Discrete Structures

Algorithms Practice Problems: Solutions
1. (a) Fill in the following table with one of the three: $O$, $\Omega$, $\Theta$.

**Remark:** If $f = \Theta(g)$ then $f = O(g)$ and $f = \Omega(g)$ are wrong answers.

<table>
<thead>
<tr>
<th></th>
<th>$f(n)$</th>
<th>$g(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$n$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>b</td>
<td>$n^2$</td>
<td>$n$</td>
</tr>
<tr>
<td>c</td>
<td>$2n$</td>
<td>$5n$</td>
</tr>
<tr>
<td>d</td>
<td>$1000000n$</td>
<td>$(1/100000)n$</td>
</tr>
<tr>
<td>e</td>
<td>$\log_2(n)$</td>
<td>$\log_2(n)$</td>
</tr>
<tr>
<td>f</td>
<td>$\log_2(n)$</td>
<td>$\log_{10}(n)$</td>
</tr>
<tr>
<td>g</td>
<td>$n\log_2(n)$</td>
<td>$n/\log_2(n)$</td>
</tr>
<tr>
<td>h</td>
<td>$2^n$</td>
<td>$n^{100}$</td>
</tr>
<tr>
<td>i</td>
<td>$2^n$</td>
<td>$3^n$</td>
</tr>
<tr>
<td>j</td>
<td>$2^n$</td>
<td>$n!$</td>
</tr>
</tbody>
</table>

(b) Which of the following 10 functions are $O(n)$? Which are $\Omega(n)$? Which are $\Theta(n)$?

**Remark:** If $f = \Theta(n)$ then $f = O(n)$ and $f = \Omega(n)$ are wrong answers.

<table>
<thead>
<tr>
<th></th>
<th>$f(n)$</th>
<th>$g(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$2^n$</td>
<td>$\Omega$</td>
</tr>
<tr>
<td>b</td>
<td>$n^2$</td>
<td>$\Omega$</td>
</tr>
<tr>
<td>c</td>
<td>$2n$</td>
<td>$\Theta$</td>
</tr>
<tr>
<td>d</td>
<td>$n/\log_2(n)$</td>
<td>$O$</td>
</tr>
<tr>
<td>e</td>
<td>$\log_2(n)$</td>
<td>$O$</td>
</tr>
<tr>
<td>f</td>
<td>$100\log_2(n)\log_2(n)$</td>
<td>$O$</td>
</tr>
<tr>
<td>g</td>
<td>$n\log_2(n)$</td>
<td>$\Omega$</td>
</tr>
<tr>
<td>h</td>
<td>$10^{100}n/100^{100}$</td>
<td>$\Theta$</td>
</tr>
<tr>
<td>i</td>
<td>$n^\pi$</td>
<td>$\Omega$</td>
</tr>
<tr>
<td>j</td>
<td>$n!$</td>
<td>$\Omega$</td>
</tr>
</tbody>
</table>
2. Express the value of \( c \) when each of the following procedures terminates with the \( \Theta \)-notation.

(a) \( f(n) \) (* \( n = k^2 \) is a positive square integer *)

\[
\begin{align*}
  c &= 0 \\
  \text{for } i = 1 \text { to } n \text { do} \\
  &\quad \text{if } i \text { is a square number} \\
  &\quad \quad \text{then } c := c + 1
\end{align*}
\]

Answer: \( c = \sqrt{n} = \Theta(n^{1/2}) \).

Explanation: \( c \) is incremented only when \( i \) is a square number. That is, for \( n = k^2 \), \( c \) is incremented when \( i = 1, 4, 9, \ldots, (k-1)^2, k^2 \). The final value of \( c \) is \( k = \sqrt{n} \) because there are exactly \( k \) square integers between 1 and \( k^2 \).

(b) \( f(n) \) (* \( n > 1000 \) is a power of 2 *)

\[
\begin{align*}
  c &= 0 \\
  \text{while } n > 512 \text { do} \\
  &\quad n := n/2 \\
  &\quad c := c + 1
\end{align*}
\]

Answer: \( \log_2 n - 9 = \Theta(\log n) \).

Explanation: Assume \( n = 2^k \). Since \( n > 1000 \) it follows that \( k \geq 10 \). Each time \( c \) is incremented by 1, \( n \) is divided by 2. Therefore, the values of \( n \) are: \( 2^k, 2^{k-1}, \ldots, 2^9 \). Once \( n = 2^9 \) the while loop stops because \( 512 = 2^{9} \). Therefore, \( c \) is incremented \( k - 9 \) times. The answer is \( \log_2 n - 9 \) because \( k = \log_2 n \).
3. Let \( A = A[1], A[2], \ldots, A[n] \) be an unsorted array containing \( n \) distinct integers. For \( n \geq 3 \), describe an efficient algorithm that finds an integer in \( A \) that is neither the smallest integer in \( A \) nor the largest integer in \( A \). What is the complexity of your algorithm?

**Idea:** It is enough to consider only three integers from the array and find their median. Since all the integers in \( A \) are distinct, it follows that this median is neither the smallest integer in \( A \) nor the largest integer in \( A \).


```plaintext
Sort(A[1], A[2], A[3])
```

**Complexity:** Since the sorting is done with at most three comparisons, it follows that the complexity of this algorithm is \( \Theta(1) \).

**Remark:** There is no need to sort \( A \) with a \( \Theta(n \log n) \)-algorithm or to find the smallest and the largest integers in \( A \) with a \( \Theta(n) \)-algorithm.
4. Let \( A = A[1] < A[2] < \cdots < A[n] \) be a sorted array containing \( n \) distinct negative and positive integers. Describe an efficient algorithm that finds, if it exists, an index \( 1 \leq i \leq n \) such that \( A[i] = i \). What is the complexity of your algorithm?

A trivial linear complexity algorithm: For all indices \( 1 \leq i \leq n \), check if \( A[i] = i \). If such an index is found, return it. Otherwise, after learning that \( A[n] \neq n \), return a message that such an index does not exist.

Algorithm \( \mathcal{X} (A) \):

\[
\text{for } i = 1 \text{ to } n \text{ do } \\
    \text{if } A[i] = i \text{ then return}(i) \\
\text{return}("A[i] \neq i for all indices } 1 \leq i \leq n \text{ in } A")
\]

Algorithm \( \mathcal{X} \) is correct because by inspecting all the \( n \) indices in \( A \), it cannot miss, if it exists, an index \( i \) for which \( A[i] = i \).

The complexity of algorithm \( \mathcal{X} \) is \( \Theta(n) \) because in the worst-case the algorithm needs to examine all the \( n \) entries in the array with complexity \( \Theta(1) \) for each entry.

Observation: \( A[i+1] - (i+1) \geq A[i] - i \) for \( 1 \leq i < n \).


A linear complexity algorithm with \( \Theta(\log n) \) comparisons: Define an array \( B \) such that \( B[i] = A[i] - i \) for \( 1 \leq i \leq n \). It follows that if \( B[i] = 0 \) for some \( 1 \leq i \leq n \) then \( A[i] = i \). The above observation implies that \( B[1] \leq B[2] \leq \cdots \leq B[n] \). Use Binary-Search to find if 0 appears in the array \( B \). Return the index \( i \) if there exists \( 1 \leq i \leq n \) such that \( B[i] = 0 \). Otherwise return the message \( A[i] \neq i \) for all \( 1 \leq i \leq n \).

Algorithm \( \mathcal{Y} (A) \):

\[
\text{for } i = 1 \text{ to } n \text{ do } B[i] = A[i] - i \\
    i = \text{Binary-Search}(B, 0) \\
    \text{if } A[i] = i \text{ then return}(i) \\
    \text{else return}("A[i] \neq i for all indices } 1 \leq i \leq n \text{ in } A")
\]

By definition of the array \( B \), it follows that if \( B[i] = 0 \) for some \( 1 \leq i \leq n \) then \( A[i] = i \). Since \( B \) is sorted, the Binary-Search procedure finds the smallest index \( i \) such that \( B[i] = 0 \). On the other hand, if 0 is not in \( B \) then the Binary-Search procedure returns an index \( i \) for which \( B[i] \neq 0 \) and therefore \( A[i] \neq i \). In this case, algorithm \( \mathcal{Y} \) returns a negative message. Both arguments prove that algorithm \( \mathcal{Y} \) is correct.

Algorithm \( \mathcal{Y} \) is using \( \Theta(\log n) \) comparisons which is the complexity of the Binary-Search procedure. However, the overall complexity of the algorithm is \( \Theta(n) \) since the for loop that defines the array \( B \) has \( n \) iterations.
A $\Theta(\log n)$ algorithm: There is no need for array $B$. The comparison $B[i] = 0$ is equivalent to the comparison $A[i] = i$. Therefore, the Binary-Search procedure can be modified to run directly on the array $A$.

Algorithm $Z(A)$:

1. $\ell = 1$
2. $u = n$
3. while $\ell \leq u$ do
   1. $m = \left\lfloor \frac{u + \ell}{2} \right\rfloor$
   2. Case $A[m] = m$ then return($m$)
   3. Case $A[m] > m$ then $u = m - 1$
   4. Case $A[m] < m$ then $\ell = m + 1$
5. return(“$A[i] \neq i$ for all indices $1 \leq i \leq n$ in $A$”)

Algorithm $Z$ is correct because it is equivalent to algorithm $Y$.

The while loop in algorithm $Z$ has at most $\lceil \log n \rceil$ iterations the same number of iterations that the Binary-Search procedure has. The complexity of algorithm $Z$ is $\Theta(\log n)$ since the complexity of each iteration is $\Theta(1)$. 


5. Let \( A = A[1] \leq A[2] \leq \cdots \leq A[n] \) be a sorted array of \( n \) integers. Let \( k \) be an integer. Describe an efficient algorithm that finds the number of times \( k \) appears in the array. What is the complexity of your algorithm?

**A trivial linear complexity algorithm:** Count the number of indices for which \( A[i] = k \) by scanning the whole array.

Algorithm \( \mathcal{X}(A) \):

\[
\text{Count} = 0 \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad \text{if } A[i] = k \\
\quad \quad \text{then } \text{Count} = \text{Count} + 1 \\
\text{return(“k appears Count times in A”)}
\]

Algorithm \( \mathcal{X} \) is correct because it examines all the integers in the array \( A \).
There are \( n \) iterations of the for loop in algorithm \( \mathcal{X} \) and the complexity of each iteration is \( \Theta(1) \). Therefore, the complexity of algorithm \( \mathcal{X} \) is \( \Theta(n) \).

**Assumption:** When the Binary-Search procedure is looking to find a key in an array, it returns its first location if it appears at least once in the array. Otherwise it returns 0 if \( k < A[1] \), returns \( n \) if \( A[n] < k \), and returns the index \( i \) for which \( A[i] < k < A[i + 1] \). The complexity of this version of Binary-Search is still \( \Theta(\log n) \).

Algorithm \( \mathcal{Y}(A) \):

- Run Binary-Search to find if \( k \) appears in \( A \).
- If \( k \) does not appear in \( A \):
  * return(“k appears 0 times in A”).
- Otherwise, assume \( A[i] = k \) while \( A[i - 1] < k \) or \( i = 1 \).
- Run Binary-Search to find if \( k + 1 \) appears in the sub array \( A[i + 1] \leq \cdots \leq A[n] \).
  * return(“k appears \( j - i + 1 \) times in A”).
- Otherwise, the Binary-Search returns \( j \) such that \( A[j] = k \) and \( A[j + 1] > k + 1 \). As in the previous case:
  * return(“k appears \( j - i + 1 \) times in A”).

The high level description of algorithm \( \mathcal{Y} \) explains why algorithm \( \mathcal{Y} \) is correct.
The complexity of algorithm \( \mathcal{Y} \) is \( \Theta(\log n) \) which is the complexity of two executions of the Binary-Search procedure.

**Remark:** Consider the following variation of algorithm \( \mathcal{Y} \) called algorithm \( \mathcal{Z} \). After finding that the first appearance of \( k \) is in \( A[i] \), algorithm \( \mathcal{Z} \) counts the number of appearances of \( k \) in \( A \) sequentially. Assume \( A[i] = k \) for some \( 1 \leq i \leq n \). Then algorithm \( \mathcal{Z} \) checks if \( A[i + 1] = k, A[i + 2] = k, \ldots \) until it finds \( j \) such that \( A[j + 1] > k \) or until \( j = n \).
Then algorithm \( \mathcal{Z} \) returns that \( k \) appears \( (j - i + 1) \) times in \( A \). While algorithm \( \mathcal{Z} \) is more efficient than algorithm \( \mathcal{Y} \) when \( (j - i + 1) \) is small, in the worst case when \( (j - i + 1) = \Theta(n) \), the complexity of algorithm \( \mathcal{Z} \) is \( \Theta(n) \).

**Example:** For $n = 11$, $k = 47$, and $A = [1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]$, the output is 6 and 8 because $A[6] + A[8] = 13 + 34 = 47$. However, for $k = 59$ the answer should be that such indices do not exist.

What is the complexity (number of comparisons) of your algorithm?

**A trivial quadratic complexity algorithm:** For each $1 \leq i \leq n$, check if $A[i] + A[j] = k$ for all $i \leq j \leq n$.

Algorithm $\mathcal{X}(A)$:

```python
for i = 1 to n - 1 do
    for j = i + 1 to n do
            then return((i, j))
return("k is not the sum of two different integers from A")
```

Algorithm $\mathcal{X}$ is correct because it examines all possible pairs of integers in the array $A$.

There are $(n - 1) + (n - 2) + \cdots + 1 = \frac{(n-1)n}{2} = \Theta(n^2)$ iterations of the two for loops in algorithm $\mathcal{X}$ and the complexity of each iteration is $\Theta(1)$. Therefore, the complexity of algorithm $\mathcal{X}$ is $\Theta(n^2)$.

**A binary search based algorithm:** For each $1 \leq i \leq n$, use the Binary-Search procedure to check if $k - A[i]$ appears in $A$. Recall, that if $x$ appears in $A$, then the Binary-Search procedure returns the index $j$ such that $A[j] = x$.

Algorithm $\mathcal{Y}(A)$:

```python
for i = 1 to n do
    j = Binary-Search(A,(k - A[i]))
        then return((i, j))
return("k is not the sum of two different integers from A")
```

Algorithm $\mathcal{Y}$ is correct because if $A[j] + A[j] = k$ then $A[j] = k - A[i]$ and the binary search procedure would find this index $j$.

The complexity of each iteration of algorithm $\mathcal{Y}$ is $\Theta(\log n)$ – the same as the complexity of the Binary-Search procedure. Since there are $n$ iterations, it follows that the complexity of algorithm $\mathcal{Y}$ is $\Theta(n \log n)$. 


**Observation:** For $1 \leq i < j \leq n$,


**Proof:** Both statements follow because the array $A$ contains $n$ distinct integers that are sorted in ascending order.

**A linear complexity algorithm:** Maintain two indices $1 \leq i < j \leq n$, starting with $i = 1$ and $j = n$, such that if $A[i] + A[j] = k$ then it must be the case that $1 \leq i' < j' \leq j$. If $A[i] + A[j] = k$ then return the indices $i$ and $j$. Otherwise, increment $i$ by 1 if $A[i] + A[j] < k$ or decrement $j$ by 1 if $A[i] + A[j] > k$. Return an unsuccessful message once $i = j$. Algorithm Z($A$):

```plaintext
i = 1
j = n
while i < j do
    case A[i] + A[j] = k then return((i, j))
    case A[i] + A[j] < k then i = i + 1
    case A[i] + A[j] > k then j = j - 1
return("k is not the sum of two different integers from A")
```

Algorithm $Z$ is correct with a proof by induction on $n - (j - i)$ that during the run of the algorithm, for $1 \leq i < j \leq n$, if $A[i'] + A[j'] = k$ then necessarily $1 \leq i' < j' \leq j$. The base case is when $i = 1$ and $j = n$ and therefore $n - (j - i) = 1$. In this case, clearly if $A[i'] + A[j'] = k$ then $1 \leq i' < j' \leq n$ because these are the only possible values for $i'$ and $j'$. The inductive step is correct due to the above observation.

- Case $A[i] + A[j] < k$: By the observation, there is no $j' < j$ such that $A[i] + A[j'] = k$ therefore $i$ is incremented.
- Case $A[i] + A[j] > k$: By the observation, there is no $j' > i$ such that $A[i'] + A[j] = k$ therefore $j$ is decremented.

There are at most $n$ iterations in the while loop of algorithm $Z$. This is because the value of $j - i$ is initially $n - 1$ and after each unsuccessful iteration it is decremented by one until it becomes zero. The complexity of algorithm $Z$ is $\Theta(n)$ since the complexity of each iteration is $\Theta(1)$. 

**A trivial quadratic complexity algorithm:** For each \( 1 \leq i \leq n \) check if \( A[i] > B[j] \) for all \( 1 \leq j \leq n \).

Algorithm \( \mathcal{X}(A, B) \):

\[
\text{Count} = 0 \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad \text{for } j = 1 \text{ to } n \text{ do} \\
\quad \quad \text{if } A[i] > B[j] \\
\quad \quad \quad \text{then } \text{Count} = \text{Count} + 1 \\
\quad \text{return(Count)}
\]

Algorithm \( \mathcal{X} \) is correct because it examines all \( n^2 \) possible pairs of integers one from \( A \) and one from \( B \).

There are exactly \( n^2 \) iterations of the two for loops in algorithm \( \mathcal{X} \) and the complexity of each iteration is \( \Theta(1) \). Therefore, the complexity of algorithm \( \mathcal{X} \) is \( \Theta(n^2) \).

**A binary search based algorithm:** For each \( 1 \leq i \leq n \), use the Binary-Search procedure to find an index \( 0 \leq j \leq n \) such that either \( B[j] < A[i] < B[j + 1] \) for \( 1 \leq j \leq n - 1 \), or \( A[i] > B[n] \) and then \( j = n \), or \( A[i] < B[1] \) and then \( j = 0 \). Recall, that if \( x \) does not appear in \( A \), then the Binary-Search procedure returns the index \( j \) such that \( B[j] < x < B[j + 1] \) if \( B[1] < x < B[n] \), returns \( n \) if \( x > B[n] \), and returns \( 0 \) if \( x < B[1] \).

Algorithm \( \mathcal{Y}(A, B) \):

\[
\text{Count} = 0 \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad j = \text{Binary-Search}(B, A[i]) \\
\quad \text{Count} = \text{Count} + j \\
\text{return(Count)}
\]

Algorithm \( \mathcal{Y} \) is correct because \( A[i] \) by definition is not in \( B \) for \( 1 \leq i \leq n \) and the index \( j \) returned by the Binary-Search procedure is exactly the number of integers in \( B \) that are smaller than \( A[i] \).

The complexity of each iteration of algorithm \( \mathcal{Y} \) is \( \Theta(\log n) \) – the same as the complexity of the Binary-Search procedure. Since there are \( n \) iterations, it follows that the complexity of algorithm \( \mathcal{Y} \) is \( \Theta(n \log n) \).
**Notations:** Let $B[0] = 0$ and $B[n + 1] = 1 + \max\{A[n], B[n]\}$. It follows that $B[0]$ is smaller than all the $2n$ distinct integers from $A$ and $B$ while $B[n + 1]$ is larger than all the $2n$ distinct integers from $A$ and $B$.

**Observation:** For $1 \leq i \leq n$, let $0 \leq j(i) \leq n$ be the index in $B[0..n + 1]$ such that $B[j(i)] < A[i] < B[j(i) + 1]$. Then,

$$0 \leq j(1) \leq j(2) \leq \cdots \leq j(n) \leq n$$

**Proof:** First note that for all $1 \leq i \leq n$ since $B[0] < A[i] < B[n + 1]$ it follows that $0 \leq j(i) \leq n$. Fix $1 \leq i \leq n - 1$. Since $A[i] < A[i + 1]$ it follows that $j(i) \leq j(i + 1)$.

**Corollary:**

$$Count = j(1) + j(2) + \cdots + j(n)$$

**Proof:** There are exactly $j(i)$ integers in $B$ that are smaller than $A[i]$.

**A linear complexity algorithm:** Maintain two indices $1 \leq i \leq n$ and $0 \leq j \leq n$, starting with $i = 1$ and $j = 0$. For a given $i$, increment $j$ by 1 as long as $j < j(i)$ until $j = j(i)$ (equivalently, $A[i] < B[j + 1]$). Next update $Count$ by adding to it $j(i)$ and increment $i$ by 1. Return the value of $Count$ once $i = n + 1$.

Algorithm $Z(A, B)$:

```latex
\begin{align*}
    & j = 0 \\
    & \text{for } i = 1 \text{ to } n \text{ do} \\
    & \quad \text{while } (A[i] > B[j + 1]) \text{ do } j = j + 1 \\
    & \quad Count = Count + j \\
    & \text{return}(Count)
\end{align*}
```

Algorithm $Z$ is correct with a proof by induction that for any $1 \leq i \leq n$, at the end of the $i$th iteration of the for loop, $Count = j(1) + j(2) + \cdots + j(i)$. As a result, after the $n$th iteration, $Count = j(1) + j(2) + \cdots + j(n)$. By the above corollary, this is the correct value of $Count$.

Assume $j(0) = 0$. Fix $1 \leq i \leq n$. The complexity of the $i$th iteration of the for loop in algorithm $Z$ is $\Theta(j(i) - j(i - 1))$. Therefore, the complexity of algorithm $Z$ is

$$\Theta(j(1) - j(0)) + \Theta(j(2) - j(1)) + \cdots + \Theta(j(n) - j(n - 1))$$

Since

$$(j(1) - j(0)) + (j(2) - j(1)) + \cdots + (j(n) - j(n - 1)) = j(n) - j(0) = j(n) \leq n$$

it follows that the complexity of algorithm $Z$ is $\Theta(n)$. 

11