Illinois Institute of Technology
Department of Computer Science

Homework 2 Solutions

CS 430 Introduction to Algorithms
Fall Semester, 2007

1. **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 $\Omega(n \log n)$.

2. **Problem 7-4 on page 188**. **Solution:**

   (a) Tail-Recursive-Quicksort (TRQ) performs the same computation as QUICKSORT (which is already shown to be correct). Both algorithms partition the array in the same way and call themselves on the left part of the array. QUICKSORT then calls itself on the right part of the array. In contrast, TRQ set $p = q + 1$ and executes another iteration of the while loop. This additional iteration is the same as calling TRQ on the right part of the array.

   (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) The problem with TRQ is that it always calls itself recursively on the left sublist regardless of its size. If we modify the algorithm so that it always recursively calls itself on the smaller of the two sublists, this will guarantee that the stack depth never exceeds $\Theta(n)$. All this requires is that after line 3 of TRQ, we check the size of both sublists and apply TRQ to the smaller of the two lists and update p appropriately. Since TRQ is always applied to the smaller of the two sublists, which has length of at most $n/2$, we will never have more than $\Theta(\log n)$ consecutive recursive calls to TRQ.

   (d) The average stack depth for regular QUICKSORT can be expressed with the following recurrence

$$S(n) = 1 + \frac{1}{n} \sum_{i=0}^{n-1} S(i)$$

The recurrence for unmodified TRQ is exactly the same. However, for our modified TRQ we have

$$S(n) = 1 + \frac{1}{\lfloor n/2 \rfloor} \sum_{i=0}^{\lfloor n/2 \rfloor - 1} S(i)$$

since the consecutive recursive calls are always applied to the smaller sublist which will have length of at most $\lfloor n/2 \rfloor$.

The following is the solution to the recurrence for regular QUICKSORT (NOTE: solving the recurrence for modified TRQ was also acceptable). Since there are no recursive function calls when the length of the list is zero, $S(0) = 1$. We can rewrite this recurrence in the following form

$$t_n = 1 + \frac{1}{n} \sum_{i=0}^{n-1} t_i$$

where $t_0 = 1$. If we multiply both sides by n we get

$$nt_n = n + \sum_{i=0}^{n-1} t_i = n + t_{n-1} + \sum_{i=0}^{n-2} t_i \tag{1}$$

If we substitute $n - 1$ for n we get

$$(n-1)t_{n-1} = (n-1) + \sum_{i=0}^{n-2} t_i \tag{2}$$

If we subtract Equation 2 from Equation 1 we now have

$$nt_n - (n-1)t_{n-1} = t_{n-1} + 1$$

After we add $(n-1)t_{n-1}$ to both sides and divide by n we have

$$t_n = t_{n-1} + \frac{1}{n}$$

If we substitute $t_{n-1} = t_{n-2} + \frac{1}{n-1}$ we have

$$t_n = t_{n-2} + \frac{1}{n-1} + \frac{1}{n}$$

By continuing this process of substitution all the way down to t_0 we eventually get

$$t_n = t_{n-n} + \frac{1}{n-(n-1)} + \frac{1}{n-(n-2)} + \ldots + \frac{1}{n-1} + \frac{1}{n} = t_0 + \sum_{i=1}^{n} \frac{1}{i} = 1 + H_n$$

where H_n is the nth harmonic number. Since $H_n = \ln n + O(1)$, the solution to our recurrence is

$$S(n) = t_n = 1 + \ln n + O(1) = \Theta(\log n)$$

3. **Problem 8.3-4 on page 200**.

   Solution: Treat the numbers as 3-digit numbers in radix n. Each digit ranges from 0 to $n-1$. Sort these 3-digit numbers with radix sort. There are 3 calls to counting sort, each taking $\Theta(n+n+n) = \Theta(n)$ time, so that the total time is $\Theta(n)$.

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

   Solution: We can use a modified radix sort to do this. Treat each letter as a number such that a<b<c... <z. Now sort all of the strings by their first letter using counting sort. Group all of the strings that have the same first letter. Remove the first letter (or ignore it in someway) and sort each group by the next letter in the strings using counting sort. If there is no next letter in a string treat the string as if it has a letter which is less than 'a'. Continue to recursively apply this scheme to within each group until all there is nothing left to sort.

   The runtime analysis for this algorithm is a little tricky as we do not know anything about the lengths of the strings nor even the number of strings. However, we do know that if we have a string s_i that has length l_i it will be sorted by $l_i + 1$ counting sorts (one for each letter plus one for the time when the string is sorted as an empty string). Furthermore, we know that if we apply counting sort to t strings it takes $\Theta(t)$ time. Therefore the total time for all counting sort calls is

$$O(\sum_{i=1}^{m} l_i + 1) = O(m + \sum_{i=1}^{m} l_i) = O(n + m) = O(n)$$

   where m is the number of strings and $m \leq n$.

5. **Problem 9.3-8 on page 223**

   Solution: Let's start out by supposing that the median (the lower median, since we know we have an even number of elements) is in X. Let's call the median value m, and let's suppose that it's in $X[k]$. Then k elements of X are less than or equal to m and $n - k$ elements of X are greater than or equal to m. We know that in the two arrays combined, there must be n elements less than or equal to m and n elements greater than or equal to m, and so there must be $n - k$ elements of Y that are less than or equal to m and $n - (n - k) = k$ elements of Y that are greater than or equal to m. Thus, we can check that $X[k]$ is the lower median by checking whether $Y[n - k] \leq X[k] \leq Y[n - k + 1]$. A boundary case occurs for $k = n$. Then $n - k = 0$, and there is no array entry $Y[0]$; we only need to check that $X[n] \leq Y[1]$. Now, if the median is in X but is not in $X[k]$, then the above condition will not hold. If the median is in $X[k']$, where $k' < k$, then $X[k]$ is above the median, and $Y[n - k + 1] < X[k]$. Conversely, if the median is in $X[k'']$, where $k'' > k$, then $X[k]$ is below the median, and $X[k] < Y[n-k]$. Thus, we can use a binary search to determine whether there is an $X[k]$ such that either $k < n$ and $Y[n - k] \leq X[k] \leq Y[n - k + 1]$ or$k = n$ and $X[k] \leq Y[n - k + 1]$; if we find such an $X[k]$, then it is the median. Otherwise, we know that the median is in Y , and we use a binary search to find a $Y[k]$ such that either $k < n$ and $X[n - k] \leq Y[k] \leq X[n - k + 1]$ or $k = n$ and $Y[k] \leq X[n - k + 1]$; such a $Y[k]$ is the median. Since each binary search takes $O(\log n)$ time, we spend a total of $O(\log n)$ time.