Illinois Institute of Technology
Department of Computer Science

Homework 2 Solutions

CS 430 Introduction to Algorithms
Spring Semester, 2016

1. **Problem 1**

   Solution:

   (a) The expected value of $\frac{1}{n}\sum_{i=1}^{n}|\pi_i - i|$ is $\frac{1}{n}\sum_{\pi_i=1}^{n}|\pi_i - i|$ averaging over i. $\frac{1}{n}\sum_{\pi_i=1}^{n}|\pi_i - i| = \frac{1}{n}[\sum_{\pi_i=1}^{i}(i-\pi_i) + \sum_{\pi_i=i+1}^{n}(\pi_i - i)] = \frac{1}{n}(\sum_{\pi_i=1}^{i-1}\pi_i + \sum_{\pi_i=1}^{n-i}\pi_i) = \frac{1}{2n}[i(i-1)+(n-i)(n-i+1)]$. Averaging over i gives $\frac{1}{2n^2}\sum_{i=1}^{n}[2i^2 - 2(n+1)i + n^2 + n] = \frac{1}{3n}(n^2-1) = \frac{n}{3} - \frac{1}{3n}$.

   (b) $\sum_{i=1}^{n}$ represents the time or the number of steps required to sort the array. π_i describes the random permutation, $|\pi_i - i|$ describes the distance of an element from its sorted position. So the sum of all these elements divided by the total number of elements gives the average distance an item will move during sorting.

   (c) The permutation given in part (a) of the problem represents algorithms which perform adjacent interchanges. So to find the find the total time taken we multiply the average distance travelled by an element with the number of elements or n in this case. Hence we get $n \times \frac{1}{3n}(n^2-1) = \frac{1}{3}(n^2-1)$ or $\Theta(n^2)$. Thus any sorting algorithm that uses only adjacent interchanges is doomed to take $\Omega(n^2)$ average time.

2. **Problem 6.4-3 on page 160**

   Solution: This problem can be deceiving. Although both cases appear to be very different, one initially appears to be a best case scenario, the other a worst case scenario, they both take $\Theta(n \log n)$ time.

   In the case of a list of items sorted in decreasing order, this is a best case scenario since BUILD-MAX-HEAP does not need to change anything, but it still runs through half of the list in the process of verifying that the list is a heap, thus it still takes $\Theta(n)$ time even though the list is a max-heap. However, the real problem is the $n-1$ calls to MAX-HEAPIFY. Each time it is called, one of the smallest values in the heap has just been placed at the top of the heap, so it must "float" down toward the bottom of the heap, which takes $\Theta(\log n)$ time.

   For a list sorted in increasing order, BUILD-MAX-HEAP must convert the list into a max-heap, which takes more time than when the items were sorted in decreasing order, but the difference is just by a constant as it still takes just $\Theta(n)$ time to build the heap. At this point it is difficult to analyze the number of comparisons exactly, because the list has been re-arranged by BUILD-MAX-HEAP. However, each call to MAX-HEAPIFY takes the worst-case time of $\Omega(\log n)$, so HEAPSORT in this case takes $\Omega(n \log n)$.

3. **Problem 7-4 on page 188**

   Solution:

   (a) The only difference between TAIL-RECURSIVE-QUICKSORT and QUICKSORT is that instead of recursively call itself on the right subarray in the form $QUICKSORT(A, q+1, r)$, it increases q by 1. For TAIL-RECURSIVE-QUICKSORT, notice that if $q+1 = r$, it means $A[q] < A[r]$ and there is no element in between. The last two elements have already been correctly sorted. If

$q+1 < r$, it will just recursively call $PARTITION(A, q+1, r)$ and $TAIL-RECURSIVE-QUICKSORT(A, q+1, r)$ in the next recursion. This is equivalent to recursively sort the right subarray. In both cases, $TAIL-RECURSIVE-QUICKSORT$ will correctly sort the right subarray just as done by quick sort.

(b) Consider an array that is already sorted increasing order. In this case each call to partition will partition around the last element in the current list which creates a right sublist that consists of just the last element in the list and the left sublist has the first $n-1$ elements. Since TRQ always calls itself recursively on the left sublist and the list is sorted in increasing order, with each recursive call, the length of the left sublist only decreases by one for each recursive call. This means that we will have $\Theta(n)$ consecutive recursive calls, which will result in a stack depth of $\Theta(n)$.

(c) Notice that the problem with $TAIL-RECURSIVE-QUICKSORT$ is that it may recursively call itself on the larger portion of the partitioned array. We can modify it such that $TAIL-RECURSIVE-QUICKSORT$ will always call itself on the smaller portions first. The algorithm is as follows:

Algorithm 1 TAIL-RECURSIVE-QUICKSORT(A,p,r)

```
while $q < r$ do
    q = PARTITION(A,p,r)
    if $q < \frac{p+r}{2}$ then
        TAIL-RECURSIVE-QUICKSORT(A,p,q-1)
        $q = q + 1$
    else
        TAIL-RECURSIVE-QUICKSORT(A,q+1,r)
        $r = q - 1$
    end if
end while
```

(d) The average stack depth is similar to that of quicksort. We have the recursive relation $S(n) = 1 + \frac{1}{n}\sum_{i=1}^{n-1} S(i)$. Given n_0 initial values $t_0, t_1, ..., t_{n}.$, we rewrite it as

$$t_n = a + \frac{1}{n}\sum_{i=1}^{n-1} t_i, \tag{1}$$

where $n > n_0$. Substituting $n-1$ for n, we have

$$(n-1)t_{n-1} = a(n-1) + \sum_{i=1}^{n-2} t_i \tag{2}$$

Subtract 2 from 1, we have $nt_n - (n-1)t_{n-1} = an - a(n-1) + t_{n-1}$. As $nt_n - nt_{n-1} = a$, we have $t_n - t_{n-1} = \frac{a}{n}$. Thus $\sum_{i=1}^{n} t_n - t_{n-1} = \sum_{i}^{n} \frac{a}{i} = a\ln n + O(1)$, which means $t_n - t_0 = a\ln n + O(1)$. This indicates $t_n = a\ln n + O(1) = \Theta(\lg n)$.

4. **Problem 8-3(a) on page 206**

Solution: We can use a revised version of radix sort to do this. Suppose the integer have at most k digits. From $i = 1$ to k, we sort the integer according to the ith digit of each string. If we find a integer has no digit on the i-th position, we considered the integer sorted and do not include it in the next loop.

Running time analysis: suppose that d_i is the number of integer that have the ith digit. We have $\sum_{i=1}^{k} d_i = n$. For loop i, there are d_i digits taking part in the sorting. Suppose the stable sorting we use took less than $C(d_i + D)$ time. Note C is a constant and D is the possible value of each digits, which is also a constant. The sorting algorithm thus take less than $\sum_{i=1}^{k} \cdot C(d_i + D)$ time according to lemma 8.3 on page 198. We have

$$\sum_{i=1}^{k} \cdot C(d_i + D) = \sum_{i=1}^{k} \cdot C \cdot d_i + \sum_{i=1}^{k} \cdot D = C \sum_{i=1}^{k} d_i + Dk \le Cn + Dk = O(n)$$

5. **Problem 5**

   Solution:

   (a) The corrupted code works sometimes. If the largest element is continuously selected as the pivot in line 3 and it is not the ith smallest element, then the resulting subarray $A[p..q]$ in line 8 will be the same as the original array $A[p..r]$ and hence the code will never terminate.

   (b) As discussed above, the worst-case running time is infinite since the code will not terminate.

   (c) The best-case behavior occurs when the ith smallest element is selected as the pivot element in the first partition. Thus the best-case running time is $\Theta(n)$ that is the time of RANDOMIZED-PARTITION.

   (d) Similar to the analysis of the original code on page 217 of CLRS3, we have

$$T(n) \le \sum_{k=1}^{n} X_k \cdot T(\max(k, n-k)) + O(n).$$

Taking expected values, we have

$$E[T(n)] \le \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max(k, n-k))] + O(n).$$

Obviously,

$$\max(k, n-k) = \begin{cases} k & \text{if } k \ge \lceil n/2 \rceil, \\ n-k & \text{if } k < \lceil n/2 \rceil. \end{cases}$$

If n is odd, each term from $T(\lceil n/2 \rceil)$ up to $T(n-1)$ appears twice and $T(n)$ appears once in the summation. If n is even, each term from $T(\lceil n/2 \rceil + 1)$ up to $T(n-1)$ appears twice, and $T(\lceil n/2 \rceil)$ and $T(n)$ appear once. Thus we have

$$E[T(n)](1 - 1/n) \le \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} E[T(k)] + O(n).$$

We must prove that $E[T(n)] \le cn$ for some constant c. Let a be the constant in the $O(n)$. Using the same inductive hypothesis as on page 218, we have

$$\begin{aligned}
E[T(n)](1-1/n) &\leq \frac{2}{n}\sum_{k=\lceil n/2\rceil}^{n-1} ck + an \\
&= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2\rceil -1} k\right) + an \\
&= \frac{2c}{n}\left(\frac{n(n-1)}{2} - \frac{\lceil n/2\rceil(\lceil n/2\rceil - 1)}{2}\right) + an \\
&\leq \frac{2c}{n}\left(\frac{n(n-1)}{2} - \frac{(n/2)\cdot(n/2-1)}{2}\right) + an \\
&= c\left(\frac{3n}{4} - \frac{1}{2}\right) + an
\end{aligned}$$

We need to choose c such that for sufficiently large n,

$$E[T(n)] \leq \frac{3cn/4 - c/2 + an}{1-1/n} \leq cn.$$

We can take $c = 5a$, whereupon the last inequality will hold for $n \geq 10$. This proves that $E[T(n)] = O(n)$.

(e) It is strange that the average-case running time of the corrupted code is still $O(n)$, though the worst-case is now infinite, but this holds with a larger constant hidden in the big-O notation. The average behavior remains linear because that the pivot is selected randomly and it occurs with extremely low probability that the same element is repeatedly selected as the pivot.