7200. Analysis of Algorithms

Midterm Exam

Solutions
- Consider the following 10 functions on positive integers $n$:
  - Which are $\Theta(n)$?
  - Which are $\Omega(n)$ but not $\Theta(n)$?
  - Which are $O(n)$ but not $\Theta(n)$?

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

- **Input:** For $n \geq 1$, a sorted array $A = [A_1 < A_2 < \cdots < A_n]$ that contains $n$ distinct integers from the range $[1..(n+1)]$. That is, exactly one of the integers $1, 2, \ldots, n+1$ is missing from $A$.

- **Examples:**
  - For $n = 5$, the missing integer in $[1, 2, 4, 5, 6]$ is $3$.
  - For $n = 6$, the missing integer in $[2, 3, 4, 5, 6, 7]$ is $1$.
  - For $n = 4$, the missing integer in $[1, 2, 3, 4]$ is $5$.

- **Task:** Design an efficient algorithm that identifies the missing integer.

- **Complexity:** What is the worst-case number of comparisons made by the algorithm?

**Trivial algorithm:** Scan the array from $1$ to $n$ until finding an index $i$ such that $A_i > i$ or until $A_n = n$. In the first case return $i$ and in the second case return $n + 1$.

**Correctness:** In the first case, $A_1 = 1, A_2 = 2, \ldots, A_{i-1} = i - 1$ and $A_i > i$. Therefore, $A_i = i + 1$ and the missing number is $i$. In the second case, $A_i = i$ for all $1 \leq i \leq n$. Therefore, the missing number is $n + 1$.

**Complexity:** For $1 \leq i \leq n$, the algorithm makes exactly $i$ comparisons when the missing number is $i$. The algorithm makes $n$ comparisons when the missing number is either $n$ or $n + 1$ which is the worst-case. Thus, its complexity is $\Theta(n)$.

The proofs of the following three propositions are implied by the definition of the array $A$:

**Proposition I:** $A_i \geq i$ for all $1 \leq i \leq n$.

**Proposition II:** For $1 \leq i \leq n$, if $A_i = i$, then the missing integer is $j$ such that $j > i$.

**Proposition III:** For $1 \leq i \leq n$, if $A_i > i$ then the missing integer is $j$ such that $1 \leq j \leq i$.

**Algorithm:** If $A_n = n$ then return $(n + 1)$. Otherwise, apply the following Binary Search recursive procedure. The search is done in a range $[\ell..r]$ of the array for some $1 \leq \ell \leq r \leq n$. Initially, $\ell = 1$ and $r = n$.

**Recursive step** for the range $[\ell..r]$ for which $\ell \leq r$:
- If $\ell = r$ then return ($\ell$).
- Let $m = \left\lfloor \frac{\ell + r}{2} \right\rfloor$ be the middle index of the range $[\ell..r]$.
- Compare $A_m$ with $m$.
  - Case $A_m = m$: continue recursively with the range $[(m + 1)..r]$. (* Proposition II *)
  - Case $A_m > m$: continue recursively with the range $[\ell..m]$. (* Proposition III *)

**Termination:** Implied by the following arguments.
- First, note that by Proposition I, there is no need to check if $A_m < m$.
- Next, observe that $\ell < m < r$ when $\ell < r - 1$ and that $\ell = m = r - 1$ when $\ell = r - 1$. Therefore, $\ell \leq m$ when the new $r$ becomes $m$ and when the new $\ell$ becomes $m + 1$.
- Finally, since $r - (m + 1) < r - \ell$ and $m - \ell < r - \ell$, it follows that the size of the range is getting smaller after each recursive call. Therefore, the algorithm eventually reaches the case of $\ell = r$ and terminates.

**Correctness:** If the missing number is $n + 1$ the algorithm identifies this before applying the Binary-Search procedure. Otherwise, $A_n > n$ and therefore initially $A_r > r$. If $A_m = m$, then $r$ is not changed and in the next recursive step still $A_r > r$. If $A_m > m$, then the new $r$ in the next recursive step is $m$ and therefore still $A_r > r$. Hence, during the run of the algorithm, it is always the case that $A_r > r$. Once $\ell = r$, the range contains only one integer which is $A_r = r + 1$. This implies that $r$ is the missing integer.

**Complexity:** The size of the range of the next recursive step is at most half of the size of the current range. Therefore, there are at most $\lceil \log(n) \rceil$ recursive steps. Since each recursive step makes exactly one comparison, it follows that the Binary-Search part of the algorithm always makes $\lceil \log(n) \rceil$ comparisons. Together with the initial comparison between $A_n$ and $n$, the algorithm makes $\lceil \log(n) \rceil + 1$ comparisons. Thus, its complexity is $\Theta(\log n)$.

**Avoiding the initial comparison “if $A_n = n$”**: Without the initial comparison, the algorithm fails when the missing number is $n + 1$ because it terminates with $\ell = r = n$ and then returns $n$ instead of $n + 1$. A somewhat sophisticated solution is to continue recursively with the range $[\ell..(m - 1)]$ instead of the range $[\ell..m]$ in the case $A_m > m$. However, the correctness proof of this version is more involved and is omitted.
Part III

• **Input:** A positive integer $k$ and two arrays $A$ and $B$ each of $n \geq 1$ distinct positive integers:
  * A sorted (in an ascending order) array $A = [A_1 < A_2 < \cdots < A_n]$.
  * A sorted (in a descending order) array $B = [B_1 > B_2 > \cdots > B_n]$.

• **Task:** Design an efficient algorithm that finds, if exist, two indices $1 \leq i, j \leq n$ such that $A_i + B_j = k$.

• **Examples:** For $n = 8$, consider $A = [1, 3, 6, 10, 15, 21, 28, 36]$ and $B = [34, 21, 13, 8, 5, 3, 2, 1]$.
  * $k = 27$: the output is $[i = 3, j = 2]$ since $A_3 + B_2 = 6 + 21 = 27$.
  * $k = 32$: the output is: NO because there are no indices $1 \leq i, j \leq n$ such that $A_i + B_j = 32$.
  * $k = 8$: the output could be either $[i = 2, j = 5]$ or $[i = 3, j = 7]$ since $A_2 + B_5 = A_3 + B_7 = 8$.

• **Complexity:** What is the worst-case number of comparisons made by the algorithm?

---

**Trivial algorithm:** For all possible pairs of integers one from $A$ and one from $B$ check if their sum is $k$.

**Correctness:** By definition.

**Complexity:** Since there are $n^2$ pairs of integers one from $A$ and one from $B$, the algorithm makes exactly $n^2$ comparisons. Thus, the complexity of the algorithm is $\Theta(n^2)$.

**Remark:** This algorithm works even if both arrays are not sorted.

---

**Binary-Search based algorithm:** For each integer in $B_j$ ($1 \leq j \leq n$) in the array $B$, using Binary-Search, check if $k - B_j$ is in the sorted array $A$.

**Correctness:** If $A_i + B_j = k$ for some $i$ and $j$ such that $1 \leq i, j \leq n$, then Binary-Search finds $A_i$ while searching if $A - B_j$ appears in $A$.

**Complexity:** Since $A$ is sorted, each application of Binary-Search makes $\lceil \log_2(n) \rceil$ comparisons. The total number of comparisons is therefore $n \lceil n \log_2(n) \rceil$. Thus, the complexity of the algorithm is $\Theta(n \log(n))$.

**Remark:** It is enough that only one of the input arrays is sorted for this algorithm to work. If both input arrays are not sorted, then sorting one of the arrays and applying this algorithm would be an efficient $\Theta(n \log(n))$-algorithm. Sorting both arrays and using the linear time algorithm described next would be less efficient although its complexity is also $\Theta(n \log(n))$. 
Efficient linear-time algorithm outline: Scan both arrays together in at most $2n$ rounds. Initially, the scanning indices are $i = 1$ for $A$ and $j = 1$ for $B$. If in some round $A_i + B_j = k$, then terminate with the indices $i$ and $j$. Otherwise, after each round for which $A_i + B_j \neq k$ for $i$ and $j$ such that $1 \leq i, j \leq n$, continue the scan with either $i$ and $j + 1$ or with $i + 1$ and $j$ depending on whether the sum is larger or smaller than $k$ respectively. If one of $i$ and $j$ reaches $n + 1$ the output is NO.

Observation I: Let $1 \leq a, b \leq n$ be two indices such that $A_a + B_b > k$. Then $A_h + B_b > k$ for all $a \leq h \leq n$ because $A_h + B_b \geq A_a + B_b > k$. Therefore, if there are two indices $a \leq i \leq n$ and $b \leq j \leq n$ such that $A_i + B_j = k$ then $j = b$ is not one of them.

Observation II: Let $1 \leq a, b \leq n$ be two indices such that $A_a + B_b < k$. Then $A_a + B_h < k$ for all $b \leq h \leq n$ because $A_a + B_h \leq A_a + B_b < k$. Therefore, if there are two indices $a \leq i \leq n$ and $b \leq j \leq n$ such that $A_i + B_j = k$ then $i = a$ is not one of them.

Algorithm: Initially, $i = j = 1$. Iterate the following procedure until either $i > n$ or $j > n$. In this case, return a NO answer (that is, there are no two indices $a$ and $b$ such that $A_a + B_b = k$ in the array).

Iteration for $1 \leq i, j \leq n$:
- Case $A_i + B_j > k$: continue recursively with $i = i$ and $j = j + 1$. (* Observation I *)
- Case $A_i + B_j = k$: return a YES answer with the indices $i$ and $j$.
- Case $A_i + B_j < k$: continue recursively with $i = i + 1$ and $j = j$. (* Observation II *)

Correctness: Implied by Observations I and II.

Termination and complexity: Let $S = i + j$. Initially, $S_{initial} = S = 2$. After each iteration, the value of $S$ is increased by 1. The maximum number of iterations happens when the answer is YES or NO when both $i = n$ and $j = n$. At this stage, $S_{final} = 2n$. Therefore, there are at most $S_{final} - S_{initial} + 1 = 2n - 1$ iterations. In each iteration there is at most one comparison. Thus, the complexity of the algorithm is $\Theta(n)$. 


Part IV

- **Input:** An arbitrary (not necessarily sorted) array \( A = [A_1, A_2, \ldots, A_n] \) of \( n \geq 1 \) distinct positive integers.

- **Task:** Design an efficient algorithm that determines if there exists a positive integer \( k \) such that both \( k \) and \( k^2 \) are in the array.

- **Model:** Sorting algorithms like Bucket-Sort, Radix-Sort, or Hashing that are based on the values of the integers cannot be used. The only type of questions (comparisons) that the algorithm may ask about integers in the array are:
  - Is \( A_i < A_j \) for some \( 1 \leq i, j \leq n \).
  - Is \( A_j = A_i^2 \) for some \( 1 \leq i, j \leq n \)

- **Examples:** For \( n = 8 \),
  - \( A = [49, 38, 2, 9, 27, 16, 4, 64] \): The answer is YES because both 4 and 16 appear in the array.
  - \( A = [49, 99, 17, 9, 25, 6, 54, 4] \): The answer is NO because the square of any integer in \( A \) is not in \( A \).

- **Complexity:** What is the worst-case number of comparisons made by the algorithm?

**Trivial algorithm:** For each integer \( A_i \) (\( 1 \leq i \leq n \)), scan the entire array to check if \( A_i^2 \) is in the array.

**Correctness:** By definition.

**Complexity:** Note that if \( A_i = 1 \) then \( A_i^2 = A_i \). Therefore, for each integer \( A_i \) (\( 1 \leq i \leq n \)), the scan should check the entire array to look for \( A_i^2 \). As a result, when the array does not contain a pair \((k, k^2)\), the algorithm makes exactly \( n^2 = \Theta(n^2) \) comparisons.

**Sort first algorithm:** First sort the array. Then for each integer \( A_i \) (\( 1 \leq i \leq n \)), check using Binary-Search if \( A_i^2 \) is in the array.

**Correctness:** By definition.

**Complexity:** This algorithm makes \( \Theta(n \log(n)) \) comparisons to sort the array (e.g., using Merge-Sort) and then makes \( \Theta(\log(n)) \) comparisons for each one of the \( n \) applications of Binary-Search. The total complexity is therefore
\[
\Theta(n \log(n)) + n \cdot \Theta(\log(n)) = \Theta(n \log(n))
\]

**Remark:** The binary search for \( A_i^2 \) must be done in the entire array because \( A_1 \) could be 1. However, the binary search for \( A_i^2 \) (\( 2 \leq i \leq n - 1 \)) can be done only in the range \([(i + 1), n]\) of the array. Moreover, for \( n \geq 2 \) there is no need to do any binary search for \( A_n^2 \). See the efficient solution for how to take advantage of these and similar observations to do all the searches with at most \( 2n - 1 \) comparisons.
A more efficient procedure to check if the sorted array contains a pair \((k, k^2)\): Scan the sorted array with two indices \(i\) and \(j\) such that \(1 \leq i \leq j \leq n\). Initially \(i = j = 1\). Iterate the following until \(j = n + 1\) and then return NO.

- Case \(A_i^2 < A_j\): set \(i = i + 1\).
- Case \(A_i^2 = A_j\): return YES with the indices \(i\) and \(j\).
- Case \(A_i^2 > A_j\): set \(j = j + 1\).

**Termination:** Assume that the array does not contain a pair \((k, k^2)\). Observe that whenever \(i = j\), only \(j\) can be incremented. Since for each \(j\) eventually \(i\) must be equal to \(j\), eventually \(j\) will be equal to \(n + 1\).

**Correctness:** The following arguments justify the decision when to increment \(i\) and when to increment \(j\).

- \(A_i^2 < A_j\): Since the array is sorted, it follows that \(A_i^2 < A_h\) for all \(j \leq h \leq n\). Therefore, \(A_i^2\) is not in the array and the search should continue with \(i + 1\).
- \(A_i^2 > A_j\): Since the array is sorted, it follows that \(A_h^2 > A_j\) for all \(i \leq h \leq n\). Therefore, \(A_j\) can not be a square of any integer in the array and the search should continue with \(j + 1\).

**Complexity:** In the worst-case, the array does not contain a pair \((k, k^2)\) and both indices \(i\) and \(j\) equal \(n\). Then after incrementing \(j\), the algorithm terminates because \(j = n + 1\). In this case, the algorithm makes \(2n - 1\) comparisons.

**Remark:** Since \(\Theta(n \log(n)) + \Theta(n) = \Theta(n \log(n))\), this more efficient implementation of this part of the overall algorithm does not improve the order of the total complexity. However, if the input array is sorted, then this double scanning search for squares would improve the complexity to \(\Theta(n)\) while \(\Theta(n \log(n))\) is the complexity implied by applying Binary-Search \(n\) times to look for squares.