Solutions to the Midterm Exam Problems

October 23, 2019
1. Let \( f \) and \( g \) be two integer functions.

**Definition I:** \( f = O(g) \) if there exist a real constant \( c > 0 \) and an integer \( n_0 > 0 \) such that \( f(n) \leq cg(n) \) for every integer \( n \geq n_0 \).

**Definition II:** \( f = \Omega(g) \) if there exist a real constant \( c > 0 \) and an integer \( n_0 > 0 \) such that \( f(n) \geq cg(n) \) for every integer \( n \geq n_0 \).

**Definition III:** \( f = \Theta(g) \) if there exist two real constants \( c', c'' > 0 \) and an integer \( n_0 > 0 \) such that \( c''g(n) \leq f(n) \leq c'g(n) \) for every integer \( n \geq n_0 \).

**Corollary I:** \( f = O(g) \) if and only if \( g = \Omega(f) \).

**Proof:**
- Assume \( f = O(g) \). By Definition I, there exist \( c > 0 \) and \( n_0 > 0 \) such that \( f(n) \leq cg(n) \) for every \( n \geq n_0 \). It follows that \( g(n) \geq (1/c)f(n) \) for every \( n \geq n_0 \). Since \( 1/c > 0 \), by Definition II \( g = \Omega(f) \).
- Assume \( g = \Omega(f) \). By Definition II, there exist \( c > 0 \) and \( n_0 > 0 \) such that \( g(n) \geq cf(n) \) for every \( n \geq n_0 \). It follows that \( f(n) \leq (1/c)g(n) \) for every \( n \geq n_0 \). Since \( 1/c > 0 \), by Definition I \( f = O(g) \).

**Corollary II:** \( f = \Theta(g) \) if and only if \( f = O(g) \) and \( f = \Omega(g) \).

**Proof:**
- Assume \( f = \Theta(g) \). By Definition III, there exist \( c', c'' > 0 \) and \( n_0 > 0 \) such that \( c''g(n) \leq f(n) \leq c'g(n) \) for every \( n \geq n_0 \). By Definition I, \( f = O(g) \) for \( c = c' \) and \( n_0 \) and by Definition II, \( f = \Omega(g) \) for \( c = c'' \) and \( n_0 \).
- Assume \( f = O(g) \) and \( f = \Omega(g) \). By Definition I, there exist \( c_1 > 0 \) and \( n_1 > 0 \) such that \( f(n) \leq c_1g(n) \) for every \( n \geq n_1 \) and by Definition II, there exist \( c_2 > 0 \) and \( n_2 > 0 \) such that \( f(n) \geq c_2g(n) \) for every integer \( n \geq n_2 \). Therefore, for \( n_0 \geq \max\{n_1, n_2\} \), it follows that \( c_2g(n) \leq f(n) \leq c_1g(n) \) for every \( n \geq n_0 \). By Definition III, \( f = \Theta(g) \) for \( c = c_1, c'' = c_2 \), and \( n_0 \).

**Proposition I:** \( n = O(n^2) \) and \( n^2 = \Omega(n) \).

**Proof:** Observe that \( n < n^2 \) for every integer \( n \geq 1 \). Therefore, for \( c = 1 \) and \( n_0 = 1 \), Definition I implies that \( n = O(n^2) \) and Definition II implies that \( n^2 = \Omega(n) \).

**Proposition II:** \( n^2 \neq O(n) \) and \( n \neq \Omega(n^2) \).

**Proof:** Observe that if \( (1/c) < n \) for a constant \( c > 0 \), then by multiplying both sides of the inequality by \( cn \) it follows that \( n < cn^2 \). Therefore, \( n < cn^2 \) for every real constant \( c > 0 \) and integer \( n \geq n_1 > (1/c) \). As a result, there are no real constant \( c > 0 \) and integer \( n_0 \) such that \( n \geq cn^2 \) for every integer \( n \geq n_0 \). Consequently, Definitions I and II cannot be applied to get \( n^2 = O(n) \) or \( n = \Omega(n^2) \).

**TRUE (T) or FALSE (F)?**

(a) \( f = \Omega(g) \) implies \( f = O(g) \): **FALSE**.
- Propositions I & II for the functions \( f(n) = n^2 \) and \( g(n) = n \) imply that \( f = \Omega(g) \) but \( f \neq O(g) \).

(b) \( f = O(g) \) implies \( f = \Theta(g) \): **FALSE**.
- Propositions I & II for the functions \( f(n) = n \) and \( g(n) = n^2 \) implies that \( f = O(g) \) but \( f \neq \Omega(g) \).
  It follows from Corollary II that \( f \neq \Theta(g) \).

(c) \( f = \Theta(g) \) implies \( f = \Omega(g) \): **TRUE**.
- By Corollary II.

(d) \( f = O(g) \) implies \( g = \Omega(f) \): **TRUE**.
- By Corollary I.

(e) \( f = O(g) \) implies \( g = O(f) \): **FALSE**.
- Propositions I & II for the functions \( f(n) = n \) and \( g(n) = n^2 \) imply that \( f = O(g) \) but \( g \neq O(f) \).

(f) \( f = \Theta(g) \) implies \( g = O(f) \): **TRUE**.
- By Corollary II, \( f = \Omega(g) \). Therefore, \( g = O(f) \) by Corollary I for \( f \leftrightarrow g \).
2. Define recurrence formulas for the following six exact solutions. Each formula should specify the initial value and the recurrence equation.

(a) \(T(n) = \log_2(n)\) for all non-negative powers of 2 \((1, 2, 4, 8, \ldots)\).

**Recurrence formula:**

\[
\begin{align*}
T(1) &= 0 \\
T(n) &= T(n/2) + 1
\end{align*}
\]

**Proof by induction:** \(T(1) = 0\) and \(\log_2(1) = 0\) for \(n = 1\). For \(n \geq 1\), assume that \(T(n) = \log_2(n)\) and prove that \(T(2n) = \log_2(2n)\).

\[
\begin{align*}
T(2n) &= T(n) + 1 \\
&= \log_2(n) + \log_2(2) \\
&= \log_2(2n)
\end{align*}
\]

**Top down proof:** Apply the recursion until \(n = 1\),

\[
\begin{align*}
T(n) &= T(n/2) + 1 \\
&= T(n/4) + 1 + 1 \\
&= T(n/8) + 1 + 1 + 1 \\
&\vdots \\
&= T(n/(2^i)) + i \\
&\vdots \\
&= T(n/(2^{\log_2(n)})) + \log_2(n) \\
&= T(1) + \log_2(n) \\
&= \log_2(n)
\end{align*}
\]

(b) \(T(n) = n\) for all positive integers \((1, 2, 3, \ldots)\).

**Recurrence formula:**

\[
\begin{align*}
T(1) &= 1 \\
T(n) &= T(n - 1) + 1
\end{align*}
\]

**Proof by induction:** \(T(1) = 1\) for \(n = 1\). For \(n \geq 1\), assume that \(T(n) = n\) and prove that \(T(n + 1) = n + 1\).

\[
\begin{align*}
T(n + 1) &= T(n) + 1 \\
&= n + 1
\end{align*}
\]

**Top down proof:** Apply the recursion \(n - 1\) times,

\[
\begin{align*}
T(n) &= T(n - 1) + 1 \\
&= T(n - 2) + 1 + 1 \\
&= T(n - 3) + 1 + 1 + 1 \\
&\vdots \\
&= T(n - i) + i \\
&\vdots \\
&= T(n - (n - 1)) + (n - 1) \\
&= T(1) + (n - 1) \\
&= n
\end{align*}
\]
(c) \( T(n) = n \log_2(n) \) for all non-negative powers of 2 \( (1, 2, 4, 8, \ldots) \).

**Recurrence formula:**

- \( T(1) = 0 \)
- \( T(n) = 2T(n/2) + n \)

**Proof by induction:** \( T(1) = 0 \) and \( 1 \cdot \log_2(1) = 0 \) for \( n = 1 \). For \( n \geq 1 \), assume that \( T(n) = n \log_2(n) \) and prove that \( T(2n) = \log_2(2n) \).

\[
T(2n) = 2T(n) + 2n \\
= 2(n \log_2(n)) + 2n \\
= 2n(\log_2(n) + 1) \\
= 2n(\log_2(n) + \log_2(2)) \\
= 2n \log_2(2n)
\]

**Top down proof:** Apply the recursion until \( n = 1 \),

\[
T(n) = 2T(n/2) + n \\
= 4T(n/4) + 2(n/2) + n \\
= 8T(n/8) + 4(n/4) + 2(n/2) + n \\
\vdots \\
= 2^i \cdot T(n/(2^i)) + n \cdot i \\
\vdots \\
= 2^{\log_2(n)} \cdot T(n/(2^{\log_2(n)})) + n \log_2(n) \\
= nT(1) + n \log_2(n) \\
= n \log_2(n)
\]

(d) \( T(n) = n^2 \) for all positive integers \( (1, 2, 3, \ldots) \).

**Recurrence formula:**

- \( T(1) = 1 \)
- \( T(n) = T(n-1) + (2n-1) \)

**Proof by induction:** \( T(1) = 1 \) and \( 1^2 = 1 \) for \( n = 1 \). For \( n \geq 1 \), assume that \( T(n) = n^2 \) and prove that \( T(n+1) = (n+1)^2 \).

\[
T(n+1) = T(n) + (2(n+1) - 1) \\
= n^2 + 2n + 1 \\
= (n + 1)^2
\]

**Top down proof:** Apply the recursion \( n - 1 \) times,

\[
T(n) = T(n-1) + (2n-1) \\
= T(n-2) + (2n-3) + (2n-1) \\
= T(n-3) + (2n-5) + (2n-3) + (2n-1) \\
\vdots \\
= T(n-i) + (2n-(2i-1)) + (2n-(2i-3)) + \cdots + (2n-3) + (2n-1) \\
\vdots \\
= 1 + 3 + 5 + \cdots + (2n-3) + (2n-1) \\
= (2 + 4 + 6 + \cdots + (2n-2) + 2n) - n \\
= 2(1 + 2 + 3 + \cdots + (n-1) + n) - n \\
= 2(n(n+1)/2) - n \\
= n^2 + n - n \\
= n^2
\]
(e) $T(n) = 2^n$ for all positive integers $1, 2, 3, \ldots$.

**Recurrence formula:**

\[
\begin{align*}
T(1) &= 2 \\
T(n) &= 2 \cdot T(n-1)
\end{align*}
\]

**Proof by induction:** $T(1) = 2$ and $2^1 = 2$ for $n = 1$. For $n \geq 1$, assume that $T(n) = 2^n$ and prove that $T(n + 1) = 2^{n+1}$.

\[
\begin{align*}
T(n + 1) &= 2 \cdot T(n) \\
&= 2 \cdot 2^n \\
&= 2^{n+1}
\end{align*}
\]

**Top down proof:** Apply the recursion $n - 1$ times,

\[
\begin{align*}
T(n) &= 2 \cdot T(n-1) \\
&= 2 \cdot 2 \cdot T(n-2) \\
&= 2 \cdot 2 \cdot 2 \cdot T(n-3) \\
&\vdots \\
&= 2^i \cdot T(n - i) \\
&\vdots \\
&= 2^{n-1} \cdot T(n - (n - 1)) \\
&= 2^{n-1} \cdot T(1) \\
&= 2^{n-1} \cdot 2 \\
&= 2^n
\end{align*}
\]

(f) $T(n) = n!$ for all positive integers $1, 2, 3, \ldots$.

**Recurrence formula:**

\[
\begin{align*}
T(1) &= 1 \\
T(n) &= n \cdot T(n-1)
\end{align*}
\]

**Proof by induction:** $T(1) = 1$ and $1! = 1$ for $n = 1$. For $n \geq 1$, assume that $T(n) = n!$ and prove that $T(n + 1) = (n + 1)!$.

\[
\begin{align*}
T(n + 1) &= (n + 1) \cdot T(n) \\
&= (n + 1) \cdot n! \\
&= (n + 1)!
\end{align*}
\]

**Top down proof:** Apply the recursion $n - 1$ times,

\[
\begin{align*}
T(n) &= n \cdot T(n-1) \\
&= n \cdot (n - 1) \cdot T(n-2) \\
&= n \cdot (n - 1) \cdot (n - 2) \cdot T(n-3) \\
&\vdots \\
&= n \cdot (n - 1) \cdots (n - (i - 1)) \cdot T(n - i) \\
&\vdots \\
&= n \cdot (n - 1) \cdots (n - (n - 1)) \cdot T(n - (n - 1)) \\
&= n \cdot (n - 1) \cdots 2 \cdot T(1) \\
&= n \cdot (n - 1) \cdots 1 \\
&= n!
\end{align*}
\]

(a) Assume that $A$ contains only 2 distinct integers. Design an algorithm that sorts the array with $O(n)$ comparisons between two array's integers. You may not use hashing or radix-sort like sorting algorithms.

Algorithm:
- Find the maximum, $Max$, and the minimum, $Min$, integers in $A$ with one scan that performs at most $n - 1$ comparisons. This is possible since there are only two different integers.
- During the scan, counts the number of times $Max$ and $Min$ appear in the array. This can be done with two counters $C_{\text{max}}$ and $C_{\text{min}}$ whose sum is $n$.
- Assign $Min$ to $A[1], \ldots, A[C_{\text{min}}]$ and $Max$ to $A[n - C_{\text{max}} + 1], \ldots, A[n]$ to get $A$ sorted.

Correctness: The algorithm assigns all the appearances of the smaller integer to the prefix of the array and assigns all the appearances of the larger integer to the suffix of the array.

Complexity: The algorithm performs exactly $n - 1$ comparisons to calculate the values of $C_{\text{min}}$ and $C_{\text{max}}$. Also, the algorithm performs $\Theta(n)$ operations to set and increment the counters and to rearrange the array. As a result, the total complexity is $\Theta(n)$.

(b) Assume that $n = 2^k - 1$ for some positive integer $k$ and that the array contains exactly $k$ different integers. Furthermore, the largest number appears $(n+1)/2$ times, the second largest number appears $(n+1)/4$ times and so on until the smallest number which appears exactly once.

Example: Let $n = 2^4 - 1 = 15$. Then $A$ contains exactly $k = 4$ distinct integers. The largest appears 8 times, the second largest appears 4 times, the third largest appears 2 times, and the smallest number appears once. $A$ could be the following array $[34, 34, 21, 8, 13, 34, 34, 21, 21, 34, 34, 34, 13, 34]$ in which $34$ appears 8 times, $21$ appears 4 times, $13$ appears twice, and $8$ appears once.

Design an algorithm that sorts the array with $O(n)$ comparisons between two array’s integers. You may not use hashing or radix-sort like sorting algorithms.

Notations: Let $m_1 > m_2 > \cdots > m_k$ be the $k$ integers in the array. By assumption, $m_h$ appears $(n+1)/2^h$ times in the array for $1 \le h \le k$. It follows that $m_h$ appears $2^{k-h}$ times in the array because

$$\frac{n+1}{2^n} = \frac{(2^k - 1) + 1}{2^k} = \frac{2^k}{2^h} = 2^{k-h}$$

Indeed, $m_1 + m_2 + \cdots + m_k = 2^{k-1} + 2^{k-2} + \cdots + 2 + 1 = 2^k - 1 = n$.

Algorithm:
- Find $m_1$, the maximum number in the array, with at most $n - 1$ comparisons.
- Scan the array and rearrange it such that all the $2^{k-1}$ appearances of $m_1$ are assined to the array entries $A[2^{k-1}], \ldots, A[n]$. This can be done with an index $i$ that runs from 1 to $n$ and an index $j$ that runs from $n$ to 1 until $i = j$. An exchange between $A[i]$ and $A[j]$ is performed only if $A[i] = m_1$ and $A[j] < m_1$.
- Continue recursively with the prefix of the array $A = A[1], A[2], \ldots, A[2^{k-1} - 1]$ that contains $k - 1$ integers $m'_1 = m_2 > m'_2 = m_3 > \cdots > m'_{k-1} = m_k$ integers such that $m_h$ appears $2^{(k-1)-h}$ times in the array for $1 \le h \le k - 1$.
- Terminate with the sorted array when the prefix has only one integer.

Correctness: The algorithm assigns all the appearances of the larger integer to the suffix of the array and then recursively sorts the prefix of the array. This can be proven by induction on $k$.

Complexity: For $k \ge h \ge 1$, when the size of the prefix is $2^h - 1$, the algorithm performs at most $2^h$ comparisons. The total number of comparisons is therefore at most

$$2^k + 2^{k-1} + \cdots + 2 + 1 = 2^{k+1} - 1 = 2n + 1 = \Theta(n).$$

Similarly, the algorithm performs $\Theta(n)$ handling the indices and rearranging the array operations. As a result, the total complexity is $\Theta(n)$. 

6

Remark: For the following two tasks, the running time should take into account both comparisons between array integers and operations between integers.

(a) Design an efficient algorithm that determines if the sum of all the integers is negative or positive? What is the running time of your algorithm?

Algorithm: Sum all the $n$ integers to find out if the sum is positive or negative.

Correctness: By definition.

Complexity: $\Theta(n)$ for one scan of the array.

Optimality: Any algorithm must examine all the $n$ integers to determine if the sum is negative or positive and therefore its complexity is $\Omega(n)$. Informally, if the algorithm “ignores” one of the integers, then there exists an assignment of values for the rest of the integers such that the sum of all numbers is negative or positive depending on the value of the ignored integer.

More formally, assume towards a contradiction that there exists an algorithm that does not examine the value of $A[k]$ for some $1 \leq k \leq n$. If $n = 1$, the algorithm must check if $A[1]$ is negative or positive. Assume that $n \geq 2$.

- Case $k = 1$: Let $A[i] = i$ for $2 \leq i \leq n$ and let $S = \sum_{i=2}^{n} A[i]$. As a result, the sum is negative if $A[1] < -S$ and the sum is positive if $-S < A[1] < 0$. Therefore, the algorithm must examine the value of $A[1]$. A contradiction.

- Case $k = n$: Let $A[i] = -n + 1 + i$ for $2 \leq i \leq n$ and let $S = \sum_{i=2}^{n} A[i]$. As a result, the sum is positive if $A[1] > -S$ and the sum is negative if $0 < A[1] < -S$. Therefore, the algorithm must examine the value of $A[n]$. A contradiction.


(b) Design an efficient algorithm that determines if the product of the integers is negative or positive? What is the running time of your algorithm?

Observation: The product is negative if and only if there are odd number of negative integers in the array.

Algorithm: If $n = 1$ then the product is negative if and only if $A[1]$ is negative. If $A[1] > 0$ then the product is positive. If $A[n] < 0$, then by the above observation, the product is negative if and only if $n$ is odd. Otherwise, $n \geq 2$ and


With a binary-search like procedure find the unique index $1 \leq k < n$ such that

$$A[k] < 0 < A[k + 1].$$

By the above observation, the product is negative if and only if $k$ is odd.

Correctness: By the definition of $k$, there are exactly $k$ negative integers in the array. Hence, the product is negative or positive depending on $k$ being odd or even.

Complexity: $\Theta(\log(n))$ for the binary search.

Optimality: There is no faster way to count the number of negative integers in the array. The answer cannot be determined without knowing this number.

---

The exact details of this assignment are omitted but they are not hard to figure out.