Lab01-Proof, Algorithm Design and Analysis

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

∗ Name: Hongyi Guo Student ID: 516030910306 Email: guohongyi@sjtu.edu.cn

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.

   We change this statement by its logically equivalence: If $a \times b$ is odd, then $a^2 + b^2 \neq c^2$.
   If $a \times b$ is odd, then obviously a is odd and b is odd. Assume $a = 2m+1, b = 2n+1 (m, n \in \mathbb{Z})$. Then $a^2 + b^2 = 4(m^2 + n^2 + m + n) + 2$. Thus, $a^2 + b^2 \equiv 2 \pmod 4$. Meanwhile, $c^2 \equiv 0$ or $1 \pmod 4$.
   Therefore, $a^2 + b^2 \neq c^2$. □

   (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.

   Hypothesise for $k > 1$ and $1 \leq n \leq k$, $P(n)$ is true, which means $p_n < 2^{2^n}$.
   Assume $p_{k+1} \geq 2^{2^{k+1}}$.
   We assign $q = 1 + \prod_{j=1}^{k} p_j$. Obviously $q > p_k$. Meanwhile,

$$q < 1 + \prod_{j=1}^{k} 2^{2^k} = 1 + 2^{2^{k+1}-2} \leq 2^{2^{k+1}} \leq p_{k+1}$$

   Thus, $p_k < q < p_{k+1}$. Then q is not a prime for there is no prime between p_k and p_{k+1}. Therefore, $\exists p \in \{p_1, p_2, \cdots, p_k\}, p \mid q$, which contradicts the hint $p_i \nmid (1 + \prod_{j=1}^{n} p_j), i = 1, 2, \cdots, n$.
   So the assumption $p_{k+1} \geq 2^{2^{k+1}}$ fails. Thus, $p_{k+1} < 2^{2^{k+1}}$. $P(k+1)$ is true.
   Together, $p_n < 2^{2^n} (n \in \mathbb{N})$. □

2. Please analyze the time complexity of Algorithm 1 with brief explanations.

Algorithm 1: 'Modified' InsertionSort

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

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

Solution.

The time complexity of Algorithm 1 is $\mathrm{O}(n^2)$. The outer **for** is executed $n-1$ times. The inner **for** is executed at most $i-1$ times. The complexity of each Binary search is $O(\log(i-1))$, which is much smaller than the complexity of inner **for** and can be ignored.

$$\sum_{i=2}^{n}(i-1) = n(n-1)/2 = O(n^2)$$

□

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 $\mathrm{O}(n)$. Hence she guesses that there may be an algorithm to solve this problem with time complexity $\mathrm{O}(nk)$, 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.

   Yes, it's easy to solve this problem with time complexity $\mathrm{O}(nk)$.

Algorithm 2: Top-k Search,$\mathrm{O}(nk)$

Input: An array $A[1, \cdots, n]$, k
Output: $B[1, \cdots, k]$ the k maximum elements of A

```
1  for i ← 1 to n do
2      chosen[i] ← false; //chosen[i]: whether A[i] is chosen as one of the k-top elements.
3  for i ← 1 to k do
4      curMax ← −∞;
5      curMaxIndex ← 1;
6      for j ← 1 to n do
7          if chosen[j] = false and A[j] > curMax then
8              curMax ← A[j];
9              curMaxIndex ← j;
10     B[i] ← curMax;
11     chosen[curMaxIndex] ← true;
```

□

(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.

Yes, the time complexity can be further reduced to $O(n \log k)$ as following steps:

step 1. Construct a min heap and push the first k elements of the array into the heap. The complexity of this step is $O(k)$.

step 2. For each element in the rest, say x, compare it with the top of the min heap, say t. If $x < t$, pop t and push x into the heap. The complexity of this step is $O(n \log k)$.

step 3. In the last, all elements in the heap are the first k maximum elements in the given array. You can pop k times to fetch them, whose complexity is $O(k \log k)$.

Analyse: We consider the elements of the min heap are the candidates of *top-k*. There can only be k candidates in one moment. The first k elements are initial candidates. But the rest are going to challenge them. Each of them just needs to defeat the current weakest candidate which is the top of the heap. If one of them successes, it'll weed out the failed candidate and become a new one. The final candidates are the top-k elements. □

(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.

Yes, the time complexity can be $O(n \log \log n)$ using AVL Tree or other balanced trees. Use AVL Tree as an example:

step 1. Build an special AVL Tree which is empty at the beginning. Each node has extra data members *cnt* and *size*. *cnt* records how many elements in the array has the same value with this node. *size* is the size of the subtree whose root is this node, which equals to $leftChild \to size + rightChild \to size + cnt$. The size of empty node is 0.

step 2. Insert every element in the array to the AVL Tree. For each element, if the tree has a node whose value is the same with it, then plus the *cnt* of that node by 1. Otherwise, insert a new node with the value of that element and the *cnt* of 1. For there are only $\log n$ different values in the array, the height of the AVL tree is $O(\log n)$. So the time complexity of this step is $O(n \log \log n)$.

step 3. With the data member *cnt* and *size*, it's easy to get the top-k elements from the AVL Tree. Different methods have different time complexity but it's definitely smaller than $O(n \log \log n)$ in step 2. □