given two non-parallel, non-vertical, infinite lines they cross at a single point, and each is hidden from that point out to infinity in one direction or the other. As a result, a visible line is visible over a single (possibly infinite) interval. Additionally, given a set of lines $1 \dots k$, exactly one of those lines is visible over the others at any x coordinate except the coordinates where two intersect.

We conduct a modified mergesort on the n lines.

Label each line with the start of its visible interval, initially $-\infty$. To merge two sets of lines A and B, assume that they have been pre-sorted so that the first elements are the lines with the left-most visible intervals.

Starting with the first line in A and B, in constant time, find the coordinate x_0 where these two lines cross. The one that is higher on the left is the new first element of the merged set with label $-\infty$. If we most recently noted visible line A_i over B_i with crossing point x_i, then the next visible line is either A_{i+1} at its current label, or B_j for some $j \geq i$ at where A_i and B_j cross. Note that we can ensure that each line in both sets is "checked" at most once by doing a sweep over the set of lines. That is, if B_i is considered not visible, then we do not recheck it in the future again. So the entire merge process can be done linearly.

(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: First, the merge process is correct as it considers all three possibility of the next visible line.

Claim: Given a set of lines as input, the algorithm returns a sorted list of visible lines.
Proof: By strong induction on k, the size of the set of lines.

- Base case: $k = 1$, correctly returns the line as it is the only visible line. $k = 2$, correctly returns both lines in order by definition of the merge process.
- Inductive step: By induction hypothesis, the two sets of lines A and B are sorted. By definition of the merge process, the correct sorted list of visible lines will be returned.

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. Here we explore the structure of some different recursion trees than the previous homework.
   (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

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

5. 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, you should be able to develop an algorithm that runs in $O(n \log n)$ time.
   Hint: What does this problem have in common with the problem of 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.

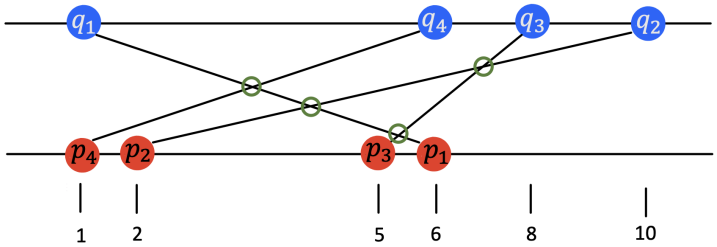


Figure 1: An example for the line intersection problem where the answer is 4

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 in C\C++, the results of some of the test cases may not fit in a 32-bit integer. If you are using C\C++, make sure you use a 'long long' to store your final answer.

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 2022

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:

- 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 $s[n]$. The schedule can be reconstructed 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:

- 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. *Kleinberg, Jon. Algorithm Design (p.333, q.26).*

Consider the following inventory problem. You are running a company that sells trucks and predictions tell you the quantity of sales to expect over the next n months. Let d_i denote the number of sales you expect in month i. We'll assume that all sales happen at the beginning of the month, and trucks that are not sold are stored until the beginning of the next month. You can store at most s trucks, and it costs c to store a single truck for a month. You receive shipments of trucks by placing orders for them, and there is a fixed ordering fee k each time you place an order (regardless of the number of trucks you order). You start out with no trucks. The problem is to design an algorithm that decides how to place orders so that you satisfy all the demands $\{d_i\}$, and minimize the costs. In summary:

- There are two parts to the cost: (1) storage cost of c for every truck on hand; and (2) ordering fees of k for every order placed.
- In each month, you need enough trucks to satisfy the demand d_i, but the number left over after satisfying the demand for the month should not exceed the inventory limit s.

(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, while if $j = d_i + j'$, there is no ordering fee. The case of $j > d_i + j'$ is not possible, as all unsold trucks must be stored.

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

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:

- 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 $m[n][j] = k + c \cdot j$ for all $j < d_n$, $m[n][j] = c \cdot j$ 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 $j > d_n$.
- Define $f(i, j, j') = 0$ if $j \ge d_i + j'$ and 1 otherwise. That is, f is an indicator function, where i is the current month, j is the storage from month $i-1$ to i, and j' is the storage from month i to $i+1$. f is 1 if we need to order more trucks (if our current storage j is not enough to cover both the demand for next month, and the number of trucks we plan to store after next month).
- Bellman Equations for $i = n-1$ to 1 and $j = 0$ to s,

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

- 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 $c \cdot j$. 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 k is included as determined by $f(i, j, j')$. 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 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. 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.

   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.

CS 577 # Assignment 8 – Dynamic Programming 2 Spring 2023

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. 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.

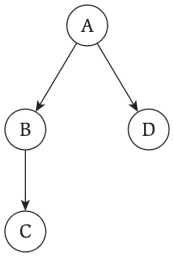


Figure 1: A hierarchy with four people. The fastest broadcast scheme is for A to call B in the first round. In the second round, A calls D and B calls C. If A were to call D first, then C could not learn the news until the third round.

The questions are on the next page.

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**
  $L.append($`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:

Let $R(v)$ denote the minimum number of rounds neede 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 give 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 considred 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 runtim 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 is some indicies $i < j$ 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:

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(`$s, X[2\ldots], Y$`)`;
end
if $s[0] = Y[0]$ **then**
  $b_Y \leftarrow$ `Interleave(`$s, X, Y[2\ldots]$`)`;
end
return $b_X \vee b_Y$;

(b) Give an efficient dynamic programming algorithm.

Solution:

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 intial 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 repeitions 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 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 $p_1, p_2, \ldots, p_m$ 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 $p_1, \ldots, p_k$ start with the edge (i, j). If $p'_1, \ldots, p'_k$ 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, 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. Implement the algorithm for the Knapsack Problem in either C, C++, C#, Java, or Python. 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 positive 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:

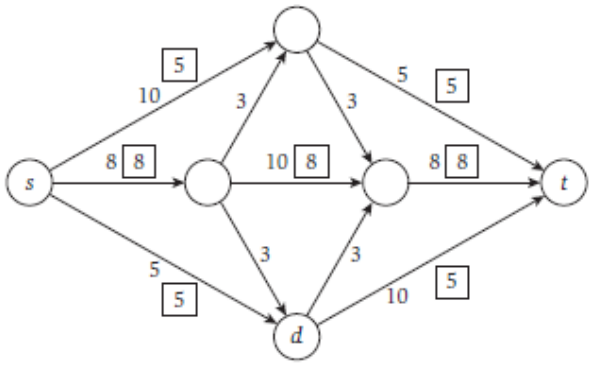
```
0
6
```

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.

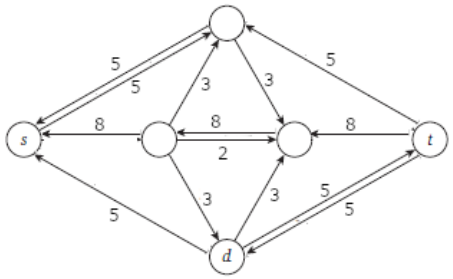


(a) What is the value of this flow?

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 $s \to (up) \to (right) \to (left) \to (d) \to t$

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)* A friend of yours has written a very fast piece of code to calculate the maximum flow based on repeatedly finding augmenting paths. However, you realize that it's not always finding the maximum flow. Your friend never wrote the part of the algorithm that uses backward edges! So their program finds only augmenting paths that include all forward edges, and halts when no more such augmenting paths remain. (Note: We haven't specified *how* the algorithm selects forward-only augmenting paths.)

   When confronted, your friend claims that their algorithm may not produce the maximum flow every time, but it is guaranteed to produce flow which is within a factor of b of maximum. That is, there is some constant b such that no matter what input you come up with, their algorithm will produce flow at least $1/b$ times the maximum possible on that input.

   Is your friend right? Provide a proof supporting your choice.

   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 blood supply on hand, 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 $|E|$ 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. The nodes are not listed separately, but are numbered $\{1 \ldots n\}$.

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

   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.

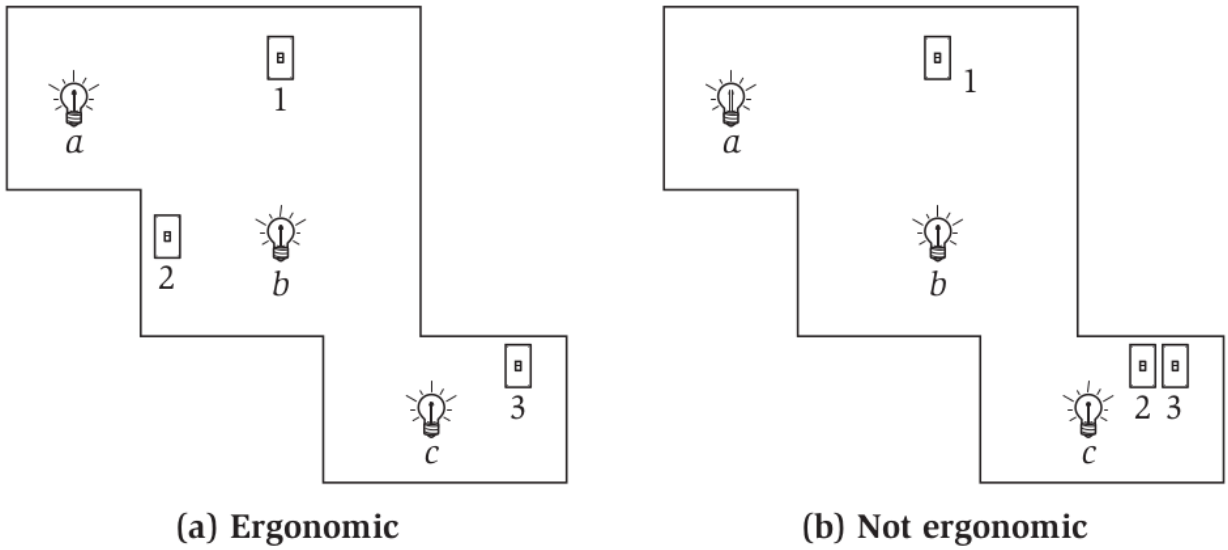


Figure 1: The floor plan in (a) is ergonomic, because we can wire switches to fixtures in such a way that each fixture is visible from the switch that controls it. (This can be done by wiring switch 1 to a, switch 2 to b, and switch 3 to c.) The floor plan in (b) is not ergonomic, because no such wiring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

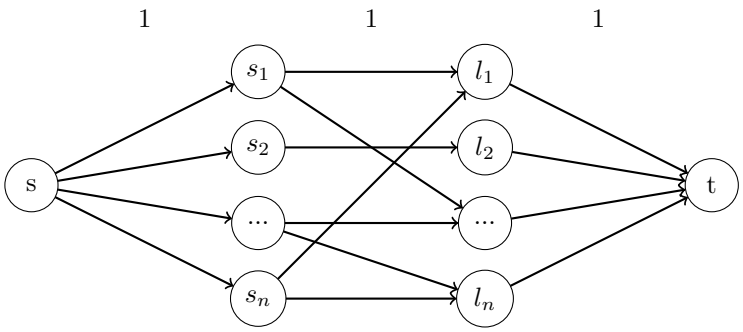
Let's call a floor plan, together with n light fixture locations and n switch locations, ergonomic if it's possible to wire one switch to each fixture so that every fixture is visible from the switch that controls it. A floor plan will be represented by a set of m horizontal or vertical line segments 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.
   - There will be edge from switch i to light j if it does not intersect any of the m walls.
   - Overall this will take $O(n^2m)$ time, i.e., check each n^2 pairs against each wall segment.
2. Run the bipartite matching max-flow algorithm: $O(n^2)$.
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 $O(n^2m)$.



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.

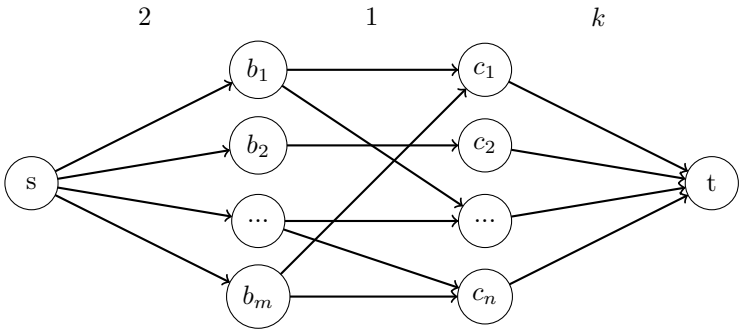
(a) Give a polynomial-time algorithm that takes the input to an instance of this problem (the n conditions, the sets S_i for each of the m balloons, and the parameter k) and decides whether there is a way to measure each condition by k different balloons, while each balloon only measures at most two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

Solution:

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

Solution 1:

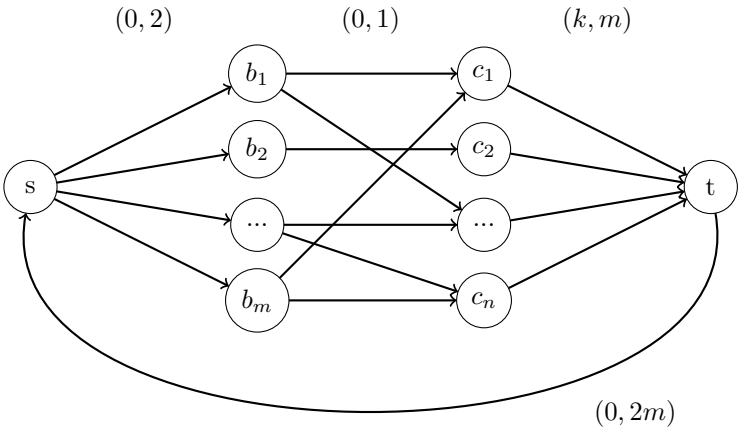
- An edge from b_i to c_j if b_i can measure c_j with capacity 1.
- A node s with edges to each balloon with capacity 2.
- A node t with edges from each condition with capacity k.
- All conditions can be measured if the max-flow is $|S| * k$ (number of conditions * number of balloons that must measure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

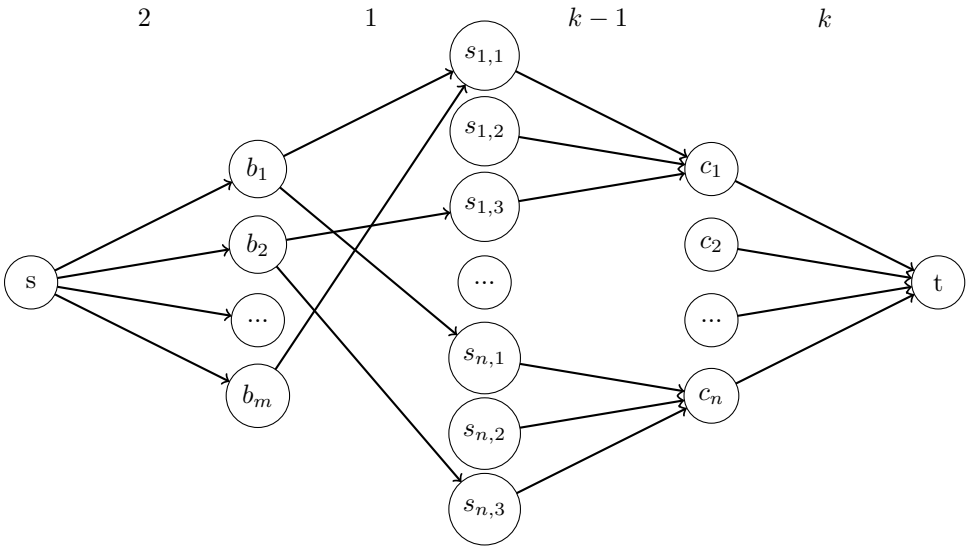
Solution:

- 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 $|S| * k$

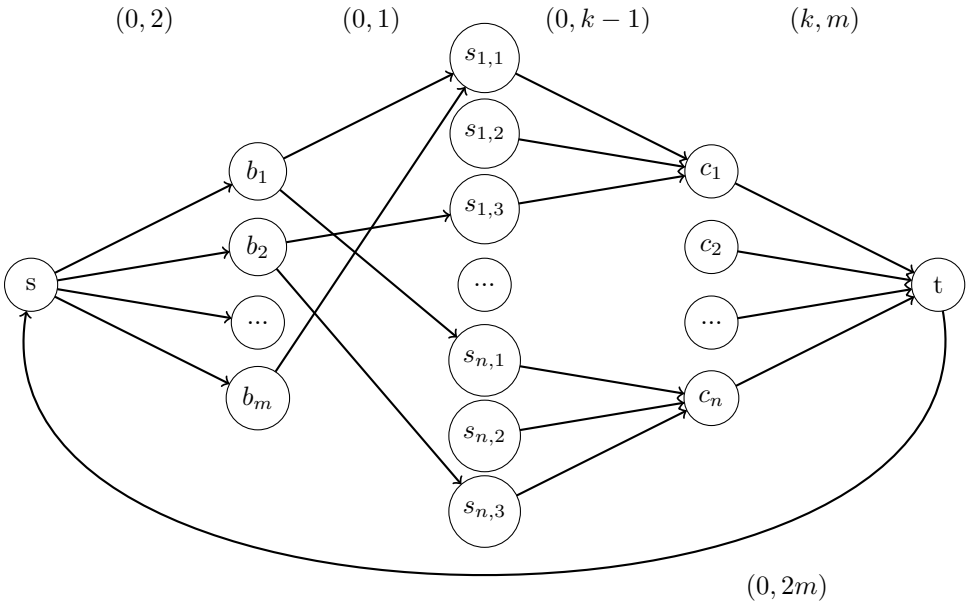
With the edge from subcontractor to condition with max capacity of $k-1$, no condition will be exclusively measured by balloons from just that subcontractor.



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.

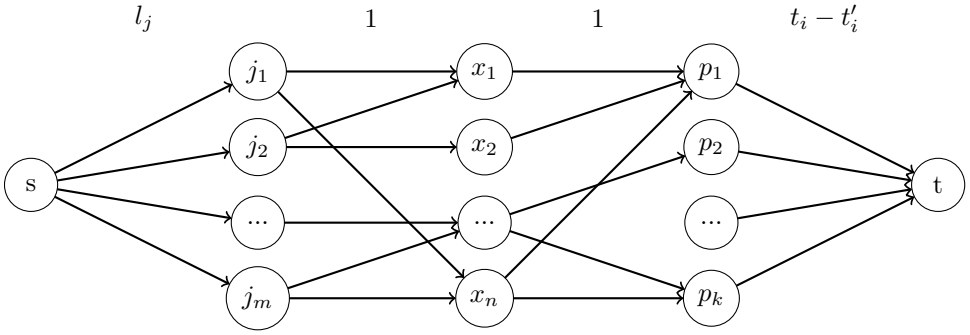
Given all this data about job requirements and processor availability, you'd like to decide whether the jobs can all be completed or not. Give a polynomial-time (in k, m, and n) algorithm that either produces a schedule completing all jobs by their deadlines or reports (correctly) that no such schedule 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 exits if max-flow is $\sum_{j=1}^{m} l_j$



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

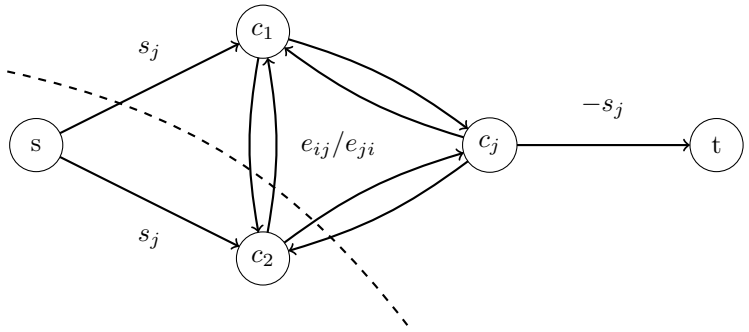
Consider the following definition. We are given a set of n countries that are engaged in trade with one another. For each country i, we have the value s_i of its budget surplus; this number may be positive or negative, with a negative number indicating a deficit. For each pair of countries i, j, we have 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 that is not equal to the full set.

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.



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 $1 \leq i \leq n$ 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. At any point in time, each process specifies a set of resources that it requests to use. Each resource might be requested by many processes at once; but it can only be used by a single process at a time. 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 of k processes, we can verify that there is no overlap in the resources required by the processes in polynomial time. For each resource, count how many selected processes have requested that resource. This can be done in time $O(km)$. If all counts are ≤ 1, the solution is valid, if not, the solution is invalid. Checking the counts takes $O(m)$. Since a proposed solution 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.
   - Let the nodes of G be the set of processes
   - Let the edges of G be the set of resources
   - A process requires a resource if and only if the node is adjacent to the edge in G.

   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-Complete, 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 requires at most one resource of each type (In other words, each process requires one specific person and one specific piece of equipment.)

Solution:

- Let S denote the set of processes which only require one resource, where every request is disjoint in resource requirements. Discard all other processes which request a resource required by a process in S. (polynomial in number of processes)
- Create two sets of nodes, T_1 and T_2, for each type of resource. For each resource r requested by processes 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 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_i, x_j, y_k, z_ℓ), do there exist n 4-tuples from C such that each element of $W \cup Y \cup X \cup Z$ 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 n 4-tuples in C, it can easily be verified in polynomial time that each element in $W \cup X \cup Y \cup Z$ belongs to exactly one of the n tuples. 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 copy one of the sets and associated elements. We can let $T' \subseteq X \times X \times Y \times Z$ be the same ordered triples, but where we had $\langle x, y, z\rangle$ before, use $\langle x, x, y, z\rangle$. This is a polynomial-time transformation. In this way, we transform an arbitrary instance of 3-D Matching into an instance of 4-D Matching.

   ($\Rightarrow$) Observe that $X \cup X \cup Y \cup Z = X \cup Y \cup Z$, and that for any $a \in X \cup Y \cup Z$, we have $a \in \langle x, y, z\rangle \iff a \in \{x, y, z\} \iff a \in \langle x, x, y, z\rangle$. If there is a set of n triples that satisfy the instance of 3-Dimensional Matching, the corresponding transformed 4-tuples will also satisfy the transformed 4-Dimensional Matching instance, since the additional information is just a copy of what was supplied for the 3-Dimensional Matching instance.

   ($\Rightarrow$) 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, similarly 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 k (or fewer) variables to set to 1, along with a boolean formula Φ, we can easily verify in polynomial time (1) if Φ is monotone and (2) if Φ is satisfied on the k identified variables set to 1 and all other variables set to 0. So Monotone Satisfiability with Few True Variables in 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 $(x_u \vee x_v)$. The formula Φ is 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 in the vertex cover, setting the corresponding variable to 1 in Φ will result in every clause evaluating to true, which means 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 they 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' 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.

CS 577 # Assignment 12 – More Intractability Spring 2023

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: _____

Reductions

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 adjacent 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 adjacent 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)*

   An arbitrary instance of 3-SAT is defined with variables $x_1, \ldots, x_m$ and k clauses. This reduction is modeled on the reduction from 3-SAT to 3D Matching discussed in class.

   For each variable x_i, create a gadget. This gadget has a core and points. The core consists of k "start" nodes $s_i^1, \ldots, s_i^k$, k "target-adjacent" nodes $u_i^1, \ldots, u_i^k$, and k "target" nodes $t_i^1, \ldots, t_i^k$, for a total of $3k$ nodes. Call the point nodes $p_i^1, \ldots, p_i^{2k}$. Draw paths $s_i^a \to p_i^{2a-1} \to u_i^a \to t_i^a$ and $s_i^a \to p_i^{2a-2} \to u_i^{a-1} \to t_i^a$ for each $a \in \{1 \ldots k\}$.

   For each clause C_j , create a start node s_j and a target node t_j. Then for each x_i in the clause, create a path $s_j \to p_i^{2j-1} \to t_j$. If $\overline{x_i}$ is in the clause, instead create a path $s_j \to p_i^{2j} \to t_j$.

   Given a 3-SAT instance with m variables and k clauses, in polynomial time this transformation generates an instance of DDP with $k(2m+1)$ paths and $|V| = k(5m+2)$.

   ($\Rightarrow$) Suppose we have a satisfying assignment. If a given variable x_i is assigned True in that assignment, let its gadget core use the "even" paths. These are disjoint by construction. If x_i is in a clause, that clause has a path from start to target through an odd point in the x_i gadget, and so this is also disjoint. If x_i is assigned False, then the gadget core can use the "odd" paths by the same logic. Because the assignment is satisfying, at least one of the variables in each clause must match, and so each clause gadget includes at least one $s \to t$ path which is disjoint from the variable gadget it touches. Thus, there exists a set of disjoint paths covering all $s \to t$ pairs in the graph.

   ($\Leftarrow$) If we have k node-disjoint paths through the graph, then for each clause gadget we have a path which selects one of the variables in that clause to hold. Let each of these variable assignments be included in our candidate satisfying assignment. By construction, each of these additions satisfies its clause. It remains to show that we never pick both x_i and $\overline{x_i}$. But because the x_i gadget core requires that we pick "odds" or "evens", we can never have clause paths that include both x_i and $\overline{x_i}$ point nodes. So our candidate satisfying assignment is satisfying, and we may complete it by assigning any unassigned variables arbitrarily.

   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.

4. *Kleinberg, Jon. Algorithm Design (p. 508, q. 9)* The *Path Selection Problem* may look initially similar to the problem from the question 3. 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 adjacent to v_i. In other words: let $e_{j_1}, e_{j_2}, \ldots, e_{j_{h(i)}}$ be the edges adjacent 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 Figure **??** for 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.

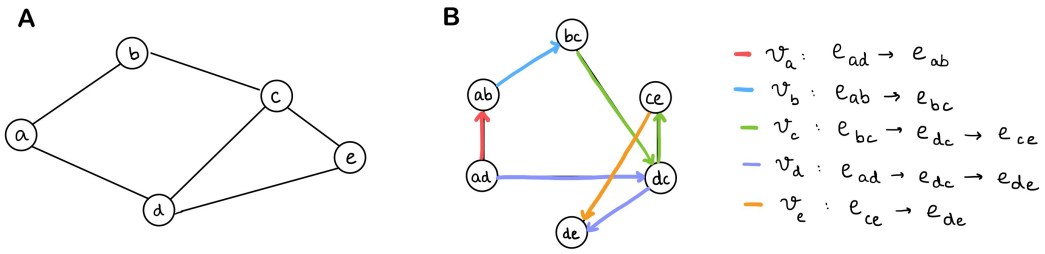


Figure 1: *Path Selection Problem.* **A** Example directed graph $G = (V, E)$. Observe that the maximum independent set for G has size 2 ($\{a, e\}, \{b, d\}, \{b, e\}$, or $\{a, c\}$). **B** We build a graph G'. For each edge in G, we create a node in G'. Then for every vertex v_i of G with adjacent 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:

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 variables are false, which has probability at most p^m (when there is a negating singleton for each variable). We further loose 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, or Python.

   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 $x_3 \wedge \overline{x_1} \wedge \overline{x_4}$. 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 $\lfloor 7/8 \rfloor m$ 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 3, so your assignment should satisfy at least $\lfloor \frac{7}{8} \times 3 \rfloor = 2$ clauses. One possible correct output to the sample input would be:

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