Final Exam Solutions

Problem 1. Asymptotics [10 points]

For each pair of functions $f(n)$ and $g(n)$ in the table below, write O, Ω, or Θ in the appropriate space, depending on whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, or $f(n) = \Theta(g(n))$. If there is more than one relation between $f(n)$ and $g(n)$, write only the strongest one. The first line is a demo solution.

We use $\lg$ to denote the base-2 logarithm.

Solution:

	n	$n \lg n$	n^2
$n \lg^2 n$	Ω	Ω	O
$2^{\lg^2 n}$	Ω	Ω	Ω
$\lg(n!)$	Ω	Θ	O
$n^{\lg 3}$	Ω	Ω	O

Problem 2. True or False [40 points] (10 parts)

Decide whether these statements are **True** or **False**. You must briefly justify all your answers to receive full credit.

(a) An algorithm whose running time satisfies the recurrence $P(n) = 1024\,P(n/2) + O(n^{100})$ is asymptotically faster than an algorithm whose running time satisfies the recurrence $E(n) = 2\,E(n - 1024) + O(1)$.

True **False**

Explain:

Solution: **True.** The first recurrence leads to a result that is polynomial in n, while the second recurrence produces a result that is exponential in n.

(b) An algorithm whose running time satisfies the recurrence $A(n) = 4\,A(n/2) + O(1)$ is asymptotically faster than an algorithm whose running time satisfies the recurrence $B(n) = 2\,B(n/4) + O(1)$.

True **False**

Explain:

Solution: **False.** Considering the recursion trees for $A(n)$ and $B(n)$, it is easy to see that the tree for A has both a smaller height ($\log_4(n)$ vs. $\log_2(n)$), and a smaller branching factor.

(c) Radix sort works in linear time **only if** the elements to sort are integers in the range $\{0, 1, \ldots, c\,n\}$ for some $c = O(1)$.

True **False**

Explain:

Solution: **False.** Radix sort also works in linear time if the elements to sort are integers in the range $\{1, \ldots, n^d\}$ for any constant d.

(d) Given an undirected graph, it can be tested to determine whether or not it is a tree in $O(V + E)$ time. A *tree* is a connected graph without any cycles.

True **False**

Explain:

Solution: **True.** Using either DFS or BFS yields a running time of $O(V + E)$.

(e) The Bellman-Ford algorithm applies to instances of the single-source shortest path problem which do not have a negative-weight directed cycle, but it does not detect the existence of a negative-weight directed cycle if there is one.

True **False**

Explain:

Solution: **False.** Bellman-Ford detects negative-weight directed cycles in its input graph.

(f) The topological sort of an arbitrary directed acyclic graph $G = (V, E)$ can be computed in linear time.

True **False**

Explain:

Solution: **True.** A topological sort can be obtained by listing the nodes in the reverse order of the exit times produced by a DFS traversal of the graph.

The DFS can also be used to detect if there is a cycle in the graph (there is no valid topological sort in that case). The running time of DFS is $O(V + E)$.

(g) We know of an algorithm to detect negative-weight cycles in an arbitrary directed graph in $O(V + E)$ time.

True **False**

Explain:

Solution: **False.**

The best solution presented in this class is the Bellman-Ford algorithm, and its running time is $O(VE)$.

(h) We know of an algorithm for the single source shortest path problem on an arbitrary graph with no negative-weights that works in $O(V + E)$ time.

True **False**

Explain:

Solution: **False.** The best solution presented in this class is Dijkstra with Fibonacci heaps, and its running time is $O(V \log V + E)$.

(i) To delete the i^{th} node in a min heap, you can exchange the last node with the i^{th} node, then do the min-heapify on the i^{th} node, and then shrink the heap size to be one less the original size.

True **False**

Explain:

Solution: **False.** The last node may be smaller than the i^{th} node's parent; min-heapify won't fix that.

(j) Generalizing Karatsuba's divide and conquer algorithm, by breaking each multiplicand into 3 parts and doing 5 multiplications improves the asymptotic running time.

True **False**

Explain:

Solution: **False.** Karatsuba's running time is $T(n) = 3T(n/2)+O(n) = O(n^{\log_2 3})$. The generalized algorithm's running time would be $T(n) = 5T(n/3) + O(n) = O(n^{\log_3 5})$.

Problem 3. Set Union [10 points]

Give an efficient algorithm to compute the union $A \cup B$ of two sets A and B of total size $|A|+|B| = n$. Assume that sets are represented by arrays (Python lists) that store distinct elements in an arbitrary order. In computing the union, the algorithm must remove any duplicate elements that appear in both A and B.

For full credit, your algorithm should run in $O(n)$ time. For partial credit, give an $O(n \lg n)$-time algorithm.

Solution: **Algorithm.** Let $\mathcal{H}$ be an initially empty hash table (Python dictionary), and R be an initially empty growable array (Python list). For each element e in A and B, do the following. If e is in $\mathcal{H}$, skip over e. Otherwise, append e to R and insert e into $\mathcal{H}$.

Correctness. Each element from A and B is considered. An element is added to $\mathcal{H}$ only when it is added to R, so the elements that are skipped must be duplicates.

Running Time. There are n total elements. In the worst case, each element is looked up in $\mathcal{H}$ once, then inserted into R and $\mathcal{H}$. All operations are constant time per element, so the total running time is $O(n)$.

Problem 4. Balanced Trees [10 points]

In the definition of an AVL tree we required that the height of each left subtree be within one of the height of the corresponding right subtree. This guaranteed that the worst-case search time was $O(\log n)$, where n is the number of nodes in the tree. Which of the following requirements would also provide the same guarantee?

(a) The number of nodes in each left subtree is within a factor of 2 of the number of nodes in the corresponding right subtree. Also, a node is allowed to have only one child if that child has no children.

This tree has worst case height $O(\lg n)$.

True **False**

Explain:

Solution: **True.** The proof is very similar to the AVL tree proof.

Let $N(h)$ be the minimum number of nodes contained in a tree with height h. The base cases are $N(0) = 0$, $N(1) = 1$, and $N(2) = 2$. Now we have the following recurrence for N:

$$N(h) = 1 + N(h-1) + \frac{1}{2}N(h-1) = 1 + \frac{3}{2}N(h-1)$$

Because a tree with height h must have one subtree of height $h-1$, and the other subtree has at least half the number of nodes in that subtree.

The solution to this recurrence is $N(h) = \Theta((\frac{3}{2})^h)$, which gives $h = \Theta(\lg N)$, as desired.

(b) The number of leaves (nodes with no children) in each left subtree is within one of the number of leaves in the corresponding right subtree.

This tree has worst case height $O(\lg n)$.

True **False**

Explain:

Solution: **False.** Consider a tree of n nodes, where node 1 is the root, and node $i > 1$ is the child of node $i - 1$. For each node, the left subtree has one leaf, whereas the right subtree has zero. This meets the "balancing" condition. The height of the tree is n.

Problem 5. Height Balanced Trees [10 points]

We define the height of a node in a binary tree as the number of nodes in the longest path from the node to a descendant leaf. Thus the height of a node with no children is 1, and the height of any other node is 1 plus the larger of the heights of its left and right children.

We define height balanced trees as follows;

- each node has a "height" field containing its height,
- at any node, the height of its right child differs by at most one from the height of its left child.

Finally we define $\texttt{Fib}(i)$ as follows,

$$\texttt{Fib}(0) = 1$$

$$\texttt{Fib}(1) = 1$$

$$\texttt{Fib}(i) = \texttt{Fib}(i-1) + \texttt{Fib}(i-2), \text{ for } i \geq 2.$$

You may use without proof that $\texttt{Fib}(n) \geq 1.6^n$ for large n.

Prove that there are at least $\texttt{Fib}(h)$ nodes in a height balanced tree of height h, for all $h \geq 1$.

Solution: Let $T(h)$ be the minimum number of nodes in a height balanced tree of height h. We proceed by induction. For the base cases note that $T(1) \geq T(0) \geq 1$, thus $T(1) \geq \texttt{Fib}(1)$ and $T(0) \geq \texttt{Fib}(0)$. Now assume that $T(h') \geq \texttt{Fib}(h')$ for all $h' < h$ Clearly, $T(h) \geq T(h-1) + T(h-2)$, hence $T(h) \geq \texttt{Fib}(h-1) + \texttt{Fib}(h-2) = \texttt{Fib}(h)$.

Problem 6. Maintaining Medians [15 points]

Your latest and foolproof (really this time) gambling strategy is to bet on the median option among your choices. That is, if you have n distinct choices whose sorted order is $c[1] < c[2] < \cdots < c[n]$, then you bet on choice $c[\lfloor (n+1)/2 \rfloor]$. As the day goes by, new choices appear and old choices disappear; each time, you sort your current choices and bet on the median. Quickly you grow tired of sorting. You decide to build a data structure that keeps track of the median as your choices come and go. Specifically, your data structure stores the number n of choices, the current median m, and two AVL trees S and T, where S stores all choices less than m and T stores all choices greater than m.

(a) Explain how to add a new choice c_{new} to the data structure, and restore the invariants that (1) m is the median of all current choices; (2) S stores all choices less than m; and (3) T stores all choices greater than m. Analyze the running time of your algorithm.

Solution: Store the sizes $|S|$ and $|T|$, and update them whenever an element is added or removed from each tree. We will maintain the invariant: $\big||S| - |T|\big| <= 1$.

If $c_{\text{new}} < m$, and $|S| <= |T|$, then insert c_{new} into S, and do not change m. If $c_{\text{new}} < m$ and $|S| = |T| + 1$, then insert c_{new} into S and insert m into T. The largest element of S is the new median, so remove the largest element from S and assign that value to m. (The case where $c_{\text{new}} > m$ is symmetric.)

Each AVL insertion and deletion requires $O(\lg n)$ time, for $O(\lg n)$ total.

(b) Explain how to remove an existing choice c_{old} from the data structure, and restore invariants (1–3) above. Analyze the running time of your algorithm.

Solution: If $c_{\text{old}} < m$, delete c_{old} from S, decrementing $|S|$. If $\big||S| - |T|\big| <= 1$, do nothing. If $|S| = |T| - 2$, we need to modify the median. Insert m into S, then extract minimum element of T and store it in m.

Each AVL insertion and deletion requires $O(\lg n)$ time, for $O(\lg n)$ total.

Problem 7. Hashing [10 points]

Suppose that we have a hash table with $2n$ slots, with collisions resolved by chaining, and suppose that n keys are inserted into the table. Assume simple uniform hashing, i.e., each key is equally likely to be hashed into each slot.

(a) What is the expected number of elements that hash into slot i?

Solution: Let X_{ij} be an indicator variable whose value would be 1 if element j hashes into slot i, or 0 otherwise.

$$E[X_{ij}] = 0 \cdot Pr[j \text{ does not hash into slot } i] + 1 \cdot Pr[j \text{ hashes into slot } i]$$

The probability that an element will hash into slot i is $\frac{1}{2n}$, assuming simple uniform hashing, so $E[X_{ij}] = \frac{1}{2n}$.
Let N_i be the number of elements that hash into slot i. $N_i = \sum_{j=1}^{n} X_{ij}$, so, by linearity of expectation, $E[N_i] = \sum_{j=1}^{n} E[X_{ij}] = \sum_{j=1}^{n} \frac{1}{2n} = \frac{1}{2}$

(b) What is the probability that exactly k keys hash into slot i?

Solution: It is the probability that some chosen k keys hash into slot i *and* the other $n - k$ keys hash into any other slot.

$$\binom{n}{k} \left(\frac{1}{2n}\right)^k \left(\frac{2n-1}{2n}\right)^{n-k}$$

Problem 8. d**-max-heap** [10 points]

A ***d-max-heap*** is like an ordinary binary max-heap, except that nodes have d children instead of 2.

(a) Describe how a d-max-heap can be represented in an array $A[1 \ldots n]$. In particular, for the internal (non-leaf) node of the d-max-heap stored in any location $A[i]$, which positions in A hold its child nodes?

Solution: The representation would use the same idea for storing binary heaps, which involves storing the nodes as they are given by a level- order traversal. For example, the root will be node 1, and its children will be nodes 2, 3, 4, and 5.

For a non-leaf node i, its children would be $d \cdot i - d + 2$, $d \cdot i - d + 1 \ldots d \cdot i$, $d \cdot i + 1$.

(b) Define the height of the heap to be the number of nodes on the longest path from the root to a leaf.

In terms of n and d, what is the height of a d-max-heap of n elements?

Solution: $\lceil \log_d n \rceil$

Problem 9. Firehose Optimization [10 points]

You have decided to apply your algorithmic know-how to the practical problem of getting a degree at MIT. You have just snarfed the course catalog from WebSIS. Assume there no cycles in course prerequisites. You produce a directed graph $G = (V, E)$ with two types of vertices $V = C \cup D$: regular *class vertices* $c \in C$ and special *degree vertices* $d \in D$. The graph has a directed edge $e = (u, v)$ whenever a class $u \in C$ is a prerequisite for $v \in V$ (either a class or a degree). For each class $c \in C$, you've computed your *desire* $w(c) \in \mathbb{R}$ for taking the class, based on interest, difficulty, etc. (Desires can be negative.)

Give an $O(V + E)$-time algorithm to find the most desirable degree, that is, to find a degree $d \in D$ that maximizes the sum of the desires of the classes you must take in order to complete the degree: $\sum\{w(c) : \text{path } c \rightsquigarrow d\}$. (For partial credit, give a slower algorithm.)

Solution: Topological sort. Then compute the aggregate desire for each node in that order. Then take the max node in S.

Problem 10. Histogram Hysterics [15 points]

Sometime in the future, you become a TA for 6.006. You have been assigned the job of maintaining the grade spreadsheet for the class. By the end of the semester, you have a list g of final grades for the n students, sorted by grade: $g[0] < g[1] < \cdots < g[n-1]$. In an attempt to draw various beautiful histograms of these grades, the (rather odd) professors now ask you a barrage of questions of the form "what is the sum of grades $g[i:j]$, i.e., $g[i] + g[i+1] + \cdots + g[j-1]$?" for various pairs (i, j). (Dividing this sum by $j - i$ then gives an average.)

To save you work computing summations, you decide to compute some of the sums $g[i:j]$ ahead of time and store them in a data structure. Unfortunately, your memory is large enough to store only $\Theta(n)$ such sums. Once these sums have been computed, can you answer each query by the professors in $O(1)$ time? If not, give the fastest solution you can.

(a) Which sums $g[i:j]$ should you compute ahead of time?

Solution: For all k, store $g[0:k]$.

(b) In what data structure should you store these sums?

Solution: They can be stored in an array, a python list, or a dictionary.

(c) How do you then compute a professors' query for an arbitrary sum $g[i:j]$, and how long does this take?

Solution:

$$g[i:j] = g[0:j] - g[0:i]$$

This only involves two lookups and a subtraction, for $O(1)$ time.

Problem 11. Wonderland [20 points]

You have just taken a job at Wonderland (at the end of the Blue Line) as an amusement-ride operator. Passengers can enter the ride provided it is not currently running. Whenever you decide, you can run the ride for a fixed duration d (during which no passengers can enter the ride). This action brings joy to the passengers, causing them to exit the ride and pay you d/t_i dollars where t_i is the amount of time passenger i spent between arriving and exiting the ride. Thus, if you start the ride as soon as a passenger arrives, then $t_i = d$, so you get \$1.00 from that passenger. But if you wait d units of time to accumulate more passengers before starting the ride, then $t_i = 2d$, so you only get \$0.50 from that passenger.

Every day feels the same, so you can predict the arrival times $a_0, a_1, \ldots, a_{n-1}$ of the n passengers that you will see. As passenger i arrives, you must decide whether to start the ride (if it is not already running). If you start the ride at time a_j, then you receive $d/(d + a_j - a_i)$ dollars from customers $i \leq j$ that have not yet ridden, and you can next start the ride at times $a_k \geq a_j + d$. Your goal is to maximize the total amount of money you make using dynamic programming.

(a) Clearly state the set of subproblems that you will use to solve this problem.

Solution: Subproblems are finding the maximum amount of money that can be gained from passengers $0 \ldots i$, such that the ride starts after passenger i arrives. Note that i ranges between 0 and $n - 1$.

(b) Write a recurrence relating the solution of a general subproblem to solutions of smaller subproblems.

Solution: We use $\mathcal{DP}[i]$ to store the answer to the subproblems described above.

$$\mathcal{DP}[i] = 1 + \max_{j<i, a_i - a_j \geq d} \left(\mathcal{DP}[j] + \sum_{k=j+1}^{i} \frac{d}{d + a_i - a_k} \right)$$

In words, when we are solving subproblem i, we know that we will be starting the ride after passenger i arrives. Then we state that j is the last time the ride was started before, and we use the results to the previously computed subproblems to find out what value of j yields the most money.

(c) Analyze the running time of your algorithm, including the number of subproblems and the time spent per subproblem.

Solution: There are n subproblems. Each subproblem takes $O(n)$ time to solve, because it is taking into consideration the solutions for $O(n)$ previously solved subproblems, and evaluating all the $O(n)$ sums can be done in $O(n)$ amortized time.
Therefore, the total running time is $O(n^2)$.

(d) Write the solution to the original problem in terms of solutions to your subproblems.

Solution: The answer is $\mathcal{A} = \mathcal{DP}[n-1]$. Since there are no passengers coming after the $(n-1)^{\text{th}}$ passenger, it will always make sense to start the ride after that last passenger arrives.
The complete solution consists of the passengers whose arrival should trigger the starting of the ride. It can be easily reconsituted by using parent pointers which store the argmax for each max computed.

Problem 12. Dance Dance Evolution [20 points]

You are training for the upcoming Dance Dance Evolution competition and decide to apply your skills in dynamic programming to find optimal strategies for playing each song.

A simplified version of the problem can be modeled as follows. The song specifies a sequence of "moves" that you must make. Each *move* is one of four buttons in the set $B = \{\boxed{\uparrow}, \boxed{\downarrow}, \boxed{\leftarrow}, \boxed{\rightarrow}\}$ that you must press with one of your feet. An example of a song is

$$\boxed{\leftarrow}, \boxed{\rightarrow}, \boxed{\uparrow}, \boxed{\downarrow}, \boxed{\rightarrow}, \boxed{\leftarrow}, \boxed{\downarrow}, \boxed{\uparrow}.$$

You have two feet. At any time, each foot is on one of the four buttons; thus, the current state of your feet can be specified by an ordered pair (L, R) where $L \in B$ denotes the button beneath your left foot and $R \in B$ denotes the button beneath your right foot.

One foot at a time: When you reach a move $M \in B$ in the song, you must put one of your feet on that button, transitioning to state (M, R) or (L, M). Note that you can change only one of your feet per move. If you already have a foot on the correct button, then you do not need to change any feet (though you are allowed to change your other foot).

Forbidden states: You are also given a list of forbidden states $\mathcal{F}$, which you are never allowed to be in. $\mathcal{F}$ might include states where both feet are on the same square, or states where you would end up facing backwards.

Your goal is to develop a polynomial-time algorithm that, given an n-move song $M_1, M_2, \ldots, M_n$, finds an initial state (L_0, R_0) and a valid sequence of transitions $(L_i, R_i) \to (L_{i+1}, R_{i+1}) \notin \mathcal{F}$, for $0 \le i < n$, where $M_{i+1} \in \{L_{i+1}, R_{i+1}\}$ and either $L_i = L_{i+1}$ or $R_i = R_{i+1}$,

(a) Clearly state the set of subproblems that you will use to solve this problem.

Solution: A subproblem is the question if it is possible to "clear" states $1 \ldots i$ and end up with the feet in the configuration (j, k), where $j, k, \in B$. To simplify the mathematical recurrence below, an answer of yes is 1, and an answer of no is 0.

(b) Write a recurrence relating the solution of a general subproblem to solutions of smaller subproblems.

Solution: We use $\mathcal{DP}[i][j][k]$ to store the answer to the subproblems described above.

$$\mathcal{DP}[i][j][k] = \begin{cases} 1 \text{ if} & i = 0 \wedge (j,k) \notin \mathcal{F} \\ 0 \text{ if} & (j,k) \in \mathcal{F} \vee M_i \notin \{j,k\} \\ \max\limits_{u,v \in B, u=j \vee v=k} \mathcal{DP}[i-1][u][v] \text{ otherwise} & \end{cases}$$

In words, we know that we can start out from any feet configuration not in $\mathcal{F}$. No sequence of moves can contain a configuration in $\mathcal{F}$, and a sequence claiming to "clear" moves $1 \ldots i$ must end in either (i, R) or (L, i).

Once these formalities are out of the way, we try to "clear" moves $1 \ldots i$ by using our answers for clearing $1 \ldots i-1$, and then moving at most one foot. Note that $\max$ functions as a logical $\vee$ (or), given our representation of boolean values.

(c) Analyze the running time of your algorithm, including the number of subproblems and the time spent per subproblem.

Solution: Let $b = |B|$ (we know that $b = 4$, but we'll keep it a variable for the analysis' sake). Then we have $n \times b \times b = nb^2$ subproblems. In the worst case, computing the answert to a subproblem requires taking the maximum over $O(b)$ previously solved subproblems ($b - 1$ moves for the left foot, $b - 1$ moves for the right foot, and 1 case where no move is required).

The total running time is $O(nb^3)$. Given that $b = O(1)$, the expression for the running time can be simplified to $O(n)$. This is optimal, given that our input size is $O(n)$.

(d) Write the solution to the original problem in terms of solutions to your subproblems.

Solution: The answer is $\mathcal{A} = max_{j,k \in B} \mathcal{DP}[n][j][k]$. The sequence of moves can be restored by using parent pointers that remember the argmax of every $\max$ taken.