CSOR 4231: Analysis of Algorithms I, Problem Set 2

Jing Qian - jq2282

October 5, 2018

Problem 1

(a). $a = 4$, $b = 2$, so $f(n) = n^2 = n^{\log_b a}$. This is the case 2 in master theorem, so $T(n) = \Theta(n^2 \log n)$.

(b). $a = 3$, $b = 4$, so $f(n) = n = n^{\log_4 3+\epsilon}$. This is the case 3 in master theorem, so $T(n) = \Theta(n)$.

(c). 1) $a = 3, b = 2$, $n^{\log_b a} = n^{\log 3}$ is not polynomially smaller or larger than $f(n) = n \log^2 n$. So we could not use master theorem here.

2) We could use recursion-tree method for this problem. Suppose $m = \log n$ and $T(n)$ could be expressed as the function of m, $R(m)$. Then we have:

$$\begin{aligned} R(m) &= 3 \times R(m-1) + 2^m \times m^2 \\ &= 3^2 \times R(m-2) + 3 \times 2^{m-1} \times (m-1)^2 + 2^m \times m^2 \\ &\cdots \\ &\approx 3^{m-1} R(1) + \sum_{k=0}^{m-2} 3^k 2^{m-k} (m-k)^2 \\ &= 3^{m-1} R(1) + 3^m \sum_{i=2}^{m} (\frac{2}{3})^i \ i^2 \end{aligned} \tag{1}$$

Since $\int (\frac{2}{3})^x x^2 dx = (\frac{2}{3})^x \ x^2 \ \frac{1}{\ln \frac{2}{3}} - (\frac{2}{3})^x \ x \ \frac{2}{(\ln \frac{2}{3})^2} + (\frac{2}{3})^x \ \frac{2}{\ln \frac{2}{3}} + \text{const}$,

$$\begin{aligned} 3^m \sum_{i=2}^{m} (\frac{2}{3})^i \ i^2 &\approx 3^m [(\frac{2}{3})^m \ m^2 \ \frac{1}{\ln \frac{2}{3}} - (\frac{2}{3})^m \ m \ \frac{2}{(\ln \frac{2}{3})^2} + (\frac{2}{3})^m \ \frac{2}{\ln \frac{2}{3}} + c] \quad (c \text{ is a constant}) \\ &= 2^m \ m^2 \ \frac{1}{\ln \frac{2}{3}} - 2^m \ m \ \frac{1}{(\ln \frac{2}{3})^2} + 2^m \ \frac{2}{\ln \frac{2}{3}} + 3^m c \end{aligned} \tag{2}$$

where $3^m c$ is the leading order which grows fastest with n. Considering $R(1)$ takes constant

time, so:

$$\begin{aligned} R(m) &\approx 3^{m-1}R(1) + 3^m \sum_{i=2}^{m} (\frac{2}{3})^i \ i^2 \\ &= \Theta(3^{m-1}) + \Theta(3^m) \\ &= \Theta(3^m) \\ T(n) &= \Theta(3^{\log n}) = \Theta(n^{\log 3}) \end{aligned} \tag{3}$$

3) We could use substitution method to validate the answer above. $T(n) = \Theta(n^{\log 3})$ means $T(n)$ is both $O(n^{\log 3})$ and $\Omega(n^{\log 3})$. To prove $T(n) = \Omega(n^{\log 3})$, the substitution method requires us to prove that $T(n) \geq cn^{\log 3}$ for an appropriate choice of the constant $c > 0$. Then $T(n/2) \geq c(n/2)^{\log 3}$. Substituting into the recurrence,

$$\begin{aligned} T(n) &\geq 3c(\frac{n}{2})^{\log 3} + n\log^2 n \\ &= 3c\frac{n^{\log 3}}{3} + n\log^2 n \\ &= cn^{\log 3} + n\log^2 n \\ &\geq cn^{\log 3} \end{aligned} \tag{4}$$

where the last step holds for any positive c. So $T(n) = \Omega(n^{\log 3})$.

On the other hand, let's assume $T(n) \leq c_1(n^{\log 3} - n\log^2 n)$ for an appropriate choice of the constant $c_1 > 0$. Then $T(n/2) \leq c_1((n/2)^{\log 3} - (n/2)\log^2(n/2))$. Substituting into the recurrence,

$$\begin{aligned} T(n) &\leq 3c_1((n/2)^{\log 3} - (n/2)\log^2(n/2)) + n\log^2 n \\ &= c_1(n^{\log 3} - n\log^2 n) - [(\frac{c_1}{2} - 1)n\log^2 n - 3c_1 n\log n + \frac{3c_1}{2}n] \\ &\leq c_1(n^{\log 3} - n\log^2 n) \end{aligned} \tag{5}$$

for $c_1 > 2$. So $T(n) = O(n^{\log 3} - n\log^2 n)$ and then $T(n) = O(n^{\log 3})$.

Since $T(n)$ is both $O(n^{\log 3})$ and $\Omega(n^{\log 3})$, $T(n) = \Theta(n^{\log 3})$.

(d). We could use recursion-tree method for this problem.

$$\begin{aligned} T(n) &= 2T(n-3) + 1 \\ &= 2^2T(n-6) + 2 + 1 \\ &\cdots \\ &= 2^iT(n-3i) + \sum_{j=0}^{i-1} 2^j \end{aligned} \tag{6}$$

where $i = n/3$ and $n - 3i = n \mod 3$. So $T(n-3i)$ takes constant time and $T(n) = 2^i\Theta(1) + 2^i - 1 = \Theta(2^i) = \Theta(2^{(n/3)})$.

(e). We could use recursion-tree method for this problem. Suppose $m = \log n$ and $T(n)$ could be expressed as the function of m, $R(m)$. Then we have $R(m) = R(m/2) + 1$. Use master theorem, $a = 1, b = 2$, $f(n) = 1 = n^{\log_2 1}$. $R(m)$ belongs to case 2 and hence $R(m) = \Theta(\log m)$. So $T(n) = \Theta(\log \log n)$.

(f). We could use substitution method for this problem. Let's assume $T(n) \leq c_1 n$ for an appropriate choice of the constant $c_1 > 0$.

$$\begin{aligned} T(n) &= T(n/2) + T(n/3) + n \\ &\leq \frac{c_1 n}{2} + \frac{c_1 n}{3} + n \\ &= \frac{5c_1 + 6}{6} n \end{aligned} \tag{7}$$

If we choose $c_1 \geq 6$, $T(n) = \frac{5c_1+6}{6} n \leq c_1 n$. So $T(n) = O(n)$.

Then let's assume $T(n) \geq c_2 n$ for an appropriate choice of the constant $c_2 > 0$.

$$\begin{aligned} T(n) &= T(n/2) + T(n/3) + n \\ &\geq \frac{c_2 n}{2} + \frac{c_2 n}{3} + n \\ &= \frac{5c_2 + 6}{6} n \end{aligned} \tag{8}$$

If we choose $c_2 \leq 6$, $T(n) = \frac{5c_2+6}{6} n \geq c_2 n$. So $T(n) = \Omega(n)$.

Since $T(n)$ is both $O(n)$ and $\Omega(n)$, $T(n) = \Theta(n)$.

Problem 2

1. The probability that $A[1] > A[2]$ is 1/2.

The number of random permutations of array A equals to the number of choosing a pair of $(A[1], A[2])$ times the permutation of $(A[1], A[2])$ and times the permutation of the rest $n-2$ elements: $n! = \binom{n}{2} \times 2! \times (n-2)!$. For every pair of $(A[1], A[2])$, there are two permutations with equal probability: one is $A[1] > A[2]$ and the other is $A[1] < A[2]$. So the probability that $A[1] > A[2]$ is 1/2.

2. The number of choosing a pair of elements $(A[i], A[j])$ from n elements where $1 \leq i < j \leq n$ is $\binom{n}{2}$. From previous discussion, the probability of $A[i] > A[j]$ is 1/2 for each pair. So the expected number of pairs of indices i, j such that $1 \leq i < j \leq n$ and $A[i] > A[j]$ is $\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4}$.

Problem 3

To solve this problem, we could build an array of distance between $A[1]$ and other elements and select the k smallest elements in the new array. Considering the distance is an absolute value and there maybe a tie, we build a 2-d array B in which every element has two parts

$B[i] = (|A[i+1] - A[1]|, A[i+1])$ for i from 1 to $n-1$. Building array B takes $\Theta(n)$ time because we pass though array A once. Then we random-select k elements with the smallest first part $|A[i+1] - A[1]|$. As we learnt in class, this random selection part takes linear time, which is $O(n)$. Then we print the second part of the selected k elements, which are the closest in value to the first element of A. This step takes $\Theta(k) = O(n)$ time. Since each part in this algorithm takes linear time at most, we could find the k elements of A that are closest in value to the first element of A in $O(n)$ time.

Problem 4

1. We start the discussion from a simple case and suppose $k = 2^m$. Then we could solve this problem recursively (use the rank i to represent the i-rank element in the array):

1). Find $\lceil \frac{n}{2} \rceil$ from $[1, n]$. Random selection takes linear time, $O(n)$.

2). Find $\lceil \frac{n}{4} \rceil$ from $[1, \lceil \frac{n}{2} \rceil)$ and $\lceil \frac{3n}{4} \rceil$ from $(\lceil \frac{n}{2} \rceil, n]$. $O(2 * \frac{n}{2}) = O(n)$.

3). Find $\lceil \frac{n}{8} \rceil$ from $[1, \lceil \frac{n}{4} \rceil)$, $\lceil \frac{3n}{8} \rceil$ from $(\lceil \frac{n}{4} \rceil, \lceil \frac{n}{2} \rceil)$, $\lceil \frac{5n}{8} \rceil$ from $(\lceil \frac{n}{2} \rceil, \lceil \frac{3n}{4} \rceil)$ and $\lceil \frac{7n}{8} \rceil$ from $(\lceil \frac{3n}{4} \rceil, n]$. $O(4 * \frac{n}{4}) = O(n)$.

$\cdots$

m). Find $\lceil \frac{n}{2^m} \rceil$ from $[1, \lceil \frac{n}{2^{m-1}} \rceil)$, $\cdots$, $\lceil \frac{(2^m-1)n}{2^m} \rceil$ from $(\lceil \frac{(2^{m-1}-1)n}{2^{m-1}} \rceil, n]$. $O(2^{m-1} * \frac{n}{2^{m-1}}) = O(n)$.

So the total running time is $T(n) = m\ O(n) = O(n \log k)$.

2. For the general case when k my not equal to 2^m, we could do the recursion similarly though this recursion tree is a little different:

1). Find $\lceil \frac{\lceil k/2 \rceil n}{k} \rceil$ from $[1, n]$. $O(n)$.

2). Find $\lceil \frac{\lceil k/4 \rceil n}{k} \rceil$ from $[1, \lceil \frac{\lceil k/2 \rceil n}{k} \rceil)$ and $\lceil \frac{\lceil 3k/4 \rceil n}{k} \rceil$ from $(\lceil \frac{\lceil k/2 \rceil n}{k} \rceil, n]$. $O(n)$.

$\cdots$

l). Find $\lceil \frac{\lceil \frac{k}{2^l} \rceil n}{k} \rceil$ from $[1, \lceil \frac{\lceil \frac{k}{2^{l-1}} \rceil n}{k} \rceil)$, $\cdots$, $\lceil \frac{\lceil \frac{(2^l-1)k}{2^l} \rceil n}{k} \rceil$ from $(\lceil \frac{\lceil \frac{(2^{l-1}-1)k}{2^{l-1}} \rceil n}{k} \rceil, n]$.

Here the height of the recursion tree is $l = \lceil \log k \rceil$.

When $k \neq 2^m$, the running time for every layer is still $O(n)$ except the final layer where some leaves are missing. For example, if $k = 3$, in layer 2, the leaf $\lceil \frac{\lceil 3k/4 \rceil n}{k} \rceil = n$ does not exist because it is larger than $\lceil \frac{(k-1)n}{k} \rceil$. So the running time for the last layer is smaller than $O(n)$, but still could be represented by $O(n)$. So $T(n) = l\ O(n) = O(n \log k)$.

In general, the time complexity is $T(n) = O(n \log k)$.

Problem 5

1. According to the lecture, in every partition we separate the array or subarray into two parts: $\leq x$ and $> x$ with respect to the pivot x. When partition a n-length array, if all elements are equal, the $\leq x$ part would have $n-1$ elements and the $> x$ part would be

empty. Since the partition takes linear time, we have:

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &\cdots \\ &= T(1) + \sum_{i=2}^{n} i \\ &= \Theta(1) + \frac{(n+2)(n-1)}{2} \\ &= \Theta(n^2) \end{aligned} \tag{9}$$

So the expected running time is $\Theta(n^2)$.

2. Following is the pseudocode for the modified PARTITION routine that partitions subarrays with respect to the pivot x into 3 parts consisting of elements $< x$, $= x$, $> x$ respectively:

MODIFIED_PARTITION

```
1  PARTITION(A, p, r)
2          x = A[r]
3          i = p - 1
4          l = p - 1
5          for j = p to r
6                  if A[j] < x
7                          i = i + 1
8                          l = l + 1
9                          exchange A[i] with A[l]
10                         exchange A[i] with A[j]
11                 else if A[j] = x
12                         l = l + 1
13                         exchange A[l] with A[j]
14         return (i, l)
```

This PARTITION returns (q_1, q_2) where elements in the final subarray $A[p \cdots q_1]$ are smaller than the pivot x, elements of $A[q_1 + 1 \cdots q_2]$ are equal to x and elements of $A[q_2 + 1 \cdots r]$ are greater than x.

Similar to the loop invariants in original PARTITION in the book, for any array index k, we have:

1. If $p \leq k \leq i$, then $A[k] < x$.
2. If $i + 1 \leq k \leq l$, then $A[k] = x$.
3. If $l + 1 \leq k \leq j$, then $A[k] > x$.

The indices between $j + 1$ and $r - 1$ are not covered by any of these three cases, so we don't know the relationship between these values and the pivot x. Before the first iteration, $i = p - 1, l = p - 1$, no value lies between p and i, or $i + 1$ and l, or $l + 1$ and j. So the loop invariants hold.

Depending on the test in line 6 and test 11, there are three possible outcomes: If $A[j] < x$, we increment i and l both by one, and swap A[i] with A[l], A[i] with A[j]. Because of the swaps, we have now $A[i] < x, A[l] = x, A[j] > x$. The loop invariants hold. If $A[j] = x$, we only increment l by one and swap A[l] with A[j]. Because of the swap, we have now $A[l] = x$. Since we don't change i, we do not need to worry about that subarray. The loop invariants hold. If $A[j] > x$, we don't do anything. i and l are not changed, so first two loop invariants hold. Loop invariant also holds

At termination, $j = r$, $A[j] = x$. So we increment l by one and swap A[l] with A[j]. The output i is taken as q_1 and l is taken as q_2. So all elements in $A[p \cdots q_1] = A[p \cdots i]$ are less than x, all elements in $A[q_1 + 1 \cdots q_2] = A[i + 1 \cdots l]$ are equal to x and all elements in $A[q_2 + 1 \cdots r] = A[l + 1 \cdots r]$ are greater than x. This is the result we want. In addition, since we only exchange the position of elements, this algorithm is in place.

3. As we discussed in question 1, even using RANDOMIZED-QUICKSORT, if all elements are equal, the running time would be $\Theta(n^2)$. We could add randomness to the algorithm developed in question 2, and get a $O(n \log n)$ method for any input array.

The RANDOMIZED-QUICKSORT and RANDOMIZED-PARTITION are the same as those in the book while we call the modified PARTITION from question 2 in the RANDOMIZED-PARTITION. Because we randomly choose the pivot element, we expect the split of the input array to be reasonably well balanced on average. That is to say, in every recursion, the size of $A[p \cdots q_1]$ and $A[q_2 + 1 \cdots r]$ are close. Since there is a third subarray $A[q_1 + 1 \cdots q_2]$ in which all elements are equal to x, we do not to need sort it. So the size of subproblem is smaller than half of the original problem. According to the lecture, RANDOMIZED-QUICKSORT without duplicate elements takes $\Theta(n \log n)$ time because of the smaller size of subproblems. So RANDOMIZED-QUICKSORT with duplicate elements takes $O(n \log n)$ time.

In conclusion, for any arbitrary input array, this RANDOMIZED-QUICKSORT runs in $O(n \log n)$.

Problem 6

a). If more than $n/2$ chips are bad, we could use G to represent the set of good chips, B to represent the set of bad chips with the same number of chips as the number of G, and R to represent the rest bad chips. So the reports from good set G and bad set B are symmetric, which make them indistinguishable. That is to say, for every report from a good chip g in the good set G, we could always find a bad chip b from the bad set B to give an opposite report. Then we are unable to judge based on these reports. In conclusion, if more than $n/2$ chips are bad, the professor is unable to decide which chips are good with this pairwise test.

b). We could use following strategy of $\lfloor n/2 \rfloor$ pairwise tests to solve this problem while reduce the problem size by half in every recursion:

1) Separate all chips randomly into pairs of two chips. The number of pairs is $\lfloor n/2 \rfloor$.

2) Discard chips based on the outcomes of $\lfloor n/2 \rfloor$ tests. If both chips report good, then we randomly keep one and discard the other. If both chips report bad or only one chip reports bad, we discard both chips. For every test of two chips, we discard at least one chip. So for

$\lfloor n/2 \rfloor$ tests, we discard at least $\lfloor n/2 \rfloor$ chips. That is to way, this $\lfloor n/2 \rfloor$ tests are sufficient to reduce the problem to one of nearly half the size.

3) Repeat steps 1) and 2) until the number of chips $n \leq 2$. The $n \leq 2$ chips are good. If n is odd before or during the recursion, the rest one chip after paring is added to the survivors of $\lfloor n/2 \rfloor$ tests and taken into next recursion. The parity would not affect the correctness and time complexity of this algorithm.

Now prove the correctness of this algorithm: Since more than $n/2$ of the chips are good in the initial state, there are more good chips than bad chips before loops. The fact that good chips are more than bad chips is a loop invariant. Then we analyze whether the loop invariant holds after each iteration. There are three possible outcomes of $\lfloor n/2 \rfloor$ pairwise tests of the n chips: If both chips report good, the conclusion we can get is both chips are actually good or both are bad. If we randomly keep one and discard the other, the probability we discard a good chip is 1/2. If both chips report bad or only one chip reports bad, the conclusion we can get is at least one is bad. Since we discard both chips, we may discard one bad chip or discard two bad chips. So after the $\lfloor n/2 \rfloor$ pairwise tests, we discard more (or same amount of) bad chips than good chips. Since there are more good chips than bad chips before this iteration, we will still have more good chips after this iteration. Finally, when the recursions end, there is only single chip (or 2 chips), it must be good. So the loop invariant still holds because now the number of bad chips is zero. So this algorithm is correct.

c). We could use the algorithm in question b) to find a single good chip, use this good chip to test all other chips and find all good chips.

According to the algorithm in question b), in a recusion starting with n chips, we do $n/2$ pairwise tests (here the difference due to the parity of n is negligible in time complexity anaylsis) and reduce the problem size to at most $n/2$. So we have:

$$\begin{aligned} T_1(n) &= T_1(n/2) + n/2 \\ &= \Theta(n) \end{aligned} \tag{10}$$

where the case 3 of master theorem applies.

Since we use $\Theta(n)$ pairwise tests to find a single good chip and $n-1$ pairwise tests between this good chip and all other chips to find other good chips, we could identify all good chips with $\Theta(n)$ pairwise tests as long as more than $n/2$ of the chips are good.