Summer CS161 Take Home Midterm Solution

Problem 1: Recurrences (20 points, 5 points each)

Give asymptotically tight upper (big O) bounds for $T(n)$ in each of the following recurrences. Prove your solution by naming the particular case of the master theorem, by iterating the recurrence, or by using the substitution method. You may use the version of the master theorem stated in Exercise 4.4-2 if you wish, and you may do so without proving it. Assume that $T(n)$ is constant for sufficiently small n.

- $T(n) = 3T(n/9) + \sqrt{n}\lg^4 n$

  This one screams master theorem, but we see logarithms, and that generally worries us, as it implies that it may fall in between either case 1 and 2, or in between case 2 and 3. Let's see.

  Here we're dealing with $\sqrt{n}\lg^4 n$ versus $n^{\log_9 3} = n^{1/2} = \sqrt{n}$. $\sqrt{n}\lg^4 n$ is asymptotically larger than $\sqrt{n}$, but only logarithmically so. The original master theorem in Chapter 4 wouldn't touch this one, but fortunately, you can assume the result of Exercise 4.4-2 was part of your bag of tricks all along, and it **does** cover this one with its revised case 2. This new and improved master theorem tells us that the answer is clear and straightforward: $T(n) = \sqrt{n}\lg^5 n$.

- $T(n) = 2T(n/3) + n\lg n$

  Also reeks of master theorem, and there are logarithms once again. Is $n\lg n$ polynomially different from $n^{\log_3 2}$. Yes! $n\lg n = \Omega(n)$, and $n = \Omega(n^{\log_3 2+\varepsilon})$, where $0 < \varepsilon < 1 - \log_3 2$, so that $n\lg n$ is polynomially larger than $n^{\log_3 2}$. Since $n\lg n$ surely satisfies the regularity condition of master theorem's case 3, our recurrence is definitely covered: $T(n) = n\lg n$.

- $T(n) = T(n/3) + T(n/5) + T(n/7) + T(n/11) + T(n/13) + n$

  Many of you went through the tedious chore of expanding a recursion tree on behalf of the recurrence. A better approach was use of the observation that

  $$\frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{11} + \frac{1}{13} = \frac{12673}{15015} < \frac{17}{20}$$

We can solve the much easier recurrence $V(n) = V\left(\frac{17n}{20}\right) + n$ using the master theorem, and then understand that $T(n) = O(V(n))$. Because the size of the recursive sub-problem modeled by the $V(n)$ problem is larger than the sum of all the sub-problems modeled by the $T(n)$ problem, we expect $V(n)$'s true solution to be larger than that for $T(n)$.

The master theorem tells us that $V(n) = \Theta(n)$, and therefore $T(n) = O(n)$. The original recurrence also tells us that $T(n) > n$, so that $T(n)$ is also $\Omega(n)$ and therefore $\Theta(n)$.

- $T(n) = T\left(\sqrt{n}\right) + 1$

Repeated substitution tells us a lot here. Basically, how many times do you need to take the square root of number before that number is less than or equal to 2. (We should never expect cascaded square roots get us to 1, since it'll never happen.)

$$\begin{aligned} T(n) &= T\left(n^{1/2}\right) + 1 \\ &= T\left(n^{1/4}\right) + 2 = T\left(n^{1/2^2}\right) + 2 \\ &= T\left(n^{1/8}\right) + 3 = T\left(n^{1/2^3}\right) + 3 \\ &\vdots \\ &= T\left(n^{1/2^k}\right) + k \end{aligned}$$

holds for all values of k, include that value for which $n^{1/2^k} = 2$. Let's see:

$$\begin{aligned} n^{1/2^k} &= 2 \\ 1/2^k &= \log_n 2 = \lg 2/\lg n = 1/\lg n \\ 2^k &= \lg n \\ k &= \lg\lg n \end{aligned}$$

So, we have that $T(n) = T\left(n^{1/2^k}\right) + k\Big|_{k=\lg\lg n} = T(2) + \lg\lg n = \Theta(\lg\lg n)$

Problem 2: Quick and Dirty Algorithms (14 points, 7 points each)

a) Provide an $\Theta(n)$-time algorithm that, given an integer array A of length n, prints all those elements between order statistics $\lceil n/4 \rceil$ and $\lfloor 3n/4 \rfloor$, inclusive. You should assume that all of the elements of A are distinct, and you needn't print the elements in any particular order.

You can take the linear time to select the element of rank $\lceil n/4 \rceil$, and then take linear time to select the element of rank $\lceil 3n/4 \rceil$. Once these two numbers have been extracted (using the $\Theta(n)$ version of Select), simply iterate over the elements of the set (again, in linear time) and print an element if and only if that elements falls in between the two values. The algorithm uses a finite number of $\Theta(n)$ substeps, so it too runs in $\Theta(n)$ time.

b) Provide an $\Theta(n + k \lg n)$ algorithm that, given an integer array A of length n, partially sorts A so that the k smallest elements are sorted and placed in positions 1 through k, and that the k largest elements are sorted and placed in positions $n - k + 1$ through n. The other elements—those ending up in positions $k + 1$ through $n - k$—needn't be in any particular order. You should **not** assume that the elements of A are distinct.

Take $\Theta(n)$ time to call **`Heapify`** on the array A, where the heap property mandates that a parent key be larger or equal to the keys of its children. Call **`Extract-Max`** on the heap k times (or a total of $\Theta(k \lg n)$, and place the results at indices n, n-1, n-2, etc. As each of these elements is extracted, these slots at indices n, n -1, and so forth are releases from the heap and are therefore safe places to place the maximum elements. Do the same thing all over again, calling **`Heapify`** on the elements that remain, but this time do so with the opposite heap property, where the parent key is always smaller than or equal to its child keys. Call **`Extract-Min`** k times, placing each at indices n - k - 1, n - k - 2, and so forth. Afterward, take the $\Theta(k)$ time to swap A[i] with A[n - k - i], for $1 \le i \le k$.

Some students came up with a lovely algorithm that ran in $\Theta(n + k \lg k)$, where they partitions the entire array first around the k^{th} order statistic, and then the n - k^{th} statistic, and then merge-sorted the first k and the last k elements. This worked provided that they used the $\Theta(n)$ version of **`Select`**, and that they used the stable version of **`Partition`** discussed in lecture. Very nice.

Problem 3: Sorting Hardware (16 points)

Professor Engler has developed a hardware priority queue for his computer. The priority queue device can store up to k records, each consisting of a key and a small amount of satellite data (such as a pointer). The computer to which it is attached can perform **`Insert`** and **`Extract-Min`** operations on the priority queue, each of which takes $\Theta(1)$, no matter how many records are stored on the device. The professor wishes to use the hardware priority queue to help implement a sorting algorithm on his computer. He has n records stored in the primary memory of his machine. If $n \le k$, the professor can certainly sort the keys in $\Theta(n)$ time by first inserting all of them into the priority queue, and then repeatedly extracting the minimum. Design an efficient

algorithm for sorting n items using the hardware priority queue, when $n > k$. Analyze your algorithm in terms of both k and n.

The best is has you modify merge sort to make use of the linear time sorting you have available to you for sorting arrays of length k.

One idea (worth a little bit of partial credit) is based on Problem 1-2 of the textbook. Since the hardware can sort k elements in $\Theta(k)$ time, we can divide the input into n/k lists of length k, sort the lists using the queue in $(n/k)\ \Theta(k) = \Theta(n)$, and then merge the lists in $\Theta(n\lg(n/k))$ time, as shown in Problem 1-2b. The total time for such an algorithm is $\Theta(n\lg(n/k))$, since the merging clearly dominates the running time.

The above approach, however, only uses the hardware for sorting the sublists; the hardware doesn't help with the merging at all. A better idea (worth full credit) is to use the hardware to implement a k-way merge that runs in $\Theta(n)$ time (for k lists containing n elements total.), and use it the same way plain old merge sort uses its $\Theta(n)$-time 2-way merge.

Algorithm:

```
Sort(A)
   if length[A] = 1
      then return;

   Divide A evenly into k subarrays, A1, A2, A3, ... , Ak
   Sort(A1), Sort(A2), Sort(A3), ..., Sort(Ak)
   K-Way-Merge(A1, A2, A3, A4, ..., Ak)
```

K-Way-Merge works as follows. Like 2-way Merge, it repeatedly moves the smallest element (the minimum of the elements at the front of the k lists) from each of the input lists to the output list. The hardware queue is used to find the minimum each time in only $\Theta(1)$ time (instead of the $\Theta(k)$ time it would normally take the scan the k list minimums. At any given time, the minimum element of each sublist.

- Start by moving the smallest element from each list into the hardware queue, along with some indicator as to which list it came from. Time: k elements, $\Theta(1)$ time for each insertion, $\Theta(k)$ time total.

- Extract-Min to get the smallest element. Say it came from the i^{th} sorted list. Move the next element (if any) from the i^{th} list into the queue to replace the one just extracted.
  Time: $\Theta(1)$ to Extract-Min and Insert

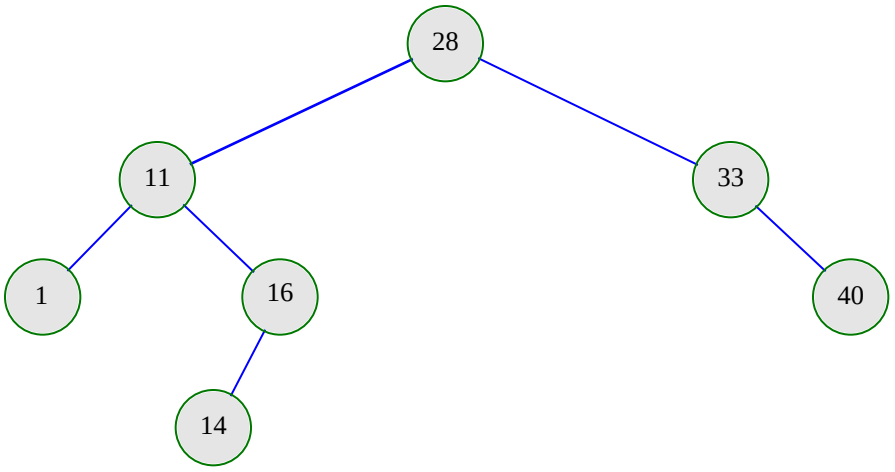
  Repeat the Extract-Min / Insert process until all n elements have been extracted.
  Time: n iterations, $\Theta(1)$ time for each, therefore $\Theta(n)$ total time.

  The running time of the K-Way-Merge is therefore $\Theta(k+n) = \Theta(n)$.

The recurrence relation modeling the running time is just like that of the unmodified Merge-Sort, save the fact that k replaces 2 everything: $T(n) = kT(n/k) + \Theta(n)$, whose solution is $T(n) = \Theta(n\log_k n)$. This recurrence needs to be solved using a recurrence tree, because the master theorem would treat k as a constant, and we don't want that. This $T(n) = \Theta(n\log_k n)$ time is asymptotically better than the $\Theta(n\lg(n/k))$ time of the first approach, since we are not treating k as a constant with respect to n. $\log_k n = \lg n / \lg k$, whereas $\lg(n/k) = \lg n - \lg k$.

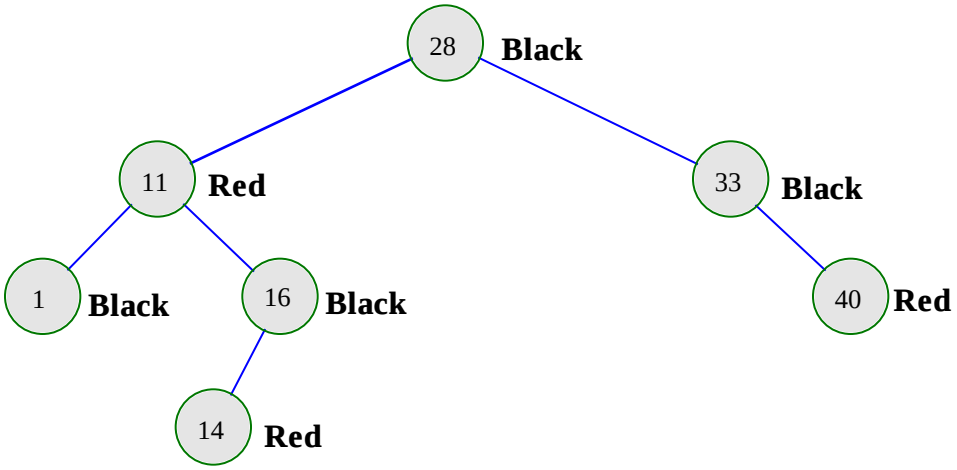
Problem 4: Transitioning from Binary Search Trees to Red-Black Trees (10 points, 3, 3, and 4)

a) Label the following binary search tree with numbers from the set $\{16,40,11,14,33,1,28\}$ so that it is a legal binary search tree.



b) Label each node in your tree above as either **Red** or **Black** so that the tree is also a legal red-black tree.

This was surprisingly non-trivial for me. Of course, you can experiment with trial and error, but the more informative approach is to try and figure out why the answer is what it is and how we should be able to color any such tree knowing it can be legally colored. We know that the two colors Red and Black are introduced more or less for bookkeeping purposes—by maintaining some invariant, we know that the black height of any path—including the shortest one and the longest one—must be the same throughout.



c) Make the left child of the root be the root by performing a single rotation. Draw the binary search tree that results, and label your tree with the keys from part **a)**. Is it possible to label the nodes with colors so that the new tree is a red-black tree? Justify you answer.