Fundamental Algorithm Homework 3 Solution

by Xing Xia

1. Exercise 7.4-5, page 159 (5+10+10+5 points)

Solution:

(i) What we can say about the ordering in A is that: the output array A can be divided into subarrays, each of size < k. The ith subarray has entries which are less than the entries in the (i+1)th subarray, for any i > 0. However, within each subarray, we don't assume any ordering on the entries.

Remark: we don't know where these subarrays' boundaries are.

(ii) Please read the method used in textbook page 156-158 for comparison.

Let $Xij = I\{z_i$ is compared to $z_j\}$

If the distance between i and j, |i-j|, is >k/3, then $Pr\{z_i$ is compared to $z_j\}$ can be bounded as in (7.3) in textbook. When j – i <= k/3, let R_{ij} be the set $\{z_h: h-j <= k/3\}$, let L_{ij} be the set $\{z_h: i-h <= k/3\}$.

|----------|-----|----------|

L_{ij} i j R_{ij}

Let EL_{ij} = the event i or j is picked before any element in L_{ij}.

Let ER_{ij} = the event i or j is picked before any element in R_{ij}.

It is easy to see

$Pr(EL_{ij}) = O(1/k)$, $Pr(ER_{ij}) = O(1/k)$, $Pr(EL_{ij} \cup ER_{ij}) <= Pr(EL_{ij}) + Pr(ER_{ij}) = O(1/k)$

Since $\{Xij=1\}$ is a subset of $EL_{ij} \cup ER_{ij}$, $E[Xij] = O(1/k)$.

$$E[X] = \sum_{i=1}^{n-1} \left(\sum_{j=1}^{n-i} 2/(j+1) - \sum_{j=1}^{k-1} (2/(j+1) - 1/k) \right) = \sum_{i=1}^{n-1} (\ln n - \ln k + O(1)) = O(n\log(n/k))$$ **Q.E.D.**

Alternative method: Let T(n) be the running time of k-truncated quicksort for n elements. Then according to the description of k-trucated quicksort, T(n) = 0 for any n < k. For any n >= k

T(n) = n + [((T(0)+T(n-1)) + (T(1) + T(n-2)) + … + (T(n-1)+T(0))]/n

= n + (2/n)[T(k) + T(k+1) + … + T(n-1)]

How to solve T(n)? Do some changes to the above equation.

nT(n) = n*n + 2[T(k) + T(k+1) + … + T(n-1)] (1)

substitue n by n-1 in (1), we have

(n-1)T(n-1) = (n-1)* (n-1) + 2[T(k) + T(k+1) + … + T(n-2)] (2)

(1)– (2),

nT(n) – (n-1)T(n-1) = 2n – 1 + 2T(n-1) =>

T(n)/(n+1) – T(n-1)/n = (2n-1)/n(n+1) =>

T(n)/(n+1) – T(n-1)/n = 2/n – 3/n(n+1)

Define S(n) = T(n)/(n+1), then

S(n) – S(n-1) <= 2/n (3)

Substitute n by n-1, n-2, …k in (3), we have

S(n-1) – S(n-2) <= 2/(n-1)

…

S(k) – S(k-1) <= 2/k

Add the above n-k+1 inequations together, we have
$S(n) < 2(1/n + 1/(n-1) + \ldots + 1/k)$
When n and k are infinite, we know that
$1/1 + 1/2 + \ldots + 1/n = \log n$ and $1/1 + 1/2 + \ldots + 1/k = \log k$
So $S(n) = T(n)/(n+1) <= 2(\log n - \log k) = 2\log(n/k)$, i.e. $T(n) = O(n\log(n/k))$ **Q.E.D.**

(iii) The key idea is that in insertion sort, each element is moved at most k times in total. Why? The reason is that each element can only move within the boundary of its own subarray, and we know from part (i) that the size of each subarray is less than k.

(iv) Consider $T(n) = C_1 n\log(n/k) + C_2 kn$ as a function of k instead of n. You can think of n as a constant. To get the minimum value of T(k), the derivative of T(k), i.e. T'(k) should be 0, i.e. $T'(k) = -C_1 n/k + C_2 n = 0$, so $k = C_1/C_2$. **Q.E.D.**

2. Exercise 9.1-1, page 185 (10 points)
Solution:
Set up a binary tree T with n leaves and height [lgn]. Place the numbers in leaves of T and use it as a "tournament" to compute the largest. It's easy to see we need n–1 comparisons to get the largest number. (Why? Image a tournament, in each round, exactly one team is eliminated, we eliminate n-1 teams in total to get the champion). Now, we can search along the path traced by the largest value to find which [logn] elements ever compared with the largest number, since the second largest number must be among those [logn] numbers. To find the "largest number among those [logn] numbers", we need [logn] –1 comparisons. So the total number of comparison is (n-1) + ([logn] – 1) = n + [logn] – 2.
Note: [m] is the smallest that is no less than m. **Q.E.D.**

3. Exercise 9.3-9, page 193 (10 points)
Solution:
For simplicity, assume the y-coordinates of points are distinct.
Generally, for any n, if n is even, we should put the main pipeline in the position such that there are exactly n/2 points above it and n/2 points below it. Otherwise, suppose there are n_a points above the main pipeline and n_b points below it, where n_a is not equal to n_b. For example, if $n_a > n_b$, we can move the main pipeline up gradually until there are $n_a - 1$ point above it and $n_b - 1$ points below it. It's easy to see the total length of spurs is decreased, so this is a contradiction that we have chosen the optimal placement.
If n is odd, we should put the main pipeline right across the median point, i.e., there are exactly (n-1)/2 points above it and (n-1)/2 points below it.
So, the problem is reduced to find the median number of an array, which can be determined in linear time (see proof in the textbook). **Q.E.D.**

4. Median analysis (10+20 points)
Solution:
(i) To show $T(n) = \Theta(n)$, we need to show two facts: $T(n) = O(n)$ and $T(n) = \Omega(n)$. $T(n) = \Omega(n)$ is trivial, since $T(n) = n + T(c_1 n) + T(c_2 n)$. Now we use an induction argument to

show that T(n) = O(n). It's easy to show T(n) >= 0. All we need is to show T(n) <=Cn for all n > n0.
Without loss of generality, suppose T(n) <= C for all n <= n_0. Note: C and n_0 are arbitary but given to us. So pick K_1, such that T(n) <= K_1n, for n = n_0.
Fix any m. Suppose T(n) <= Kn for any n such that m > n >= 1 (K is not determined yet). Let's consider the situation when n = m.
T(m) = m + T(c_1m) + T(c_2m)
<= m + Kc_1m + Kc_2m (by assumption and the fact c_1m<m, c_2m<m)
= (1+ K (c_1+c_2))m
To let T(m) <= Km, we need to pick K such that 1+ K(c_1+c_2) < K, since c_1+c_2<1, we have K >= 1/(1-(c_1+c_2)). Let K_2 = 1/(1-(c_1+c_2)). We need to pick K = max(K_1, K_2).
By induction, we know T(n) <= Kn for all n >= n_0. So T(n) = O(n).

(ii) First we guess T(n) = Θ(nlogn). Why? If you set c_1=1/2 and c_2=1/2, and use master theorem to get the answer. So this is a reasonable initial guess.
To prove T(n) = Θ(nlogn), we need to prove two facts: T(n) = Ω(nlogn) and T(n) = O(nlogn).
First, we show T(n) = Ω(nlogn).
Suppose T(n) >= K_1n, for any m > n >= n_0. Now we use induction to show when n = m, T(n) >= K_1n.
T(m) = m + T(c_1m) + T(c_2m)
>= m + c_1m log(c_1m) + $c_2$mlog(c_2m) (by assumption and the fact c1m<m c_2m<m)
= m + (c_1+c_2)mlogm + $c_1$mlogc_1 + $c_2$mlogc_2
> mlogm (since c_1+c_2=1)
By induction, we know T(n) >= K_1n for all n>=n_0. So T(n) = Ω(nlogn).
Similarly we can show T(n) = O(nlogn). So T(n) = Θ(nlogn). **Q.E.D.**

5. Exercise 11.3-2, page 236 (10 points)
Implementing the division method for a radix-128 number.
Solution:
Let $N_r = x_r x_{r-1} \ldots x_2 x_1$ be the string of r characters, treated as a radix-128 number. If we compute N_r mod m without any tricks, it's easy to see the space needed to store N_r is Θ(r), not a constant number of words of storage. Now let's see how to compute N_r mod m in a constant number of words of storage.

$$N_r = \sum_{i=1}^{r} x_i(128)^{i-1} = 128N_{r-1} + x_1, \text{ where } N_{r-1} = \sum_{i=2}^{r} x_i(128)^{i-2}$$

N_r mod m = (128*N_{r-1} + x_1) mod m
= (128*(N_{r-1} mod m) + x_1) mod m
^^^^^^^^^^ this is the trick
Here we use the fact that (a*b) mod m = ((a mod m) * b) mod m.
The above equation implies that to compute N_r mod m, we can first compute N_{r-1} mod m; Similarly to compute N_{r-1} mod m, we can compute N_{r-2} mod m, …The idea is to keep all the intermediate results (N_{i-1} mod m) in the same space. In step (r-i) of computation, we store the value of N_i mod m, which only need one 32-bit word, and we use this value to compute N_{i+1} mod m. So all the bits we need to compute Nr are 7(for constant 128) +

32(for N_i mod m) + 7(for x1) + 32(for m), which obviously is a constant number of words. **Q.E.D.**

6. Perfect Hashing (Section 11.5 page 245) (20 + 10 + 15 points)

Solution:

(i) Data structure (please refer to figure 11.6 in page 246 of the textbook)

Basically, we use an array T for the primary table, and for each secondary table, use a separate array S. i.e.

```
entry{
        int m;
        int a;
        int b;
        int[] S;
}
T: array[n] of entry;
```

Hash search algorithm:

For a key k,

(1) Compute h(k), say, h(k) = i, where h is the primary hash function.
(2) Retrieve m_i, a_i, b_i.
(3) Compute $h_i(k) = (a_ik+b_i) \bmod m_i$

If $T[i].S[h_i(k)] = k$, return "found"; else return "not found"

(ii) According to the hints given by professor Yap at http://www.cs.nyu.edu/~yap/classes/funAlgo/01f/hw/hw3/hints.html, we can find a hash function h* that maps U to $\{0, 1, \ldots n^2\}$ such that h*(x) is different from h*(y) for all distinct elements x, y in K, i.e. h* is perfect for K.

We add three arrays, C, L and S, where C is used to store all the ASCII strings of K; L is used to store the lengths of the string where L[i] is the length of the i_{th} string; S is used to store the beginning position of each string in array C. e.g. (Note that L is somewhat redundant, since we can compute the length of some specific string by the information give by array S. We just want to use L to make the solution clear).

K = {"this", "that", "ok"}

C = ['t', 'h', 'i', 's', 't', 'h', 'a', 't', 'o', 'k']

L = [4, 4, 2]

S = [1, 5, 9]

Now, the i_{th} key (i=1,2,..n) has length L[i] and is stored in the subarray of C[S[i], …, S[i] + L[i] – 1].

The idea is you now use h*(k) instead of k (k is a ASCII string) as the parameter of primary hash function h, and you store the index of the k in array C (e.g. In the example above, the index of "this" is 1, the index of "that" is 2) instead of k itself into the secondary hash table. Why? Suppose given a query string k, after you compute the hash functions: h*, h and h_i, you find the entry in the secondary hash table where k should be stored. Suppose I (the index of some key) is now stored in that entry, all you need to do is to compare the subarray C[S[i], …, S[i] + L[i] – 1] and query string k.

(iii) Question A: how many times do you expect to do this test until it is passed?

Apply Markov's inequality, for random variable X, $Pr\{X>=t\} <= E[X]/t$, we have

$$Pr\{\sum_{i=0}^{n-1}|h^{-1}(i)|^2 >= 4n\} <= 1/2$$

According to the hint given by professor Yap, the expected number of times to do the test is

$$<= 1/Pr\{\sum_{i=0}^{n-1}|h^{-1}(i)|^2 >= 4n\} = 2.$$

So, the times you expect to do this test is 2.

Question B: Give more detils about how you would implement this test, and say how much time is needed for each test.

```
Repeat
        T[i] = 0, for 0 <= i <= n-1.
        For any k in K, T[h(k)]++;
        S=0;
        For i = 0 to n-1
                S += T[i]^2;
Until S <4n;
```

Obviously, the time for each test is O(n). **Q.E.D.**