Final Examination Solutions

Problem 1. Miscellaneous True/False [18 points] (6 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 If the load factor of a hash table is less than 1, then there are no collisions.
Explain:

Solution: False. The table contains fewer elements than it has slots, but that doesn't prevent elements from hashing to the same slot.

(b) T F If $SAT \leq_P A$, then A is NP-hard.
Explain:

Solution: True. SAT is NP-complete. This follows from the definitions of NP-hard and NP-complete.

(c) T F The longest common subsequence problem can be solved using an algorithm for finding the longest path in a weighted DAG.
Explain:

Solution: True. Shown in lecture.

(d) T F Applying a Givens rotation to a matrix changes at most one row of the matrix.
Explain:

Solution: **False.** It changes two rows.

(e) T F The problem of finding the shortest path from s to t in a directed, weighted graph exhibits optimal substructure.
Explain:

Solution: **True.** The shortest path to t can be easily computed given the shortest paths to all vertices from which there is an edge to t.

(f) T F A single rotation is sufficient to restore the AVL invariant after an insertion into an AVL tree.
Explain:

Solution: **False.** Restoring the invariant may require $\Theta(\lg n)$ rotations.

Problem 2. More True/False [18 points] (6 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 Using hashing, we can create a sorting algorithm similar to COUNTING-SORT that sorts a set of n (unrestricted) integers in linear time. The algorithm works the same as COUNTING-SORT, except it uses the hash of each integer as the index into the counting sort table.
Explain:

Solution: False. Counting sort requires that the elements in its table are ordered according their indices into the table. Hashing the integers breaks this property

(b) T F There exists a comparison-based algorithm to construct a BST from an unordered list of n elements in $O(n)$ time.
Explain:

Solution: False. Because we can create a sorted list from a BST in $O(n)$ time via an in-order walk, this would violate the $\Omega(n \lg n)$ lower bound on comparison-based sorting.

(c) T F It is possible for a DFS on a directed graph with a positive number of edges to produce no tree edges.
Explain:

Solution: True. Consider a graph with two vertices, a and b, and a single edge (a, b). Starting the DFS at b produces no tree edges. (When the search then starts at a, the only edge is a cross edge.)

(d) T F A max-heap can support both the INCREASE-KEY and DECREASE-KEY operations in $\Theta(\lg n)$ time.
Explain:

Solution: True.
INCREASE-KEY: Change the node's key, then swap it with its parent until the heap invariant is restored.
DECREASE-KEY: Change the node's key, then MAX-HEAPIFY on the changed node.

(e) T F In a top-down approach to dynamic programming, the larger subproblems are solved before the smaller ones.
Explain:

Solution: False. The larger problems depend on the smaller ones, so the smaller ones need to be solved first. The smaller problems get solved (and memoized) recursively as part of a larger problem.

(f) T F Running a DFS on an undirected graph $G = (V, E)$ always produces the same number of cross edges, no matter what order the vertex list V is in and no matter what order the adjacency lists for each vertex are in.
Explain:

Solution: True. DFS in an undirected graph never produces cross edges.

Problem 3. Miscellaneous Short Answer [24 points] (4 parts)

You should be able to answer each question in no more than a few sentences.

(a) Two binary search trees t_1 and t_2 are *equivalent* if they contain exactly the same elements. That is, for all $x \in t_1$, $x \in t_2$, and for all $y \in t_2$, $y \in t_1$. Devise an efficient algorithm to determine if BSTs t_1 and t_2 are equivalent. What is the running time of your algorithm, assuming $|t_1| = |t_2| = n$?

Solution: One solution is to read the elements of each tree into a sorted list using an in-order walk, and compare the resulting lists for equality. This algorithm runs in $\Theta(n)$.

Another $\Theta(n)$ solution is to put all of the elements of each tree into a hash table, and compare the resulting tables.

(b) You are given a very long n-letter string, S, containing transcripts of phone calls obtained from a wiretap of a prominent politician. Describe, at a high level, an efficient algorithm for finding the 4-letter substring that appears most frequently within S.

Solution: Iterate through S, hashing each four-letter substring and keeping track of frequencies in a hash table. Either a normal hash or a rolling hash works here; each requires $\Theta(n)$ time to process the entire string ($\Theta(1)$ per substring).

(c) A *word ladder* is a sequence of words such that each word is an English word, and each pair of consecutive words differs by replacing exactly one letter with another. Here is a simple example:

TREE, FREE, FRET, FRAT, FEAT, HEAT, HEAP

Assume that you are supplied with a function GET-WORDS(w) that returns, in $\Theta(1)$ time, a list of all English words that differ from w by exactly one letter (perhaps using a preprocessed lookup table). Describe an efficient algorithm that finds a word ladder, beginning at w_1 and ending at w_2, of minimum length.

Solution: The GET-WORDS function defines the edges in an implicit graph where words are vertices. Use a (bidirectional) BFS on this graph.

(d) What procedure would you use to find a longest path from a given vertex s to a given vertex t in a weighted directed acyclic graph, when negative edge weights are present?

Solution: Dynamic programming. The fact that some edges have negative edge weights doesn't affect anything.

Problem 4. Changing Colors [15 points]

Consider a directed graph G where each edge $(u, v) \in E$ has both a weight $w(u, v)$ (not necessarily positive) as well as a color $color(u, v) \in \{\text{red}, \text{blue}\}$.

The weight of a path $p = v_1 \to v_2 \to \ldots \to v_k$ is equal to the sum of the weights of the edges, *plus* 5 for each pair of adjacent edges that are not the same color. That is, when traversing a path, it costs an *additional* 5 units to switch from one edge to another of a different color.

Give an efficient algorithm to find a lowest-cost path between two vertices s and t, and analyze its running time. (You may assume that there exists such a path.) For full credit, your algorithm should have a running time of $O(VE)$, but partial credit will be awarded for slower solutions.

Solution: Construct a new graph G' as follows: for each vertex $v \in V$, create *two* vertices, a red vertex v_R and a blue vertex v_B, in G'. For each edge $(u, v) \in E$, if $color(u, v) = \text{red}$, create (u_R, v_R) in G'. Otherwise, create (u_B, v_B). In either case, the new edge should have the same weight as the original. Finally, for each vertex $v \in V$, create the edges (v_R, v_B) and (v_B, v_R) in G', each with weight 5.

This works because all the red edges run between red vertices in G', and all the blue edges run between blue vertices. Switching colors requires traversing an edge in G' between a red and a blue vertex, incurring the extra cost of 5 units.

Now, run Bellman-Ford twice: once starting from s_R and once starting from s_B. Examine the paths ending at t_R and t_B. Determine which of the four paths ($s_R \rightsquigarrow t_R$, $s_R \rightsquigarrow t_B$, $s_B \rightsquigarrow t_R$, $s_B \rightsquigarrow t_B$) is shortest. Strip the subscripts to recover the desired path in G.

Constructing G' takes $O(V + E)$ time. G' has $2V$ vertices and $V + E$ edges. Running Bellman-Ford on G' takes $O(V'E') = O((2V)(V + E)) = O(VE)$ time. Thus the total running time is $O(VE)$.

Problem 5. DAG Paths [20 points]

Consider two vertices, s and t, in some directed acyclic graph $G = (V, E)$. Give an efficient algorithm to determine whether the number of paths in G from s to t is odd or even. Analyze its running time in terms of $|V|$ and $|E|$.

Solution: Here we present a dynamic programming algorithm that finds the number of paths from s to t. Determining whether that number is odd or even is easy.

Produce a topological ordering $v_1, v_2, \ldots, v_n$ of the vertices of G, and without loss of generality, let $v_1 = s$ and $v_n = t$ (because that is the only part of the graph we care about).

Define subproblems as follows: let $S[i]$ be the number of paths from v_i to $v_n = t$. We can define a recurrence expressing the solution to these subproblems in terms of smaller subproblems:

$$S[i] = \sum_{j \in \mathrm{adj}(i)} S[j]$$

where $\mathrm{adj}(i)$ is the set of all vertices v s.t. $(i, v) \in E$. (That is, all vertices to which there is an edge from i.) The base case is $S[n] = 1$.

The solution to the general problem is the number of paths from $v_1 = s$ to $v_n = t$, or $S[1]$. The total time required to solve the $O(V)$ subproblems is $O(E)$, and the topological sort requires $O(V + E)$ time, so the total running time is $O(V + E)$.

Problem 6. Square Depth [25 points]

Imagine starting with the given number n, and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The *square-depth* $S(n)$ of n is defined to be the maximum number of square numbers you can arrange to see along the way. For example, $S(32492) = 3$ via the sequence

$$32492 \rightarrow \mathbf{3249} \rightarrow \mathbf{324} \rightarrow 24 \rightarrow \mathbf{4}$$

since 3249, 324, and 4 are squares, and there is no better sequence of chops giving a larger score (although you can do as well other ways, since 49 and 9 are both squares).

Describe an efficient algorithm to compute the *square-depth*, $S(n)$, of a given number n, written as a d-digit decimal number $a_1 a_2 \ldots a_d$. Analyze your algorithm's running time.

Your algorithm should run in time polynomial in d. You may assume a function IS-SQUARE(x) that returns, in constant time, 1 if x is square, and 0 if not.

Solution: We can solve this using dynamic programming.

First, we define some notation: let $n_{ij} = a_i \ldots a_j$. That is, n_{ij} is the number formed by digits i through j of n. Now, define the subproblems by letting $D[i, j]$ be the square-depth of n_{ij}.

The solution to $D[i, j]$ can be found by solving the two subproblems that result from chopping off the left digit and from chopping off the right digit, and adding 1 if n_{ij} itself is square. We can express this as a recurrence:

$$D[i, j] = \max(D[i + 1, j], D[i, j - 1]) + \text{IS-SQUARE}(n_{ij})$$

The base cases are $D[i, i] = \text{IS-SQUARE}(a_i)$, for all $1 \leq i \leq d$. The solution to the general problem is $S(n) = D[1, d]$.

There are $\Theta(d^2)$ subproblems, and each takes $\Theta(1)$ time to solve, so the total running time is $\Theta(d^2)$.

Problem 7. Least-Squares Minimization [30 points] (2 parts)

(a) Some least-squares minimizations can be solved by substitution even though the coefficient matrix is not upper triangular. For example, the following problem can be solved by substitution:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -7 & -2 \\ 0 & 3 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 8 \\ 4 \end{pmatrix} .$$

One way to solve such problems is to find an ordering of the rows and an ordering of the columns: find x_1 from the third constraint, then substitute x_1 and find x_2 from the second constraint, and so on.

Describe an efficient algorithm to find suitable orderings for the unknowns and the constraints. For simplicity, the algorithm is allowed to assume that there is such a reordering.

The input matrix is given to the algorithm as a Python list of m rows, where each row is a list of tuples; each tuple contains a column index (0 to $n-1$) and a nonzero value. Zero matrix entries are not represented at all. The output should be a list of n (unknown index, constraint index) pairs, giving the substitution ordering. The algorithm should run in time $\Theta(n+m+k)$, where k the number of nonzeros in the entire matrix (which is also the number of tuples in the input).

Solution: We can perform a substitution for any row in which only one unsolved unknown remains. Start an empty queue S. This queue contains the rows for which there is exactly one unsolved unknown, so we can solve for that unknown and substitute the result into the other constraints.

Allocate two arrays: let $C[i]$ be the number of unknowns in constraint i, and $U[j]$ be a list of constraint/row numbers in which the unknown x_j appears.

Read in the constraints (rows). For each column index j in constraint i, increment $C[i]$ and append i to $U[j]$. For each row i that contains only one entry, append i to S; we can immediately perform the substitution in this row.

Now repeat the following until all constraints are resolved (i.e., S becomes empty): extract the first row number, i, from S. Solve for the unknown in constraint i and save the result. Now that we have solved for this unknown, we are one step closer to solving each constraint in which it appears: for each i in $U[j]$, decrement $C[i]$. If $C[i]$ is now equal to 1, there is only one unsolved unknown left, so append i to S.

Because there are m constraints to solve, n unknown values to solve, and k substitutions to perform, and each of these operations takes $O(1)$ time, the total time is $O(n+m+k)$.

(b) For many problems, substitution alone is not sufficient; Givens rotations are necessary.

One interesting challenge in performing a sequence of rotations efficiently is to compute unions of pairs of integer sets.

Describe an algorithm that repeatedly takes two sets of integers represented as lists of integers and and generates a list of the union of the two sets. Also describe any data structures that your algorithm uses.

The integers are all between 0 and $n-1$, there are no duplicates in the input lists, and there should be no duplicates in the output list. The pairs of sets are unrelated.

The algorithm can use $O(n)$ preprocessing time before computing the first union. After the preprocessing phase, computing each union should take $O(k+k')$ *worst-case* time, where k and k' are the sizes of the two input lists.

Solution: Take the first set and append just those elements from the second set that are not already present, then return the resulting set.

To do this, use a list, A, of n booleans, to keep track of the elements in the first list. Initialize them all to `False`. This preprocessing requires $O(n)$ time.

For i in the first input list, set $A[i]$ to `True`. Now, for j in the second input list, if $A[j]$ is `False`, append j to the first list. Return the resulting list.

Using hashing to perform the lookups is not correct because it requires $O(k+k')$ *expected* time, when worst-case time was required.

Problem 8. Sorting on Disks [30 points] (3 parts)

You need to sort a large array of n elements that is stored on disk (in a file). Files support only two operations:

1. READ(f) returns the next number stored in file f.
2. WRITE(f, p) writes the number p to the end of file f.

These operations are slow, so we wish to sort the given array into another file while incurring as few READs and WRITEs as possible. Your algorithm may create new files if you desire.

If we had unlimited RAM, we could solve this problem by simply reading the entire array into memory, sorting it, then writing it to disk. Unfortunately, the computer can only store a limited number of numbers in memory, so this simple approach is infeasible.

(a) If our computer can store at most $\frac{n}{2}$ numbers in memory, how would you sort the file? Assume the computer has enough working memory to store any auxiliary information your algorithm needs. How many READs and WRITEs does your algorithm incur?

Solution: READ the first $\frac{n}{2}$ elements of the file, sort them, then WRITE the sorted array to a new file. READ and sort the remaining $\frac{n}{2}$ elements, then merge the two sorted arrays. This process incurs $\frac{3n}{2}$ READs and $\frac{3n}{2}$ WRITEs.

(b) How would you sort the file if the computer can store at most $\frac{n}{4}$ numbers in memory? Again assume adequate working memory. How many READs and WRITEs does your algorithm incur?

Solution: Repeat the first step of part (a) 3 times, then perform a 4-way merge. This incurs $\frac{7n}{4}$ READs and $\frac{7n}{4}$ WRITEs.

(c) Now imagine the computer uses some technology (e.g., flash memory) where the READ operation is fast, but the WRITE operation is expensive. Modify your algorithm from part (b) to minimize the number of WRITEs. The computer can still store at most $\frac{n}{4}$ numbers in memory.

Solution: Perform a selection sort: select the smallest $\frac{n}{4}$ elements from the file, sort them, then WRITE them to the new file. Repeat this process for each successively larger set of $\frac{n}{4}$ elements until all the elements of the original array are sorted. This incurs n WRITEs.

Problem 9. Star Power! [20 points]

Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which (along with making you temporarily invincible) lets you traverse the edge of your choice for free. In other words, you may change the weight of any one edge to zero.

Give an efficient algorithm to find a lowest-cost path between two vertices s and t, given that you may set one edge weight to zero. Analyze your algorithm's running time. For full credit, your algorithm should have a running time of $O(E + V \lg V)$, but partial credit will be awarded for slower solutions.

Solution 1: Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t. Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta(u, v)$.

Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \to v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \to v \rightsquigarrow t$ is the desired path.

The algorithm requires two invocations of Dijkstra, and an additional $\Theta(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra: $\Theta(E + V \lg V)$.

Solution 2: Construct a new graph G' as follows: for each vertex $v \in V$, create *two* vertices, v_0 and v_1, in G'. The v_0 vertices represent being at v in G and having not yet used the Star. The v_1 vertices represent being at v, and having already used the Star.

For each edge $(u, v) \in E$, create three edges in G': (u_0, v_0) and (u_1, v_1) with the original weight $w(u, v)$, and (u_0, v_1) with a weight of zero. An edge, (u_0, v_1), crossing from the "0" component to the "1" component of the graph represents using the Star to traverse (u, v).

Now, run Dijkstra's algorithm with source s_0 and destination t_1. Drop the subscripts from the resulting path to recover the desired answer. (Observe that, because all edge weights are positive, it is always optimal to use the Star somewhere.)

G' has $2V$ vertices and $3E$ edges. Thus, constructing G' takes $\Theta(V+E)$ time, and running Dijkstra on G' takes $\Theta(E + V \lg V)$ time. Thus the total running time is $\Theta(E + V \lg V)$.