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 $\Theta$, for this algorithm, as a function of $n$.

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:

<table>
<thead>
<tr>
<th>$J$</th>
<th>$PrevOcc(1)$</th>
<th>$PrevOcc(2)$</th>
<th>$PrevOcc(3)$</th>
<th>$PrevOcc(4)$</th>
<th>$T$</th>
<th>$J-T+1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-12</td>
<td>1</td>
<td>-12</td>
<td>-12</td>
<td>-12</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>-12</td>
<td>1</td>
<td>2</td>
<td>-12</td>
<td>-12</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>-12</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-12</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>-12</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>-12</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>-12</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>-12</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>-12</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>-12</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>-12</td>
<td>7</td>
<td>6</td>
<td>3</td>
<td>-12</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>7</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>


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.
Proof: We start by proving the following

$I$ : For each iteration $j$ of the loop, and for each $V \in \{1..k\}$, $\text{PrevOcc}(V)$ is the last position before $j$ where $V$ occurs in $A$, or is $-n$ if no such position exists. More precisely, if $\text{PrevOcc}(V) \neq -n$, $A[\text{PrevOcc}(V)] = \iff$, and, for each $j'$ with $\iff < j' \leq \iff V V V$, and if $\text{PrevOcc}(V) = -n$, then for every $1 \leq j' \leq j$, $A[j'] V V V$.

In the base case, $j = 0$, this statement is true, since every $\text{PrevOcc}(V) = \iff V V V$ 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 $V \neq A[j + 1]$, $\text{PrevOcc}(V)$ does not change through the loop. If $\text{PrevOcc}(V) = -n$, 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 $\iff$, as required. If $\text{PrevOcc}(V) \neq -n$, then $A[\text{PrevOcc}(V)] = \iff$ by the invariant for $j$. Also by the invariant for $j$, $V \neq A[j']$ for any $\iff$, and since $A[j + 1] \neq V$, $V \neq A[j']$ for any $\iff$, as required for the invariant at $j + 1$. For $V = A[j + 1]$, we set $\text{PrevOcc}(V)$ to $\iff$, and the invariant holds, since $A[\text{PrevOcc}(V)] = A[\iff] = \iff$, and there are no $j'$ with $\text{PrevOcc}(V) = j + 1 < j' \leq j + 1$.

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

At each time $j$, let $t = \text{min}_V \text{PrevOcc}(V)$. If $t = -n$, there is some $V$ so that $\text{PrevOcc}(V) = -n$. 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 $A[t,i]$ 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 $\iff$ so that $A[j'] = V$. Let $j' = \iff$, and since $A[j',i]$ is the smallest such subarray. First, we need to show that there is no smaller subarray $A[t',j]$ with $t' > t$, containing each $V$. Since $t = \text{min}_V \text{PrevOcc}(V)$, $\iff \leq \text{PrevOcc}(V) \leq j$. Second, we need to show that there is no smaller subarray $A[t',j]$ with $t' < t$, containing each $V$. Since $t = \text{min}_V \text{PrevOcc}(V)$, we can choose $V$ so that $t = \iff$. Then by the invariant, for each $j'$ with $t < j' \leq i$, $\iff$. 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 $\iff$ 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.

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 $A[I_1] + A[I_2] + \ldots + A[I_k]$ with $1 \leq I_1 < I_2 < I_3 < I_k \leq n$ so that no two elements are consecutive, i.e., $I_{j+1} > I_j + 1$ for each $1 \leq j \leq k$. (Note: here $k$ is any length, not an input parameter.)

MaxNonConsSum$[A[1..n]]$

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

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.