CSE 241 Algorithms and Data Structures

Practice Exam Solutions

Assigned: November 21, 2013

Here is a collection of problems to help you get some practice for the final exam. This is not the correct length of the exam. The format and the length will be similar to the midterm.

Problem 1. True, False and Justify [4-5 each points]

For each of the following statements, indicate whether the statement is true or false. If the statement is correct, briefly state why. If the statement is wrong, explain why. Just a true or false solution will not get you any points, the justification is worth all of the points. The more accurate your justification, the higher your grade. If you have two justifications, one correct and one incorrect, you will still not receive all the points. So be complete, but concise.

(a) Checking of there is a pair of non-equal elements in an array takes $\Omega(n \lg n)$ time.

Solution: False. This can be done in linear time by comparing the first element to all the other elements. If all elements are equal, then the first element is equal to all others. If there is a pair which is non-equal, then the first element is not equal to one or both of them. Therefore, if an unequal element is found, return true; otherwise, return false.

(a) Let T be the minimum spanning tree of a graph $G = (V, E)$. Then, for any pair of vertices s and t, the shortest path from s to t in G is the path from s to t in T.

Solution: False. The minimum spanning tree does not necessarily contain shortest paths. An easy counter example is a graph with 3 vertices s, a and t, such that $w(s, a) = 2$, $w(a, t) = 3$ and $w(s, t) = 4$. The minimum spanning tree $T = \{(s, a), (a, t)\}$. However, the shortest path from s to t is using the edge (s, t).

(b) Assume that you are given a magical priority queue data structure, which performs extract-min, insert and decrease-key in $O(1)$ time each. Then you can implement Dijkstra's algorithm to run in $O(V + E)$ time on a graph with V vertices and E edges.

Solution: True. Dijkstra's algorithm is analyzed in terms of these three operations. As we saw in class, it performs V inserts and extract-min operations and E decrease key operations. Therefore, if each of these operations runs in $O(1)$ time, then Dijkstra's algorithm would run in $O(V + E)$ time.

(c) Let $f(n)$ and $g(n)$ be two asymptotically positive functions. If $f(n) = \Theta(g(n))$, then $|f(n) - g(n)| = \Theta(f(n) + g(n))$.

Solution: False. Consider the case when $f(n) = g(n) = n$. Then the LHS is 0, while the RHS is $f(n)$ which may be non-zero.

(d) Suppose a hash table has m slots and only one of these slots contains a single element with key k and the remaining slots are empty. Say we search in this table for x various other keys none of which are equal to k. Assuming simple uniform hashing, the probability that at least one of these searches probes the slot containing k is x/m.

Solution: False. The probability that at least one of the searches probes the slot containing k is 1 minus the probability that none of the searches probes that slot. The probability that a particular search probes this slot is $1/m$. Therefore, the probability that a particular search does not probe this slot is $1 - 1/m$. Since all searches are independent, the probability that none of the searches probe the slot is $(1 - 1/m)^x$. Therefore, the probability that at least one search probes the slot is $1 - (1 - 1/m)^x$

(e) If $T(n) = 5T(n/4) + n$, then $T(n) = O(n \log^{5/4} n)$.

Solution: False. By the master method, we have $T(n) = \Theta(n^{\log_4 5}) \neq O(nlog^{5/4}n)$.

(f) The expected time to search for an element in a skip list is $\Theta(\lg n)$.

Solution: True. The search time on skip list is $c_1 \lg n$ with probability at least $1 - 1/n$ for some constant c_1. In addition, in the worst case, the search time is $\Theta(n) < c_2 n$. Therefore, the expected search time is $\leq c_1 \lg n \times (1 - 1/n) + c_2 n \times (1/n) \leq c_1 \lg n + c_2 = \Theta(\lg n)$.

(g) Deleting an element from a binary search tree takes $O(\lg n)$ time in the worst case.

Solution: False. It takes $O(h)$ time and for an ordinary binary search tree, h can be $\omega(\lg n)$.

(h) Depth first search is asymptotically faster than breadth first search.

Solution: False. They both take $O(V + E)$ time.

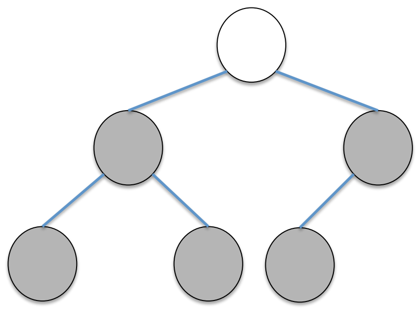
Problem 2. Short Answers [5-7 each points]

(a) Say we have a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. The edge weights of the graph are integers between 1 and 10. Give an efficient algorithm to find the shortest paths from a given source vertex s to all other vertices.

Solution: Create a new graph G'. G' has all the nodes from G and also some "dummy nodes". For every edge (u, v) in G, if $(u.v)$ has weight i, then add $i - 1$ "dummy nodes", $u_1, u_2, ..., u_{i-1}$, and add i edges such that there is an edge from u to u_1, from u_j to u_{j+1} for all $1 \leq j < i - 1$, and from u_{i-1} to v. By construction, shortest paths in G' are the same as shortest paths in G. The number of nodes in G' is at most $V' = V + 9E$, and the number of edges in G' is at most $E' = 10E$. All edges in G' have weight 1 and we can run BFS to find shortest paths. The running time is $O(V' + E') = O(V + E)$.

(b) Consider a min-heap with 6 elements. Draw this heap and shade in all the positions that can be occupied by the third-smallest element.

Solution:



(c) Say you are trying to augment a red-black tree with a particular attribute a. What property should this attribute have so that you can easily maintain it?

Solution: In order to be maintainable through rotations, this attribute a for a particular node x in the tree should depend only on the information stored in the children of x in the tree.

(d) Which of the following are greedy algorithms? You do not need to explain, just mark the greedy algorithms with a check-mark or a circle them.

1. Prim's algorithm for MST.
2. Dijkstra's algorithm for shortest paths.

3. Bellman-Ford algorithm for shortest paths.
4. Finding the longest common subsequence of two sequences.

Solution: Prim's algorithm and Dijkstra's algorithm are greedy. The other two are not.

(e) Solve the recurrence $T(n) = T(9n/10) + \log n$.

Solution: By Master Method, case 2, we get $T(n) = \Theta(\log^2 n)$.

(f) Prove that the solution to the recurrence $T(n) = 2^n T(n-1)$ is $T(n) = \Theta((\sqrt{2})^{n(n+1)})$.

Solution: For base case, say $T(0) = c_1$. Assume as Inductive hypothesis that $T(k) \leq c_2(\sqrt{2})^{k(k+1)}$ for all $k < n$. Therefore, we get

$$\begin{aligned} T(n) &= 2^n T(n-1) \\ &\leq c_2 2^n (\sqrt{2})^{(n-1)n} \\ &= c_2(\sqrt{2})^{(n-1)n+2n} \\ &= c_2(\sqrt{2})^{(n+1)n} \end{aligned}$$

Problem 3. Package Shipping You work for a manufacturing company. The owner decides that the company is spending too much money on shipping items from the factory (where the items are produced) to the warehouse (where the items are stored). You have been asked to use your algorithms knowledge to figure out the best shipping strategy.

Every day the factory produces n items and the same items are produced in the same order every day. As the items arrive at the loading dock over the course of the day they must be packaged into boxes and shipped out. Items are boxed in contiguous groups according to their arrival order; for example, items $1, 2, 3, 4$ might be placed in the first box, items $5, 6, 7$ in the second, items $8, 9, 10, 11, 12, 13$ in the third, and so on. Alternatively, items $1, 2, 3$ may be placed in the first box, $4, 5, 6.7$ in the second and so on. You may not skip items or ship them out of order. For example, items $1, 2, 5$ in the first box and $3, 7, 8$ in the second box, and so on, is illegal. Each item i has two attributes, value $v(i)$ and and weight $w(i)$, and you know the values and weights of all items.

There are two types of shipping options available to you:

- **Limited-Value Boxes:** One of your shipping companies offers insurance on boxes and hence requires that any box shipped through them must contain no more than V units of value. Therefore, if you pack items into such a limited-value box, you can place as much weight in the box as you like, as long as the total value in the box is at most V. Each box of this type costs a dollars to ship.
- **Limited-Weight Boxes:** Another of your shipping companies lacks the machinery to lift heavy boxes, and hence requires that any box shipped through them must contain no more than W units of weight. Therefore, if you pack items into such a limited-weight box, you can place as much value in the box as you like, as long as the total weight inside the box is at most W. Each box of this type costs b dollars to ship.

You may assume that every individual item has value at most V and weight at most W. Your job is to determine the optimal way of packaging items into boxes with specified shipping options, so that shipping costs are minimized. You may choose different shipping options for different boxes.

For partial credit, solve the problem when $a = b$.

Solution: Use dynamic programming with a running time of $O(n)$. For each item i, $C[i]$ is the minimum cost to ship all items from i to n. Our final solution is $C[1]$.

For all i, calculate $A(i)$ as the last item you can pack into a limited value box if you started at i. Therefore, $\sum_{j=i}^{A(i)} v(i) \leq V$ and $\sum_{j=i}^{A(i)+1} v(i) > V$. Similarly, $B(i)$ is the last item you can pack into a limited value box if you started at i. Therefore, $\sum_{j=i}^{B(i)} w(i) \leq W$ and $\sum_{j=i}^{B(i)+1} w(i) > W$.

Therefore $C(i) = \min\{C[A(i)+1] + a, C[B(i)+1] + b\}$ As a base case $C[n+1] = 0$. This DP calculation takes $O(n)$ time.

Now to calculate $A(i)$ and $B(i)$ for each i in $O(n)$ time.

```
$vbegin \leftarrow 0$
$tval \leftarrow 0$
**for** $i \leftarrow 1$ to $n$
    **do** $tval \leftarrow tval + v_i$
       **while** $tval > V$
         **then** $A(vbegin) \leftarrow i - 1$
              $tval \leftarrow tval - v_{vbegin}$
              $vbegin++$
```

The loop runs at most $O(n)$ times since it sets at least one $A(i)$ every time it runs and never resets it. You can use a similar algorithm for $B(i)$.

The partial credit problem when $a = b$ admits a greedy strategy. A the boxes arrive, keep putting it in the current box as long as current weight is less than W or current value is less than V. The greedy choice property here is that at any point, you have used the minimum number of boxes, which is the right thing to do, since all boxes cost the same amount of money.

Problem 4. Suppose you are given an array $A[1..n]$ that either contains all ones or $3n/4$ ones and $n/4$ zeroes. Your problem is to determine if A contains any zeroes.

(a) Give a lower bound in terms of n on the worst case running time of any deterministic algorithm that solves this problem.

Solution: Any correct deterministic algorithm must check at least $3n/4 + 1$ locations, since otherwise, it might miss a zero even if one exists.

(b) Give a randomized algorithm that runs in $O(1)$ time and gives the correct answer with probability at least $1/2$.

Solution: Pick any three random locations and check if any of those is a 0. If any of those is a 0, return "has zeroes", otherwise you return "does not have zeroes". Obviously, this algorithm takes $O(1)$ time. In addition if there are no 0's in the array, then it can not find any and always returns the correct answer. If there are 0's, then the probability that one probe (check at a random location) finds a 0 is $1/4$. Therefore, the probability that the probe finds 1 is $3/4$. The probability that all three probes find 1's is $(3/4)^3 = 27/64 < 1/2$. Therefore, you will return the correct answer with probability $> 1/2$.

Problem 5. You are given an unsorted array of $n = 2^k$ distinct numbers, where k is a non-negative integer. You want to find the elements of rank $1, 2, 4, 8, ..., 2^{k-1}$. Recall that an element of rank r is the element that has r elements smaller than or equal to itself. Give an algorithm to solve this problem in $\Theta(n)$ time.

Solution: You can solve the problem recursively. First use Select algorithm to find the element of rank 2^{k-1} and partition around this element. You will have divided the array into two approximately equal arrays. We can then recurse in the smaller half of the array to find the element of rank 2^{k-2} and so on. The recurrence is

$$T(n) = T(n/2) + \Theta(n) = \Theta(n)$$

by the master method.

Problem 6. Interval Scheduling

You are given a set of n intervals, $a_1, a_2, ..., a_n$, where each interval has a start time s_i, end time e_i and weight w_i. An interval a_i overlaps with interval a_j if $s_i < e_j$ and $s_j < e_i$. We want to find a set M of non-overlapping intervals such that the sum of the weights of the intervals in M is maximized, that is, $\sum_{a_i \in M} w_i$ is maximized.

(a) Solve the problem when all weights $w_i = 1$, that is, you want to maximize the cardinality of the set M.

Solution: This problem admits a greedy solution. We first sort all intervals by finishing time and then greedily always pick the compatible interval with the earliest finishing time and add it to our set M. The running time of this algorithm is $n \lg n$ since you have to sort the intervals by finishing time. After that you just run through the sorted list and pick the next interval that begins after the end of the previous interval you had picked.

There are a few different ways of arguing the correctness of this greedy choice. We want to prove that this algorithm picks the optimal solution; that is, if this algorithm picks $M = \{i_1, i_2, i_3, ..., i_m\}$ then no other algorithm can pick a set of compatible intervals $L = \{j_1, j_2, ..., j_l\}$ such that $l > m$. WLOG, we sort both sets by finishing time and first prove the following lemma:

Lemma 1 *i_k finishes before or at the same time as j_k for all k.*

Proof. Prove by induction. Base case is obvious since the greedy algorithm first picks the interval that finishes at the earliest time.

For the inductive step, assume that it is true for $k-1$, that is, i_{k-1} ends before or at the same time as j_{k-1}. Now the greedy algorithm will pick as i_k, the interval that starts after i_{k-1} finishes and has the earliest finishing time. Since i_{k-1} finishes before or at the same time as j_{k-1}, the algorithm that is picking L can not pick an interval that ends earlier than i_k since if such an interval existed, the greedy algorithm would also have picked it.

Theorem 2 $|M| \geq |L|$.

Proof. Assume $m < l$ for contradiction. By previous lemma, we know that i_m finishes before or at the same time as j_m. Therefore, if there was another compatible interval that finishes after j_m, that interval would also be compatible with i_m and therefore, the greedy algorithm would also have picked it.

There are also various graph theoretic solutions to this problem. I will let you think about those.

(b) Now solve the problem when the weights are arbitrary.

Solution: This problem no longer admits a greedy strategy. You must solve it using dynamic programming. Again, we sort the intervals by finishing time. For every interval i, we calculate p_i, which is the last interval in this sorted order that is compatible with i, that is, it ends before i begins. We can then define $m(i)$ as the maximum weight we can gather from intervals that end by the time i ends. The final solution we want is simply $m(i_n)$, where i_n is the last interval in the list of intervals sorted by finishing time. We can write the recurrence for $m(i)$ as

$$m(i) = \max\{m(i-1), m(p(i)) + w(i)\}$$

Why is this correct? Well, you either not include i in your set — in this case, the weight you collect at the end of i is the same as the weight you collect by the end of $i-1$, represented by the first term in the recurrence— or you include i in your set — in this case, the latest previous interval you could have included was $p(i)$ and the total weight you get to collect is the weight of i and whatever weight you could have collected by the end of $p(i)$.

The running time is $O(n \lg n)$ again. You must still sort all intervals by finishing time. The only interesting part is the calculation of $p(i)$, but you can calculate $p(i)$ for each interval by doing a binary search in $O(\lg n)$ time, for a total of $O(n \lg n)$ for n intervals.