Final Exam

- 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 every page of this quiz booklet.
- You have 180 minutes to earn 180 points.
- **You are allowed a 3-page cheat sheet**. No calculators or programmable devices are permitted. No cell phones or other communications devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem.
- Do not waste time and paper rederiving facts that we have studied. Simply cite them.
- Throughout the exam, you can utilize the Simple Uniform Hashing Assumption **unless otherwise specified in the problem**.
- Running time analyses can be either expected or worst-case; **clearly state which one.**
- When describing an algorithm, a description in English suffices. (No pseudocode required.)

Problem	Parts	Points	Grade	Grader
1	15	30		
2	10	30		
3	3	15		
4	1	20		
5	1	15		
6	2	15		
7	2	15		
8	1	20		
9	1	20		
Total		180		

Name: _____

Circle your recitation:	R01 Kevin King **10AM**	R02 Christos Tzamos **11AM**	R03 Kevin Chen **12PM**	R04 Amol Bhave **12PM**	R05 Hongyu Yang **1PM**	R06 Matthew Chang **1PM**	R07 Daniel Manesh **2PM**	R08/09 Danil Tyulmankov **2,3PM**	R10 Siyao Xu **4PM**

Problem 1. True or False [30 points] (15 parts)

For each of the following questions, circle either **T** (True) or **F** (False). There is no need to justify the answers. Each problem is worth 2 points.

(a) T F The number of comparisons a sorting algorithm makes is represented by the total number of edges in a decision tree corresponding to the algorithm.

Solution: False. The number of steps is represented by the length of a path from the root to a leaf.

(b) T F If we insert n numbers $1 \dots n$ into an empty AVL tree (in an increasing order) then the depth of the resulting AVL tree might be $\Omega(n)$.

Solution: False

(c) T F When creating a new heap out of an array A, we do not need to call MAX-HEAPIFY on the last $\lceil n/2 \rceil$ elements, i.e., $A[\lceil n/2 \rceil : n]$.

Solution: True

(d) T F We have $(\log n)^{\log \log n} = O((\log \log n)^{\log n})$.

Solution: True.

(e) T F The runtime of Insertion Sort is $\Omega(n)$.

Solution: True

(f) T F Suppose you have designed a hash table with load factor $\alpha = 0.25$. Then, open addressing with linear probing would ensure that a search operation would take expected constant time.

Solution: False. Cluster sizes can be as large as $\Theta(\log n)$.

(g) T F The hash function $h(k) = k \mod m$ is guaranteed to have no collisions if and only if m is a prime number.

Solution: False. If keys are larger than m then collisions could well occur.

(h) T F Given a long text string T of length n and a pattern string P of length $\sqrt[3]{n}$, identifying if P occurs in T requires $\Theta(n^{\frac{4}{3}})$ time.

Solution: False (Rabin-Karp tells us that we can do it in $\Theta(n)$ time).

(i) T F To maintain a dynamically varying graph G — where the vertices are fixed, but edges can be inserted and deleted — the adjacency matrix representation enables more efficient updates than the adjacency list representation.

Solution: True. Deletion takes $O(1)$ time for adjacency matrices but can cost up to $\Theta(n)$ time for lists.

(j) T F One of the algorithms seen in the class solves the single source *longest* path problem for an arbitrary graph $G = (V, E)$, with nonnegative edge weights, in time $O(|E| + |V| \log |V|)$.

Solution: False. Optimal substructure does not hold.

(k) T F In using Newton's method for finding square roots, the number of correct digits in the solution *squares* in each iteration.

Solution: False.

(l) T F Newton's method for finding cube roots might not always converge to the correct answer.

Solution: True. You need to start at a "sufficiently close" initial point.

(m) T F It is asymptotically faster to *square* a d-digit integer than to *multiply* two distinct d-digit integers.

Solution: False, since $ab = ((a + b)^2 - a^2 + b^2)/2$.

(n) T F One can solve the SSSP problem on a graph with negative edge weights by simply adding the smallest edge weight to all other edge weights, and running Dijkstra's algorithm.

Solution: False - easy counterexamples.

(o) T F The minimum vertex cover problem can be solved in time polynomial in the size of the input, if the input graph is a tree.

Solution: True (we did it in class).

Problem 2. Algorithmic Techniques [30 points] (10 parts)

For each of the following problems, indicate which algorithm or algorithmic technique you would choose in order to solve it using the **fastest** asymptotic running time.

(a) As the TA for "6.666: Defense against the Dark Arts", you are asked to sort a stack of n final exams in decreasing order of their score. Each score is a positive integer between 1 and 180.

Circle one: **Heap Sort** or **Merge Sort** or **Radix Sort**

Solution: Radix Sort

(b) You want to dynamically maintain a collection of all the soft drinks that you have tasted so far. Each drink has a sweetness index. Given a new unseen drink, you want to quickly identify the drink in the collection that is closest in terms of the sweetness index.

Circle one: **Sorting** or **Hashing** or **BST**

Solution: BST

(c) Maintain a hash table under arbitrary inserts and deletes without table-resizing.

Circle one: **Chaining** or **Linear Probing** or **Double Hashing**

Solution: Chaining

(d) Given a long string of DNA represented using the letters $\{\text{A, T, C, G}\}$, count the number of occurrences of the fragment 'GATTACA' in the string.

Circle one: **Dynamic Programming** or **Rabin-Karp** or **Bellman-Ford**

Solution: Rabin-Karp

(e) Given a unweighted undirected graph G, detect whether there is a cycle in the graph.

Circle one: **BFS** or **DFS** or **Dijkstra** or **Bellman-Ford**

Solution: DFS

(f) Given a weighted directed graph G, detect whether there is a positive weight cycle in the graph.

Circle one: **BFS** or **DFS** or **Dijkstra** or **Bellman-Ford**

Solution: Bellman-Ford

(g) Solve the single-source shortest paths problem on a DAG with nonnegative integer edge weights from the range $1 \ldots W$, where W is constant.

Circle one: **BFS** or **Dijkstra** or **DFS**

Solution: BFS

(h) Calculate the Golden Ratio $\alpha = (1 + \sqrt{5})/2$ up to 1,000,000,000,000 digits.

Circle one: **Binary Search** or **Newton's method** or **Euclid's algorithm**

Solution: Newton's method

(i) You are given a map of a region with three (3) countries containing all their cities, roads, and road lengths. Compute the shortest paths between the **capital** city of each country to every other city in the region.

Circle one: **Dijkstra** or **Bellman-Ford** or **Floyd-Warshall**

Solution: Dijkstra

(j) You are the CEO of a startup with an initial budget to make new hires. There is an abundant number of applications from a variety of candidates to join the startup. Each candidate fits a profile that will provide a certain amount of value to the company, but also take up a certain amount of the budget.

You know the cost and value of each candidate profile. You want to find a set of new hires that will add the most (cumulative) value to the staff, but at the same time, do not wish to exceed the hiring budget.

Circle one: **Sorting** or **Shortest Paths** or **Dynamic Programming**

Solution: Dynamic Programming

Problem 3. Short Answer Problems [15 points] (3 parts)

(a) [5 points] Given a set P of n numbers, we say that two numbers $p, q \in P$ form a *closest pair* if $|p - q| = \min_{u,v \in P, u \neq v} |u - v|$. Give an $O(n \log n)$ time algorithm that finds a closest pair in a given set P.

Solution: We sort P, and then calculate the closeness of each adjacent pair with a linear scan, returning the closest pair.

(b) [5 points] Give an algorithm for HEIGHT(x) which returns the height of a binary tree rooted at x. Assume that $x.left$ and $x.right$ points to the left and right child of x respectively or is NULL if the respective child does not exist.

Solution: HEIGHT$(x) = \max($HEIGHT$(x.left),$ HEIGHT$(x.right)) + 1$, where HEIGHT$(NULL) = 0$.

(c) [5 points] Suppose we are trying to solve SSSP on a weighted graph G with n vertices and m edges. We know that Bellman-Ford takes $O(nm)$ time to run. We want to find out whether this running time is polynomial in the size of the input.
You are given:

1. the number of vertices n,
2. the number of edges m, and
3. an adjacency matrix with n^2 entries each containing a edge weight. Let the maximum edge weight be w.

Calculate the the input size in terms of $n, m,$ and w. Argue that the running time of Bellman-Ford is polynomial in the size of the input.

Solution: Storing the number of vertices requires $\lg V$ bits
Storing the number of edges requires $\lg E$ bits
Storing V^2 edge weights requires $V^2 \lg W$ bits
Thus we have total input size: $O(V^2 \lg W + \lg E + \lg V)$, but the first term dominates so we have total size $O(V^2 \lg W)$
We then accepted 2 different models of computation, one where addition of two b-bit numbers requires b time, and one where all arithmetic is $O(1)$ time.

For linear addition, the runtime of Bellman Ford is $O(VE \lg W)$, or $O(V^3 \lg W)$ when $E = \Theta(V^2)$. This runtime is clearly polynomial in the input size.

For constant addition, the runtime of Bellman Ford is $O(VE)$, or $O(V^3)$, which is also clearly polynomial in $V^2 \lg W$.

Problem 4. Cross Distance [20 points]

Consider a new distance metric for points on a plane called CROSS-DISTANCE, which is defined as the standard distance between two points if the points share at least one coordinate, and otherwise infinity. Formally, we compute the CROSS-DISTANCE between two points (x_1, y_1) and (x_2, y_2) as follows:

1. If $x_1 = x_2$, return $|y_1 - y_2|$.
2. If $y_1 = y_2$, return $|x_1 - x_2|$.
3. Otherwise, return ∞.

Design a data structure to support the following operations on points in the plane. All operations should run in $O(\log n)$ time, where n is the number of elements in the structure directly before the operation is performed.

- INSERT(x, y): If (x, y) is not already in the structure, insert it. Otherwise, do nothing.
- DELETE(x, y): If (x, y) is in the structure, remove it. Otherwise, do nothing.
- NEAREST(x, y): Return the distance of the point closest to (x, y) in the structure, measured according to CROSS-DISTANCE. Note that (x, y) may or may not already be in the structure.

Solution: Maintain an AVL tree T_x keyed by x, where each node of the AVL tree is another AVL tree keyed by y (consisting of all elements with that x coordinate). Maintain another AVL tree T_y keyed by y, where each node is another AVL tree keyed by x.

For INSERT(x, y), just search for x in T_x. If it already exists, insert y into the AVL tree at that node. Otherwise, create a new node which is an AVL tree with the single element y. Symmetrically insert the point in T_y. This takes $O(\log n)$ overall because it is at most 4 AVL insertions. The implementation for DELETE is basically the same.

To perform QUERY(x, y), we first check if (x, y) exists in the structure. If so, return that point. Otherwise, we need to find the nearest point on each side of the cross, and return the closest one. To find the nearest point directly above (x, y), search T_x for x. If that node exists, the successor of y in that AVL tree corresponds to the closest point. The closest point below can be found symmetrically using the predecessor of y. The closest points to the left and right can be found using the same method on T_y. Then just return the distance of the closest of the four points.

Problem 5. The Walking Undead [15 points]

A curious disease is afflicting the land. Symptoms include walking around with stiffly outstretched arms and glazed expressions, and muttering incomprehensible proofs of the Riemann Hypothesis.

The government has determined that the disease is caused by *outbreaks*. Each outbreak is an event in which a virus suddenly originates at some location with coordinates (x, y) (here x and y can be arbitrary real numbers). Once an outbreak happens, everyone within a radius of 5 units is affected. To monitor the disease, the goverment has installed *beacons* at **all** locations with **integer** coordinates in the 2D plane. A beacon is activated everytime an outbreak happens within a distance of 5 units from its location.

Suppose that there are m outbreaks at locations $(x_i, y_i),\ i = 1, \ldots m$. Develop an algorithm to quickly identify the beacon that is activated the **maximum** number of times. Your algorithm should run in $O(m)$ time.

Solution: Create a hash table of size $100m$ that will contain all the beacons that have been activated at least one time. Each time an outbreak occurs at a point (x_i, y_i), we compute all the integer beacon locations that are within a distance of 5 from the outbreak point. We can do this by iterating through all points (x, y) for $x = \lfloor x_i \rfloor - 5, \lfloor x_i \rfloor - 4, ..., \lceil x_i \rceil + 5$ and for $y = \lfloor y_i \rfloor - 5, \lfloor y_i \rfloor - 4, ..., \lceil y_i \rceil + 5$ and then checking if they are within distance 5 from (x_i, y_i). For each such outbreak point, we increment the counter of all the close beacon locations by one (if the entry doesn't exist, we create it and we initialize it to 1). There are at most $100m$ operations that we need to perform in the hash table and under the SUHA, each of them takes $O(1)$ in expectation since the load factor is at most 1. Therefore, the total running time for insertion is $O(m)$.

Finally, we go through the hash table once more to find the beacon location with the largest counter. This takes an additional $O(m)$ time and thus the complexity of the algorithm is $O(m)$ in both runtime and memory.

Problem 6. Speeding up Dijkstra [15 points]

A d-ary min heap is similar to a binary min heap, except that each node now has d children instead of just two children. Consider a graph $G = (V, E)$ with nonnegative edge weights, where $|E| \geq 2|V|$. You are asked to solve the single-source shortest path problem on G, and decide to implement DIJKSTRA using a d-ary min-heap instead of a standard binary min-heap.

(a) [10 points] Write down the running time of DIJKSTRA on G using a d-ary min-heap.

Solution: It takes $O(\log_d(|V|)$ to perform DECREASE-KEY, because the tree has height $O(\log_d |V|)$. (You only ever decrease the value of a key, so it only needs to be swapped with its parent pointer.) It takes $O(d \log_d |V|)$ to perform EXTRACT-MIN, because when you call MAX-HEAPIFY, you have to repeatedly swap the element with the minimum of its children. This leads to an overall runtime of

$$O(|E| \cdot \log_d |V| + |V| \cdot d \log_d |V|).$$

(b) [5 points] What value of d should you choose in order to minimize the runtime?

Solution: To minimize the runtime we set the two terms equal to equal other, so we have

$$|E| \cdot \log_d |V| = |V| \cdot d \log_d V,$$

which simplifies down to $|E| = |V| \cdot d$, so $d = |E|/|V|$.

Problem 7. The Pursuit of Happiness [15 points]

Squeaky the Squirrel comes across a large walnut tree containing n branch points (nodes). Each node v in the tree contains $h(v)$ nuts. Every walnut increases his Happiness by 1 unit. However, each node also contains $g(v)$ thorns that Squeaky has to sweep off the tree with his feet. Every such thorn decreases his Happiness by 1 unit. Happiness is additive; if Squeaky chooses to visit node v, then his net Happiness gain would be $h(v) - g(v)$.

You are given the tree $T = (V, E)$ and the functions $h(v), g(v)$. You want to calculate the maximum net Happiness gained by Squeaky when he begins at the root. You correctly deduce that this problem can be solved using structural DP. To solve it, you initialize a table $DP[v]$ that contains the Happiness of the optimal subtree rooted at node v.

(a) Write down a recurrence for $DP[v]$. Ensure that you also cover the base cases.

Solution:
If we define $DP[v]$ so that you must include node v, we have:

$$DP[v] = h(v) - g(v) + \sum_{u \in v.children} \max(DP[u], 0).$$

If v has no children, it follows that $DP[v] = h(v) - g(v)$. In this case, the maximum net Happiness is just $\max(DP[root], 0)$.

Alternatively, if $DP[v]$ does not imply we include node v, we have:

$$DP[v] = \max(0, (h(v) - g(v) + \sum_{u \in v.children} DP[u]))$$

If v has no children, it follows that $DP[v] = \max(0, h(v) - g(v))$. The maximum net Happiness in this case is simply $DP[root]$.

(b) Briefly analyze the recurrence, and establish a running time of $O(n)$.

Solution: There are $O(n)$ subproblems (one for each node), and for each node v, the subproblem takes $O(degree(v))$. $\sum_{v \in V} O(degree(v)) = O(n)$, so the total runtime is $O(n)$.

Common mistakes: A large number of students set up the DP as:

$$DP[v] = h(v) - g(v) + \max_{u \in v.children} DP[u].$$

However, this only finds the optimal *path* from root to leaf (and not the optimal subtree). We ended up giving partial credit for this since it is not entirely clear from the problem statement if Squeaky is allowed to traverse any given node multiple times.

Problem 8. Finding the Largest Black Square [20 points]

A black-and-white *image* can be represented as a square matrix of size n by n, where each element is either 1 (representing a black pixel) or 0 (representing a white pixel). Design an algorithm that takes such an image as input and returns the size of the largest black square, that is, the largest **square** sub-matrix that contains all 1's.

For example, given the following matrix, your algorithm should return 16.

0	1	1	0	1	0
0	1	1	1	1	0
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
0	1	1	0	1	0

Describe your algorithm, and provide a running time analysis. (You do not need to provide a correctness proof.) For full points, your algorithm should run in $O(n^2)$ time. A slower but correct algorithm will receive partial credit.

Solution: Let c_{ij} denote the pixel value at square (i, j) (row i, column j).

Subproblem Definition: Let $DP[i][j]$ denote the side length of the largest all-one square matrix in the upper left corner (rows 1 to i, columns 1 to j) that includes the square (i, j). There are n^2 subproblems total.

Recurrence: $DP[i][0] = c_{i0}$, $DP[0][j] = c_{0j}$. When $i, j \geq 1$, $DP[i][j] = min(DP[i-1][j], DP[i-1][j-1], DP[i][j-1]) + 1$ if $c_{ij} = 1$, $DP[i][j] = 0$ if $c_{ij} = 0$. Time per subproblem: $O(1)$.

Order to solve subproblems: Start from square $(0, 0)$, following the increasing order of $i + j$.

Getting the final result: Find the maximum $DP[i][j]$, square it.

Total time: $O(n^2)$.

Problem 9. 1D Dominos [20 points]

A domino can be represented by two integers, (l, r), representing the number of dots on the left and right half of the domino. A *chain* is a sequence of dominos $[(l_1, r_1), (l_2, r_2), \ldots, (l_k, r_k)]$ where $r_i = l_{i+1}$ for $i = 1, 2, \ldots, k-1$ and k is the length of the chain.

(a) [10 points] Consider the case where you cannot flip the dominos, so dominos $(3, 1)$ and $(1, 3)$ are different. Define the *weight* of a chain of dominos to be $\sum_{i \in \text{chain}}(l_i + r_i)$. So, for example, the chain $[(1, 4), (4, 2), (2, 7), (7, 3)]$ has weight $(1 + 4) + (4 + 2) + (2 + 7) + (7 + 3) = 30$.

Given a set of n dominos, a start domino d_s, and an end domino d_t, design an algorithm to find the weight of the **lowest-weight chain** of dominos beginning with d_s and ending with d_t and analyze its runtime. Assume d_s and d_t exist in the set of dominos. If no chain exists, your algorithm should return ∞.

(b) [10 points] Now consider the case where we are allowed to flip dominos, so $(3, 1)$ is indistinguishable from $(1, 3)$. Given an ordered list of n dominos, design an algorithm that returns the length of the **longest subsequence** of dominos that form a chain and analyze its runtime. Because we are looking for a subsequence, if the i^{th} input domino comes before the j^{th} input domino in the chain, we need $i < j$. For example, if our input list of dominos is $[(1, 2), (4, 2), (5, 1), (4, 3)]$, the longest chain is $[(1, 2), (2, 4), (4, 3)]$.

SCRATCH PAPER