

CSE 101 Homework 1

Background (Order and Recurrence Relations), correctness proofs, time analysis, and speeding up algorithms with restructuring, preprocessing and data structures.

Due Tuesday, April 19

100 points total = 10 %

Solve each problem. For algorithm problems, if the problem only specifies that you need to give a proof of correctness, then no time analysis is required. If it specifies that you need to give an efficient implementation, then you do not need to give a correctness proof for the basic strategy (just explain why your version actually carries out the strategy). If it says to do both, or doesn't specify what parts you need, you need to give both a proof of correctness and time analysis.

- Order (5 points each)**
1. Is $\log(n^2) \in \Theta(\log n)$? Why or why not?
 2. If Alg1 takes time $\Theta(n^2)$ and produces an output of length $n \log n$, and Alg2 takes time $\Theta(n^2)$, what is the time complexity of an algorithm that runs Alg1 and then runs Alg2 on the output of Alg1? Explain your answer.
 3. Is $n! \in \Theta(n^n)$? Why or why not?
 4. If on a graph with n nodes and m edges, Alg1 takes time $O(nm)$ and Alg2 takes time $O(n^2 \log n)$, when is it better to use Alg1? Explain.

Merging k sorted lists (20 pts) You are given k sorted linked lists, and want to merge them into a single sorted list containing all of the elements in all of the lists. There are n elements total in all of the lists, but the lists are not necessarily of the same size. Give an efficient algorithm for this problem. Be sure to give a correctness proof and proof of the time analysis.

Last larger element, 20 points total Consider the following problem: given an array of integers $A[1..n]$, for each $1 \leq I \leq n$, find the last position $1 \leq J \leq I$ with $A[J] > A[I]$ (or find 0 if no such J exists), and store it as $B[I]$.

Analyze the obvious algorithm, 5 points Here is the most obvious algorithm for this problem: (**LastLargerElement**[**A**[1..n]: array of integers])

1. FOR $I = 1$ to n do:
2. $J \leftarrow I - 1$.
3. While $J > 0$ and $A[J] \leq A[I]$ do $J \leftarrow J - 1$
4. $B[I] \leftarrow J$.

5. Return B

Give a worst-case time analysis, up to Θ , for this algorithm, as a function of n . (Since there may be some inputs on which this algorithm is faster than its worst-case, be sure to provide an example of inputs on which its performance matches the analysis.)

Correctness proof for better strategy, 10 pts Here's a high-level algorithmic strategy for the same problem:

LastLargerElement($A[1..n]$):

1. $B[1] \leftarrow 0$.
2. Initialize S as $\{1\}$.
3. For $I = 2$ TO n do:
4. $J \leftarrow$ the largest element of S .
5. While $J > 0$ and $A[J] \leq A[I]$ do:
6. Delete the largest element of S .
7. IF $S \neq \emptyset$
8. THEN $J \leftarrow$ the largest element of S
9. ELSE $J \leftarrow 0$
10. $B[I] \leftarrow J$.
11. Insert I into S .
12. Return B

For example, say that $A[1..6] = [6, 2, 1, 3, 0, 5]$. $B[1] = 0$. Then S would contain 1. In the loop when $I = 2$, we would compare $A[2]$ to $A[1]$, and immediately set $B[2] = 1$ and add 2 to S . Next, we compare $A[3]$ to $A[2]$, set $B[3] = 2$ and add 3 to S . In the iteration when $I = 4$, we compare $A[4]$ to first $A[3]$ then to $A[2]$, deleting both 4 and 3 from S , but then compare it to $A[1]$ which is larger. We set $B[4] = 1$, and S becomes $\{1, 4\}$. In the next iteration, we set $B[5] = 4$ and add 5 to S . Finally, we compare $A[6] = 5$ to $A[5] = 4$, $A[4] = 3$ and $A[1] = 6$, setting $B[6] = 1$ and returning B .

Prove that this strategy solves the problem. Here's an outline for how such a proof might go: Let $A[1..n]$ be an input array. Say that J is *unblocked* at I if $A[J] > A[K]$ for all $J < K \leq I$. Let S_I be the set of unblocked positions at I .

1. Prove that if J is the last larger element to $A[I+1]$, then $J \in S_I$.
2. Prove that if $1 \leq J_1 < J_2 < \dots < J_k \leq I$ are the elements of S_I , then $A[J_1] > A[J_2] > \dots > A[J_k]$.
3. Prove the following loop invariant: after the loop I , $S = S_I$.
4. Use this to conclude that each $B[I]$ is the position of the last larger element before $A[I]$ (or 0 if none exists).

Efficient versions of algorithm, 5 pts Give an efficient algorithm to find the last larger element based on the given strategy. Specify clearly the data structures and preprocessing used, and give pseudo-code or a clear description of all steps in terms of these data structure operations. Give an informal explanation for why your algorithm follows the given strategy. Give a complete time analysis of your algorithm. Some of your grade will be based on the efficiency of your algorithm.

Maximum bandwidth path, 20 points Consider a directed graph with positive edge weights $w(e) > 0$, intuitively representing the bandwidth of a connection between the endpoints. The *bandwidth* of a path p from u to v is the minimum weight of an edge along the path. The maximum bandwidth path problem is to find, given G , u and v , a path of maximum possible bandwidth from u to v in G . Find, prove correct, and analyze an efficient algorithm for this problem. (Hint: How could you decide whether there is a path in G from u to v of bandwidth at least B ? A good algorithm has time approximately $O((n + m) \log n)$ where the graph has n nodes and m edges, but there is also an $O(n + m)$ algorithm.)

Implementation-20 points Implement a naive $O(n^2)$ time sorting algorithm (such as bubble sort) and heap-sort. You can use heaps from a standard library to implement heap-sort. Plot their performance on random arrays of n integers with values between 1 and n , for $n = 2^6, 2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}$. Plot their performance on a log-log scale. Is heap-sort always better than bubble-sort? Why or why not?