Solution: Final

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet.
- You have 180 minutes to earn a maximum of 180 points. Do not spend too much time on any one problem. Skim them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed three double-sided letter-sized sheet with your own notes**. No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. Pages will be scanned and separated for grading. If you need more space, write "Continued on S1" (or S2, S3, S4, S5, S6, S7) and continue your solution on the referenced scratch page at the end of the exam.
- Do not waste time and paper rederiving facts that we have studied in lecture, recitation, or problem sets. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. Be sure to argue that your **algorithm is correct**, and analyze the **asymptotic running time of your algorithm**. Even if your algorithm does not meet a requested bound, you **may** receive partial credit for inefficient solutions that are correct.
- **Pay close attention to the instructions for each problem**. Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points
1: Information	2	2
2: Decision Problems	10	40
3: Sorting Sorts	2	24
4: Pythagorean Quad	1	14
5: Animal Counting	1	20
6: Limited Connections	1	14
7: On the Street	1	14
8: RGB Graph	1	18
9: Separated Subsets	1	16
10: A Feast for Crowns	1	18
Total		180

Name: _____

School Email: _____

Problem 1. [2 points] **Information** (2 parts)

(a) [1 point] Write your name and email address on the cover page.
Solution: OK!

(b) [1 point] Write your name at the top of each page.
Solution: OK!

Problem 2. [40 points] **Decision Problems** (10 parts)

For each of the following questions, circle either **T** (True) or **F** (False), and **briefly** justify your answer in the box provided (a single sentence or picture should be sufficient). Each problem is worth 4 points: 2 points for your answer and 2 points for your justification. **If you leave both answer and justification blank, you will receive 1 point**.

(a) **T** **F** $2^{2n} \in \Theta(2^n)$.

Solution: False. This statement is equivalent to saying $k^2 \in O(k)$ for $k = 2^n$. Constants in exponents matter asymptotically!

(b) **T** **F** If $T(n) = \frac{9}{4}T\left(\frac{2}{3}n\right) + n^2$ and $T(1) = \Theta(1)$, then $T(n) = O(n^2)$.

Solution: False. This is an example of Case II of Master Theorem, since $a = \frac{9}{4}$, $b = \frac{3}{2}$, $f(n) = n^2$ and $n^2 = \Theta(n^{\log_{3/2} 9/4} \log^0 n)$. Thus, the recurrence evaluates to $T(n) = \Theta(n^2 \log n)$, which is not $O(n^2)$.

(c) **T** **F** Performing an $O(1)$ amortized operation n times on an initially empty data structure takes worst-case $O(n)$ time.

Solution: True. This is the definition of amortization.

(d) **T** **F** Given an array A containing n comparable items, sort A using merge sort. While sorting, each item in A is compared with $O(\log n)$ other items of A.

Solution: False. As a counter example, during the final merge step between two sorted halves of the array, each of size $\Theta(n)$, a single item from one array may get compared to all the items from the other list.

(e) **T** **F** Given a binary min-heap storing n items with comparable keys, one can build a Set AVL Tree containing the same items using $O(n)$ comparisons.

Solution: False. If such an algorithm A existed, we would be able to sort an array of comparable items in $O(n)$ time, which would contradict the $\Omega(n \log n)$ comparison sort lower bound. Specifically, we could build a binary min-heap from the array using $O(n)$ comparisons, use A to construct a Set AVL Tree in $O(n)$ comparisons, and then return its traversal order.

(f) **T** **F** Given a directed graph $G = (V, E)$, run breadth-first search from a vertex $s \in V$. While processing a vertex u, if some $v \in \text{Adj}^+(u)$ has already been processed, then G contains a directed cycle.

Solution: False. BFS can't be used to find directed cycles. A counterexample is $V = \{s, a, b, t\}$ and $E = \{(s, t), (s, a), (a, b), (b, t)\}$. Running BFS from s will first process vertices in levels $\{s\}$, then $\{a, t\}$, then $\{b\}$. When processing vertex b, vertex $t \in \text{Adj}^+(b)$ has already been processed, yet G is a DAG.

(g) **T** **F** Run Bellman-Ford on a weighted graph $G = (V, E, w)$ from a vertex $s \in V$. If there is a **witness** $v \in V$, i.e., $\delta_{|V|}(s, v) < \delta_{|V|-1}(s, v)$, then v is on a negative-weight cycle of G.

Solution: False. A witness is only guaranteed to be **reachable** from a negative-weight cycle; it may not actually be **on** a negative-weight cycle.

(h) T F Floyd–Warshall and Johnson's Algorithm solve all-pairs shortest paths in the same asymptotic running time when applied to weighted **complete** graphs, i.e., graphs where every vertex has an edge to every other vertex.

Solution: True. A complete graph is dense, i.e., $|E| = \Theta(|V|^2)$, so Johnson's algorithm runs in $O(|V|^2 \log |V| + |V||E|) = O(|V|^3)$ time, which is the same as Floyd–Warshall.

(i) T F If there is an algorithm to solve 0-1 Knapsack in polynomial time, then there is also an algorithm to solve Subset Sum in polynomial time.

Solution: True. Subset Sum is the special case of 0-1 Knapsack. Specifically, one can (in linear time) convert an instance (A, T) of Subset Sum into an equivalent instance of 0-1 Knapsack, with an item i for each integer $a_i \in A$ having size $s_i = a_i$ and value $v_i = a_i$, needing to fill a knapsack of size T; and then solve the instance via the polynomial-time algorithm for 0-1 Knapsack.

(j) T F Suppose a decision problem A has a pseudopolynomial-time algorithm to solve A. If P $\neq$ NP, then A is not solvable in polynomial time.

Solution: False. A problem could have a pseudopolynomial-time algorithm and a polynomial-time algorithm. In fact, any polynomial-time algorithm is also a pseudopolynomial-time algorithm!

Problem 3. [24 points] **Sorting Sorts**

(a) [12 points] An integer array A is ***k*-even-mixed** if there are exactly k even integers in A, and the odd integers in A appear in sorted order. Given a k-even-mixed array A containing n distinct integers for $k = \lceil n/\lg n \rceil$, describe an $O(n)$-time algorithm to sort A.

Solution: Scan through A and put all even integers in order into an array A_E and all odd integers in order into an array A_O (where $|A_E| = k$ and $|A_O| = n - k$). A_O is sorted by definition, and we can sort A_E in $O(k \log k) = O((n/\lg n)\log(n/\lg n)) = O(n)$ time, e.g., via merge sort. Then we can merge sorted A_E and A_O back into A in $O(n)$ time using the merge step of merge sort, using $O(n)$ time in total.

Common Mistakes:

- Using insertion sort
- Splitting into a non-constant number of subarrays and trying to merge
- Using binary search to insert evens into odds (but with linear shifts)
- Using radix or counting sort

(b) [12 points] Let A be an array of n pairs of positive integers (x_i, y_i) with $x_i, y_i < n^2$ for all $i \in \{0, \ldots, n-1\}$. The **power** of pair (x, y) is the integer $x + n^y$. Describe an $O(n)$-time algorithm to sort the pairs in A increasing by power.

Solution: First note that $x < n^y$ for any integer $y > 1$ and for any $x \in \{0, \ldots, n^2-1\}$. Scan through A and put all pairs having $y = 1$ into array A_1, and all other pairs into array A_2. Sort A_1 directly by computing and comparing their respective powers $x + n$. Since these values are bounded above by $O(n^2)$, sort A_1 in $O(n)$ time using Radix sort. To sort A_2, use tuple sort, sorting first by x values and then by y values (since power is more sensitive to changes in y). Since the x and y values are both bounded above by $O(n^2)$, we can use Radix sort for tuple sort's stable sorting algorithm to sort A_2 in $O(n)$ time. Then merge A_1 and A_2 back into A in $O(n)$ time using the merge step of merge sort, using $O(n)$ time in total.

Common Mistakes:

- Not handling $y = 1$ (x may be larger or smaller than n^1)
- Explicitly computing powers (which may have exponential size)
- Concatenating instead of merging two overlapping lists

Problem 4. [14 points] **Pythagorean Quad**

A **Pythagorean Quad** consists of four integers (a, b, c, d) such that $d = \sqrt{a^2 + b^2 + c^2}$. Given an array A containing n distinct positive integers, describe an $O(n^2)$-time algorithm to determine whether four integers from A form a Pythagorean Quad, where integers from A may appear more than once in the Quad. State whether your running time is worst-case, expected, and/or amortized.

Solution: First, we observe that it suffices to find (a, b, c, d) such that $a^2 + b^2 = d^2 - c^2$. Let P be the set of n^2 ordered pairs of integers from A, where integers in A may be repeated in a pair. Construct an empty hash table H, and for each pair $(a, b) \in P$, compute and insert value $a^2 + b^2$ into H. Then for each pair $(c, d) \in P$, compute and lookup value $d^2 - c^2$ in H. If the value is in H, then some $a^2 + b^2$ equals some $d^2 - c^2$, so return that a Pythagorean Quad exists. Otherwise, if no $d^2 - c^2$ exists in H, then return that a Pythagorean Quad does not exist. Each $a^2 + b^2$ or $d^2 - c^2$ value takes constant time to compute, so computing them all takes worst-case $O(n^2)$ time, while inserting them into or looking them up in the hash table takes expected $O(n^2)$ time, so this algorithm runs expected in $O(n^2)$ time in total.

Common Mistakes:

- Computing non-integers involving logarithms or square roots
- Saying a worst-case running time is amortized
- Trying to write an incorrect dynamic program
- Trying to write an incorrect four-finger algorithm

Problem 5. [20 points] **Animal Counting**

PurpleRock Park is a wildlife reserve, divided into zones, where each zone has a park ranger who records current **sightings** of animals of different **species** over time. Old animal sightings are periodically removed from the database. A species s is **common** if current park records contain at least 100 sightings of species s within any single zone of the park.

Describe a database to store animal sightings, supporting the following four operations, where n is the number of sightings stored in the database at the time of the operation. State whether your running times are worst-case, expected, and/or amortized.

`initialize()`	Initialize an empty database in $O(1)$ time
`add_sighting(s, i)`	Record a newest sighting of species `s` in zone `i` in $O(\log n)$ time
`remove_oldest()`	Remove the oldest sighting stored in the database in $O(\log n)$ time
`is_common(s)`	Return whether species `s` is common based on sightings that have not yet been removed from the database in $O(1)$ time

Solution: To implement the database, maintain the following data structures:

- A hash table H mapping each species s to a Set AVL tree T_s
- Each Set AVL Tree T_s stores pairs (i, c_i) of zone numbers i and the count c_i representing the number of sightings of species s in zone i, keyed by zone number.
- Augment each node x in each T_s by the maximum number of sightings $x.m$ of any zone in the subtree of x. $x.m$ can be maintained in $O(1)$ time from the augmentations of x's children, specifically $x.m = \max\{x.left.m, x.key, x.right.m\}$.
- A doubly-linked list L of all current sightings (s, i) in the order in which they were added to the database (oldest at the front).

To implement `initialize()`, initialize an empty H and empty L in worst-case $O(1)$ time.

To implement `add_sighting(s, i)`, lookup s in H to find T_s in expected $O(1)$ time (if s does not exist in H, insert s mapping to an empty T_s in expected amortized $O(1)$ time). Then find zone i in T_s. If zone i is not in T_s, insert $(i, 1)$ into T_s. Otherwise, i is in T_s, so remove (i, c_i) from T_s and reinsert $(i, c_i + 1)$. It takes worst-case $O(\log n)$ time to remove or insert items from T_s while maintaining augmentations (since at most n sightings could exist for species s). Lastly, Insert (s, i) to the back of L in worst-case $O(1)$ time. Thus this operation takes $O(\log n)$ expected amortized time, and maintains the invariants of the database directly.

To implement `remove_oldest()`, remove the oldest pair (s, i) from the front of L in worst-case $O(1)$ time. Lookup s in H to find T_s in expected $O(1)$ time; then lookup i in T_s and decrease c_i by one. If c_i is decreased to zero, remove i from T_s. If T_s becomes empty, remove s from H in expected amortized $O(1)$ time. This operation takes $O(\log n)$ expected amortized time, and maintains the invariants of the database directly.

To implement `is_common(s)`, simply lookup s in H and return whether s is in H and the stored max at the root of T_s is 100 or greater in expected $O(1)$ time. This operation is correct based on the invariants of the data structure.

Common Mistakes: Continued on S1

Problem 6. [14 points] **Limited Connections**

For any weighted graph $G = (V, E, w)$ and integer k, define $\boldsymbol{G_k}$ to be the graph that results from removing every edge in G having weight k or larger.

Given a connected undirected weighted graph $G = (V, E, w)$, where every edge has a unique integer weight, describe an $O(|E| \log |E|)$-time algorithm to determine the largest value of k such that G_k is not connected.

Solution: Construct an array A containing the $|E|$ distinct edge weights in G, and sort it in $O(|E| \log |E|)$ time, e.g., using merge sort. We will binary search to find k. Specifically, consider an edge weight k' in A (initially the median edge weight), and run a reachability algorithm (e.g., Full-BFS or Full-DFS) to compute the reachability of an arbitrary vertex $x \in V$ in $O(|E|)$ time. If exactly $|V|$ vertices are reachable from x, then $G_{k'}$ is connected and $k > k'$; recurse on strictly larger values for k'. Otherwise, $G_{k'}$ is not connected, so $k \leq k'$; recurse on non-strictly smaller values for k'. By dividing the search range by a constant fraction at each step (i.e., by always choosing the median index weight of the unsearched space), binary search will terminate after $O(\log |E|)$ steps, identifying the largest value of k such that G_k is not connected. This algorithm takes $O(|E| \log |E|)$ time to sort, and computes reachability of a vertex in $O(|E|)$ time, $O(\log |E|)$ times, so this algorithm runs in $O(|E| \log |E|)$ time in total.

Common Mistakes:

- Filtering linearly on all edge weights, rather than using binary search
- Using Dijkstra/Bellman-Ford to inefficiently solve reachability
- Simply stating 'Use Dijkstra'

Problem 7. [14 points] **On the Street**

Friends Dal and Sean want to take a car trip across the country from Yew Nork to Fan Sancrisco by driving between cities during the day, and staying at a hotel in some city each night. There are n cities across the country. For each city c_i, Dal and Sean have compiled:

- the positive integer expense $h(c_i)$ of **staying at a hotel** in city c_i for one night; and
- a list L_i of the at most 10 other cities they could drive to **in a single day** starting from city c_i, along with the positive integer expense $g(c_i, c_j)$ required to drive directly from c_i to c_j for each $c_j \in L_i$.

Describe an $O(nd)$-time algorithm to determine whether it is possible for Dal and Sean to drive from Yew Nork to Fan Sancrisco in at most d days, spending at most b on expenses along the way.

Solution: Let $C = \{c_0, \ldots, c_{n-1}\}$, and let c_s denote Yew Nork and let c_t denote Fan Sancrisco. Construct a graph G with:

- a vertex (c_i, d') for each city $c_i \in C$ and day $d' \in \{0, \ldots, d\}$,
  representing staying the night in city c_i on the night before day d'; and
- a directed weighted edge $((c_i, d'), (c_j, d' + 1))$ with weight $g(c_i, c_j) + h(c_j)$
  for each city $c_i \in C$, $c_j \in L_i$ and $d' \in \{0, \ldots, d - 1\}$.

Then the weight of any path in G from vertex $(c_s, 0)$ to any vertex (c_t, d') for $d' \in \{0, \ldots, d\}$ corresponds to the expenses incurred along a driving route from Yew Nork to Fan Sancrisco in at most d days (assuming they stay the night upon reaching c_t; other assumptions are also okay). G is acyclic, since each edge always connects a vertex from a smaller day to a larger day, so run DAG Relaxation to compute single-source shortest paths from $(c_s, 0)$ in G. Then return whether $\delta((c_s, 0), (c_t, d')) \leq b$ for any $d' \in \{0, \ldots d\}$. G has $O(nd)$ vertices and $O(nd)$ edges (since $|L_i| \leq 10$ for all $i \in \{0, \ldots, n - 1\}$), so DAG relaxation runs in $O(nd)$ time and checking all destination values takes $O(d)$ time, leading to $O(nd)$ time in total.

Common Mistakes:

- Not considering paths on fewer than d days (or considering paths on more than d days)
- Not enforcing staying at a hotel every night
- Not including hotel costs

Problem 8. [18 points] **RGB Graph**

Let $G = (V, E, w)$ be a weighted directed graph. Let $c : V \to \{r, g, b\}$ be an assignment of each vertex v to a **color** $c(v)$, representing red, green, or blue respectively. For $x \in \{r, g, b\}$,

- let V_x be the set of vertices with color x, i.e., $V_x = \{v \in V \mid c(v) = x\}$; and
- let E_x be the set of edges outgoing from vertices in V_x, i.e., $E_x = \{(u, v) \in E \mid u \in V_x\}$.

Suppose graph G and coloring c have the following properties:

1. Every edge in E either connects two vertices of the same color, goes from a red vertex to a green vertex, or goes from a green vertex to a blue vertex.
2. $|V_r| = |E_r| = O(|V|)$, and edges in E_r have identical positive integer weight w_r.
3. $|V_g| = |E_g| = O(|V|^{0.99})$, and edges in E_g have nonnegative integer weights.
4. $|V_b| = |E_b| = O(\sqrt{|V|})$, and edges in E_b can have positive or negative integer weights.

Given G, c, a red vertex $s \in V_r$, and a blue vertex $t \in V_b$, describe an $O(|V|)$-time algorithm to compute $\delta(s, t)$, the minimum weight of any path from s to t.

Solution: Any path from s to t is a path through edges in E_r, followed by a path through edges in E_g, followed by a (possibly empty) path through edges in E_g. So we compute minimum weight distances in G incrementally, first using edges in E_r, then using edges in E_g, then edges in E_b.

Step 1: Construct unweighted graph $G' = (V', E')$ composed of the edges $E' = E_r$ and the vertices appearing in those edges, specifically $V' = \bigcup_{(u,v)\in E_r} \{u, v\}$ (which contains vertices from V_r and V_g). Run breadth-first search from s in G' to compute unweighted distances. Then the minimum weight distance in G from s to any green vertex in $V' \cap V_g$ is w_r times the unweighted distance computed. G' has size $O(|V|)$, so this step takes $O(|V|)$ time.

Step 2: Now construct weighted graph $G'' = (V'', E'')$ composed of vertex s with a new directed edge to each green vertex in $V' \cap V_g$ weighted by its distance found in Step 1 (i.e., the minimum weight of any path from s to that vertex), along with weighted edges E_g and the vertices appearing in those edges. All the weights in G'' are positive, so run Dijkstra from s in G'' to compute minimum weight distances. Then the computed distance to any blue vertex v in $V'' \cap V_b$ is the minimum weight of any path from s to v in G that traverses only red or green edges. G'' has size $O(1 + |V_g| + |E_g|) = O(|V|^{0.99})$, so this step takes $O(|V|^{0.99} \log |V|^{0.99}) = O(|V|)$ time.

Step 3: Now construct a weighted graph $G''' = (V''', E''')$ composed of vertex s with a new directed edge to each blue vertex in $V'' \cap V_b$ weighted by its distance found in Step 2 (i.e., the minimum weight of any path from s to that vertex), along with weighted edges E_b and the vertices appearing in those edges. Weights in G''' may be positive or negative, so run Bellman-Ford from s in G''' to compute weighted minimum weight distances. Then the computed distance to t is the minimum weight of any path from s to t in G, as desired. G''' has size $O(1 + |V_b| + |E_b|) = O(\sqrt{|V|})$, so this step takes $O(\sqrt{|V|}\sqrt{|V|}) = O(|V|)$ time, leading to $O(|V|)$ time in total.

Common Mistakes: Continued on S1.

Problem 9. [16 points] **Separated Subsets**

For any set S of integers and for any positive integers m and k, an **(m, k)-separated subset** of S is any subset $S' \subseteq S$ such that S' sums to m and every pair of distinct integers $a, b \in S'$ satisfies $|a - b| \geq k$. Given positive integers m and k, and a set S containing n distinct positive integers, describe an $O(n^2 m)$-time algorithm to **count the number** of (m, k)-separated subsets of S.

(When solving this problem, you may assume that a single machine word is large enough to hold any integer computed during your algorithm.)

Solution:

1. **Subproblems**
   - First sort the integers in S increasing into array A in $O(n \log n)$ time, e.g., via merge sort
   - where $A = (a_0, \ldots, a_{n-1})$
   - $x(i, j)$: the number of (j, k)-separated subsets of suffix $A[i:]$
   - for $i \in \{0, \ldots, n\}$ and $j \in \{0, \ldots, m\}$
2. **Relate**
   - Sum the number of (j, k)-separated subsets using $A[i]$ with the ones that do not use $A[i]$
   - If $A[i] \leq j$ is used:
     - Then no integer in $A[i:]$ smaller than $A[i] + k$ may be used
     - Let $f(i)$ be the smallest index greater than i such that $A[f(i)] - A[i] \geq k$
     - Then recrusively count $x(f(i), j - A[i])$
   - Otherwise, $A[i]$ is not used and we can recursively count $x(i + 1, j)$
   - $x(i, j) = \sum \left\{ \begin{array}{ll} x(f(i), j - A[i]) & \text{if } A[i] \leq j, \\ x(i + 1, j) & \text{always} \end{array} \right\}$
3. **Topo**
   - Subproblem $x(i, j)$ only depends on strictly larger i, so acyclic
4. **Base**
   - $x(n, 0) = 1$, the empty subset can always be acheived
   - $x(n, j) = 0$ for $j > 0$, empty sets cannot sum to a positive number
5. **Original**
   - $x(0, m)$, the number of (m, k)-separated subsets of A
6. **Time**
   - # subproblems: $(n + 1)(m + 1) = O(nm)$
   - Work per subproblem: $O(n)$ to find $f(i)$ by linear scan
   - $O(n^2 m)$ time in total
   - (Note that it is possible to compute $f(i)$ in $O(\log n)$ time via binary search, or in amortized $O(1)$ time from $f(i-1)$, but these optimizations are not necessariy for full points.)

Common Mistakes: Continued on S1.

Problem 10. [18 points] **A Feast for Crowns**

Ted Snark is arranging a feast for the Queen of Southeros and her guests, and has been tasked with seating them along one side of a long banquet table.

- Ted has a list of the $2n$ guests, where each guest i has a known distinct positive integer f_i denoting the guest's **favor** with the Queen.
- Ted must seat the guests **respectfully**: the Queen must be seated in the center with n guests on either side so that guests' favor monotonically decreases away from the Queen, i.e., any guest seated between a guest i and the Queen must have favor larger than f_i.
- Additionally, every guest hates every other guest: for every two guests i, j, Ted knows the positive integer **mutual hatred** $d(i, j) = d(j, i)$ between them.

Given Ted's guest information, describe an $O(n^3)$-time algorithm to determine a respectful seating order that minimizes the sum of mutual hatred between pairs of guests seated next to each other. **Significant partial credit** will be awarded to correct $O(n^4)$-time algorithms.

Solution:

1. **Subproblems**
   - Sort the guests increasing by favor in $O(n \log n)$ time into $F = (f_0, \ldots, f_{2n-1})$
   - Any partition of F into two length-n subsequences corresponds to a respectful seating
   - $x(i, j_L, j_R, n_L)$: minimum total hatred of adjacent guests possible by respectfully seating the $n - i$ guests from suffix $F[i :]$ next to the Queen, with n_L guests to the left and $n_R = (n-i) - n_L$ guests to the right, where guest $j_L < i$ has already been seated $n_L + 1$ places to the left, and guest $j_R < i$ has already been seated $n_R + 1$ places to the right.
   - for $i \in \{0, \ldots, 2n\}$, $j_L, j_R \in \{-1, \ldots, 2n-1\}$ and $n_L \in \{0, \ldots, n\}$ where either $j_L = i - 1$ or $j_R = i - 1$
   - Let $d(-1, i) = d(i, -1) = 0$ for all $i \in \{0, \ldots, 2n-1\}$ (no hatred at the end of table)
2. **Relate**
   - Guess whether guest i is seated on the right or left
   - Sitting next to j_L costs hatred $d(i, j_L)$; sitting next to j_R costs hatred $d(i, j_R)$
   - $x(i, j_L, j_R, n_L) = \min \left\{ \begin{array}{ll} d(i, j_L) + x(i+1, i, j_R, n_L - 1) & \text{if } n_L > 0, \\ d(i, j_R) + x(i+1, j_L, i, n_L) & \text{if } (n-i) - n_L > 0 \end{array} \right\}$
3. **Topo**: Subproblem $x(i, j_L, j_R, n_L)$ only depends on strictly larger i, so acyclic
4. **Base**: $x(2n, j_L, j_R, 0) = 0$ for all $j_L, j_R \in \{0, \ldots, 2n\}$ (no hatred if no guests)
5. **Original**: $x(0, -1, -1, n)$, min hatred of adjacent guests by respectfully seating all guests
6. **Time**
   - # subproblems: though there are four parameters, there are only $O(n^3)$ subproblems for which either $j_L = i - 1$ or $j_R = i - 1$
   - Work per subproblem: $O(1)$, so $O(n^3)$ time in total

Common Mistakes: Continued on S1.

SCRATCH PAPER 1. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S1" on the problem statement's page.

Common Mistakes: (for Problem 5)

- Not removing records for sightings or zones containing zero sightings
- Not accounting for existence or non-existence of a key in a data structure
- Algorithm is linear in number of zones or does account for zones at all
- Forgetting to say amortized for dynamic hash table operations
- Confusing Sequence/Set AVL Trees, or dynamic/direct-access arrays
- Not maintaining or updating number of sightings per zone per species correctly

Common Mistakes: (for Problem 8)

- Correctly finding shortest paths within each color, but not connecting them properly
- Assuming an optimal path from s to t goes through the green vertex closest to s
- Solving APSP on the red or green graph (inefficient)
- Trying to apply DAG relaxation to a possibly cyclic graph
- Assuming that if $|V| = |E|$, graph has at most one cycle (only true if connected)

Common Mistakes: (for Problem 9)

- Maintaining a subset of used items (yields exponential state)
- Unconrstrained subproblem for relation that relies on a constraint
- Solving the decision problem rather than the counting problem (or max instead of sum)
- Missing a base case

Common Mistakes: (for Problem 10)

- Not ensuring that n guests are seated to either side of Queen
- Not keeping track of guests on either end
- Defining $\Omega(n^3)$ subproblems whose parameters are not independent
- Maintaining a subset of used items (yields exponential state)

SCRATCH PAPER 2. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S2" on the problem statement's page.

SCRATCH PAPER 3. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S3" on the problem statement's page.

SCRATCH PAPER 4. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S4" on the problem statement's page.

SCRATCH PAPER 5. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S5" on the problem statement's page.

SCRATCH PAPER 6. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S6" on the problem statement's page.

SCRATCH PAPER 7. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S7" on the problem statement's page.

6.006 Introduction to Algorithms

Spring 2020