Discrete Structures

Algorithms Practice Problems: Solutions
1. 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>$O$</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>$1000000n$</td>
<td>$\Theta$</td>
</tr>
<tr>
<td>e</td>
<td>$\log_2(n)$</td>
<td>$O$</td>
</tr>
<tr>
<td>f</td>
<td>$\log_2(n)$</td>
<td>$\Theta$</td>
</tr>
<tr>
<td>g</td>
<td>$n\log_2(n)$</td>
<td>$\Omega$</td>
</tr>
<tr>
<td>h</td>
<td>$2^n$</td>
<td>$\Omega$</td>
</tr>
<tr>
<td>i</td>
<td>$2^n$</td>
<td>$O$</td>
</tr>
<tr>
<td>j</td>
<td>$2^n$</td>
<td>$O$</td>
</tr>
</tbody>
</table>

2. 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>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$2^n$</td>
</tr>
<tr>
<td>b</td>
<td>$n^2$</td>
</tr>
<tr>
<td>c</td>
<td>$2n$</td>
</tr>
<tr>
<td>d</td>
<td>$n/\log_2(n)$</td>
</tr>
<tr>
<td>e</td>
<td>$\log_2(n)$</td>
</tr>
<tr>
<td>f</td>
<td>$100\log_2(n)\log_2(n)$</td>
</tr>
<tr>
<td>g</td>
<td>$n\log_2(n)$</td>
</tr>
<tr>
<td>h</td>
<td>$10^{100}n/100^{100}$</td>
</tr>
<tr>
<td>i</td>
<td>$n^\pi$</td>
</tr>
<tr>
<td>j</td>
<td>$n!$</td>
</tr>
</tbody>
</table>

3. Match the following 8 functions as 4 pairs: if $f(n)$ is paired with $g(n)$ then $f(n) = \Theta(g(n))$.

$\frac{2^{n+1}}{n} ; \frac{n}{\log_2(n^2)} ; \frac{n^2}{2^n} ; \log_2(n) ; \log_2(2^n) ; 100n^2 - 500n ; \log(2^n) ; 2^{n+1}$

**Answer:**

- $\log_2(n^2) = 2\log_2(n) \implies \log_2(n^2) = \Theta(\log_2(n))$
- $\log_2(2^n) = n\log_2(2) = n \implies \log_2(2^n) = \Theta(n)$
- $100n^2 - 500n = \Theta(100n^2) \implies 100n^2 - 500n = \Theta(n^2)$
- $2^{n+1} = 2 \cdot 2^n \implies 2^{n+1} = \Theta(2^n)$
4. Let $P$ be a problem whose input is an array of size $n$ for $n \geq 1$. Order the following ten algorithms from the most efficient to the least efficient.

- Algorithm A solves $P$ with complexity $\Theta(n)$.
- Algorithm B solves $P$ with complexity $\Theta(2^n)$.
- Algorithm C solves $P$ with complexity $\Theta(n \log(n))$.
- Algorithm D solves $P$ with complexity $\Theta(n^2)$.
- Algorithm E solves $P$ with complexity $\Theta(n^2)$.
- Algorithm F solves $P$ with complexity $\Theta(1)$.
- Algorithm G solves $P$ with complexity $\Theta(n^n)$.
- Algorithm H solves $P$ with complexity $\Theta(n^{100})$.
- Algorithm I solves $P$ with complexity $\Theta(\log(n))$.
- Algorithm J solves $P$ with complexity $\Theta(\sqrt{n})$.

**Answer:** The following is the hierarchy among the ten functions:

$$1 = o(\log(n)) = o(\sqrt{n}) = o(n) = o(n \log(n)) = o(n^2) = O(n^{100}) = o(2^n) = o(n!) = o(n^n)$$

Therefore, using $X < Y$ to indicate that algorithm $X$ is more efficient than algorithm $Y$, it follows that the order among the ten algorithms from the most efficient one to the least efficient one is as follows:

$$F < I < J < A < C < E < H < B < D < G$$

5. For each of the following four parts, give an example of a function that satisfies the criteria or state that none exist.

(a) A function that is $O(n/2)$ and also $\Omega(2n)$.

**Answer:** Both $n/2 = \Theta(n)$ and $2n = \Theta(n)$. Therefore, $n = O(n/2)$ and $n = \Omega(2n)$.

(b) A function that is both $\Omega(10n)$ and $O(n^2/100)$.

**Answer:** Observe that $10n = o(n^2/100)$, $10n = \Theta(n)$, and $n^2/100 = \Theta(n^2)$. Therefore, any function that is $\Omega(n)$ and $O(n^2)$ is a correct answer. In particular, both $n$ and $n^2$ are correct answers. But also $n \log_2(n)$, $\sqrt{n}$, and $n / \log_2(n)$ are correct answers. In fact, more accurately, the latter three functions are $\omega(2n)$ and $o(n/2)$.

(c) A function that is $O(5n)$ but not $\Theta(n/3)$.

**Answer:** Such a function must be $O(n)$ but not $\Theta(n)$ and as a result it must be $o(n)$. The functions $\sqrt{n}$ and $\log(n)$ are two examples.

(d) A function that is $\Omega(2^n)$ but not $\Theta(2^n)$.

**Answer:** Such a function must be $\omega(n)$. The functions $3^n$, $n!$, and $n^n$ are three examples.

6. A problem $P$ has an **upper bound** complexity $O(n^2)$ and a **lower bound** complexity $\Omega(n)$.

(a) Could someone design an algorithm that solves the problem with complexity $n^3$?

**Answer:** Yes because $n^3 = \Omega(n)$ and it is always possible to design inefficient algorithms.

(b) Could someone design an algorithm that solves the problem with complexity $0.5n$?

**Answer:** Yes because $0.5n = \Theta(n)$ and therefore $0.5n = O(n^2)$ and $0.5n = \Omega(n)$.

(c) Could someone design an algorithm that solves the problem with complexity $100 \log(n)$?

**Answer:** No because $100 \log(n) = o(n)$ and as such $100 \log(n) \neq \Omega(n)$. 


7. Express the value of \( c \) when each of the following procedures terminates with the \( \Theta \)-notation.

**Bonus:** Try to find then exact value of \( c \) when each of the following procedures terminates.

(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 \( k = \log_2 n \).

8. Consider the following procedure:
\[
\begin{align*}
f(x, y) \quad (* \text{ a positive multiple of } 3 \text{ integer } x \text{ and a positive integer } y *) \\
c &= 0 \\
\text{for } i = 1 \text{ to } x/3 & \text{ do} \\
& \quad \text{for } j = 1 \text{ to } 6y^2 & \text{ do} \\
& \quad \quad \text{then } c := c + 1
\end{align*}
\]
(a) As a function of \( x \) and \( y \), what is the **exact** value of \( c \) when the program terminates?

**Answer:** The variable \( c \) is incremented \((x/3)(6y^2)\) times. Therefore, when the program terminated \( c = 2xy^2 \).

(b) Define \( x \) and \( y \) as functions of \( n \) such that \( c = \Theta(n^3) \) when the program terminates.

**Answer:** When \( n = 2x \) and therefore \( x = n/2 \) and \( y = n \), by part (a) the program terminates with \( c = 2xy^2 = 2(n/2)(n^2) = n^3 \). Trivially, \( n^3 = \Theta(n^3) \). This can be generalized for any \( x = \Theta(n) \) and \( y = \Theta(n) \).

There are infinitely many other answers. For example, \( x = \Theta(n^2) \) and \( y = \Theta(\sqrt{n}) \).

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

```plaintext
for i := 1 to n do
  if A[i] = i then return(i)
return(“A[i] ≠ i for all indices 1 ≤ i ≤ n 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 and $n \cdot \Theta(1) = \Theta(n)$.

**Remark:** Algorithm $\mathcal{X}$ is correct with the same linear complexity even if the array is not sorted.

**Observation:** $A[i+1] - (i+1) ≥ A[i] - i$ for $1 ≤ i < n$.


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

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

```plaintext
for i := 1 to n do B[i] := A[i] - i
  i := Binary-Search(B, 0)
  if A[i] = i then return(i)
else return(“A[i] ≠ i for all indices 1 ≤ i ≤ n in A”)
```

By definition of the array $B$, it follows that if $B[i] = 0$ for some $1 ≤ i ≤ n$ 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 $B[i] ≠ 0$ and therefore $A[i] ≠ i$. 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 $\Theta(\log(n))$-complexity algorithm: In fact, 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 $\mathcal{Z}(A)$:

```plaintext
\ell := 1 and u := n
while \ell < u do
  m := \left\lfloor \frac{u + \ell}{2} \right\rfloor
  If A[m] ≥ m then u := m
  else \ell := m + 1
if A[\ell] = \ell then return(\ell)
else return(“A[i] ≠ i for all indices 1 ≤ i ≤ n in A”)
```

Algorithm $\mathcal{Z}$ is correct because it is equivalent to algorithm $\mathcal{Y}$.

The while loop in algorithm $\mathcal{Z}$ has at most $\lceil \log_2(n) \rceil$ iterations the same number of iterations that the Binary-Search procedure has. The complexity of algorithm $\mathcal{Z}$ is $\Theta(\log(n))$ since each iteration has a $\Theta(1)$-complexity and $\Theta(\log(n)) \cdot \Theta(1) = \Theta(\log(n))$. 

5
10. For $n \geq 1$, let $A$ be an array of size $n$ for which the first $k$ entries contain positive integers and the rest of the array is all zeros. The value of $n$ is known but the value of $k$, which can be any number between 0 and $n$, is unknown.

Examples:
- $[34, 13, 21, 0, 0, 0, 0, 0]: k = 3$ in this array of length 8.
- $[0, 0, 0, 0, 0, 0, 0, 0]: k = 0$ in this array of length 7.
- $[55, 8, 34, 13, 21, 89]: k = 6$ in this array of length 6.

Describe an efficient algorithm that determines the value of $k$ which is the number of positive integers in $A$. What is the complexity of the algorithm?

A trivial linear complexity algorithm: Scan the array starting with the first entry in the array until either finding a zero or reaching the end of the array.

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

```plaintext
k := 0
while (k < n) and (A[k + 1] > 0) do
    k := k + 1
return(k)
```

Algorithm $\mathcal{X}$ is correct because by inspecting all the $n$ indices in $A$, the algorithm identifies the last non-zero entry if it exists or returns 0 if $A$ contains only zeros.

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 and $n \cdot \Theta(1) = \Theta(n)$.

A $\Theta(\log n)$-complexity algorithm: Run the Binary-Search procedure to find the last zero in the array $A$. The rules for the binary-search are that if $A[i] = 0$ then it must be the case that $k < i$ while if $A[i] > 0$ it must be the case that $k \geq i$.

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

```plaintext
A[0] := 1
\ell := 0
r := n
while (\ell < r) do
    m := \left\lfloor \frac{\ell + r}{2} \right\rfloor
    if A[m] = 0
        then r := m
    else \ell := m + 1
return(\ell)
```

Algorithm $\mathcal{Y}$ is correct because the binary-search will find the last appearance of a positive integer in $A$ which always exists after defining $A[0] = 1$.

The number of iterations in algorithm $\mathcal{Y}$ is $\Theta(\log(n + 1)) = \Theta(\log(n))$ which is the complexity of the Binary-Search procedure on an array of length $n + 1$. Consequently, the complexity of algorithm $\mathcal{Y}$ is $\Theta(\log(n))$ since each iteration has a $\Theta(1)$-complexity and $\Theta(\log(n)) \cdot \Theta(1) = \Theta(\log(n))$. 

Describe an efficient algorithm that finds the number of times $k$ appears in the array. What is the complexity of the algorithm?

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

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

$\text{Count} := 0$

for $i := 1$ to $n$

if $A[i] = k$

then $\text{Count} := \text{Count} + 1$

return("$k$ appears $\text{Count}$ 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 $\Theta(n)$ because $n \cdot \Theta(1) = \Theta(n)$.

**Remark:** Algorithm $\mathcal{X}$ is correct with the same linear complexity even if the array is not sorted.

**A $\Theta(\log n)$-complexity algorithm:** Run the Binary-Search procedure twice to search in $A$ for $k$ and $k + 1$ and then deduce the number of times $k$ appears in the array.

**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 $(j - i)$ times in $A$").
  - * 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 $A[i + 1] = k$, $A[i + 2] = k$, $\cdots$ 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 $\Theta(n)$.
12. For \( n \geq 2 \), let \( A = A[1] < A[2] < \cdots < A[n] \) be a sorted array with \( n \) distinct positive integers from the range \( 1, 2, \ldots, n + 1 \). That is, exactly one of the integers from this range is missing in the array \( A \).

**Examples:** The missing integer in the array \([1, 2, 3, 4, 5, 7, 8, 9]\) is 6, the missing integer in the array \([1, 2, 4, 5]\) is 3, the missing integer in the array \([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15]\) is 12, the missing integer in the array \([2, 3, 4, 5, 6, 7]\) is 1, and the missing integer in the array \([1, 2, 3, 4, 5, 6, 7, 8, 9]\) is 10.

Describe an efficient algorithm that finds the missing integer. The only questions about the integers in the arrays that your algorithm may ask are of the type “is \( A[i] = i? \)” for some integer \( 1 \leq i \leq n \).

What is the worst-case complexity (number of questions asked) of the algorithm?

**A trivial linear complexity algorithm:** For all indices \( 1 \leq i \leq n \), check if \( A[i] = i \). The missing integer is \( i \) when \( A[i] \neq i \) which implies that \( A[i] = i + 1 \). If at the end \( A[n] = n \) then the missing integer is \( n + 1 \).

Algorithm \( X(A) \):

```plaintext
for i := 1 to n do
  if A[i] = i then continue
  else return(i)
return(n + 1)
```

Algorithm \( X \) is correct because it inspects all the \( n \) indices in \( A \). If the missing number is \( i < n + 1 \), then it must be found during the scan because in this case \( A[i] = i + 1 \). If the missing number is \( n + 1 \), then the procedure finishes the scan and returns \( n + 1 \).

The complexity of algorithm \( X \) 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 and \( n \cdot \Theta(1) = \Theta(n) \).

**A \( \Theta(\log n) \)-complexity algorithm:** Run the Binary-Search procedure to find the first index in the array \( A \) for which \( i < A[i] \).

Algorithm \( Y(A) \):

```plaintext
A[n + 1] := n + 2
ℓ := 1
r := n + 1
while (ℓ < r) do
  m := \left\lfloor \frac{ℓ + r}{2} \right\rfloor
  if A[m] = m
    then ℓ := m + 1
  else r := m
return(ℓ)
```

Algorithm \( Y \) is correct because the binary-search will find the first index \( i \) in \( A \) such that \( A[i] = i + 1 \) which always exists after defining \( A[n + 1] = n + 2 \).

The number of iterations in algorithm \( Y \) is \( \Theta(\log(n + 1)) = \Theta(\log(n)) \) which is the complexity of the Binary-Search procedure on an array of length \( n + 1 \). Consequently, the complexity of algorithm \( Y \) is \( \Theta(\log(n)) \) since each iteration has a \( \Theta(1) \)-complexity and \( \Theta(\log(n)) \cdot \Theta(1) = \Theta(\log(n)) \).

**Remark:** There is a way to solve this problem without comparisons if the algorithm may apply addition operations involving entries of the array. First, find the sum \( S \) of all the integers in the array. If all the integers between 1 and \( n + 1 \) were in the array, then the sum would have been \( 1 + 2 + \cdots (n + 1) = \frac{(n + 1)(n + 2)}{2} \). As a result the missing number is

\[
\frac{(n + 1)(n + 2)}{2} - S
\]

For example, for the array \([1, 2, 3, 4, 5, 7, 8, 9]\) the sum is \( S = 39 \). The missing number is 6 because

\[
\frac{9 \cdot 10}{2} - 39 = 45 - 39 = 6
\]
13. For \( n \geq 2 \), let \( A = A[1], \ldots, A[n] \) be an array of \( n \) positive integers. Let the sum of all the integers in the array be \( M = A[1] + \cdots + A[n] \). For \( 1 \leq i \leq n \), let \( S[i] \) be the sum of all the numbers in the array except \( A[i] \).

\[
\]

**Example:** Let \( A = [16, 2, 128, 64, 1, 8, 32, 4] \). Then \( M = 255 \) and \( S = [239, 253, 127, 191, 254, 247, 223, 251] \).

Design a linear time algorithm (\( \Theta(n) \)) to compute \( S[1], \ldots, S[n] \) only with plus operations (it is not allowed to use minus operations).

What is the exact number of plus operations used by the algorithm?

**A by-definition \( \Theta(n^2) \)-Algorithm:** For \( 1 \leq i \leq n \), compute \( S[i] = A[1] + \cdots + A[i-1] + A[i+1] + \cdots + A[n] \).

**Complexity:** For \( 1 \leq i \leq n \), computing \( S_i \) is done by exactly \( n - 2 \) plus operations. Therefore, the total number of plus operations in this algorithm is \( n(n - 2) = n^2 - 2n = \Theta(n^2) \).

**A \( \Theta(n) \)-Algorithm:**

- Compute the prefix-sum of the first \( n - 1 \) integers in \( A \). For \( 1 \leq i \leq n - 1 \), let \( P[i] = \sum_{j=1}^{i} A[j] \).
- Compute the suffix-sum of the last \( n - 1 \) integers in \( A \). For \( n \geq i \geq 2 \) let \( Q[i] = \sum_{j=i}^{n} A[j] \).
- Compute the array \( S \)

\[
S[i] = \begin{cases} Q[2] & \text{for } i = 1 \\ P[n-1] & \text{for } i = n \\ P[i-1] + Q[i+1] & \text{for } 2 \leq i \leq n-1 \end{cases}
\]

**Correctness:** By definition, \( S_1 = Q[2] \) and \( S_n = P[n-1] \). Fix \( 2 \leq i \leq n-1 \). Then \( P[i-1] = A[1] + \cdots + A[i-1] \) and \( Q[i+1] = A[i+1] + \cdots + A[n] \). Therefore,

\[
\]

**Remark:** Note that there is no need to compute the last values of the prefix-sum (\( P[n] \)) and the last value of the suffix-sum (\( Q[1] \)) because they are not required for the computations of \( S[1], S[2], \ldots, S[n] \).

**Complexity:** The \( n - 1 \) prefix-sum values can be computed with \( n - 2 \) plus operations and so are the \( n - 1 \) suffix-sum values. Then, for \( 2 \leq i \leq n - 1 \), all the \( S_i \) values are computed with \( n - 2 \) plus operations. The total number of plus operations in this algorithm is

\[
(n - 2) + (n - 2) + (n - 2) = 3n - 6 = \Theta(n)
\]

**Example:**

\[
A = [16, 2, 128, 64, 1, 8, 32, 4] \\
P = [16, 18, 146, 210, 211, 219, 251, *, *] \\
Q = [* , 239, 237, 109, 45, 44, 36, 4] \\
S = [239, 253, 127, 191, 254, 247, 223, 251]
\]

\[
\]

Describe an efficient algorithm that finds two integers from the array whose sum is even. What is the complexity the algorithm?

**A by-definition algorithm:** Examine the sums of all possible \( \binom{n}{2} \) pairs of integers from the array until finding a pair whose sum is even.

Algorithm \( X(A) \):

```
for \( i := 1 \) to \( n - 1 \) do
    for \( j := i + 1 \) to \( n \) do
        if \( A[i] + A[j] \) is even
            then return \( (A[i], A[j]) \)
    return ("there are no two integers in \( A \) whose sum is even")
```

Algorithm \( X \) is correct because it inspects the sums of all possible pairs in \( A \).

The complexity of algorithm \( X \) is \( O(n^2) \) because in the worst-case the two loops terminate when there are no two integers in \( A \) whose sum is even. However, if \( A[1] \) is even then \( i \) will be 2 only if the rest of the integers in the array including \( A[2] \) and \( A[3] \) are odd. But then \( A[2] + A[3] \) is even and \( i \) is never greater than 2. Similarly, if \( A[1] \) is odd then \( i \) will be 2 only if the rest of the integers in the array including \( A[2] \) and \( A[3] \) are even. But then \( A[2] + A[3] \) is even and \( i \) is never greater than 2. As a result, the worst-case complexity of algorithm \( X \) is \( \Theta(n) \).

**A constant time algorithm:** The \( \Theta(n) \) analysis of algorithm \( X \) works even if only the first three integers in \( A \) are examined. This is because two of the integers among \( A[1], A[2], A[3] \) must have the same parity and as a result their sum is even.

Algorithm \( Y(A) \):

```
```

There are constant number of operations in algorithm \( Y \) and therefore its complexity is \( \Theta(1) \).

**Remark:** Algorithm \( Y \) works with any three integers from the array \( A \) and it works even if \( A \) is not sorted.