

CSE 101
Practice Midterm: Winter, 2005
Answer Key

Answer all questions. Give informal (at least) proofs for all answers. Grading will be on completeness and logical correctness, and if applicable, efficiency, as well as correctness.

Analyzing loops-10pts Consider the following algorithm, that given two sequences of symbols $w_1..w_n$ and $v_1..v_n$ with $n \geq m$, returns the largest suffix of w that is also a prefix of v , i.e., the biggest I so that $v_1..v_I = w_{n-I+1}..w_n$.

PrefixSuffixMatch($w_1..w_n, v_1..v_n$)

1. $Best \leftarrow 0$
2. FOR $I = 1$ to n do:
3. $K \leftarrow 1; J \leftarrow n - I + 1$
4. While $v_K = w_J$ and $K \leq I$ do $K ++; J ++$
5. IF $K = I + 1$ THEN $Best \leftarrow I$
6. Return $Best$

Give a worst-case time analysis, up to Θ , for this algorithm, as a function of n .

When the outermost FOR loop has $I = i$, the inner while loop executes at most i times, since K is incremented each loop and it stops when K reaches $I + 1$. Since $i \leq n$, and all other commands in the FOR loop are constant time, the inside of the FOR loop is $O(n)$. The FOR loop is executed n times, giving a total of $O(n^2)$, and the rest of the algorithm is constant time, so the overall time is $O(n^2)$.

When run on two strings that are all the same symbol, say both n a's, the first condition in the while loop is always met, so the while loop executes exactly i times. Since there are $n/2$ values of i greater than $n/2$, this gives a total time of $\Omega(n^2)$, so the algorithm's time is $\Theta(n^2)$.

Correctness proofs You are given an array $A[1..n]$ of n integers in the range $1..k$. You want to find the smallest consecutive subarray, $A[I..J]$ that contains each of the k elements, if such a subarray exists.

Here's a high-level algorithmic strategy for this problem:

Small Consecutive Subarray Containing All Values ($A[1..n], k$)

1. Define $PrevOcc[1..k]$ as an array of integers. For each $j \in \{1..k\}$, initialize $PrevOcc(j)$ to $-n$. Initialize $ShortestSubarray$ to (NIL, NIL) , and $BestLength$ to $n + 1$.
2. For $J = 1$ TO n do:
3. $PrevOcc(A[J]) \leftarrow J$;
4. Let T be the $\min_K PrevOcc(K)$;
5. If $J - T + 1 < BestLength$ then $ShortestSubarray \leftarrow (T, J)$; $BestLength \leftarrow J - T + 1$.
6. If $BestLength \leq n$ return $ShortestSubarray$, else return "Not all present".

For example, say $k = 4$ and the input array were $A[1..12] = 2, 3, 4, 3, 2, 3, 2, 1, 3, 3, 4$. Then the values of the $PrevOcc$ would evolve as:

J	PrevOcc(1)	PrevOcc(2)	PrevOcc(3)	PrevOcc(4)	T	J-T+1
1	-12	1	-12	-12	-12	14
2	-12	1	2	-12	-12	15
3	-12	1	2	3	-12	16
4	-12	1	4	3	-12	17
5	-12	5	4	3	-12	18
6	-12	5	6	3	-12	19
7	-12	7	6	3	-12	20

8	8	7	6	3	3	6
9	8	7	9	3	3	7
10	8	7	10	3	3	8
11	8	7	10	11	7	5

giving best subarray $A[7..11]$.

Below, there's a proof that this algorithm works with some gaps missing. The gaps are labelled with Roman numerals. For each gap, supply the missing phrase. **Answers are in bold print**

Proof: We start by proving the following **loop invariant**: For each iteration j of the loop, and for each $V \in \{1..k\}$, $PrevOcc(V)$ is the last position before j where V occurs in A , or is $-n$ if no such position exists. More precisely, if $PrevOcc(V) \neq -n$, $A[PrevOcc[V]] = \mathbf{V}$, and, for each j' with $\mathbf{PrevOcc}[\mathbf{V}] < j' \leq \mathbf{j}$, $A[j'] \neq V$; and if $PrevOcc(V) = -n$, then for every $1 \leq j' \leq j$, $A[j'] \neq V$.

In the base case, $j = 0$, this statement is true, since every $PrevOcc(V) = -\mathbf{n}$ and there are no $1 \leq j' \leq 0$.

For the induction step, assume that the invariant holds after the loop when $J = j$, and we will prove that it is still true after loop $J = j+1$. For each $V \neq A[j+1]$, $PrevOcc(V)$ does not change through the loop. If $PrevOcc(V) = -n$, then by the invariant for j , $V \neq A[j']$ for any $1 \leq j' \leq j$. Then since also $V \neq A[j+1]$, $V \neq A[j']$ for any $1 \leq j' \leq j+1$, as required. If $PrevOcc(V) \neq -n$, then $A[PrevOcc(V)] = \mathbf{V}$ by the invariant for j . Also by the invariant for j , $V \neq A[j']$ for any $\mathbf{PrevOcc}(\mathbf{V}) < \mathbf{j}' \leq \mathbf{j}$, and since $A[j+1] \neq V$, $V \neq A[j']$ for any $\mathbf{PrevOcc}(\mathbf{V}) < \mathbf{j}' \leq \mathbf{j}+1$ as required for the invariant at $j+1$.

For $V = A[j+1]$, we set $PrevOcc(V)$ to $\mathbf{j}+1$, and the invariant holds, since $A[PrevOcc(V)] = A[\mathbf{j}+1] = \mathbf{V}$, and there are no j' with $PrevOcc(V) = j+1 < j' \leq j+1$.

Thus, by induction, the invariant holds for all j , $1 \leq j \leq n$.

At each time j , let $t = \min_V PrevOcc(V)$. If $t = -n$, there is some V so that $PrevOcc(V) = -n$. Then by the invariant, $A[j'] \neq V$ for any $1 \leq j' \leq j$, so there is no subarray ending at j that contains all k elements. If $t \neq -n$, then we claim that $A[t..j]$ is the smallest such subarray. First, we need to show that it is such a subarray, that is, for each V , we need to show that there is a j' with $\mathbf{t} \leq \mathbf{j}' \leq \mathbf{j}$ so that $A[j'] = V$. Let $\mathbf{j}' = \mathbf{PrevOcc}[\mathbf{V}]$, since by the invariant, $A[\mathbf{PrevOcc}[\mathbf{V}]] = V$. Since $t = \min_V PrevOcc(V)$, $\mathbf{t} \leq PrevOcc(V) \leq j$. Second, we need to show that there is no smaller subarray $A[t', j]$ with $t' > t$, containing each V . Since $t = \min_V PrevOcc(V)$, we can choose V so that $t = \mathbf{PrevOcc}[\mathbf{V}]$. Then by the invariant, for each j' with $t < t' \leq j' \leq i$, $A[\mathbf{j}'] \neq \mathbf{V}$. Thus, V is not in such an interval, so there is no smaller interval containing all values.

Thus, our algorithm computes, for each j , the smallest subarray of the form (t, j) that contains all V . It returns the **smallest** such interval, which must be the smallest subarray of A containing each V .

Data structures and efficient versions of algorithms 10 pts: For the problem above, give an efficient algorithm to compute the minimum length subarray that contains all $1 \leq J \leq k$. Base it on the strategy given, but 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 a time analysis of your algorithm, in terms of both n and k . Some of your grade will be based on the efficiency of your algorithm, as well as correctness.

We need to keep track of the set of $PrevOcc(V)$ for $V = 1, ..k$ and in each iteration, find the smallest element of this set. This argues for using a min-heap. However, when we set $PrevOcc(A[I])$ to I , we need to be able to find the element corresponding to $A[I]$, delete it, and insert a replacement. This means that we should use a NamedHeap from class. The NamedHeap will always have exactly k elements, one for each $1 \leq V \leq k$. So heap operations will take time $O(\log k)$ for insert and deletebyname, and $O(1)$ for FindMinimum. In terms of these operations, the algorithm becomes: Small Consecutive Subarray Containing All Values ($A[1..n]$, k)

1. Initialize a named min-heap with items named $1..k$, each in the heap with value $-n$. Initialize *ShortestSubarray* to (NIL, NIL) , and *BestLength* to $n+1$.
2. For $J=1$ TO n do:
3. DeleteByName($A[J]$);

4. Insert an item with value J and name $A[J]$
5. $T \leftarrow FindMin$
6. If $J - T + 1 < BestLength$ then $ShortestSubarray \leftarrow (T, J)$; $BestLength \leftarrow J - T + 1$.
7. If $BestLength \leq n$ return $ShortestSubarray$, else return "Not all present".

Initialization takes time $O(k) \in O(n)$ since $k \leq n$. Since each heap operation takes time $O(\log k)$, the inside of the FOR loop takes $O(\log k)$ time, so the total time for the for loop is $O(n \log k)$, which dominates the rest of the algorithm.

Divide-and-Conquer Recurrence: 10 points Consider the following recursive algorithm. Its input is an array of positive integers. $A[1..n]$ The goal is to find the maximum possible sum of a sub-sequence $A[I_1] + A[I_2], \dots + A[I_k]$ with $1 \leq I_1 < I_2 < I_3 < \dots < I_k \leq n$ so that no two elements are consecutive, i.e., $I_{j+1} > I_j + 1$ for each $1 \leq j < k$. (Note: here k is any length, not an input parameter.)

MaxNonConsSum[A[1..n]]

1. IF $n = 0$ return 0.
2. IF $n = 1$ return $A[1]$.
3. IF $n = 2$ return $\max(A[1], A[2])$.
4. $Case1 \leftarrow MaxNonConsSum(A[1..n/2 - 2]) + A[n/2] + MaxNonConsSum(A[n/2 + 2..n])$ {If we include $A[n/2]$ we can't include $A[n/2 - 1]$ or $A[n/2 + 1]$.}
5. $Case2 \leftarrow MaxConsSum(A[1..n/2 - 1]) + MaxNonConsSum(A[n/2 + 2..n])$ {If we don't include $A[n/2]$ we can include any of the others.}
6. Return $\max(Case1, Case2)$

Give a recurrence for the time $T(n)$ taken by the above algorithm. Use the recurrence to give a time analysis up to order. Be sure to justify all of your answers by referring to the algorithm description.

The algorithm makes 4 recursive calls, two each in lines 4 and 5. The size of the input in these recursive calls are $n/2 - 2$ and $n/2 - 1$ respectively, both less than $n/2$. The rest of the algorithm is constant time.

Thus, $T(n) \leq 4T(n/2) + O(1)$, so using the Main Recurrence Theorem with $a = 4, b = 2$ and $k = 0$, since $a = 4 > l = b^k$, we are in the bottom-heavy case, and $T(n) \in O(n^{\log_b a}) = O(n^{\log_2 4}) = O(n^2)$.