CS4311 Design and Analysis of Algorithms

Homework 1 (Suggested Solution)

1. (a) **Ans.** There exists $\varepsilon > 0$ such that

$$n^3 = O\big(n^{\log_2 9-\varepsilon}\big) \qquad \text{(by choosing } \varepsilon = \log\,(9/8)\text{)}.$$

By Master Theorem (case 1),

$$T(n) = \Theta\big(n^{\log 9}\big).$$

(b) **Ans.** There exists $\varepsilon > 0$ such that

$$n^3 = O\big(n^{\log_2 7+\varepsilon}\big) \qquad \text{(by choosing } \varepsilon = \log\,(8/7)\text{)}.$$

Also, there exists positive $c < 1$ such that

$$7(n/2)^3 \leq cn^3 \qquad \text{(by choosing } c = 7/8\text{)}, \quad \text{for every } n \geq 2.$$

By Master Theorem (case 3),

$$T(n) = \Theta\big(n^3\big).$$

(c) **Ans.** Set $m = \log n$. Then we have

$$T(2^m) = T(2^{m/2}) + m.$$

Next, let $S(m) = T(2^m)$, so we obtain

$$S(m) = S(m/2) + m.$$

By Master Theorem (case 3) or by recursion-tree method,

$$S(m) = \Theta(m).$$

So,

$$T(n) = T(2^m) = S(m) = \Theta(m) = \Theta(\log n).$$

(d) **Ans.** For ease of exposition, assume $n = 2^k$ for some positive integer k.

$$\begin{aligned}
T(n) &= \frac{1}{2}\,T\Big(\frac{n}{2}\Big) + n \\
&= \frac{1}{2}\left(\frac{1}{2}\,T\Big(\frac{n}{4}\Big) + \frac{n}{2}\right) + n \;=\; \frac{1}{4}\,T\Big(\frac{n}{4}\Big) + \frac{n}{4} + n \\
&= \frac{1}{4}\left(\frac{1}{2}\,T\Big(\frac{n}{8}\Big) + \frac{n}{4}\right) + \frac{n}{4} + n \;=\; \frac{1}{8}\,T\Big(\frac{n}{8}\Big) + \frac{n}{16} + \frac{n}{4} + n \\
&\;\vdots \\
&= \frac{1}{n}\,T(1) + \frac{1}{n} + \frac{4}{n} + \frac{16}{n} + \cdots + \frac{n}{16} + \frac{n}{4} + n \;=\; \Theta(n).
\end{aligned}$$

(e) **Ans.** Since

$$n/3 = \Theta\big(n^{\log_3 3}\big),$$

by Master Theorem (case 2),

$$T(n) = \Theta(n \log n).$$

2. **Ans.** We give two proofs based on two definitions of ω notation.
(First proof:) Because $f(n) \in \omega(g(n))$, by definition we know that for all positive c, $f(n) > cg(n)$ for all large enough n. Now, assume on the contrary that $f(n) \in O(g(n))$. This implies that there exists a positive c such that $f(n) \leq cg(n)$ for all large enough n. Contradiction occurs, so our assumption must not be true. In summary,

$$\text{if } f(n) \in \omega(g(n)),\ f(n) \notin O(g(n)).$$

(Second proof:) Because $f(n) \in \omega(g(n))$, by definition we know that

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0.$$

Now, assume on the contrary that $f(n) \in O(g(n))$. This implies that there exists a positive c such that $f(n) \leq cg(n)$ for all large enough n. Consequently,

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \geq \frac{1}{c} \neq 0.$$

Contradiction occurs, so our assumption must not be true. In summary,

$$\text{if } f(n) \in \omega(g(n)),\ f(n) \notin O(g(n)).$$

3. **Ans.** We can merge the three sorted parts into one sorted part by the method similar to merging two sorted parts. Let the three sorted parts be A, B, and C. The process is as follows:

Step 1: Create an empty list D.

Step 2: Compare the first element of A, B, and C, and determine the minimum. (If a sequence is empty, just ignore that sequence.)

Step 3: Remove the minimum from the sequence containing the minimum. Append the minimum to end of D.

Step 4: Repeat Steps 2 and 3 until A, B, and C are all empty.

Step 5: Output D.

Since each round we remove exactly one element from A, B or C, the running time of the merge method is $\Theta(n)$.

The time complexity of three-part merge sort can be described by the following recurrence:

$$T(n) = 3\ T(n/3) + \Theta(n).$$

By Master Theorem (case 2), $T(n) = \Theta(n \log n)$. Thus, two-part merge sort and three-part merge sort have the same asymptotic running time.

4. **Ans.** The outer loop iterates for n rounds. At the kth iteration, the inner loop runs for $\lfloor n/k \rfloor$ steps. Thus, the total running time is:

$$n + n/2 + n/3 + ... + 1 = n\,(1 + 1/2 + 1/3 + ... + 1/n) = \Theta(n \log n).$$

5. **Ans.** (Correctness:) We first use induction to show that after the ith round of `Greedy-Pick` and merge, the array B is sorted. The base case is that B is empty. Since B has no element, it is sorted. For the inductive case, we assume that B is sorted after the kth round. Then, at the $(k+1)$th round, `Greedy-Pick` will get an increasing sequence from A, so the sequence is sorted. And we know from lecture that after merging two sorted sequence, the result is also sorted. Thus, B is sorted after $(k+1)$th round. (This completes the proof of induction.)

   Further, `Greedy-Pick` will pick at least one element from A, and A is finite, so the the algorithm runs in finite number of rounds. In the end, B must contain all elements of A; also, B must be sorted by the above arguments. Thus, the algorithm is correct.

   (Worst-case input:) The worst case will occur when the input is a decreasing sequence. At each round of the algorithm, `Greedy-Pick` will only pick one element from A and merge to B. It takes $\Theta(n)$ rounds to complete the sorting. In each round, it takes $\Theta(n)$ time to perform `Greedy-Pick` and merge. Therefore, the total running time is $\Theta(n^2)$.[§]

6. (a) **Ans.** We can find out the missing integer as follows: First, we examine all the bits of the $\lceil n/2 \rceil$th element (i.e., the middle element) in the array. There are two cases:

   (Case 1:) If this element is $\lceil n/2 \rceil - 1$, then the missing element must be in the right part (subarray $A[\lceil n/2 \rceil + 1..n]$), whose value is between $\lceil n/2 \rceil$ and n.

   (Case 2:) Else, the missing element must be in the left part (subarray $A[1..\lceil n/2 \rceil]$), whose value is between 0 and $\lceil n/2 \rceil$.

   In either case, we are left with a subproblem of searching a missing number in a sorted array of k distinct elements, whose values are from $k+1$ contiguous integers. This is exactly the same as the original problem, except the problem size is *halved*.

   Thus, by using recursion, we can find the missing number in $\Theta(\log n)$ steps. As each step requires $\Theta(\log n)$ questions to find out all the bits of the middle element, total number of questions is $\Theta(\log^2 n)$.[‖]

   (b) **Ans.** We first create a list D with $n+1$ distinct integers, from 0 to n. We can find the missing integer as follows:

   For $i = 1, 2, \ldots, \log n$, perform Step 1 to Step 3:

   (Step 1:) Look at the ith least-significant-bit (LSB) of all integers.

   (Step 2:) If total number of items with ith LSB = 1 is not correct (this implies: ith LSB of missing number is 1), then remove all integers in D whose ith LSB is 0.

   (Step 3:) Else, remove all integers in D whose ith LSB is 1.

   In the end, only one number remains in D, and this number must be the missing number. The number of questions asked in the ith round is $\Theta(n/2^i)$, so that there are altogether $\Theta(n)$ questions.

[§] On the other hand, the running time of the sorting algorithm is $O(n^2)$, since there are $O(n)$ rounds, and each round takes $O(n)$ time. This confirms that the chosen input is indeed a worst-case input.

[‖] In fact, $O(\log n)$ questions are sufficient to solve this problem. The key observation is that in each step, we just need to check the *least significant bit* of the middle element, whose value is sufficient to help us reduce the problem size by half. Try to work this out at home!

7. **Ans.** We observe that if a bolt is smaller than some nut, this bolt cannot be the largest bolt. Similarly, if a nut is smaller than some bolt, this nut cannot be the largest nut. This observation gives us an easy way to find out the largest bolt and nut:

   (Step 1:) Compare a bolt and a nut.

   (Step 2:) Discard the smaller one. (If the same, discard an arbitrary one.)

   (Step 3:) If a nut is discarded, we find another nut from the remaining ones. Else, if a bolt is discarded, we find another bolt from the remaining ones. Repeat Step 2 unless all nuts or all bolts are discarded.

   After at most $2n-1$ comparisons, we must be left with either the largest nut or the largest bolt. Then, using a further $\Theta(n)$ comparisons, we can easily find its counterpart. Thus, total number of comparisons is $\Theta(n)$.