## Algorithms and Data Structures

**COMP 251 SEC 001**

18:30 pm December 13, 2017

<table>
<thead>
<tr>
<th>EXAMINER:</th>
<th>Hamed Hatami</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSOC. EXAMINER:</td>
<td></td>
</tr>
</tbody>
</table>

### INSTRUCTIONS

- **CLOSED BOOK** ☑
- **OPEN BOOK** ☐
- **SINGLE-SIDED** ☑
- **PRINTED ON BOTH SIDES OF THE PAGE** ☐

- **MULTIPLE CHOICE** ☐
  
  Note: The Examination Security Monitor Program detects pairs of students with unusually similar answer patterns on multiple-choice exams. Data generated by this program can be used as admissible evidence, either to initiate or corroborate an investigation or a charge of cheating under Section 16 of the Code of Student Conduct and Disciplinary Procedures.

- **ANSWER IN BOOKLET** ☐
- **EXTRA BOOKLETS PERMITTED**: YES ☑ NO ☐
- **ANSWER ON EXAM** ☐

- **SHOULD THE EXAM BE**: RETURNED ☑ KEPT BY STUDENT ☐

### CRIB SHEETS:

- **NOT PERMITTED** ☑
- **PERMITTED** ☐
  
  e.g. one 8 1/2X11 handwritten double-sided sheet

### DICTIONARIES:

- **TRANSLATION ONLY** ☐
- **REGULAR** ☑
- **NONE** ☐

### CALCULATORS:

- **NOT PERMITTED** ☑
- **PERMITTED (Non-Programmable)** ☐

### ANY SPECIAL INSTRUCTIONS:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>/10</td>
<td>/10</td>
<td>/15</td>
<td>/15</td>
<td>/15</td>
<td>/20</td>
<td>/100</td>
<td></td>
</tr>
</tbody>
</table>
1. True or False? (Prove or Disprove) \( T(n) = O(2^n) \) if \( T(n) = T(n/2) + 2^n \), and \( T(1) = 1 \). You may assume that \( n = 2^m \) for a positive integer \( m \).

\textbf{Solution:} True. Using the recursive formula, we have
\[
T(2^m) = T(2^{m-1}) + 2^{2^m} = T(2^{m-2}) + 2^{2^{m-1}} + 2^{2^m} = T(2^{m-3}) + 2^{2^{m-2}} + 2^{2^{m-1}} + 2^{2^m} = 1 + 2^1 + 2^2 + \ldots + 2^{2^m}.
\]

Recalling \( n = 2^m \), this can be obviously bounded from above as
\[
1 + 2^1 + 2^2 + \ldots + 2^{2^m} \leq 2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1 \leq 2 \times 2^n = O(2^n).
\]
2. Explain how binary search works, and analyze its running time by writing a recursive formula of the form 

\[ T(n) = aT(n/b) + O(n^d) \]

and applying master theorem. You can assume that \( n = 2^m \) for a positive integer \( m \).

**Solution:** In the binary search problem, we have an array of sorted numbers, and we want to find an element \( k \) in the array. We recursively divide the array into two equal size parts, compare \( k \) with the element that separates those two parts to decide in which part to look for \( k \). In other words if \( k \) is less than the middle element we look at the part in the left, and otherwise we look into the right part. Since we only consider one of left or right, the recursion is 

\[ T(n) = 1 \times T(n/2) + O(1) \]

where \( O(1) \) is for comparing \( k \) with the middle element. Hence to apply the master theorem, we notice that \( a = 1 \), \( b = 2 \), and \( d = 0 \), and since \( \log_b a = 0 = d \), the running time is \( O(n^d \log n) = O(\log n) \).
3. We want to choose the smallest number of intervals among given $n$ intervals $[s_1, f_1], \ldots, [s_n, f_n]$ such that their union covers all of the interval $[0, m]$. Design and analyze a fast algorithm that given integers $m, s_1, \ldots, s_n, f_1, \ldots, f_m$ finds this number.

**Solution:** The idea is to cover the interval $[0, m]$ from left to right, and each time pick an interval that covers the largest possible interval.

---

**Algorithm 1 Solution to Problem**

1. Set $s := 0$
2. **while** $s \leq m$ **do**
3.   **if** there are intervals with $s_i \leq s \leq f_i$ **then**
4.       Pick such an interval with largest finishing point $f_i$, and set $s = f_i$.
5.   **else**
6.       Output “No solution” and terminate

This can be implemented in $O(n^2)$, as $s$ gets updated at most $n$ times and each time we have to find an interval with largest $f_i$ such that $s_i \leq s \leq f_i$, requiring another multiplicative factor of $O(n)$.

To see that this algorithm is correct, among all the optimal solution consider the one that departs from our algorithm at the latest step. Consider the first interval $[s_i, f_i]$ that is picked by our algorithm that is not in this optimal solution. This means that the algorithm uses $[s_i, f_i]$ to cover the point $s$ with $s_i \leq s \leq f_i$. The optimal solution must cover $s$ with a different interval $[s_j, f_j]$ with $s_j \leq s \leq f_j$. But by the nature of the algorithm $f_i \geq f_j$, and thus we can remove $[s_j, f_j]$ from the optimal solution and replace it with $[s_i, f_i]$. This is still a valid cover as the points $[0, s]$ were already covered by the algorithm (and the optimal solution that matched the algorithm up to that point), and $[s_i, f_i]$ covers also all the points in $[s, f_j]$ since $f_i \geq f_j$. This altered optimal solution matches our algorithm in one more step which contradicts the assumption that our initial optimal solution departed from the algorithm at the latest possible point.
4. We are given a number \( n \), and integer deadlines \( d_1, \ldots, d_n \) of \( n \) unit-length tasks, such that each deadline \( d_i \) satisfies \( 1 \leq d_i \leq n \). We are also given nonnegative penalties \( w_1, \ldots, w_n \). We have a single processor and each task takes 1 (continuous) unit of time on the processor. If the \( i \)-th task does not finish before its deadline \( d_i \) we incur a penalty of \( w_i \). We want to minimize the total penalty incurred for missed deadlines. Prove that the following algorithm finds the optimal schedule: Consider the \( n \)-th time slots starting at times 0, 1, \ldots, \( n-1 \). Sort the jobs in decreasing order of penalties so that \( w_1 \geq w_2 \geq \ldots \geq w_n \). For \( j = 1, \ldots, n \) if there is a time slot available before \( d_j \), assign the \( j \)-th to latest available time slot before time \( d_j \), otherwise assign the \( j \)-th job to the latest available time slot.

**Solution:** Consider an optimal solution \( S \), and let \( r \) be the smallest index such that the \( r \)-th task is assigned to a different time slot by \( S \) compared to the algorithm. Suppose that the \( r \)-th task is assigned to time slot \( s_r \) in \( S \) and \( a_r \) in the algorithm.

Case 1: If \( a_r \geq d_r \), then it means that at the time that the algorithm assigned \( a_r \), there was no time slot available before \( d_r \). Since both the algorithm and the optimal solution match before this time, it means that \( s_r \) misses the deadline too: \( s_r \geq d_r \). Now in \( S \) we can swap \( s_r \) with the job that is assigned to the time slot \( a_r \) (if there is no such job, just move \( s_r \) to that time slot). Since \( a_r \) was the latest available time slot, this swap will not cause \( S \) to lose any new deadlines, and this altered \( S \) matches our algorithm for one more step.

Case 2: Now consider the case \( a_r < d_r \), in which case our algorithm is gaining a profit of \( w_r \) from this task. If this time slot is empty in \( S \), then we can simply move the \( r \)-th job to the time slot \( a_r \) in \( S \) without ending up with any new penalties, and making \( S \) to follow our algorithm for one more step. Now consider the more interesting case where the time slot \( a_r \) is occupied by some other task \( t \) in \( S \). By the nature our algorithm we know that \( w_t \leq w_r \), as the optimal and the algorithm’s solution matched for the first \( r \) tasks. Now let us switch the time slots of the \( r \)-th and \( t \)-th tasks in \( S \). Either the \( r \)-th task was assigned to a time slot earlier than \( a_r \) in which case this swap cannot increase the penalty (the \( t \)-th job is moving to sometime earlier, and the \( r \)-th task is still meeting its deadline), or the \( r \)-th task was assigned to sometime after its deadline (since \( a_r \) was the latest available slot before the deadline). In this latter case, after the swap we gained a profit of \( w_r \) by meeting the deadline of the \( r \)-th job (which was previously missed), and even if we miss the deadline of the \( t \)-th job, the total amount of penalties will decrease: \( w_r - w_t \geq 0 \). Hence again we altered \( S \) to an optimal solution that matches our algorithm for one more step.

We can continue in this manner until the two solutions are identical.
5. Design and analyze an algorithm that given a sequence of numbers $a_1, \ldots, a_n$, finds the length of the largest increasing subsequence. For example the largest increasing subsequence of $(8, 4, 5, 2, 7)$ is $(4, 5, 7)$, and thus in this case the answer is 3.

**Solution:** Let $D[i]$ be the length of the longest increasing subsequence that ends at $i$. We have

$$D[i] = 1 + \max_{j: j < i, a_j < a_i} D[j],$$

where the maximum is defined to be 0 if no $j$ satisfies those conditions.

The optimal solution is the maximum of $D[1], \ldots, D[n]$. 
6. We are given as input a directed graph $G$ without any directed cycles, and two specific vertices $s$ and $t$ in $G$. Design and analyze an efficient algorithm that returns the number of directed paths from $s$ to $t$.

Solution: Since the graph $G$ is acyclic we can use DFS to find a topological sorting of the vertices $v_1, \ldots, v_n$ such that all the edges are directed from vertices from smaller indexes to larger indexes. Let $s = v_{i_0}$. Now for every $i \in \{1, \ldots, n\}$, let $D[i]$ be the number of directed paths from $s$ to $v_i$. We have $D[i_0] = 1$, and $D[i] = 0$ for all $i < i_0$, and for $i > i_0$, we have

$$D[i] = \sum_{j: v_j v_i \in E} D[j].$$

The solution is now $D[j_0]$ where $t = v_{j_0}$. 

7. We are given a directed graph \( G = (V, E) \), and \( n \) subsets \( S_1, \ldots, S_n \subseteq V \) (these sets are not necessarily disjoint). We want to compute the \( n \times n \) matrix \( M \) such that \( M_{ij} \) is the number of edges \((u, v)\) with \( u \in S_i \) and \( v \in S_j \). Design an algorithm based on matrix multiplication that computes \( M \) in time \( O(n^{\log_2 7}) \).

**Solution:** Label the vertices of \( G \) as \( v_1, \ldots, v_n \), and let \( A \) be the adjacency matrix of \( G \) using this order. That is \( A_{ij} = 1 \) if \( v_i v_j \) is an edge.

Let \( X \) be the \( n \times n \) matrix such that \( X_{ij} = 1 \) if \( v_i \in S_j \), and otherwise \( X_{ij} = 0 \). Let \( X^T \) denote the transpose of \( X \). Note that

\[
[X^TAX]_{ij} = \sum_{t} \sum_{r} X^T_{i,t} A_{t,r} X_{r,j} = \sum_{t} \sum_{r} X_{t,i} A_{t,r} X_{r,j}.
\]

Investigating the term in the sum, we see \( X_{t,i} A_{t,r} X_{r,j} = 1 \) if \( v_t \in S_i, v_r \in S_j, v_t v_r \in E \), and \( X_{t,i} A_{t,r} X_{r,j} = 0 \) otherwise. Hence indeed the \( ij \)-th entry of \( M = X^TAX \) is the number of edges \((u, v)\) with \( u \in S_i \) and \( v \in S_j \).

Using Strassen's algorithm we can compute \( X^TAX \) in \( O(n^{\log_2 7}) \) by two matrix multiplications, each requiring \( O(n^{\log_2 7}) \).