Final Exam Solutions

Problem 1. True or False [24 points] (8 parts)

For each of the following questions, circle either T (True) or F (False). **Explain your choice.** (No credit if no explanation given.)

(a) T F There exists an algorithm to build a binary search tree from an unsorted list in $O(n)$ time.
Explain:

Solution: False. Because an in-order walk can create a sorted list from a BST in $O(n)$ time, the existence of such an algorithm would violate the $\Omega(n \lg n)$ lower bound on comparison-based sorts.

(b) T F There exists an algorithm to build a binary heap from an unsorted list in $O(n)$ time.
Explain:

Solution: True. The standard BUILD-HEAP algorithm runs in $O(n)$ time.

(c) T F To solve the SSSP problem for a graph with no negative-weight edges, it is necessary that some edge be relaxed at least twice.
Explain:

Solution: False. Dijkstra's algorithm solves the SSSP problem relaxing each edge at most once.

(d) T F On a connected, directed graph with only positive edge weights, Bellman-Ford runs asymptotically as fast as Dijkstra.
Explain:

Solution: False. Bellman-Ford requires $\Theta(VE)$, regardless of the edge weights. Dijkstra runs in $\Theta(E + V \lg V)$. Because the graph is connected, $E = \Omega(V)$, so $\Theta(VE) = \Omega(V^2)$, which is clearly worse than Dijkstra.

(e) T F A Givens rotation requires $O(1)$ time.
Explain:

Solution: False. It may take $O(1)$ time if the matrix is sparse, but in general a Givens rotation requires $O(n)$ for a matrix with n columns.

(f) T F In the worst case, merge sort runs in $O(n^2)$ time.
Explain:

Solution: True. Merge sort runs in $O(n \lg n)$ time which is $O(n^2)$.

(g) T F There exists a stable implementation of merge sort.
Explain:

Solution: True. If the items in the two lists being merged are equal, choose the item in the "left" half (the half that came first in the original array).

(h) T F An AVL tree T contains n integers, all distinct. For a given integer k, there exists a $\Theta(\lg n)$ algorithm to find the element x in T such that $|k - x|$ is minimized.
Explain:

Solution: True. INSERT k, then find the PREDECESSOR and SUCCESSOR of k. Return the one whose difference with k is smaller. All three methods take $\Theta(\lg n)$ time.

Problem 2. Short Answer [32 points] (4 parts)

(a) The *eccentricity* $\epsilon(u)$ of a vertex u in a connected, undirected, *unweighted* graph G is the maximum distance from u to any other vertex in the graph. That is, if $\delta(u, v)$ is the shortest path from u to v, then $\epsilon(u) = \max_{v \in V} \delta(u, v)$.

Give an efficient algorithm to find the eccentricity of a given vertex s. Analyze its running time.

Solution: Run BFS starting at s. Keep track of how many "levels" have been explored. The number of levels required to explore the entire graph is equal to $\epsilon(s)$. This requires $O(V + E)$ time.

(b) What is the asymptotic cost of solving a linear system of equations with $n-1$ equations of the form

$$a_{i,i}x_i + a_{i,i+1}x_{i+1} = b_i \quad i = 1, \ldots, n-1$$

and one equation of the form

$$a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n = b_n$$

(none of the a's in this equation are zero). The a's and b's are given numbers and the x's are unknowns. Assume that we use Givens rotations to reduce the coefficient matrix to a triangular form. Justify your answer.

Solution: If we form a matrix and apply Givens rotations, eliminating the first column will fill in all the elements below the main diagonal, so we will need to perform $\Theta(n^2)$ rotations at a total cost of $\Theta(n^3)$. If we move the last equation (the one without zero coefficients) to the top, we only need to perform one Givens rotation per column, so the total cost reduces to $\Theta(n^2)$.

(c) Suppose a hash function h maps arbitrary keys to values between 0 and $m-1$, inclusive. We hash n keys, $k_1, k_2, \ldots, k_n$. If $h(k_i) = h(k_j)$, we say that k_i and k_j collide. Assuming simple uniform hashing, how should we choose m (in terms of n) such that the expected number of collisions is $O(1)$? Justify your answer.

Solution: Any two keys collide with probability $1/m$. There are $\binom{n}{2} = O(n^2)$ pairs of elements, so the expected number of collisions is $O(n^2/m)$. Thus, let $m = \Omega(n^2)$.

(d) Suppose you have a directed acyclic graph with n vertices and $O(n)$ edges, all having nonnegative weights. Propose an efficient method of finding the shortest path to each vertex from a single source, and give its running time in terms of n.

Solution: The given conditions allow us to use either the DAG shortest paths algorithm or Dijkstra's. DAG shortest paths runs in $\Theta(V + E) = \Theta(n)$, and Dijkstra's runs in $O((V + E) \lg V)$, or $O(V \lg V + E)$ if using a Fibonacci heap. In either case, Dijkstra's runs in $O(n \lg n)$. DAG shortest paths is faster in this case.

Problem 3. Judge Jill [24 points] (3 parts)

Judge Jill has created a web site that allows people to file complaints about one another. Each complaint contains exactly two names: that of the person who filed it and that of the person he/she is complaining about.

Jill had hoped to resolve each complaint personally, but the site has received so many complaints that she has realized she wants an automated approach.

She decides to try to label each person as either good or evil. She only needs the labeling to be consistent, not necessarily correct. A labeling is consistent if every complaint labels one person as good and the other person as evil, and no person gets labeled both as good and evil in different complaints.

(a) [8 points] Propose a way to model the consistent labeling problem as a graph problem.

Solution: We can model the problem using an undirected graph in which every edge represents one complaint and every vertex represents one name (one person).

(b) [10 points] Propose an efficient algorithm to consistently label all the names as good or evil, or to decide that no such classification exists. Use the graph model you proposed in the previous part of the problem. Analyze the running time of the algorithm.

Solution: The problem is quivalent to testing whether we can color the vertices of graph with two colors, or equivalently, whether the graph is bipartite. This can be done using DFS in $Theta(V+E)$ time. We label the root of DFS traversals arbitrarily good or bad. When we first visit a vertex, we label it differently than its parent. When we encounter back or cross edges, we check for consistency; if we find an inconsistency, the graph cannot be labeled consistently.

(c) [6 points] Later, Judge Jill wants to be more thorough. She will interview some people to figure out who is good and who is evil. She can always determine whether a person is good or evil by interviewing him or her. Assuming that one person in every complaint is good and the other is evil, what is the minimum number of people she needs to interview to correctly classify all the people named in the complaints?

Solution: She needs to interview one person in each connected component of the complaint graph.

Problem 4. Bitdiddle Bins [24 points] (3 parts)

Ben Bitdiddle has devised a new data structure called a Bitdiddle Bin. Much like an array or a set, you can INSERT values into it, and you can LOOKUP values to see if they are contained in the structure. (He'll figure out DELETE later.)

A Bitdiddle Bin is implemented as a pair of lists (arrays), designated the *neat list* and the *messy list*, with these properties:

- The *neat list* is always in sorted order. (The messy list may or may not be sorted.)
- The *messy list* has a size of at most $\sqrt{n}$, where n is the total number of values in the entire Bitdiddle Bin.

The LOOKUP algorithm for a Bitdiddle Bin is as follows:

1. Use binary search to look for the value in the neat list.
2. If it wasn't in the neat list, iterate over the entire messy list to look for the value.

The INSERT algorithm is as follows:

1. Append the value to the messy list.
2. If the messy list is now too big, CLEANUP.

This CLEANUP subroutine is run whenever the messy list grows beyond $\sqrt{n}$ items:

1. Sort the messy list.
2. Merge the messy list with the neat list.
3. The merge result is the new neat list. The new messy list is empty.

(a) [4 points] What is the worst-case asymptotic runtime of LOOKUP on a Bitdiddle Bin?

Solution: Binary search on the neat list takes $\Theta(\log n)$, and iterating over the messy list takes $\Theta(\sqrt{n})$, for a total time of $\Theta(\sqrt{n})$.

(b) [8 points] What is the worst-case asymptotic runtime of INSERT on a Bitdiddle Bin? Explain.

Solution: Appending is $\Theta(1)$. Sorting the messy list is $\Theta(\sqrt{n}\log\sqrt{n})$. Merging is $\Theta(n)$. The $\Theta(n)$ term dominates, so the asymptotic runtime is $\Theta(n)$.

(c) [12 points] What is the *amortized* asymptotic runtime of each INSERT operation, when inserting n values into an empty Bitdiddle Bin? Explain.

Solution: Use the aggregate method. The cost is $\Theta(1)$ to append to the messy list, plus $\Theta(n)$ per CLEANUP. Because CLEANUP is called $\Theta(\sqrt{n})$ times, the total cost for the n insertions is $O(n\sqrt{n})$, so the amortized cost per operation is $O(n\sqrt{n}/n) = O(\sqrt{n})$.

Problem 5. Local Minimum [24 points]

Consider an array A containing n distinct integers. We define a *local minimum* of A to be an x such that $x = A[i]$, for some $0 \leq i < n$, with $A[i-1] > A[i]$ and $A[i] < A[i+1]$. In other words, a local minimum x is less than its neighbors in A (for boundary elements, there is only one neighbor). Note that A might have multiple local minima.

As an example, suppose $A = [10, 6, 4, 3, 12, 19, 18]$. Then A has two local minima: 3 and 18.

Of course, the absolute minimum of A is always a local minimum, but it requires $\Omega(n)$ time to compute.

Propose an efficient algorithm to find some local minimum of A, and analyze the running time of your algorithm.

Solution: Use a divide-and-conquer algorithm. Let $m = n/2$, and examine the value $A[m]$ (that is, the element in the middle of the array).

Case 1: $A[m-1] < A[m]$. Then the left half of the array must contain a local minimum, so recurse on the left half. We can show this by contradiction: assume that $A[i]$ is not a local minimum for each $0 \leq i < m$. Then $A[m-1]$ is not a local minimum, which implies that $A[m-2] < A[m-1]$. Similarly, $A[m-3] < A[m-2]$. Continuing in this fashion, we obtain $A[0] < A[1]$. But then $A[0]$ is a local minimum, contrary to our initial assumption.

Case 2: $A[m+1] > A[m]$. Then the right half of the array must contain a local minimum, so recurse on the right half. This is symmetrical to Case 1.

Case 3: $A[m-1] > A[m]$ and $A[m+1] < A[m]$. Then $A[m]$ is a local minimum, so return it.

The running time recurrence is $T(n) = T(n/2) + \Theta(1)$, which yields $T(n) = \Theta(\log n)$.

Problem 6. Straits [24 points]

Let G be a connected, weighted, undirected graph. All the edge weights are positive. An edge e is a *strait* between vertices u and v if it has weight ℓ, is on a path from u to v whose other edges weigh ℓ or more, and every path between u and v contains an edge of weight ℓ or less. In other words, to go from u to v you must cross an edge of weight ℓ, but you do not need to cross edges lighter than ℓ.

Describe an efficient algorithm for finding the weight of the strait between every pair of vertices in G. Analyze the running time of the algorithm.

Hint: Use a $|V|$-by-$|V|$ table to keep track of the weights of all the straits. Initialize it so that it is correct for a graph with no edges. Then add the edges one by one.

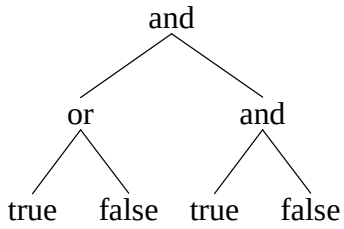
Solution: The initial table has infinity on the diagonal and zero everywhere else. Suppose the table T is correct for $E \setminus \{u, v\}$. We define the table T' for E using the rule

$$T'[x, y] = \max\big[\, T(x, y),\ \min\big(T(x, u), w(u, v), T(v, y)\big),\ \min\big(T(x, v), w(v, u), T(u, y)\big)\,\big].$$

The cost of this update is constant. We perform $V(V-1)$ such updates for every edge, so the total cost is $\Theta(EV^2)$.

Problem 7. The Cake Is a Lie! [24 points]

At Aperture Bakeries, every cake comes with a binary boolean-valued tree indicating whether or not it is available. Each leaf in the tree has either a *true* or a *false* value. Each of the remaining nodes has exactly two children and is labeled either *and* or *or*; the value is the result of recursively applying the operator to the values of the children. One example is the following tree:



If the root of a tree evaluates to *false*, like the one above, the cake is a lie and you cannot have it. Any *true* cake is free for the taking. You may modify a tree to make it *true*; the only thing you can do to change a tree is to turn a *false* leaf into a *true* leaf, or vice versa. This costs \$1 for each leaf you change. You can't alter the operators or the structure of the tree.

Cake is good. Cheap cake is even better. Describe an efficient algorithm to determine the minimum cost of a cake whose tree has n nodes, and analyze its running time.

Solution: One can compute the truth value of each node in $O(n)$ time by starting with the leaves and going up. The precise order can be computed in $O(n)$ time using either a topological sort or BFS. If the root is *true*, the cake is free, so return $\$0$.

Otherwise, we can use dynamic programming to determine this cost. Augment each node with a field C, the minimum cost to make the node *true*. Each node corresponds to the subproblem of determining C at that node.

The base case consists of the leaves. For each $true$ leaf, set C to $\$0$. For each $false$ leaf, set C to $\$1$.

Recursive cases:

For all *and* nodes: Set C to the sum of the children's C values.

For all *or* nodes: Set C to the smaller of the children's C values.

After computing C for all nodes, in the same order in which the truth values were computed initially, return the C value of the root.

Since there are $O(n)$ subproblems, and the cost of each subproblem is constant, the total running time is $O(n)$.

Problem 8. I Am Locutus of Borg, You Will Respond To My Questions [24 points]

Upon arrival at the planet Vertex T, you and Ensign Treaps are captured by the Borg. They promptly throw the ensign out of the airlock. You had better solve their problem, lest you share his fate.

The Vertex T parking lot is an $n \times n$ matrix A. There are already a number of spaceships parked in the lot. For $0 \leq i, j < n$, let $A[i][j] = 0$ if there is a ship occupying position (i, j), and 1 otherwise.

The Borg want to find the ***largest square parking space*** in which to park the Borg Cube. That is, find the largest k such that there exists a $k \times k$ square in A containing all ones and no zeros. In the example figure, the solution is 3, as illustrated by the 3×3 highlighted box.

	0	1	2	3	4
0	0	1	1	1	0
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	0	0	0
4	1	1	1	0	1

Describe an efficient algorithm that finds the size of the largest square parking space in A. Analyze the running time of your algorithm.

Hint: Call $A[0][0]$ be the *top-left* of the parking lot, and call $A[n-1][n-1]$ the *bottom-right*. Use dynamic programming, with the subproblem $S[i, j]$ being the side length of the largest square parking space whose bottom-right corner is at (i, j).

Solution: The base cases are the positions along the top and left sides of the parking lot. In each of these positions, you can fit a 1×1 square parking space if and only if the space is unoccupied. Thus, for the base cases ($i = 0$ or $j = 0$), we have $S[i, j] = A[i][j]$.

Now for the general case. Again, if $A[i][j] = 0$, then $S[i, j] = 0$, so we'll only consider the case where $A[i][j] = 1$. There is a parking space of size x with bottom-right corner at (i, j) if and only if there are three (overlapping) spaces of size $x - 1$ at each of the locations $(i - 1, j)$, $(i, j - 1)$, and $(i - 1, j - 1)$. Thus, the largest possible parking space is

$$S[i, j] = 1 + \min(\ S[i-1, j]\ ,\ S[i, j-1]\ ,\ S[i-1, j-1]\ )$$

The desired answer is $\max_{i,j} S[i, j]$, which requires $O(n^2)$ to compute.

There are n^2 subproblems, and each takes $O(1)$ time to solve, for a time of $O(n^2)$. This, added to the $O(n^2)$ to extract the desired answer, gives a total $O(n^2)$ running time, which is clearly optimal because all the data must be examined.