Solutions

1. (10%) Use the definition of “big-O” to prove that \( 12n^5 + 0.5n^4 \log(n) \) is \( O(n^5) \).

   **Solution:** \( 12n^5 + 0.5n^4 \log(n) \leq 12n^5 + 0.5n^5 \quad \forall n \geq 2. \) Therefore \( 12n^5 + 0.5n^4 \leq 12.5n^5 \quad \forall n \geq 2. \) Take \( c = 12.5 \) and \( n_0 = 2 \) and we have the proof according to the definition of “big-O”.

2. (10%) Use the definition of “big-O” to prove that \( n^2 \log(n) \) is not \( O(n^2) \).

   **Solution:** Suppose \( n^2 \log(n) \) is \( O(n^2) \). Then \( \exists c > 0 \) and \( n_0 \geq 0 \) s.t. \( n^2 \log(n) \leq c \cdot n^2 \ \forall n \geq n_0. \) Dividing both sides by \( n^2 \) we get \( \log(n) \leq c \ \forall n \geq n_0. \) Take \( m = \max\{n_0 + 1, 2^{c+1}\}. \) Clearly \( m > n_0 \) and \( \log(m) \geq \log(2^{c+1}) > c. \) We get a contradiction and therefore \( n^2 \log(n) \) is not \( O(n^2) \).

3. (10%) Let \( f(n), g(n), \) and \( h(n) \) be non-negative functions. Let \( f(n) \) be \( O(h(n)) \) and \( g(n) \) be \( O(h(n)) \). Use the definition of “big Oh” to prove that \( f(n)g(n) \) is \( O((h(n))^2) \).

   **Solution:** By definition of ‘big-O’, \( \exists c_f, c_g, n_f, n_g \) s.t. \( f(n) \leq c_f \cdot h(n) \ \forall n \geq n_f, \) and \( g(n) \leq c_g \cdot h(n) \ \forall n \geq n_g. \) Combining these two inequalities, we get:

   \[
   f(n)g(n) \leq c_f c_g \cdot h(n)h(n) \quad \forall n \geq \max\{n_f, n_g\}.
   \]

   Therefore take \( c = c_f c_g \) and \( n = \max\{n_f, n_g\} \) and we have the desired result.

4. (20%) Order functions

   \( 2.2^n, \log(n^{10}), 2^{2012}, 25n \cdot \log(n), 1.1^n, 2n^{5.5}, 4 \cdot \log(n), 2^{10}, n^{1.02}, 5n^5, 76n, 8n^5 + 5n^2 \)

   by their asymptotic growth rate, in non-decreasing order. Indicate by circling those functions that are “big-Theta” of each other.

   **Solution:** Here I underlined functions that are “big-Theta” of each other: \( 2^{2012}, 2^{10}, \log(n^{10}), 4 \cdot \log(n), 76n, 25n \cdot \log(n), n^{1.02}, 5n^5, 8n^5 + 5n^2, 2n^{5.5}, 1.1^n, 2.2^n. \)

   Most common mistakes:

   - Some said that \( 1.1^n \) and \( 2.2^n \) are ‘big-O’ of each other. It is easy to see that \( 2.2^n = (1.1 \cdot 2)^n = 1.1^n \cdot 2^n. \) Thus \( 2.2^n \) grows much faster than \( 1.1^n. \) Or, if one wants a formal proof, argue by contradiction. Suppose \( 2.2^n \) is \( O(1.1^n). \) Then \( \exists c, n_0 > 0 \) s.t. \( 2.2^n \leq c \cdot 1.1^n \ \forall n \geq n_0. \) Then \( 2^n \cdot 1.1^n \leq c \cdot 1.1^n \ \forall n \geq n_0. \) Dividing both sides by \( 1.1^n, \) we get \( 2^n \leq c \ \forall n \geq n_0. \) This is obviously false, take \( m = \max\{n_0, c\}. \)

   - Some said that \( 25n \log(n) \) grows faster than \( n^{1.02}. \) Any polynomial function with degree larger than one grows faster than a logarithmic function. Let’s prove this. As always, argue by contradiction. Let \( d > 1 \) and suppose \( n^d \) is \( O(n \log(n)). \) Then \( \exists n_0, c > 0 \) s.t. \( n^d \leq c \cdot n \log(n) \) for all \( n \geq n_0. \) Dividing both sides by \( n \) we get \( n^{d-1} \leq c \cdot \log(n) \) for all \( n \geq n_0. \) Since \( d > 1 \) we have that \( d - 1 > 0. \) Let \( p = d - 1. \) We get that \( n^p \leq c \log(n) \) for all \( n \geq n_0. \) Take \( m = \max\{n_0, (2^{n/c})^{1/p}\}. \) Obviously \( m \geq n_0 \) and also \( m^p = 2^{n/c} > c \log(m) = 1. \) The last inequality holds because 2 to any positive power is
larger than 1, and \( p/c > 0 \). Obviously I did not expect anyone to come up with this proof, but I did mention in class that \( \log(n) \) grows slower than any polynomial. If you were not sure, you could ask me or find the answer on the web.

5. (20%) Give the best asymptotic (“big-Oh”) characterization of the best and worst case time complexity of the algorithm \( \text{Count}(A, B, n) \). Explain how you computed the complexity.

<table>
<thead>
<tr>
<th>Algorithm Count(A,B,n)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Arrays ( A ) and ( B ) of size ( n ).</td>
</tr>
<tr>
<td>( A, B ) store positive integers.</td>
</tr>
<tr>
<td>( i \leftarrow 0 )</td>
</tr>
<tr>
<td>( \text{sum} \leftarrow 0 )</td>
</tr>
<tr>
<td><strong>while</strong> ( i &lt; n ) <strong>do</strong></td>
</tr>
<tr>
<td>( \text{if } A[i] &lt; n \text{ then} )</td>
</tr>
<tr>
<td><strong>for</strong> ( j \leftarrow 0 \text{ to } A[i] \text{ do} )</td>
</tr>
<tr>
<td>( \text{sum} \leftarrow \text{sum} + B[j] )</td>
</tr>
<tr>
<td>( i \leftarrow i+1 )</td>
</tr>
<tr>
<td><strong>return</strong> ( \text{sum} )</td>
</tr>
</tbody>
</table>

**Solution:** In the pseudo-code above, the first column shows primitive operations in the worst case, the second columns primitive operations in the best case. The worst case is when each \( A[i] = n \), the best case is when each \( A[i] > n \). Summing up the primitive operations, in the worst case we get \( O(n^2) \) time complexity and in the best case \( O(n) \) time complexity. Some students said that the best case is when arrays \( A \) and \( B \) have size 0. This does not work. The best case is still a function of the array size \( n \), you cannot fix it to 0. Best case is about analyzing the time when the composition (i.e. what is inside) of the arrays is the most favorable.

6. (30%) Let \( A \) be an array storing integers. Write in pseudocode an algorithm to find the subarray with the largest sum. That is your algorithm takes as input an array \( A \), its size \( n \), and returns two array indexes \( i \) and \( j \), \( i \leq j \), such that the sum \( A[i] + A[i+1] + \ldots + A[j-1] + A[j] \) is as large as possible. For example, if \( A = \{-1, 5, -3, 7, -2\} \), your algorithm should return 1 and 3. Compute the time complexity of your algorithm in the worst case. Explain how you computed complexity.
Solution:

**Algorithm** FindLargestSum\(A, n)\)

**Input:** Array \(A\) of size \(n\).

**Output:** \(i, j\) array subindexes between which the sum is the largest.

\[
largestSum \leftarrow A[0], \; i \leftarrow 0, \; j \leftarrow 0
\]

\[
\text{for } k = 0 \text{ to } n - 1 \text{ do}
\]

\[
\text{newSum} \leftarrow 0
\]

\[
\text{for } m = k \text{ to } n - 1
\]

\[
\text{newSum} \leftarrow \text{newSum} + A[m]
\]

\[
\text{if } \text{newSum} > \text{largestSum} \text{ then}
\]

\[
\text{largestSum} \leftarrow \text{newSum}
\]

\[
i \leftarrow m, \; j \leftarrow k
\]

\[
\text{return } i, j
\]

Solution: There are many possible solutions to this problem. The most straightforward one is to iterate over all possible pairs of \(i \leq j\) and sum up array \(A\) between \(i\) and \(j\). This will result in \(O(n^3)\) complexity. The solution I have above is more efficient, \(O(n^2)\) because in computing sum between indexes \(k\) and \(m\), it reuses the already computed sum between \(k\) and \(m - 1\).

There is an even more efficient solution, which is \(O(n)\). The algorithm is due to Kadane (CMU). It requires a bit of thinking, and I did not expect anyone to come up with it. Suppose \([i...j]\) is the maximum-sum subarray. Then any sub-array \([i...k]\) with \(i \leq k < j\) must be positive, otherwise the subarray \([k+1...j]\) would have a sum larger than the sum of the sub-array \([i...j]\), and this is a contradiction. There exist an \(l, 1 \leq l < i\) such that the sub-array \([l...i-1]\) has negative sum, otherwise the sub-array \([l...j]\) would have a sum larger than the one \([i...j]\). As a consequence, start linear scan from index 0, adding up array values. If during this scan the sum is negative or zero, we can safely start a new segment. This algorithm works only if the largest sub-array sum is larger than 0. Here is the pseudo-code:

**Algorithm** FindLargestSumLinear\(A, n)\)

**Input:** Array \(A\) of size \(n\).

**Output:** \(i, j\) array subindexes between which the sum is the largest.

\[
sum \leftarrow A[0], \; maxSum \leftarrow A[0], \; i \leftarrow 0, \; j \leftarrow 0
\]

\[
\text{for } k = 1 \text{ to } n - 1 \text{ do}
\]

\[
sum \leftarrow sum + A[k]
\]

\[
\text{if } sum \leq 0 \text{ then}
\]

\[
sum \leftarrow 0
\]

\[
i \leftarrow k + 1
\]

\[
\text{if } sum > maxSum \text{ then}
\]

\[
maxSum \leftarrow sum
\]

\[
j \leftarrow k
\]

\[
\text{return } i, j
\]
7. (Just for fun) An evil king has $n$ bottles of wine, and a spy has poisoned one of them. Just one drop of poison kills, but it takes a full month for the poison to take effect. Design a scheme for determining exactly which one of the bottles is poisoned in just one month’s time while expending only $O(\log n)$ testers.

**Solution:** Let us number the bottles from 1 to $n$ using binary arithmetic, that is each bottle has a binary representation like 010111...1. The total number of binary digits we need to represent number $n$ is $\log(n)$. Now let taster 1 taste all the bottles whose first digit in binary representation is 1. Let the second taster taste all the bottles whose second digit in the binary representation is 1,..., let the $\log(n)$ taster taste all the bottles whose last digit, that is $\log_n$th digit is 1. Note that each of the $\log n$ tasters will taste exactly half of the bottles. In a month, see which tasters die. If the first taster died, then the poisonous bottle number in binary representation starts with 1, since he must have tasted from the poisonous bottle and he only tasted from the bottles starting with 1 in binary representation. By similar logic, if the first taster did not die, then the poisonous bottle number in binary representation starts with 0. And so on. That is if the $i$th taster dies, then the $i$th digit of the binary representation of the poisonous bottle starts with 1, and if the $i$th taster lives, then the $i$th digit of the binary representation of the poisonous bottle starts with 0.