CS4311 Design and Analysis of Algorithms

Homework 1 (Solution Sketch)

1. (a) We shall find the position using the binary search strategy. Let B denote the array after John's modification, and let $x = B[1]$ denote the value of its first element. Thus for any entry $B[j]$, we have $B[j] < x$ if and only if this entry belongs to A_{left}.

   Our target is to find which entry of B corresponds to the beginning of A_{left}. We shall first compare the middle element $B[m]$ of B with x. In case $B[m]$ is smaller than x, we can deduce that the target entry is in the subarray $B[1..m]$ so that we recursively search for such element in $B[1..m]$. Otherwise, we can deduce that the target entry is in the subarray $B[m+1..n]$ and we recursively search for such element in $B[m+1..n]$.

   After each step, the problem size (number of entries to be searched) is reduced by half, so that the search will stop after $O(\log n)$ steps. Thus, the running time is $O(\log n)$.

   (b) The correctness of this algorithm follows from the following statement (why?), which can be shown easily by induction (how?): After each step, the subarray to be searched must contain the target entry.

2. (a) By examining the code, we see that the value of `count` is equal to either 0 or 1. More precisely, it is 0 if the number of factors of n is an even number, and 1 otherwise. By our high school mathematics, n has an even number of factors if and only if n is not a perfect square. (The reason is: For each factor x smaller than $\sqrt{n}$, there is a distinct factor, n/x, larger than $\sqrt{n}$. Thus, the number of factors smaller than $\sqrt{n}$ is exactly equal to the number of factors larger than $\sqrt{n}$. This implies that the number of factors is an even number, unless $\sqrt{n}$ happens to be an integer; in such case, $n = (\sqrt{n})^2$ is a perfect square.)

   Thus, to compute `count` is equivalent to checking whether n is a perfect square or not. (If so, we return 1. Otherwise, we return 0.) To solve the latter problem, our method is to find the largest integer $j \in [1, n]$ such that $j^2 \leq n$ by the binary search strategy. Then if $j^2 = n$, n must be a perfect square; else, we must have $j^2 < n < (j+1)^2$ so that n is not a perfect square.

   We start with the middle element m, $m = \lceil n/2 \rceil$, and check if $m^2 < n$. If so, we can deduce that $j < m$ so that we recursively search $[1, m-1]$. Otherwise, we can deduce that $j \geq m$ so that we recursively search $[m, n]$.

   After each step, the problem size (number of entries to be searched) is reduced by half, so that the search will stop after $O(\log n)$ steps. Thus, the running time is $O(\log n)$.

   (b) The correctness of this algorithm follows from the following statement (why?), which can be shown easily by induction (how?): After each step, the subarray to be searched must contain the target j.

3. (a) The correctness of this algorithm follows from the following statement (why?), which can be shown easily by induction (how?): After the kth phase, the k largest elements are in the correct positions.

   (b) Each swap can remove at most 1 inverted pair. Since the final output (sorted sequence) does not contain any inverted pairs, we must have: # of swaps $\geq$ # of inverted pairs.

   After a swap, an inverted pair formed by the swapping entries disappear; moreover, after a swap, no new inverted pair can be created. Thus each swap must correspond to an *original* inverted pair, so that we must have: # of swaps $\leq$ # of inverted pairs.

   In summary, # of swaps $=$ # of inverted pairs.

(c) The number of inverted pairs can be counted by a modified version of merge sort. Consider dividing the an array B into the left half B_{left} and the right half B_{right}. We say an inverted pair is *crossing* if one element is from B_{left} and the other is from B_{right}. We have two key observations.

Observation 1: The number of crossing inverted pairs remains the same even if B_{left} and B_{right} both are sorted (why?).

Observation 2: If B_{left} and B_{right} are sorted, counting the crossing inverted pairs can be done at the same time when we merge B_{left} and B_{right}. This can be done in linear time (how?).

Based on these observations, we shall design a function, called "sort-and-count" for any array B, which sorts B and count the inverted pairs in B as follows:

i. recursively sort-and-count B_{left};
ii. recursively sort-and-count B_{right};
iii. merge B_{left} and B_{right} and count the crossing inverted pairs;
iv. return the sum of the inverted pairs counted by (i), (ii), and (iii).

We can show by induction on i that the above algorithm correctly sorts any array and counts its inverted pairs, where i is the length of the array.

Let $T(n)$ denote the running time of the above algorithm. Thus, we have $T(n) = 2T(n/2) + \Theta(n)$, and hence $T(n) = \Theta(n \log n)$ by Master Theorem.