Lab01-Proof, Algorithm Design and Analysis

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2018.

∗ If there is any problem, please contact TA Xinyu Wu.
∗ Name:Juncheng Wan   Student ID:516021910620   Email: 578177149@qq.com

1. (a) **(Proof by Contrapositive)** Suppose $a, b, c \in \mathbb{Z}$. Please prove: If $a^2 + b^2 = c^2$, then $a \times b$ is even. (Hint: $\forall m \in \mathbb{N}, m^2 \mod 4 = 0$ or 1.)

   Proof. $a \times b$ is odd $\Longrightarrow$ a is odd and b is odd. $\Longrightarrow a = 2k+1, b = 2t+1, k, t \in \mathbb{N}.$ $\Longrightarrow$ $c^2 = a^2 + b^2 = (2k+1)^2 + (2t+1)^2 = 4 \times (k^2 + k + t^2 + t) + 2.$ $\Longrightarrow c^2 \mod 4 = 2.$
   It contradicts with $\forall m \in \mathbb{N}, m^2 \mod 4 = 0$ or 1. □

   (b) **(Course-of-Values Induction)** Let $P = \{p_1, p_2, \cdots\}$ the set of all primes. Suppose that $\{p_i\}$ is monotonically increasing, i.e., $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, $\cdots$. Please prove: $p_n < 2^{2^n}$. (Hint: $p_i \nmid (1 + \prod_{j=1}^{n} p_j), i = 1, 2, \cdots, n.$)

   Proof. First, when $n = 1$, $p_1 = 2 < 2^{2^1} = 4$. Second, assume that $\forall n \le k$, $p_k = 2 < 2^{2^k}$. Third, when $n = k+1$, I consider two conditions:
   If $p_{k+1} \le (1 + \prod_{j=1}^{k} p_j)$, $p_{k+1} \le 1 + \prod_{j=1}^{k} p_j < 1 + \prod_{j=1}^{k} 2^{2^j} = 1 + 2^{\sum_{j=1}^{k} 2^j} = 1 + 2^{2^{k+1}-2} < 2^{2^{n+1}}$. $\Longrightarrow p_{k+1} < 2^{2^{k+1}}$.
   If $p_{k+1} \ge (1 + \prod_{j=1}^{k} p_j)$, I prove that it is impossible by proof by contrapositive. $p_{k+1} \ge (1 + \prod_{j=1}^{k} p_j) \Longrightarrow p_{k+1}$ is not a prime. $\Longrightarrow \exists p_i \in \{p_1, p_2, \cdots, p_n\}$ s.t. $p_i \mid p_{k+1}$. It contradicts with $p_i \nmid (1 + \prod_{j=1}^{n} p_j), i = 1, 2, \cdots, n.$ □

2. Please analyze the time complexity of Algorithm 2 with brief explanations.

Algorithm 1: 'Modified' InsertionSort

Input: An array $A[1, \cdots, n]$
Output: $A[1, \cdots, n]$ sorted nonincreasingly

```
for i ← 2 to n do
    x ← A[i];
    k ← BinarySearch(A[1, ··· , i − 1], x); //Finding k such that
        A[k] ≥ x ≥ A[k + 1] by binary search (A[1] ≥ x or x ≥ A[i − 1] for two
        boundary points).
    for j ← i − 1 downto k do
        A[j + 1] ← A[j];
    A[k] ← x;
```

Solution. If $A[1] < A[2] < \cdots < A[n]$, this algorithem will have the worst case. For the kth iteration, time complexity is $\lfloor logk \rfloor$.
$\sum_{k=1}^{n} \lfloor logk \rfloor = 1 + 2 + \cdots + \lfloor logn \rfloor = \frac{(\lfloor logn \rfloor + 1)(\lfloor logn \rfloor)}{2} \Longrightarrow O((logn)^2)$ □

3. **Top-k Search:** In reality, we sometimes intend to identify the first k maximum (minimum) elements in an array with size n. This problem is commonly called *Top-k Search*. Suppose that the array we consider is $\{a_1, a_2, \cdots, a_n\}$ and we intend to find the k maximum elements. A common approach is to use sorting on the array from maximum to minimum and select the first k elements. Please answer the following questions:

(a) Ana, a student of course CS214, wonders if the time complexity can be lowered. She is enlightened that we only need to identify these k elements but do not need to sort them in the requirement of this problem. She notices that when $k = 1$, the time complexity decreases to $O(n)$. Hence she guesses that there may be an algorithm to solve this problem with time complexity $O(nk)$, lower than the insertion sort. Please tell Ana whether her guess is realizable. If so, please design such an algorithm written in pseudo code; If not, please tell her the reason.

Solution. I just identify the first k maximum elements in an array with size n. For the case of mimimum, it is similar.

Algorithm 2: The Second 'Modified' InsertionSort

Input: An array $A[1, \cdots, n]$, k

Output: $A[1], A[2], \cdots, A[k]$, the first k maximum elements in $A[1, \cdots, n]$

for $i \leftarrow 1$ *to* k **do**
    $x \leftarrow A[i]$;
    for $j \leftarrow i$ *to* n **do**
        if $x < A[j]$ **then**
            $y \leftarrow x$;
            $x \leftarrow A[j]$;
            $A[j] \leftarrow y$;
return $A[1], A[2], \cdots, A[k]$

In the worst case, the time complexity is calculated as follows:

$$\sum_{j=1}^{j=k}(n - j + 1) = nk - \frac{k(k+1)}{2} + k \Longrightarrow O(nk)$$

□

(b) If you answer 'Yes' in Problem 3a, then please consider: Whether the time complexity can be further reduced to $O(n \log k)$? You can just write your ideas without the need to write an algorithm. (Hint: Consider better data structure.)

Solution. Select $A[1], A[2], \cdots, A[k]$ to build a minimum heap of k nodes. Set the smallest number of the heap X. Then, for $A[k+1]$, if it is larger than X, throw X from the heap and insert $A[k+1]$ to this heap. If it is not larger than X, just throw it away. Repeat this process for $n - k$ times. Finally, the k elements of this heap is the first k maximum elements in $A[1, \cdots, n]$. For the first k minimum elements in $A[1, \cdots, n]$, the algorithm is similar.

Because the time complexity of each inserting is $O(logk)$ and the times of inserting is $O(n)$, the whole time complexity is $O(nlogk)$. □

(c) (Optional Sub-question with Bonus) Consider a special case where there are many repeated elements in the array. Specifically, we suppose there are $O(\log n)$ different values in the array. Then whether the time complexity can be $O(n \log \log n)$, which further lowers the complexity for $k = \omega(\log n)$? You can just write your ideas without the need to write an algorithm. (Hint: Construct AVL Tree.)

Solution. Constuct an AVL Tree by $A[1], A[2], \cdots, A[k]$. If $A[i] = A[j]$, store them in the same node in this AVL Tree. This just records the number without orther operation.

There are $\mathrm{O}(\log n)$ different values in the array $\Longrightarrow$ The number of nodes in this AVL Tree is $\mathrm{O}(\log n)$. $\Longrightarrow$ The height of this AVL Tree is $\mathrm{O}(\log \log n)$. $\Longrightarrow$ The time complexity of construction of this AVL Tree is $\mathrm{O}(n \log \log n)$. $\Longrightarrow$ To get the first k maximum elements in an array with size n, the time complexity is $\mathrm{O}(n \log \log n)$. $\square$