CS-UY 2413: Design&Analysis of Algorithms—Spring 2017 Prof. Boris Aronov
Department of Computer Science and Engineering Prof. Lisa Hellerstein

Homework 4

Due Feb. 28 at the start of lecture (Hellerstein's sections).
Due March 1 at the start of lecture (Aronov's section).
This homework should be handed in on paper. No late homeworks accepted. Contact your professor for special circumstances.

Policy on collaboration on this homework: The policy for collaboration on this homework is the same as in HWs I and II. By handing in this homework, you accept that policy. Remember: A maximum of 3 people per group.

Notes: (i) Every answer has to be justified unless otherwise stated. Show your work! All performance estimates (running times, bit op counts, etc) should be in asymptotic notation unless otherwise noted.

(ii) You may use any theorem/property/fact proven in class or in the textbook. You do not need to re-prove any of them.

(iii) Some problems are designated as "Extra thinking" problems. These are optional, additional problems. They may be harder than the required problems. You are welcome to try to think about the extra thinking problems. You will not get credit for them. But you will learn something interesting. Do not hand in your answers to these problems. (They will not be graded.)

(iv) In this homework, unless explicitly said otherwise, you may ignore all the rounding issues: when I write $n/3$ or $\sqrt{n}$ in a recurrence, you may pretend that they are integers, so that $T(n/3)$ and $T(\sqrt{n})$ make sense. A more careful analysis would be to write $T(\lceil n/3 \rceil)$ and $T(\lceil \sqrt{n} \rceil)$, respectively.

1. The Master Theorem, as given in our textbook, is also true if you replace each big-Oh by θ. Call this the *Master Theorem (Tight Version)*.

   For each of the following recurrences, determine whether the Master Theorem (Tight Version) can be applied to the recurrence. If so, use it to give the solution to the recurrence in Θ notation. If not, write "Master Theorem (Tight Version) does not apply."

   Note: Master Theorem in the book is stated for any *constant* $a > 0$, $b > 1$, and $d \geq 0$; they do not have to be integers.

   (a) $T(n) = \log n \cdot T(n/\log n) + n$
   Solution: Since $\log n$ is not constant, the Master Theorem (Tight Version) does not apply.

   (b) $T(n) = 9T(n/3) + \sqrt{n}$
   Solution: We have $a = 9$, $b = 3$, and $d = 1/2$ (since $\sqrt{N} = n^{1/2}$). Then we have that $T(n) = aT(n/b) + \Theta(n^d)$. Since $\log_3(9) = 2 > 1/2$, we have that $T(n) = \Theta(n^2)$.

(c) $T(n) = 5T(n/25) + n^2$

Solution: We have $a = 5$, $b = 25$, and $d = 2$. Then since $\log_{25}(5) = 1/2 < 2$, the we have that $T(n) = \Theta(n^2)$.

(d) $T(n) = 8T(n/4) + n^2$

Solution: We have $a = 8$, $b = 4$, and $d = 2$. Then $\log_4(8) = 3/2 < 2$, so we have that $T(n) = \Theta(n^2)$.

(e) $T(n) = 9T(n/3) + 3n^2$

Solution: Letting $a = 9$, $b = 3$, and $d = 2$, we have $T(n) = aT(n/b) + \Theta(n^d)$ (since $3n^2 = \Theta(n^2)$). Then we see that $\log_3(9) = 2 = d$, so the Master Theorem (Tight Version) tells us that $T(n) = \Theta(n^2 \log n)$.

(f) $T(n) = 4T(n/8) + n \log n$

Solution: Since there is no constant d such that $n \log n = \Theta(n^d)$, the Master Theorem (Tight Version) does not apply.

Note: If we do not require the Tight Version, we can still technically apply the Master Theorem as given in the book. Note that $n \log n = O(n^2)$ so we can let $d = 2$. Then with $a = 4$, and $b = 8$, we find that $\log_8(4) = 2/3 < 2$. Then the Master Theorem as given in the textbook states that $T(n) = O(n^2)$. This is correct, but is not a tight bound (i.e. $T(n) \neq \Theta(n^2)$). The Master Theorem (Tight Version) does *not* hold in this case. Note also that for any $d > 1$, $n \log n = O(n^d)$, so we can get a bound of $O(n^{1+\epsilon})$ for any $\epsilon > 0$.

(g) $T(n) = nT(n/5) + n$

Solution: The Master Theorem (Tight Version) does not apply, since $a = n$ is not constant.

(h) $T(n) = T(n/5) + n^3$

Solution: We have $a = 1$, $b = 5$, and $d = 3$. Then, applying the Master Theorem (Tight Version), we get $\log_3(1) = 0 < 3$ and so $T(n) = \Theta(n^3)$.

2. For each of the following recurrences, compute the value of $T(9)$. Then solve the recurrence EXACTLY. That is, express $T(n)$ as a function of n, without using big-Oh or theta notation. Test your solution by plugging in $n = 9$ and seeing whether you get the same value for $T(9)$ that you computed above. An example is given to you on the last page of this homework.

(a) $T(n) = 2T(n/3) + 1$; $T(1) = 1$.

Solution: First we find $T(3) = 2T(1)+1 = 3$. Then $T(9) = 2T(3)+1 = 2 \cdot 3+1 = 7$.

Using the approach shown in the example, we can draw the tree for this recurrence. The value at each level decreases by a factor of 3 as you go down. So the value in the nodes at Level i would be $n/3^i$. The bottom nodes have the value 1 (since the base case is $n = 1$). Solving for i, we get that the bottom level is Level $\log_3 n$. The number of nodes doubles at each level, so the number of nodes at

Level i is 2^i. In the levels above the bottom, the local contribution at each node is 1 (from the +1 term in the recurrence). The local contribution at the nodes in the bottom level is also 1, because $T(1) = 1$. So the total contribution over all nodes at level i is $2^i \times 1 = 2^i$. Summing the total contribution at each level results in $2^0 + 2^1 + \cdots + 2^{\log_3 n} = 2^{\log_3 n+1} - 1$. So we get $T(n) = 2^{\log_3 n+1} - 1 = 2 \times (3^{\log_3 2})^{\log_3 n} - 1 = 2 \times (3^{\log_3 n})^{\log_3 2} = 2n^{\log_3 2} - 1$.

Testing, we can find $T(9) = 2 \cdot 9^{\log_3 2} - 1 = 2 \cdot (3^{\log_3 2})^2 - 1 = 2^3 - 1 = 7$. This matches the value we obtained earlier.

(b) $T(n) = 2T(n/3)$; $T(1) = 2$.

Solution: We have $T(3) = 2T(1) = 4$. Then $T(9) = 2T(3) = 8$.

The solution here is very similar to the solution of the previous part. The difference is that there is no local contribution in nodes above the bottom level. However, the local contribution in nodes at the bottom level is 2. The total contribution of these nodes is therefore $2^{\log_3 n} \times 2 = 2^{\log_3 n+1}$. Then we get that $T(n) = 2^{\log_3 n+1} = 2 \times (3^{\log_3 2})^{\log_3 n} = 2n^{\log_3 2}$

Testing with $n = 9$, we get $T(9) = 2 \cdot 9^{\log_3 2} = 8$. This matches the value we obtained earlier.

(c) $T(n) = 2T(n-1)$; $T(1) = 2$.

Solution: $T(9) = 2T(8) = 4T(7) = 8T(6) = \cdots = 2^8 \cdot T(1) = 2^9 = 512$. We can start to see from this that the value of $T(n)$ doubles each time we increase n by one. Specifically, we can see that $T(n) = 2^n$. We can verify this by looking at the tree. The value at each node decreases by one each level, so the value in the nodes at Level i is $n - i$. Since the value at the bottom level is 1, we have that the bottom level is Level $n - 1$. Further, at each level, we double the number of nodes, so the number of nodes at Level i is 2^i. The local contribution in the nodes above the bottom level is 0, but the local contribution in the nodes at the bottom level is 2. The total contribution is $2 \times$ (# nodes in the bottom level) $= 2 \cdot 2^{n-1} = 2^n$. Testing, we see that $T(9) = 2^9 = 512$ just as before.

3. Solve each of the following recurrences *without* using Master Theorem. You should try getting an asymptotically tight bound (that is, a θ bound). If you instead give an upper bound that is not tight, or a lower bound that is not tight, you may be eligible for partial credit.

   The following facts may be useful:

   (a) If $r > 1$ and $f(i) = r^0 + r^1 + r^2 + \ldots + r^i$, then $f(i) = \theta(r^i)$.

   (b) If $0 < r < 1$ and $f(i) = r^0 + r^1 + r^2 + \ldots + r^i$, then $f(i) = \theta(1)$.

   That is, the sum is theta of either the first or last element of the sum, depending on whether $0 < r < 1$ (summing the elements of a decreasing geometric series) or $r > 1$ (summing the elements of an increasing geometric series).

The base case of every recurrence, except where noted, is assumed to be $T(1) = 1$.

Note: The optional "Extra thinking" problems are marked with "$**$".

(a) $T(n) = T(n-2) + 5n$; $T(1) = T(2) = 1$

Solution:

$$T(n) = \begin{cases} 1 + 5 \cdot (3 + 5 + 7 + \cdots + (n-2) + n) & \text{if } n \text{ is odd} \\ 1 + 5 \cdot (4 + 6 + 8 + \cdots + (n-2) + n) & \text{if } n \text{ is even} \end{cases}$$

This can be simplified to:

$$T(n) = \begin{cases} 1 + \frac{5}{4}(n-1)(n+3) & \text{if } n \text{ is odd} \\ 1 + \frac{5}{4}(n-1)(n+4) & \text{if } n \text{ is even} \end{cases}$$

In both cases (n odd or n even), $T(n)$ is quadratic, so $T(n) = \Theta(n^2)$.

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

Solution: We can substitute to get:

$$\begin{aligned} T(n) &= T(n/4) + \sqrt[3]{n} \\ T(n) &= T(n/16) + \sqrt[3]{n/4} + \sqrt[3]{n} \\ T(n) &= T(n/4^3) + \sqrt[3]{n/16} + \sqrt[3]{n} \\ &\vdots \\ T(n) &= T(n/4^k) + \sqrt[3]{n/4^{k-1}} + \cdots + \sqrt[3]{n} \end{aligned}$$

for a positive integer k.

We can simplify to get:

$$T(n) = T(n/4^k) + n^{1/3} \left(\sum_{i=0}^{k-1} \frac{1}{\sqrt[3]{4}} \right)^i$$

Using the fact from above, we notice that the sum is $\Theta(1)$ since $r = \sqrt[3]{4} < 1$. This makes the additive term $\Theta(n^{1/3})$ and so $T(n) = T(n/4^k) + \Theta(n^{1/3})$. If we set $k = \log_4 n$ then $n/4^k = 1$ and $T(n) = T(1) + \Theta(n^{1/3}) = \Theta(\sqrt[3]{n})$.

(c) $T(n) = T(n/3) + 5n$

Solution: We can use the same technique of repeatedly substituting:

$$\begin{aligned} T(n) &= T(n/3) + 5n \\ T(n) &= T(n/9) + 5n/3 + 5n \\ &\vdots \\ T(n) &= T(n/3^k) + 5n/3^{k-1} + \cdots + 5n/3^0 \end{aligned}$$

for a positive integer k.

Once again we can simplify to get

$$T(n) = T(n/3^k) + 5n \sum_{i=0}^{k-1} \left(\frac{1}{3}\right)^i$$

and apply the useful fact (with $r = 1/3 < 1$) to find that the sum is $\Theta(1)$. Setting $k = \log_3 n$ we get $T(n) = T(1) + \Theta(n) = \Theta(n)$.

(d) $T(n) = T(\sqrt{n}) + 1$; $T(2) = 1$ $**$

(e) $T(n) = T(\sqrt{n}) + \sqrt{n}$; $T(2) = 1$ $**$

(f) $T(n) = \sqrt[4]{n}T(\sqrt{n}) + n$; $T(2) = 1$ $**$

(g) $T(n) = T(\log n) + 1$ $**$

(h) $T(n) = 2T(n-1) + g(n)$; $T(1) = 2$ $**$ — determine the maximum growth rate of $g(n)$ so that it does not affect the asymptotic solution of this recurrence. In other words, first set $g(n) = 0$ and solve the recurrence (see 2(c)). Now determine how large you can make $g(n)$ without asymptotically changing the solution to the recurrence.

4. Suppose we run the fast integer multiplication algorithm from class on the two numbers $x = 87$ ($= 1010111_2$ in binary) and $y = 67$ ($= 1000011_2$).

   The algorithm appears in the textbook as function *multiply*(x, y) in Figure 2.1 of Chap. 2. It is sometimes called Karatsuba's algorithm, after its inventor.

   Solution: There is actually an error in the algorithm as printed in the textbook. The algorithm fails when n is odd. See `http://cseweb.ucsd.edu/~dasgupta/book/errata.pdf` for the corrected version.

   A useful trick here is to add a leading 0 to both inputs to make the number of bits even. This way x_L and x_R (and similarly y_L and y_R) have the same number of bits, and we avoid having to worry about the floors and ceilings. You could also use the corrected version above, but note that using the algorithm as given in the textbook without padding (so $n = 7$) will produce the wrong answer.

   (a) For the above values and x and y, what are x_L, x_R, y_L, and y_R? Give these values in binary and decimal.

   Solution: If we do as suggested above and make the inputs 8 bits long (with a leading zero), we get $x_L = 0101_2$ and $x_R = 0111_2$ (the binary values). In decimal, this gives $x_L = 5$ and $x_R = 7$. Similarly, we get $y_L = 0100_2 = 4_{10}$ and $y_R = 0011_2 = 3_{10}$.

   (b) The algorithm works by making 3 recursive calls. For the above values of x and y, what are the arguments to the 3 recursive calls? Show the arguments in binary

and decimal. Make sure to give 2 arguments for each recursive call (for x and y), each in both binary and decimal.

Solution: Our recursive calls are:

$$P_1 = \text{multiply}(x_L, y_L) = \text{multiply}(5_{10}, 4_{10}) = \text{multiply}(0101_2, 0100_2)$$
$$P_2 = \text{multiply}(x_R, y_R) = \text{multiply}(7_{10}, 3_{10}) = \text{multiply}(0111_2, 0011_2)$$
$$P_3 = \text{multiply}(x_L + x_R, y_L + y_R) = \text{multiply}(12_{10}, 7_{10}) = \text{multiply}(1100_2, 0111_2)$$

(c) What are the values returned from each of the 3 recursive calls, *in decimal*? (You do not have to show the further execution of the algorithm on these values – just do the multiplication yourself however you would like.)

Solution: We get $P_1 = 5 \times 4 = 20$, $P_2 = 7 \times 3 = 21$, and $P_3 = 12 \times 7 = 84$.

(d) The algorithm takes the 3 returned values and plugs them into a formula to compute the final answer. Show this calculation *in decimal*, and give the final answer *in decimal*. (Check it: It should equal $87 \cdot 67$.)

Solution: Note that since we added a leading zero to the inputs, we have $n = 8$. Then, according to the algorithm, $xy = P_1 \times 2^8 + (P_3 - P_1 - P_2) \times 2^4 + P_2 = 20 \cdot 256 + 43 \cdot 16 + 21 = 5120 + 688 + 21 = 5829$. We check this by computing $87 \cdot 67 = 5829$, which is the same value computed by the algorithm.

5. A new super-advanced divide-and-conquer fast integer multiply algorithm does the following: split each of the input n-bit integers (viewed as bit strings) into 3 integers with $n/3$ bits each; perform $O(n)$ additional bit operations; and finally execute M recursive calls, each to multiply some pair of $(n/3)$-bit numbers. $M \geq 0$ is an integer constant. Let $T(n)$ be the number of bit operations taken by this algorithm to multiply two n-bit integers.

(a) Write down the recurrence describing $T(n)$. Your recurrence should contain M.

Solution: $T(n) = M \cdot T(n/3) + O(n)$.

(b) The solution to the recurrence, expressed as a function of n, depends on the value of the constant M. List the different solutions to the recurrence, and for each one, indicate which values of M would produce that solution. (For example, something like $T(n) = O(n^2)$ if $1 \leq M \leq 3$, $T(n) = O(n^4 \log n)$ if $4 \leq M \leq 17$, and $T(n) = O(n^{22})$ if $M > 17$.)

Solution: In the Master Theorem, we have $a = M$, $b = 3$, and $d = 1$. We have $\log_b a = \log_3 M$. The three cases of the Master Theorem state:

$$T(n) = \begin{cases} O(n) & \text{if } \log_3 M < 1 \\ O(n \log n) & \text{if } \log_3 M = 1 \\ O(n^{\log_3 M}) & \text{if } \log_3 M > 1 \end{cases}$$

We can solve each of these cases for M. In the first case, we have $M < 3$. Since $M \geq 0$ is an integer, this case amounts to $0 \leq M < 3$, or $M = 0, 1, 2$. The second case occurs only when $M = 3$. The final case occurs when $M > 3$.

(c) For which values of M will this algorithm be asymptotically faster than fast integer multiplication algorithm described in class and in the text book?

Solution: The fast integer multiplication algorithm from the textbook has a running time of $O(n^{\log_2 3})$. In the third case above ($M > 3$), $T(n) = O(n^{\log_3 M})$. In this case, the algorithm performs better than the textbook algorithm when $\log_3 M < \log_2 3$. This occurs when $M < 3^{\log_2 3}$, which is roughly 5.7. Since M must be an integer, we have $M \leq 5$. Note that in the other two cases, when $M \leq 3$, this algorithm still performs better asymptotically than the textbook's fast multiplication algorithm. So for all $M \leq 5$, this algorithm is asymptotically faster than the fast integer multiplication algorithm from the textbook.

6. Consider the following algorithm:

```
1   maybe_sort(A) // A is an array with at least one element
2     n = size(A) // compute the number of elements of A
3                 // Assume A is indexed from 1 to n
4     if n = 1: return
5     if n = 2: if A[1]>A[2]: swap(A[1],A[2]) // exchange the two values
6                 return
7     else:
8       Let B be the part of A containing its first 2n/3 elements.
9       Let C be the part of A containing its last 2n/3 elements.
10      Recursively call maybe_sort(B)
11      Recursively call maybe_sort(C)
12      return
```

An "expert" claims `maybe_sort` correctly sorts the array A in place. For the purposes of this problem, ignore rounding issues (treat $2n/3$ as $\lceil 2n/3 \rceil$ where necessary).

You may assume that when a subarray is passed to a recursive call, no array copying occurs, but the recursive call operates on a portion of the original array (this can be implemented by passing a reference to the original array and the indices delimiting the subarray).

(a) Give two arrays of 3 numbers: one for which, after the call of `maybe_sort` the array is sorted, one for which it is not.

The two arrays must *not* be sorted before the call to `maybe_sort`.

Solution: For $A = [1, 3, 2]$, the algorithm correctly sorts A: After the first recursive call, we have $A = [1, 3, 2]$, and after the second recursive call we have $A = [1, 2, 3]$.

However, for $A = [3, 2, 1]$, the algorithm does not correctly sort the array A. After the first recursive call, we get $A = [2, 3, 1]$, and after the second recursive call, we get $A = [2, 1, 3]$, which is not sorted.

(b) Ignoring the costs of lines 2, 8, and 9, write a recurrence that describes the running time $T(n)$ of `maybe_sort` on an array of size n. As discussed above, subarrays can be passed to recursive calls using constant time and storage.

(Because you are describing running time, the additive term in your recurrence should be in big-Oh or theta notation, e.g., $T(n) = 7T(n/29) + O(n^3)$.)

Solution: $T(n) = 2T(2n/3) + O(1)$.

(c) Solve your recurrence, giving the solution in big-Oh notation. You may use any method you like to solve the recurrence.

Solution: Perhaps the easiest way to solve the recurrence is with the Master Method. We have $a = 2$, $b = 3/2$, and $d = 1$. Since $1 < \log_{3/2} 2 \approx 1.71$, the Master Theorem tells us that $T(n) = O(n^{\log_{3/2} 2}) = O(n^{1.71})$.

(d) Now add before line 12, the following *new line*:

```
Recursively call maybe_sort(B)
```

i. Prove the resulting new algorithm actually sorts A correctly.
Solution: Lets begin by examining where the algorithm fails (without the additional recursive call).
The first recursive call does not touch any elements in the final third of the input array. So any elements in the final third of the array will remain in the final third (and the order unchanged) after the first recursive call. Further note that the second recursive call cannot move any elements to the first third of the input array.
So why is this bad? If any elements in the final third of the input array belong in the first third of the sorted array, they will never be sorted into their final position. This is because after the first recursive call, these elements are still in the final third of the array. When `maybe_sort` is called recursively the second time, these elements must be moved to the first third of the array, but as we've already noted, the second recursive call cannot move any elements into this portion of the array. So the algorithm fails.
Why does the third recursive call remedy the issue? Because after the second recursive call, these problem elements will be moved to the beginning of array C, which places them in the second third of the original input array. The final recursive call will then be able to move these elements from the second third of the array to the first third, where they belong.
Note that while this is a rough intuitive explanation of why the additional recursive call fixes a shortcoming in the original algorithm, this is *not* a formal proof. We may update this solution set later to include a full proof.

ii. Give a recurrence describing its running time.
Solution: The additional recursive call roduces a recurrence of $T(n) = 3T(2n/3) + O(1)$.

iii. Solve the recurrence. How does the algorithm compare to MergeSort?

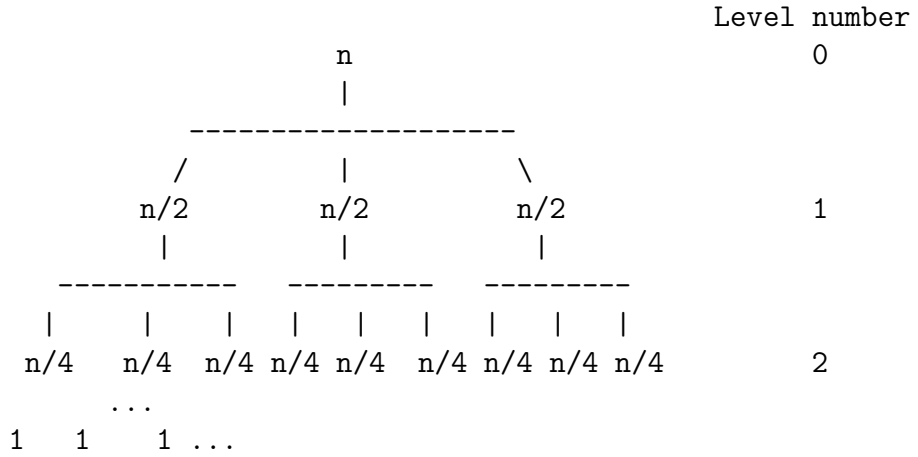
Solution: Using Master Theorem, we get $\log_b a = \log_{3/2} 3 \approx 2.71 > 1$, so we have $T(n) = O(n^{\log_{3/2} 3}) = O(n^{2.71})$. This is quite a bit slower than MERSESORT's $O(n \log n)$ running time (and indeed is even worse than other inefficient sorting algorithms such as Bubble Sort and Insertion Sort, which both have running time $O(n^2)$).

Example for Problem 2

Problem: $T(n) = 3T(n/2) + 4; T(1) = 4$; check at $T(8)$.

Solution: $T(8) = 3T(4) + 4$. $T(4) = 3T(2) + 4$. $T(2) = 3T(1) + 4 = 3 \cdot 4 + 4 = 16$. Working back up we get $T(4) = 3 \cdot 16 + 4 = 52$. So $T(8) = 3 \cdot 52 + 4 = 160$.

We represent the recurrence using a tree. The values in the nodes are the arguments to T.



The value in the nodes at Level 0 is n, at Level 1 is $n/2$ and at Level 2 is $n/4$. The value decreases by a factor of 2 as you go down. So the value in the nodes at Level i would be $n/2^i$. Since the bottom level nodes have the value 1, the level number at the bottom satisfies $n/2^i = 1$. Solving for i, we get that the bottom level is Level $\log_2 n$.

The number of nodes at level i is 3^i. In the levels above the bottom, the "local contribution" at each node is 4, because of the +4 in the recurrence. The "local contribution" at the nodes in the bottom level is also 4, because $T(1) = 4$.

Therefore, the total local contribution over all nodes at level i is (# nodes at level i)$\times$(local contribution at each node of level i) $= 3^i \times 4$.

Summing the total local contribution over all levels, starting from the top, we get

$$\begin{aligned}
T(n) &= 4 \times 3^0 + 4 \times 3^1 + 4 \times 3^2 + ... + 4 \times 3^{\log_2 n} \\
&= 4(3^0 + 3^1 + 3^2 + ... + 3^{\log_2 n}) \\
&= 4(\frac{3^{\log_2 n+1} - 1}{3 - 1}) \\
&= 2(3^{\log_2 n+1} - 1) \\
&= 2(3 \times 3^{\log_2 n} - 1) \\
&= 2(3 \times (2^{\log_2 3})^{\log_2 n} - 1) \\
&= 2(3 \times n^{\log_2 3} - 1).
\end{aligned}$$

Checking for accuracy: According to this solution, $T(8) = 2(3 \times 8^{\log_2 3} - 1)$. Using the fact that $8^{\log_2 3} = (2^3)^{\log_2 3} = (2^{\log_2 3})^3 = 3^3 = 27$, we can plug that in to our expression for $T(8)$ to get $T(8) = 2(3 \times 27 - 1) = 160$. Which is the same as the value we calculated above.