

CSE 101 Homework 2

Spring 2013

Algorithm analysis, correctness proofs, Graph representations, graph search

Due Tuesday, April 16

80 points total

Correctness Proof: 40 points, 2 points per blank An array of distinct integers $A[1..n]$ is k -almost sorted if for each $1 \leq I \leq n$, the I 'th smallest element is in positions $A[1], \dots, A[I + k]$. Here's a high level algorithm to sort an input which is k -almost sorted:

AlmostSorted[A[1..n],k]: B[1..n].

1. Let $S = \{A[1], \dots, A[k]\}$.
2. FOR $J = k + 1$ TO $n + k$ do:
3. IF $J \leq n$ Add $A[J]$ to S .
4. $B[J - k] \leftarrow$ the smallest element of S .
5. Delete the smallest element of S
6. Return B .

Fill in the blanks in the following proof that the above strategy correctly sorts a k -almost sorted list A .

We begin by proving a simple **I** : Let S_j be the set S after the iteration when $J = j$, and let S_k be the set S is initialized to. We will show that $\{B[t] | 1 \leq t \leq j - k\} \cup S_j = \{A[t] | 1 \leq t \leq j\}$ for each $1 \leq j \leq n + k$.

The proof is by induction. For the base case, before the loop begins, i.e., when $j = k$, $S =$ **II** and there are no $1 \leq t \leq k - k$. Thus,

III $\cup S = S = \{A[t] | 1 \leq t \leq k\}$, and the invariant holds.

Assume the invariant holds after the iteration when $J = j$, i.e., that **IV** . We wish to prove that, after the iteration, the invariant still holds, i.e., that **V** .

If $j + 1 \leq n$, then in this iteration we add one element to S , **VI** and delete the element that becomes $B[\text{VII}]$ Thus, $S_{j+1} =$ **VIII** . Then $\{B[t] | 1 \leq t \leq \text{IX} \} \cup S_{j+1} = \{B[t] | 1 \leq t \leq j - k\} \cup S_j \cup \{ \text{X} \} = \{A[t] | 1 \leq t \leq j\} \cup \{ \text{X} \} = \{A[t] | 1 \leq t \leq j + 1\}$.

If $j + 1 > n$, then in this iteration, we only delete $B[j + 1 - k]$ from S . Thus $\{B[t] | 1 \leq t \leq j + 1 - k\} \cup S_{j+1}$ is the same as $\{ \text{XI} \}$ which by the induction hypothesis is $\{A[1], \dots, A[n]\}$.

So in either case, the invariant still holds after the $j + 1$ 'st iteration, so by induction it holds for all iterations.

We now prove by strong induction that each $B[t]$ is the t 'th smallest element of A . Assume that for all $1 \leq t \leq m$, $B[t]$ is the t 'th smallest element of A . Let b be the $m+1$ 'st smallest element of A . We need to prove that **XII**. By the invariant proved above, after the loop when $J = m+1+k$, **XIII**. By the definition of k almost sorted, the $m+1$ 'st largest element of A is in the set **IV**, so b is either in S_{m+k+1} or in $B[1..m+1]$. Since by the strong induction hypothesis, the m smallest elements of A are **XV**, this means that $B[m+1] = b$ or $b \in S_{m+k+1}$. Assume $b \neq B[m+1]$. Since the algorithm defines $B[m+1]$ as the smallest element in S , $B[m+1]$ **XVI** b . On the other hand, $B[m+1]$ is in the array A by the invariant. This **XVII** the definition of the $m+1$ 'st smallest element of A , since b is larger than each of $B[1], \dots, B[\text{XVIII}]$, all of which are in A . From this contradiction, we must have **XIX**, which is what we wanted to prove.

Thus, by strong induction, $B[m]$ is the m 'th smallest element of A for each **XX**, so B is the correctly sorted version of A .

We begin by proving a simple **loop invariant**: Let S_j be the set S after the iteration when $J = j$, and let S_k be the set S is initialized to. We will show that $\{B[t] | 1 \leq t \leq j-k\} \cup S_j = \{A[t] | 1 \leq t \leq j\}$ for each $1 \leq j \leq n+k$.

The proof is by induction. For the base case, before the loop begins, i.e., when $j = k$, $S = \{A[t] | 1 \leq t \leq k\}$ and there are no $1 \leq t \leq k-k$. Thus, $\emptyset \cup S = S = \{A[t] | 1 \leq t \leq k\}$, and the invariant holds.

Assume the invariant holds after the iteration when $J = j$, i.e., that $\{B[t] | 1 \leq t \leq j-k\} \cup S_j = \{A[t] | 1 \leq t \leq j\}$. We wish to prove that, after the iteration, the invariant still holds, i.e., that $\{B[t] | 1 \leq t \leq j+1-k\} \cup S_{j+1} = \{A[t] | 1 \leq t \leq j+1\}$.

If $j+1 \leq n$, then in this iteration we add one element to S , $A[j+1]$ and delete the element that becomes $B[j+1-k]$. Thus, $S_{j+1} = S_j \cup \{A[j+1]\} - \{B[j+1-k]\}$. Then $\{B[t] | 1 \leq t \leq j+1-k\} \cup S_{j+1} = \{B[t] | 1 \leq t \leq j-k\} \cup S_j \cup \{A[j+1]\} = \{A[t] | 1 \leq t \leq j\} \cup \{A[j+1]\} = \{A[t] | 1 \leq t \leq j+1\}$.

If $j+1 > n$, then in this iteration, we only delete $B[j+1-k]$ from S . Thus $\{B[t] | 1 \leq t \leq j+1-k\} \cup S_{j+1}$ is the same as $\{B[t] | 1 \leq t \leq j-k\} \cup S_j$ which by the induction hypothesis is $\{A[1], \dots, A[n]\}$.

So in either case, the invariant still holds after the $j+1$ 'st iteration, so by induction it holds for all iterations.

We now prove by strong induction that each $B[t]$ is the t 'th smallest element of A . Assume that for all $1 \leq t \leq m$, $B[t]$ is the t 'th smallest element of A . Let b be the $m+1$ 'st smallest element of A . We need to prove that $B[m+1] = b$. By the invariant proved above, after the loop when

$J = m+1+k$, $\{B[t] | 1 \leq t \leq m+1\} \cup S_{m+1+k} = \{A[t] | 1 \leq t \leq m+1+k\}$. By the definition of k almost sorted, the $m+1$ 'st largest element of A is in the set $\{A[t] | 1 \leq t \leq m+1+k\}$ so b is either in S_{m+1+k} or in $B[1..m+1]$. Since by the strong induction hypothesis, the m smallest elements of A are $\{B[1], \dots, B[m]\}$, this means that $B[m+1] = b$ or $b \in S_{m+1+k}$. Assume $b \neq B[m+1]$. Since the algorithm defines $B[m+1]$ as the smallest element in S , $B[m+1] < b$. On the other hand, $B[m+1]$ is in the array A by the invariant. This **contradicts** the definition of the $m+1$ 'st smallest element of A , since b is larger than each of $B[1], \dots, B[m+1]$, all of which are in A . From this contradiction, we must have $B[m+1] = b$, which is what we wanted to prove.

Thus, by strong induction, $B[m]$ is the m 'th smallest element of A for each $1 \leq m \leq n$, so B is the correctly sorted version of A .

Triangle detection A *triangle* in an undirected graph $G = (V, E)$ is a triple of distinct nodes u, v, w so that $\{u, v\}, \{v, w\}, \{u, w\}$ are all in E , i.e., there are direct edges in the graph between any two of the three nodes. Give an algorithm that, given an undirected graph in adjacency matrix representation, decides whether it has a triangle. Then give an algorithm for the same problem, when the graph is given in adjacency list format. As always, you need to give an argument that your algorithm is correct (this could be short, in this case), and a time analysis (in terms of the number of nodes $n = |V|$ and the number of edges, $m = |E|$). A good algorithm in either case has time $O(nm + n^2)$.

(30 points, 15 points each representation. For each, 3 points correctness and correctness argument, 5 points for a correct time analysis of the given algorithm, and 7 points for efficiency (no points for efficiency unless you are better than the obvious $O(n^3)$ algorithm, and full credit for matching the $O(nm + n^2)$ bound.)

The obvious method is to look at all triples of nodes and check whether they are all connected. This would be $O(n^3)$ to check. But we can do better for sparse graphs, $m \in o(n^2)$. The high level strategy is, instead of checking for pairs of nodes, check for each edge and each node not on the edge, whether the two endpoints and the additional node form a triangle. In the matrix representation, this would work as in the following pseudo-code.

Triangles[M:[1..n][1..n]: matrix of Booleans]: list of triples of nodes (integers in the range 1..n)

1. $Found \leftarrow False$.
2. FOR $I = 1$ TO $I = n - 2$ do:
3. FOR $J = I + 1$ TO $J = n - 1$ do:

4. IF $M[I, J] = 1$ THEN do:
5. FOR $K = J + 1$ TO $K = n$ do:
6. IF $M[I, K]$ and $M[J, K]$ then $Found \leftarrow True$
7. Return $Found$;

While there are three nested loops, each taking $O(n)$ time, we can do better than $O(n^3)$ in our time analysis. The loop in lines 5 and 6 is only performed if $M[I, J]$, i.e., at most once per edge $\{I, J\}$. Thus, it is really an $O(n)$ loop executed m times, so the total time spent on lines 5 and 6 is $O(mn)$. The time spent on the rest of the algorithm is just $O(n^2)$ from the two nested loops, with the $O(1)$ time to check whether $M[I][J]$ in line 4. Thus the total time is $O(mn + n^2)$. Since $m \in O(n^2)$, this is always $O(n^3)$ as above, but it is faster if m is small.

If the graph is in adjacency list format, we can create the adjacency matrix in $O(n^2)$ time, and run the above algorithm. Thus, the total time will be $O(mn + n^2)$.

Odd length paths Consider the problem of, given a directed graph G , and a node u , list all the nodes v so that there is a path using an odd number of edges from u to v . (Note that such a path might have to have a cycle, and need not be the shortest path.) Give an efficient algorithm (with correctness proof and time analysis) for this problem. My algorithm runs in time $O(n + m)$. Hint: consider replacing each node with two nodes, one representing arrival via an even path, and the other via an odd path. (30 points, 10 for correctness and the correctness proof, 10 for a correct analysis of your algorithm, and 10 for efficiency, with full points for efficiency for the $O(n + m)$ time above.)

Following the above hint, given the graph $G = (V, E)$, we construct a graph $G' = (V', E')$ where for every node $v \in V$ we have two nodes $v^0, v^1 \in V'$ and for every edge $e = (u, v) \in E$, we add the two edges (u^0, v^1) and (u^1, v^0) to E' .

To create G' from the adjacency list for G , we just need to define an array of adjacency lists with $2n$ entries, and copy over each adjacency list twice, once appending 1 to the name of every node, once appending zero. This will take $O(n + m)$ time.

Then we run a graph search algorithm on G' , from s^0 . I claim that a node v^1 is reachable from s^0 in G' if and only if there is an odd length path from s to v in G . If p' is a path from s^0 to v^1 in G' , then the edges of the path are of the form $(s^0, u_1^1), (u_1^1, u_2^0) \dots (u_k^0, v^1)$, where $(s, u_1), (u_1, u_2) \dots (u_k, v)$ are edges in E . Since the bit in the superscript alternates between 0 and 1, and the first bit is 0, and the last 1, the path p' must have an odd

number of edges. Thus, the path $p = (s, u_1)(u_1, u_2) \dots (u_k, v)$ is an odd length path from s to v .

In the converse direction, if $p = (s, u_1)(u_1, u_2) \dots (u_k, v)$ is an odd length path from s to v in G , then $p' = (s^0, u_1^1), (u_1^1, u_2^0) \dots (u_k^0, v^1)$, is a path from s^0 to v^1 in G' .

So there is an odd length path from s to v in G if and only if there is any path from s^0 to v^1 in G' . Thus, if we do a graph search from s^0 and report v for all of the reachable nodes of the form v^1 , that will be a complete list of all the nodes in G that are reachable from s via odd length paths. Since the number of nodes in G' is $n' = 2n$ and the number of edges in G' is $m' = 2m$, the graph search (either DFS or BFS) will take time $O(n' + m') = O(n + m)$. Creating G' also took $O(n + m)$ time, so the total time is $O(n + m)$.