CS170–Spring 2017 — Homework 1Solutions

Michael Fan, SID 26697596

Collaborators: Andrew Peng, Julie Han, Charlie Tian

1. Course Syllabus

(a) Not OK

(b) Not OK

(c) OK

(d) OK

(e) OK

2.

(c) A possible f_1 is $f_1(x) = 1$. Then

$$\sum_{i=1}^{n} f_1(i) = n * 1 = n \in \Theta(n)$$

Because n and n are identical polynomials, they are trivially tight bounds of each other. A possible f_2 is $f_2(x) = x$. Then

$$\sum_{i=1}^{n} f_2(i) = 1 + 2 + ... + n = \frac{n(n+1)}{2} \in \Theta(n^2)$$

We prove the tight bound: $n^2 < n(n+1)/2 < 3n^2$

3.

(a)

$$T(n) = 2T(\frac{n}{2}) + \sqrt{n} = 2T(\frac{n}{2}) + n^{\frac{1}{2}}$$

By the Master's Theorem, because $\frac{1}{2} < \log_2 2 = 1$

$$T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$$

. The version of the Master's Theorem proved in class gives an upper bound to recurrence problems, but it is trivially shown that the same process can be used to show a tight bound if the input equation is of the form $aT(\frac{n}{b}) + \Theta(n)$.

(b)

$$T(n) = T(n-1) + c^n$$

$$T(n) = T(n-2) + c^{n-1} + c^n$$

$$T(n) = T(n-3) + c^{n-2} + c^{n-1} + c^n$$

$$...$$

$$T(n) = 1 + c + c^2 + c^3 + ... + c^{n-2} + c^{n-1} + c^n$$

$$T(n) = \Theta(c^n)$$

via properties of geometric series proven in class.

(c)

$$T(n) = 2T(\sqrt{n}) + 3$$

We begin by setting $x = \log n$. Then, let $T(n) = F(x)$. It follows that

$$\begin{aligned} T(n) = F(x) &= T(2^x) \\ &= 2T(2^{x/2}) + 3 \\ &= 2F(x/2) + 3 \end{aligned}$$

This problem is now solvable via the Master's Theorem, which gives us that $F(x) \in \Theta(x)$. We work backwards and conclude that

$$\begin{aligned} T(n) = F(x) &= \Theta(x) \\ &= \Theta(logn) \end{aligned}$$

4.

(a)

$$\begin{aligned}
f2(n) &= f2(n/2) + f2(n/4) \\
&= 2f2(n/4) + f2(n/8) \\
&= 3f2(n/8) + 2f2(n/16) \\
&= 5f2(n/16) + 3f2(n/32) \\
&... \\
&= fib(\log_2 n)f2(k) + fib((\log_2 n) - 1)f2(k), k < 3 \\
f2(n) &\in \Theta(fib(\log n))
\end{aligned}$$

(b) In the worst case, assume that input n is odd and that adding 1 to n and dividing by 2 always results in another odd number. Then,

$$\begin{aligned}
f3(n) &= f3(n+1) \\
&= f3((n+1)/2) \\
&= f3((n+3)/2) \\
&= f3((n+3)/4) \\
&= f3((n+5)/4) \\
&= f3((n+5)/8) \\
&...
\end{aligned}$$

In the best case, input n is a power of 2, so the function will end after $\log_2 n$ operations. However, even in the worst case, the number of operations are only roughly doubled - 1 extra call on every value of n to make it even. Thus, we can ignore this scaling factor and conclude that

$$f3(n) \in \Theta(\log n)$$

5.

(a) 1. Main idea: To find the median of an input list "arr", one can use the quick select algorithm that we learned about in class with pivots selected by calling the "GAM" method on the input arr with given constant γ. This allows us to partition the input arr into 3 subarrays, one containing elements less than the pivot, one containing elements greater than the pivot, and one containing elements equal to the pivot. We then recursively call this method on the subarray that contains the index of the median with an adjusted index input until the program stops recursing when the input list is of length less than $1/\gamma$ and we can return the median trivially.

2. Pseudocode:

```
function linearMedian(list arr, double y, index n): //n is run initially as len(arr)/2
    if len(arr) * y < 1:
        return (arr[0] + arr[len(arr)-1])/2
    list sl, sv, sr
    pivot = GAM(arr)
    for int i in pivot
        if i < pivot
            sl.append(i)
        else if i == pivot
            sv.append(i)
        else if i > pivot
            sr.append(i)
    if (len(sl) > len(arr)/2):
        return linearMedian(sl, y, len(arr)/2)
    else if (len(sl) < len(arr)/2 and len(arr)/2 < len(sl) + len(sr)):
        return pivot
    else:
        return linearMedian(sr, y, len(arr)/2 - len(sl) - len(sv))
```

3. Proof of correctness:
We have shown in class that the quick-select function for finding a median in an array is correct. This is the exact algorithm except that we use GAM to select a pivot to guarantee a good pivot in the middle len(arr)/2 elements instead of randomly selecting a pivot.

4. Runtime Analysis:
Take the length of the list to be n. It takes $\Theta(n)$ time to split the array into the 3 subarrays. Given the fact that GAM always produces a good pivot, the length of the array that linearMedian is recursively called on is 3len(arr)/4 in the worst case (if the pivot is the element found at the 25th or the 75th percentile). Then, the recurrence relation is $T(n) = T(3n/4) + O(n)$. By Master's Theorem, linearMedian $\in \Theta(n)$.

(b) We can find the median of a length 5 array in $\Theta(5 \log 5)$ time via mergesort. There are $n/5$ such arrays to be sorted, so the runtime is $\Theta(5 \log 5 * n) = \Theta(n)$.
Consider a list of length n that is split into $n/5$ lists of length 5 with known medians. Then,

let the median of these medians be referred to as the "King Median." Thus, there are approximately n/10 lists with medians less than the King Median and approximately n/10 lists with medians greater than the King Median. For the lists with medians less than the King Median, we know that the two elements in each list that are less than the list's median are also less than the King Median. This is a total of n/10 * 3 = 3n/10 elements less than the King median guaranteed. Similarly, in each list with a median greater than the King median, the list is guaranteed to have 3 elements greater than the King median (the median and the 2 elements greater than the median). Thus, we have found that $\gamma = 3/10$. We need n_0 to be large enough such that γn returns a number greater than 1, or else every element will be a *gamma* approximate median. Thus, $n_0 > 10/3$ meaning we have a minimal array length of 4.

6.

1. Overview: Because each of k arrays of length l is already sorted, we can get the median of each list in constant time. Then, sort the medians and find the median of medians (the King Median). Use the King Median as a pivot and binary search each array for it, saving the last index to effectively split each list into subarrays sl, sv, and sr where all elements in sl are less than the King Median, all elements in sv are equal to the King Median, and all elements in sr are greater than the King Median. Let a be the lengths of all the sl's together, let b be the lengths of all the sv's together, and let c be the lengths of all the sr's together. We use the quick select deletion process in which we know that the median occurs at index kl/2 overall, so if kl/2 $\leq$ a, we throw away all sv's and sr's and only recurse on the sl's, if a+b $\geq$ kl/2 $> a$ then we return the King Median, and otherwise we throw away all the sv's and sl's and only recurse on the sr's. We recursively call the method on the new arrays and remember that now, we have to take into account of the fact that the lists are not of the same size. We find the new medians of the arrays and sort them by these medians. We then iterate down arrays until m/2 elements have passed, where m is the total number of elements remaining. We will choose the the median of this array to be the new King Median and continue until the median is found.

2. Pseudocode:

```
function helperFindMedian(arr):
    if len(arr) is even:
        return (arr[len(arr)/2] + arr[len(arr)/2 + 1])/2
    return arr[len(arr)/2]

function findMedian(array BigArray of k arrays of length l, index)://index initialized to kl/2
    sort_by_median(BigArray)
    int kingMedian = helperFindMedian(BigArray[len(BigArray)/2])
    sl = [], sr = [], sv = [], totalLen = 0
    for each arr in BigArray:
        totalLen += len(arr)
        index = binarySearch(arr)
        sl.append(arr[0, index])
        sr.append(arr[index+1])
        if (arr[index] == kingMedian):
            sv.append(arr[index])
        else if (arr[index] < kingMedian):
            sl.append(arr[index+1])
        else if (arr[index] > kingMedian):
            sr.append(arr[index])
    medianIndex = totalLen/2
    if (totalLen <= len(sl))
        BigArray = sl
    else if (len(sl) < totalLen and totalLen <= len(sl) + len(sv)
        return kingMedian
    else
        BigArray = sv.extend(sr)
        medianIndex -= (len(sl) + len(sv))
```

```
return findMedian(BigArray, medianIndex)
```

3. Proof of Correctness:
   By finding the King Median and partitioning elements like we did in the quick select method of finding a median, we guarantee that we never delete the median itself. The proof of correctness follows exactly the way it does for the validity of the quick select median algorithm.

4. Runtime analysis:
   It takes constant time to find a median of any particular sub array as they are all sorted. It takes $k \log k$ time to sort BigArray by medians. It takes $logl$ time for each binary search so $k \log l$ time total to search through all lists. The appending of the arrays should be done in constant time in the right programming language (C) as it is just a pointer manipulation. By the same logic as used in 5, the King Median is guaranteed to be in the middle half of the sorted elements, so in the quick select process we can throw away 1/4 of the total elements per recursive call. Thus, the recurrence relation is as follows:

   $$T(n) = T(3n/4) + k \log k + k \log l$$

   The algorithm effectively ends when each sub array is of length 1 or 0, which will take up to $O(\log l)$ recursive calls. Thus, we conclude that the algorithm is bound by $O(\log l * (k \log k + k \log l))$ which is asymptotically better than $O(n)$.