Solution Set 2: Algorithms and Asymptotics

Problem 1: Solving Recurrences

Give asymptotically tight upper (big O) bounds for $T(n)$ in each of the following recurrences. Justify your solution by naming the particular case of the master theorem, by iterating the recurrence, or by making an intelligent guess and substituting. Assume that $T(n) = \Theta(1)$ for small enough n.

- $T(n) = T(n-2) + 1$

  The master theorem only applies to recurrence relations of the form $T(n) = aT(n/b) + f(n)$, where the size of the recursive subproblem is a constant fraction of the original problem size, not a constant distance like that here. Our only prayer is to use either substitution or iteration, and fortunately, either one works well here. Repeated substitution works here just fine:

  $$\begin{aligned} T(n) &= T(n-2) + 1 \\ &= T(n-4) + 2 \\ &= T(n-6) + 3 \\ &\dots \\ &= T(n-2k) + k \\ &= T(n\%2) + \lceil n/2 \rceil \\ &\le T(1) + n/2 = \Theta(n) \end{aligned}$$

  Alternatively, you could be really smart and guess that $T(n) = n/2$ and substitute to show that it works.

  $$\begin{aligned} T(n) &= T(n-2) + 1 \\ &= (n-2)/2 + 1 \\ &= n/2 - 2/2 + 1 \\ &= n/2 - 1 + 1 = n/2 \end{aligned}$$

  Voila!

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

  The master method looks oh so tempting here, since it has all the right features. But because $n\lg^2 n$, while asymptotically larger than $n^{\log_b a} = n$, is not **polynomially** larger than n, we can't really use the master method. This problem falls in the gap between case 2 and case 3, and isn't covered by the proof. We need something a little more mathy to get

this one done. The next best thing is to iterate and see if we can detect a pattern. That's what I do here:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n\lg^2 n = 2T\left(\frac{n}{2}\right) + n(\lg n)^2 \\
&= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\lg^2\frac{n}{2}\right) + n(\lg n)^2 = 4T\left(\frac{n}{4}\right) + n\lg\frac{n}{2}^2 + n(\lg n)^2 \\
&= 4T\left(\frac{n}{4}\right) + n(\lg n - 1)^2 + n(\lg n)^2 = 4T\left(\frac{n}{4}\right) + n\left((\lg n)^2 + (\lg n - 1)^2\right)
\end{aligned}$$

I need to interrupt just to remind you what logarithmic identities I relied on in there. Note that $\lg\frac{n}{2} = \lg n - \lg 2 = \lg n - 1$. If you see that and trust that the same type of thing with come up each time, then we can just go with it until we see the pattern.

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n(\lg n)^2 \\
&= 4T\left(\frac{n}{4}\right) + n\left((\lg n)^2 + (\lg n - 1)^2\right) \\
&= 8T\left(\frac{n}{8}\right) + n\left((\lg n)^2 + (\lg n - 1)^2 + (\lg n - 2)^2\right) \\
&\vdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + n\sum_{0 \le i < k} (\lg n - i)^2
\end{aligned}$$

This relationship is true for all values of k, and in particular, for $k = \lg n$. Substituting $k = \lg n$ into the above, we get:

$$\begin{aligned}
T(n) &= 2^k T\left(\frac{n}{2^k}\right) + n\sum_{0 \le i < k} (\lg n - i)^2 \\
T(n)\Big|_{k=\lg n} &= nT(1) + n\sum_{0 \le i < \lg n} (\lg n - i)^2 = \Theta(n) + n\sum_{1 \le i \le \lg n} i^2 \\
&= \Theta(n) + n\Theta\left(\lg^3 n\right) = \Theta\left(n\lg^3 n\right) \\
T(n) &= \Theta\left(n\lg^3 n\right)
\end{aligned}$$

- $T(n) = 9T(n/4) + n^2$

This one maps to the master method very nicely. Let $a = 9$, $b = 4$, and $f(n) = n^2$. To guarantee that one function is polynomially larger than the other, we need to compare $n^{\log_b a} = n^{\log_4 9}$ to n^2. $n^2 = n^{\log_4 16} = \Omega\left(n^{\log_4 9 + \varepsilon}\right)$, so that $f(n) = n^2$ is polynomially larger than $n^{\log_b a} = n^{\log_4 9}$. Ergo, we're dealing with case 3 of the master theorem, and that case says that $T(n) = \Theta\left(n^2\right)$.

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

  Case 1 this time: $n^{\log_b a} = n^{\log_2 3} = \Omega\left(n^{1+\varepsilon}\right)$, for $\varepsilon > \lg 3 - 1 > 0$. Therefore, $T(n) = \Theta\left(n^{\log_2 3}\right)$.

- $T(n) = T\left(n/2 + \sqrt{n}\right) + n$

  This was easily the hardest one to solve for, since none of our techniques really cover recurrences that look like this one. In order to hazard a guess, we can approximate the recurrence for very large n (that's where asymptotics really matters: on the number line for arbitrarily large n) by realizing that $n/2 + \sqrt{n} \approx n/2$. So, we expect the answer to our recurrence to behave primarily like that of

  $$T(n) = T(n/2) + n,$$

  which the master theorem (case 1) tells us is solved by $T(n) = \Theta(n) = O(n)$. Now this doesn't mean that we're done. What we've shown here is that $T(n) = O(n)$ might be a really good guess for our real recurrence. Master theorem won't say so, and I don't want to try any repeated substitution on this one, but I can certainly **guess** that $T(n) \le cn$ for large enough n.

  Assume that $T(k) \le ck$ for all values of $k : 0 \le k < n$, and show that $T(n) \le cn$.

  $$\begin{aligned} T(n) &= T\left(n/2 + \sqrt{n}\right) + n \\ &\le cn/2 + c\sqrt{n} + n \le cn \end{aligned}$$

  where the last inequality holds provided

  $$\begin{aligned} cn/2 + c\sqrt{n} + n &\le cn \\ c\sqrt{n} &\le \frac{(c-2)n}{2} \\ \sqrt{n} &\ge \frac{2c}{(c-2)} \\ n &\ge \frac{4c^2}{(c-2)^2} \end{aligned}$$

  for whatever choice of c we make (provided it's positive and works for small cases.) Therefore, $T(n) = \Theta(n)$.

Problem 2: More Heapifying

Consider the code for **`Build-Heap`** from Chapter 7, which operates on a heap stored in an array $A[1...n]$.

```
Build-Heap(A)
   heap-size[A] ← length[A]
   for i ← ⌊length[A]/2⌋ downto 1
      do Heapify(A,i)
```

- Show how this procedure can be implemented as a recursive divide-and-conquer procedure **`Build-Heap(A,i)`** where **`A[i]`** is the root of the subheap to be built. To build the entire heap, one would call `Build-`**`Heap(A,1).`**

  Recall that **`Heapify(A,i)`** works on the precondition that the structures rooted at indices **`2i`** and **`2i+1`** are already heaps. The structure rooted at index **`i`** would then be at most one heap-property violation away from being a true heap. That violation can be removed by allowing **`A[i]`** to drift down a path until it finds a home where the heap property is satisfied between it and its children (or until it lands gently along the fringe.)

  We can build a heap at index i by first building a heap of the structures rooted at indices **`2i`** and **`2i+1`**. Once the two child structures are valid heaps, we can rid of any potential heap property violations by calling **`Heapify`** on A at index `i`. Of course, if a particular structure contains only one node, then it is automatically a heap and we needn't bother making any recursive calls at all.

  Here's the recursive algorithm:

```
Build-Heap(A,i)
   if i ≤ ⌊length[A]/2⌋
      then Build-Heap(A, 2i)
           Build-Heap(A, 2i+1)
           Heapify(A, i)
```

- Give a recurrence that describes the worst-case running time of your procedure.

  Recall from lecture the heap data structure is as balanced a binary tree structure as they come. In the best scenario, the left and right subheaps of the entry rooted at index 1 are each about half the size of the original, and in the worst case, the left subheap is about twice as large as the right one. Foreseeing the need to solve the recurrence using the master method, we're justified in choosing the best scenario, where we have two recursive subproblems of half the size (just like **`Mergesort`**), and we have a post-recursion time that is $\Theta(\lg n)$. That means the recurrence is is given by:

$$T(n) = \begin{cases} \Theta(1) & n \le 1 \\ 2T(n/2) + \Theta(\lg n) & \text{otherwise} \end{cases}$$

Some of you argued quite convincingly that we don't know how balanced the heap is, and that while in one extreme case, we have a perfectly balanced heap (so that each of the recursive calls deals with exactly half the remaining elements), there is the scenario where the fringe of the heap, as described in the textbook, is exactly halfway full, resulting in a two thirds-to-one third ratio. Technically, the recurrence relation for any particular call is more accurately modeled as:

$$T(n) = \begin{cases} \Theta(1) & n \le 1 \\ T(\alpha n) + T((1-\alpha)n) + \Theta(\lg n) & \text{otherwise} \end{cases}$$

where $\frac{1}{3} \le \alpha \le \frac{1}{2}$.

- Solve the recurrence using the master method.

Two recursive subproblems, right? Problem size gets divided by two, no? Clearly $a = b = 2$, so that $n^{\log_b a} = n^{\log_2 2} = n$. Case 1 of the master method applies, since $\Theta(\lg n)$ is definitely polynomially smaller than n. Therefore, $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta(n)$. Woof!

The master method can't be applied to the α-based recurrence at all, since it's not of the proper form. Because the problem asked you to use the master method, you were more or less required to assume that $\alpha = \frac{1}{2}$, but those that insisted on working with the general recurrence proved that the running time was $T(n) = \Theta(n)$ by using a recursion tree, or by using induction.

Problem 4: Quick and Dirty Data Structures

For each of the following data structures, assume that an infinite amount of memory is available, and that you needn't free memory even when it is no longer needed.

Presenting Algorithms:

There are four things you need to do when you present an algorithm:

1. Describe the algorithm in English.
2. Specify an algorithm as a program in a pseudo-code, if clarity demands it.
3. Indicate (or prove) the correctness of the algorithm.
4. Analyze the running time of the algorithm.

You must do steps 1, 3, and 4 when you present an algorithm with your homework submissions. If you have questions, feel free to ask either Jerry or Hiroshi, although the first few pages of Chapter 1 will likely clarify any problems with the terms "correctness", "running time", etcetera.

- Show how to implement a queue that efficiently supports **Enqueue**, **Dequeue**, **Extract-Min** as well as **Minimum**. **Enqueue**, **Dequeue** and **Extract-Min** should run in logarithmic time—that is, $O(\lg n)$ time—whereas **Minimum** should run in constant time.

  Keeping track of a minimum suggests using a heap like that discussed in lecture, except the heap property maintains the opposite invariant: the parent has a lower or equal value than any of its children. Supporting **Enqueue** and **Dequeue** (assuming traditional FIFO behavior) requires that a queue-like structure be maintained behind the scenes as well. Hrm... heap or queue? Which one do we keep?

  We keep both! All operations can be supported in the required logarithmic time if we keep the elements in a queue (doubly linked list with constant time access to the front and rear) and keep elements in a separate heap as well. We must always ensure that both the queue and the heap are in sync insomuch that additions and deletions to one are properly reflected in the other. At the very least, this would constrain the node of each to maintain pointers to the corresponding entry in the other. That way, an Extract-Min call can splice the proper node out of the queue, whereas **Enqueue** and **Dequeue** can properly include or remove an entry from the heap.

- Show how to efficiently implement a priority queue that supports **Insert**, **Extract-Min**, and **Extract-Max** to all run in $O(\lg n)$ time.

  Maintain two heaps and cross reference the nodes as in the queue-heap hybrid above. One heap maintains the heap property that presses the maximum elements to the root, whereas the other maintains the opposite heap property where smaller elements are bubbled toward its root.