CSE 3101: Design and Analysis of Algorithms
Tutorial 6

Problems:

1. Suppose $T(n)$ is a constant for $n \leq 2$. Solve for $T(n)$ asymptotically and justify your answer.

$$T(n) = 7T(n/3) + n^2.$$

Solution: First, since $a = 7, b = 3$, so $n^{\log_b a} = n^{\log_3 7} = o(n^2)$, since $\log_3 7 < \log_3 9 = 2$. This suggests that we check if the third case of the Master Theorem holds. Indeed $n^2 = \Omega(n^{\log_b a+\epsilon}) = \Omega(n^{\log_3 7+\epsilon})$ for $\epsilon = (2 - \log_3 7)/2$. Finally, we have to check the technical condition that $af(n/b) \leq cf(n)$ for some $c < 1$ and sufficiently large n. Indeed, $7(n/3)^2 \leq cn^2$ for any c satisfying $1 > c \geq 7/9$. Therefore by the third case of the Master Theorem, we have $T(n) = \Theta(n^2)$.

2. Given the recurrence $T(1) = 1$ and for $n > 1$, $T(n) = 2T(n/2) + n/\lg n$, can we apply the Master theorem? Justify your answer.

Solution:*NO*. We know that $\lg n \notin \Omega(n^\epsilon)$ for any $\epsilon > 0$. So $n/\lg n \notin O(n^{1-\epsilon})$ for any $\epsilon > 0$.

3. Solve the following recurrence to get a big-O bound, where $T(1) = 1$ and for $n > 1$,

$$T(n) = 2\sqrt{n}T(\sqrt{n}) + 4n$$

You may ignore floors and ceilings.

Solution: Let us first substitute $S(n) = T(n)/n$. Then, we have $S(n) = 2S(\sqrt{n}) + 4$. We can now substitute $n = 2^m$, so that $m = \lg n$. This yields $S(2^m) = 2S(2^{m/2}) + 4$. Now define $R(m) = S(2^m)$. Then we have the following recurrence for $R(m)$:

$$R(m) = 2R(m/2) + 4$$

Case 1 of the Master Theorem holds, since $n^{\log_b a} = n^1$ and $f(n) = 4 = 4n^0 = 4n^{1-1}$ so $\epsilon = 1$ works. Using case 1 of the Master Theorem, we get $R(m) = O(m)$. Therefore $S(2^m) = O(m)$, or $S(n) = O(\lg n)$. Therefore $T(n) = O(n \lg n)$.

4. Solve the following recurrences to get tight bounds, where $T(1) = 1$ and for $n > 1$. You may ignore floors and ceilings.

   (a) $T(n) = 9T(n/9) + \log^3 n$.

   Solution: We can use the Master Theorem for this problem. Here $a = 9, b = 9$, and so $n^{\log_9 9} = n^1 = n$. Also $f(n) = \log^3 n$. We will prove that $f(n) = O(n^{1-\epsilon})$ for some $\epsilon > 0$. Choose $\epsilon = 1/2$. It is actually easier to prove something stronger, viz., $f(n) = o(n^{0.5})$. This can be proved using the limit formula like we did in assignment 1. So we wish to show that

$$L = \lim_{n\to\infty} \frac{\log^3 n}{n^{0.5}} = 0.$$

   By successive application of L'Hospital's rule, we get

$$\begin{aligned} L &= \lim_{n\to\infty} \frac{\log^3 n}{n^{0.5}} \\ &= \lim_{n\to\infty} \frac{3\log^2 n}{0.5n^{0.5}} \\ &= \lim_{n\to\infty} \frac{6\log n}{0.25n^{0.5}} \\ &= \lim_{n\to\infty} \frac{6}{0.125n^{0.5}} \\ &= 0 \end{aligned}$$

Therefore, case 1 of the master method applies. So $T(n) = \Theta(n)$.

Solution 2: We can also solve it from scratch. Let $n = 9^m$. So $\log^3 n = cm^3$ for some constant c. Then $T(9^m) = 9T(9^{m-1}) + cm^3$. Define $S(m) = T(9^m)$. Thus, $S(0) = 1$ and $S(m) = 9S(m-1) + cm^3$. By repeated substitution

$$\begin{aligned}
S(m) &= 9S(m-1) + cm^3 \\
&= 9[9S(m-2) + c(m-1)^3] + cm^3 \\
&= 9^2 S(m-2) + c[m^3 + 9(m-1)^3] \\
&= 9^2[9S(m-3) + c(m-2)^3] + c[m^3 + 9(m-1)^3] \\
&= 9^3 S(m-3) + c[m^3 + 9(m-1)^3 + 9^2(m-2)^3] \\
&= \dots \\
&= 9^m S(0) + c\sum_{i=0}^{m}(m-i)^3 9^i \\
&= 9^m S(0) + c\sum_{i=1}^{m} i^3 9^{m-i} \\
&= 9^m + c9^m \sum_{i=1}^{m} i^3 9^{-i} \\
&= \Theta(9^m) \\
&= \Theta(n)
\end{aligned}$$

(b) $T(n) = 4T(\sqrt[3]{n}) + n$.

Solution: We can't use the master method directly because the argument to T on the RHS is not of the form n/b. Changing the variable, we rename $n = 2^m$. So $T(2^m) = 4T(2^{m/3}) + 2^m$. Define $S(m) = T(2^m)$. Then $S(m) = 4S(m/3) + 2^m$. We can use the Master Theorem to solve this recurrence, **provided** $S(1)$ is defined. It is important to note that we need $S(1)$ as a boundary condition if we ignore floors, because for any finite k, $m/3^k > 0$. We can compute $S(1)$ using $S(1) = T(2) = 4T(1) + 2 = 6$.

Here, $a = 4, b = 3$, so $n^{\log_b a} = n^{\log_3 4}$. We will show that $f(m) = 2^m = \Omega(n^{\log_3 4+\epsilon})$. Choose ϵ such that $n^{\log_3 4+\epsilon} = n^2$. We can prove that $2^m = \omega(n^2)$ (you can prove this using limits as in the last problem). We check the regularity condition next. This requires us to prove that $af(n/b) < cf(n)$ for some $c < 1$ and sufficiently large n. Since $4 \cdot 2^{n/3} = 2^{n/3+2} = \frac{1}{2}2^{n/3+3} < \frac{1}{2}2^n$ when $n/3 + 3 < n$ or $n \geq 5$.

Thus case 3 of the master method applies. Hence, $S(m) = \Theta(2^m)$. Changing back from $S(m)$ to $T(n)$, we obtain $T(n) = \Theta(n)$.

(c) $T(n) = 3T(n/2) + n \log n$.

Solution: Here $a = 3, b = 2$, and $n^{\log_b a} = n^{\log_2 3}$. We will prove that $f(n) = n \log n = O(n^{\log_2 3 - \epsilon})$ for some $\epsilon > 0$. Choose ϵ such that $n^{\log_2 3-\epsilon} = n^{1.1}$. As in part a, it is actually easier to prove $f(n) = o(n^{1.1})$. This can be proved using the limit formula like we did in assignment 1. So we wish to show that $L = \lim_{n\to\infty} \frac{n \log n}{n^{1.1}} = 0$.

$$\begin{aligned}
L &= \lim_{n\to\infty} \frac{n \log n}{n^{1.1}} \\
&= \lim_{n\to\infty} \frac{\log n}{n^{0.1}} \\
&= \lim_{n\to\infty} \frac{1}{0.1 \cdot n^{0.1}} \text{ by L'Hospital's rule} \\
&= 0
\end{aligned}$$

Therefore, case 1 of the master method applies: so, $T(n) = \Theta(n^{\log_2 3})$.

5. For the following segment of code, find the exact value of $T(n)$.

```
T(n)
1  if n = 1
2     then return 1
3     else  sum = 0;
4           for i = 1 to n − 1
5           do sum = sum + T(i)
6              return(sum + n)
```

Solution: Let us first get a recurrence for $T(n)$. This is $T(1) = 1$ and for $n > 1$,

$$T(n) = \sum_{i=1}^{n-1} T(i) + n.$$

This recurrence can be solved using the differencing technique in Chapter 4. Since

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

We can easily prove using induction that $T(n) = 2^n - 1$. This is true for $n = 1$. Assuming that the hypothesis is true for $n = k$, we have

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

6. Describe an algorithm that, given a set S distinct numbers, determines the minimum possible difference of 2 distinct numbers in S. You must state the running time of your algorithm and justify it. Also, add a sentence to argue the correctness of your algorithm.

Solution: We claim that this problem is solved by sorting the numbers in descending order and looking at the differences between successive numbers in the sorted list. The minimum of these differeces is the answer.

Proof of claim: If the minimum possible difference is not that of successive numbers in the sorted list, let the minimum difference be of indices $i, j, i < j$ in the sorted list. Since i, j are not consecutive, there is an index k, $i < k < j$ in the sorted list. Since $A[j] > A[k]$, so $A[j] - A[i] > A[k] - A[i]$, which contradicts the assumption that the difference between $A[i], A[j]$ is the minimum possible.

Algorithm and runtime analysis: The algorithm is the following three steps. Note that I deliberately chose to give an English description rather than pseudo-code.

(a) Sort the input array in descending order.

(b) Find all the differences between successive items in the sorted list.

(c) Return the minimum of these differences.

The running time of step 1 is $\Theta(n \lg n)$ if we choose an appropriate sorting method. The running time of steps 2 and 3 combined is $\Theta(n)$. So the total running time is $\Theta(n \lg n)$.

Notes:
1. The brute force algorithm looks at all possible differences and takes time $\Omega(n^2)$.

2. The proof of the claim above is not a complete proof of correctness but rather the main reason that the algorithm works. The question asked for similar ideas when it asked for arguments for correctness. Observe that the 3 steps of the algorithm are simple and one does not have to write down assertions for those steps to see that the algorithm is correct.

7. Consider the following code for BUILD-HEAP, which operates on a heap stored in an array $A[1 \ldots n]$:

BUILD-HEAP(A)
1 $heap - size[A] \leftarrow length[A]$
2 **for** $i = \lfloor length[A]/2 \rfloor$ down to 1
3 **do** MAX-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 sub-heap to be built. To build the entire heap, we would call BUILD-HEAP($A, 1$).

Solution: Recall that MAX-HEAPIFY(A, i) requires that the two subtrees of node i must already be heaps (the iterative version given above starts building the heap bottom-up for this reason). The recursive version makes the two subtrees of node i heaps by calling itself recursively, and then invokes MAX-HEAPIFY on node i.

BUILD-HEAP-R(A, i)
1 **if** $i \leq \lfloor length[A]/2 \rfloor$
2 **then** BUILD-HEAP-R($A, 2i$)
3 BUILD-HEAP-R($A, 2i + 1$)
4 MAX-HEAPIFY(A, i)

8. Give a recurrence that describes the worst-case running time of the previous procedure. Solve the recurrence using the Master theorem.

Solution: Since MAX-HEAPIFY takes time $O(\lg n)$ and there are two recursive calls to BUILD-HEAP-R, the running time can be expressed using the following equations:
$T(1) = O(1)$, and for $n > 1$,

$$T(n) = 2T(n/2) + O(\lg n)$$

Solution of the recurrence: First, we must show that case 1 of the Master Theorem applies. Since $a = 2, b = 2$, so $n^{\log_b a} = n^{\log_2 2} = n$, and $f(n) = \lg n$, we can use the Master Theorem using $\epsilon = 0.5$ (for example). The proof that $\lg n = O(n^0.5)$ is omitted. One way to do this is by showing that $\lg n = o(n^0.5)$ using the limit definition of $o()$.

Then, $T(n) = \Theta(n)$ by case 1 of the Master theorem.