Practice Final Exam

Problem 1. Recurrences (4 parts)

For each of the recurrences below, do the following:

- Give the solution using $\Theta$-notation. You need not provide a proof or other justification.
- Name a recursive algorithm we’ve seen during the term whose running time is described by that recurrence.

(a) $T(n) = T(n/2) + \Theta(1)$
   Solution: $\Theta(\log n)$. Binary search.

(b) $T(n) = 2T(n/2) + \Theta(n)$
   Solution: $\Theta(n \log n)$. MERGE-SORT.

(c) $T(n) = T(n/5) + T(7n/10) + \Theta(n)$
   Solution: $\Theta(n)$. SELECT

(d) $T(n) = 7T(n/2) + \Theta(n^2)$
   Solution: $\Theta(n^{\log 7})$. Strassen’s matrix-multiplication algorithm.

Problem 2. Algorithms and running times (5 parts)

Match each algorithm below with the tightest asymptotic upper bound for its worst-case running time by inserting one of the letters A, B, . . ., into the corresponding box. Some running times may be used multiple times or not at all.
You need not justify your answers.

<table>
<thead>
<tr>
<th>Algorithm/Structure</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra’s implemented using Fibonacci heap</td>
<td>$O(E + V \lg V)$</td>
</tr>
<tr>
<td>Dijkstra’s implemented using binary heap</td>
<td>$O((V + E) \lg V)$</td>
</tr>
<tr>
<td>Building a BST</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BUILD-HEAP</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>Viterbi decoding in HMM with $n^{1/2}$ states and $n$ emissions</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td></td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Problem 3. Design Techniques and Data Structures (5 parts)**

For each of the following design techniques and data structures, name an algorithm covered this term that uses it.

(a) **Divide and conquer:**
   **Solution:** MERGE-SORT uses divide and conquer. It divides the problem into two problems of half the size (the left and right halves of the array), conquers the subproblems by running MERGE-SORT recursively, and combines the results by merging the subarrays together.

(b) **Dynamic programming:**
   **Solution:** The typesetting problem used dynamic programming.

(c) **Greedy:**
   **Solution:** Prim’s algorithm for minimum spanning tree is a greedy algorithm.

(d) **Binary search tree:**
   **Solution:** The dynamic maximum-prefix problem from problem set 4 used an augmented red-black tree.

(e) **FIFO queue:**
   **Solution:** Breadth-first search uses a FIFO queue.
Problem 4.  True or False, and Justify (12 parts)
Circle T or F for each of the following statements to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is wrong, explain why. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

(a)  T F  If \( f(n) \) is asymptotically positive, then \( f(n) + o(f(n)) = \Theta(f(n)) \).

   Solution: True.
   Clearly, \( f(n) + o(f(n)) \) is \( \Omega(f(n)) \), so let us prove that \( f(n) + o(f(n)) = O(f(n)) \). Let \( g(n) \in o(f(n)) \). Then for any \( c > 0 \), there exists \( n_0 \) such that \( g(n) \leq c(f(n)) \) for all \( n \geq n_0 \). Hence, \( f(n) + g(n) \leq (c + 1)f(n) \) for all \( n \geq n_0 \), which means that \( f(n) + g(n) = O(f(n)) \).

(b)  T F  An adversary can provide an input to randomized quicksort that will elicit its \( \Theta(n^2) \) worst-case running time.

   Solution: False. The worst-case behavior of quicksort happens due to bad coin flips; it has nothing to do with the adversary’s choice of inputs.

(c)  T F  Any comparison sort of 5 elements requires at least 7 comparisons in the worst case.

   Solution: True. The decision tree for sorting 5 elements has 5! = 120 leaves. Any comparison sort can only distinguish at most \( 2^6 = 64 \) different elements with fewer than 7 comparisons and is therefore unable to sort 7 elements correctly all the time.

(d)  T F  Consider a sequence of \( n \) operations on an initially empty dynamic set. Suppose that the amortized running time of each operation is \( O(1) \). Then, the \( n \) operations take \( O(n) \) time in the worst case.

   Solution: True.

(e)  T F  In an HMM, let \( x_j \) be the emission observed at time \( j \). Given a series of observed emissions \( x_1, x_2, \ldots, x_n \), the most likely state at time \( i \) is independent of emissions \( x_{i+1}, x_{i+2}, \ldots, x_n \).

   Solution: False

(f)  T F  Prim’s algorithm, Dijkstra’s algorithm, and the Bellman-Ford algorithm are all examples of greedy algorithms.

   Solution: False. Not BF.

(g)  T F  For the all-pairs shortest-paths problem on an edge-weighted graph \( G = (V, E) \) with \( E = \Theta(V^{1/2}) \), the Floyd-Warshall algorithm is asymptotically at least as fast as Johnson’s algorithm.

---

1. *elicit* transitive verb 1: to draw forth or bring out (something latent or potential) (hypnotism elicited his hidden fears) 2: to call forth or draw out (as information or a response) (her performance elicited wild applause) — *Merriam-Webster’s Collegiate Dictionary*, Tenth Edition, 1993.
Solution: False. Floyd-Warshall runs in $O(V^3)$ time. Johnson’s runs in $O(V^2 \log V + V E)$. If $E = \Theta(V^{3/2})$, then Johnson’s runs in $O(V^2 \log V + V^{5/2}) = o(V^3)$.

(h) T F Suppose that the constraint graph $G = (V, E)$ of a linear-programming system of difference constraints is acyclic. Then, a solution always exists and can be found in $O(V + E)$ time.

Solution: True. Dag shortest paths.

(i) T F Let $G = (V, E)$ be an edge-weighted digraph, where edge weights are given by the function $w : E \rightarrow \mathbb{R}$. Define another edge-weight function $w' : E \rightarrow \mathbb{R}$ by

$$w'(u, v) = w(u, v) - \text{out-degree}(u) + \text{out-degree}(v).$$

Then, $G$ contains a negative-weight cycle under $w$ if and only if $G$ contains a negative-weight cycle under $w'$.

Solution: True. The total weight on a cycle $C$ under $w'$ is

$$W_C = \sum_{(u,v) \in C} (w(u, v) - \text{out-degree}(u) + \text{out-degree}(v)).$$

Since this summation telescopes, we have $W_C = \sum_{(u,v) \in C} w(u, v)$.

(j) T F Suppose that all edge capacities in a flow network are integer multiples of 3, but that the value of a flow between the source $s$ and the sink $t$ is not a multiple of 3. Then, an augmenting path from $s$ to $t$ exists.

Solution: True. Consider the minimum cut. It is made up of edges with capacities that are multiples of 3, so the capacity of the cut (sum of capacities of edges in the cut) must be a multiple of 3. By the Maxflow-Mincut theorem, the maximum flow has the same value.

(k) T F Given a maximum flow $f$ on a flow graph $G = (V, E)$ with source $s$ and sink $t$, a minimum cut separating $s$ from $t$ can be found in $O(V + E)$ time.

Solution: True. BFS or DFS.

(l) T F The Karp-Rabin algorithm always reports a match of a pattern in a text string if one exists.

Solution: True. It may have false positives, but no false negatives.

Problem 5. Set Equality

Let $S$ and $T$ be two sets of numbers represented as unordered lists of distinct numbers. All you have are pointers to the heads of the lists, but you do not know the list lengths. Describe an $O(\min \{|S|, |T|\})$-expected-time algorithm to determine whether $S = T$. You may assume that any operation on one or two numbers can be performed in constant time.

Solution: First, check that both sets are the same size. If they are not, then they cannot be equal. To do this check in $O(\min \{|S|, |T|\})$ time, just iterate over both lists in parallel. That is, advance
one step in $S$ and one step in $T$. If both lists end, the lengths are the same. If one list ends before the other, they have different lengths.

If both lists are the same size, then we want to check whether the elements are the same. We create a hash table of size $\Theta(|S|)$ using universal hashing with chaining. We iterate over $S$, adding each element from $S$ to the hash table. Then we iterate over $T$. For each element $x \in T$, we check whether $x$ belongs to the hash table (that is, whether it is also in $S$). If not, then we return that the sets are not identical. If so, then continue iterating over $T$.

Any sequence of $|S| = |T|$ INSERT and SEARCH operations in the table take $O(|S|)$ time in expectation (see CLRS p.234), so the total runtime is $O(\min \{ |S|, |T| \})$ in expectation.

**Problem 6. Minimum Spanning Tree**

Let $G = (V, E)$ be a connected undirected graph with edge-weight function $w : E \rightarrow \mathbb{R}$. Consider the following algorithm:

1. while there exists a cycle $C$ in $G$
2. do find an edge $e \in C$ such that $w(e) = \max_{e' \in C} \{ w(e') \}$
3. $G \leftarrow (V, E - \{ e \})$

Prove that when this algorithm terminates, $G$ forms a minimum spanning tree of the original input graph.

**Solution:** This algorithm repeatedly removes a maximum-weight edge from a cycle.

First, let’s argue that the graph $G$ forms a spanning tree. That is, that $G$ is acyclic and connects all the vertices. The first part is obvious—since we continue removing edges until there are no cycles, the resulting $G$ must be acyclic. Since the only edges removed are in cycles, removing an edge never disconnects any two vertices. Thus, the output $G$ remains connected as well, and forms a spanning tree.

Now we argue that the spanning tree $G$ is minimum. If it is not minimum, then there is some edge $(u, v) \in E$ that can be removed to partition the graph into two connected components such that $(u, v)$ is not a minimum-weight edge crossing the cut. Suppose that $(x, y)$ is the minimum-weight edge crossing this cut. The algorithm removed $(x, y)$, so there must have been some cycle such that $(x, y)$ was a maximum-weight edge on the cycle. Since any cycle including $(x, y)$ crosses the cut in two places, the other edge that was left behind must have had weight at most $w(x, y)$. Thus, by induction, we must have $w(u, v) \leq w(x, y)$.

**Problem 7. Woody the Woodcutter (3 parts)**

Given a log of wood of length $k$, Woody the woodcutter will cut it once, in any place you choose, for the price of $k$ dollars. Suppose you have a log of length $L$, marked to be cut in $n$ different locations labeled $1, 2, \ldots, n$. For simplicity, let indices $0$ and $n+1$ denote the left and right endpoints of the original log of length $L$. Let the distance of mark $i$ from the left end of the log be $d_i$, and assume that $0 = d_0 < d_1 < d_2 < \cdots < d_n < d_{n+1} = L$. The wood-cutting problem is the
problem of determining the sequence of cuts to the log that will (1) cut the log at all the marked places, and (2) minimize your total payment to Woody.

(a) Give a small example illustrating that two different sequences of cuts to the same marked log can result in two different costs.

Solution: Suppose that we have a log of length 4, and we want to cut at \( d_1 = 1, d_2 = 2, \) and \( d_3 = 3 \). Then cutting \( d_1, d_2, d_3 \) results in a total cost of \( 4 + 3 + 2 = 9 \). Cutting at \( d_2, d_1, d_3 \) results in a total cost of \( 4 + 2 + 2 = 8 \).

Let \( c(i, j) \) be the minimum cost of cutting a log with left endpoint \( i \) and right endpoint \( j \) at all its marked locations.

(b) Complete the following recursive definition, and briefly justify your answer:

Solution:

\[
c(i, j) = \min_{i < k < j} \{ c(i, k) + c(k, j) + (d_j - d_i) \}
\]

First off, the length of the log segment is \( d_j - d_i \), so Woody charges $d_j - d_i$ for the cut regardless of the location.

As for the rest of the answer, the problem exhibits optimal substructure. If the optimal answer involves cutting the log at position \( k \), then we use the optimal answers for segment \( i \) to \( k \) and segment \( k \) to \( j \) to perform the rest of the cuts (use a cut & paste argument for completeness). We just try all possible locations \( k \) for the next cut and choose the best one.

(c) Using part (b), describe an efficient algorithm to solve the wood-cutting problem. What is the running time of your algorithm?

Solution: Build a table \( C \) of size \((n+1) \times (n+1)\) to hold the values \( C[i][j] = c(i, j) \). We initialize the table first by setting \( C[i][i] \leftarrow 0 \) and \( C[i][i+1] \leftarrow 0 \). Then we just fill along the diagonals to fill in the \( C[i][j] \) for segments whose endpoints are two positions away, then three, etc. We are filling the table in order of size of log segments.

Pseudocode for this algorithm looks like

1  for \( i \leftarrow 0 \) to \( n \)
2     do \( C[i][i] \leftarrow 0 \)
3         \( C[i][i+1] \leftarrow 0 \)
4  for \( s \leftarrow 2 \) to \( n \)
5     do for \( i \leftarrow 0 \) to \( n \)
6         do \( j \leftarrow i + s \)
7             \( C[i][j] \leftarrow \min_{i < k < j} \{ c(i, k) + c(k, j) + d_j - d_i \} \)

The important part is that when we fill compute \( C[i][j] \), all the necessary values to max over have already been computed. This is because we max over smaller-size log
segments, and we compute entries in order of log-segment size. The answer is found at $C[0][n]$.
This algorithm takes $O(n^3)$ time.

Problem 8. Edge Covering

Given an undirected graph $G = (V, E)$ with no isolated vertices (vertices with degree 0), an edge cover is a set $C \subseteq E$ of edges such that for all $u \in V$, there exists a $v \in V$ such that $(u, v) \in C$. The edge-covering problem is the problem of finding an edge cover of minimum cardinality.

Describe an $O(1)$-approximation algorithm for the edge-covering problem. Analyze your algorithm’s running time and its ratio bound (by what factor worse than the optimal is the approximation your algorithm produces?).

Solution: We assume that the graph is in the adjacency-list representation. We keep auxiliary information associated with each vertex to check whether a vertex is already covered. In the beginning, no vertex is covered. Iterate over all vertices. When considering vertex $u$, check if it is already covered. If so, ignore it and move to the next vertex. If not, let $v$ be the first element of $Adj[u]$. Add $(u, v)$ to $C$ and mark both $u$ and $v$ as being covered. Then continue on to the next vertex.

Each vertex is considered once and marked as covered once. Total amount of work spent on a vertex is $O(1)$, so the total time of the algorithm is $O(V)$.

This algorithm finds a 2-approximation. The optimal covering must have cardinality at least $|V|/2$, because each edge covers at most 2 new vertices. Our covering has at most $|V| - 1$ edges, which is off by less than a factor of 2.

Problem 9. Radix Sort

We have seen that COUNTING-SORT requires $O(n+k)$ time to sort $n$ numbers in the range $1, \ldots, k$, while RADIX-SORT requires $O(nd)$ time to sort $n$ number of $d$ digits each. By a judicious combination of these algorithms, we can get a linear running time, as in COUNTING-SORT, when operating on elements from a wider range, as in RADIX-SORT.

(a) Given a number $x$, show how to get the $i$th digit of its base-$r$ representation in $O(1)$ time. (The 1st digit is the least-significant one.)

Solution: The $i$th digit is $\lfloor x/r^{i-1} \rfloor \mod r$. To see this, note that dividing by $r^{i-1}$ and taking the floor is just “shifting right” the base-$r$ representation of $x$ by $i−1$ digits, and taking that value mod $r$ is just keeping the least significant remaining digit. If we can compute $r^{i-1}$ in $O(1)$ time, then an algorithm for computing the $i$th digit is straightforward. Alternatively, we can get the digits in sequence by computing $1, r, r^2, r^3, \ldots$, which yields $O(1)$ time per digit.

(b) What is the running time of RADIX-SORT on an array of $n$ numbers in the range $0, \ldots, n^5 − 1$, when using base-10 representations?
Solution: Using base 10, the numbers have \( d = \log n^5 = 5 \log n \) digits. Each COUNTING-SORT call takes \( \Theta(n+10) = \Theta(n) \) time, so the running time of RADIX-SORT is \( \Theta(nd) = \Theta(n \log n) \).

(c) Give an algorithm to sort an array of \( n \) numbers in the range \( 0, \ldots, n^5 - 1 \) in only \( O(n) \) time. Does your technique extend to wider ranges of elements? Explain.

Solution: If we use base \( n \), the numbers have \( d = \log_n n^5 = 5 \) digits, and by part (a) we can compute each digit in \( O(1) \) time. Therefore each COUNTING-SORT call takes \( \Theta(n + n) = \Theta(n) \) time, so the running time of RADIX-SORT is \( \Theta(nd) = \Theta(n) \). This method extends to numbers in the range \( 0, \ldots, n^c - 1 \) for any constant \( c \).