CSE103 Introduction to Algorithms

TD4: More Complexity Analysis and Recurrences, First Exercises on Program Correctness

April 8th, 2021

Solving recurrences via a change of variable. Sometimes a daunting-looking recurrence may be solved very quickly by changing variables. Take for instance

$$T(n) = 2\,T\left(\sqrt{n}\right) + \log n.$$

The Master Theorem does not apply here ($\sqrt{n}$ is not of the form $\frac{n}{b}$, and even if it were, $\log n$ is not $\Theta(n^c)$ for any c). If you try to apply the recursion tree method, you get swamped pretty quickly in a jungle of iterated logs and square roots. However, $\sqrt{n}$ *is* equal to $n^{\frac{1}{2}}$, it's just that the $\frac{1}{2}$ is at the "exponent level" and not at "ground level"...

Let

$$S(m) := T(2^m).$$

By definition, this implies

$$S\left(\frac{m}{2}\right) = T\left(2^{\frac{m}{2}}\right),$$
$$S(\log n) = T\left(2^{\log n}\right) = T(n).$$

Let's unravel the definition of S using the recurrence defining T and the first equation above:

$$S(m) = T(2^m) = 2\,T\left(\sqrt{2^m}\right) + \log 2^m = 2\,T\left(2^{\frac{m}{2}}\right) + m = 2\,S\left(\frac{m}{2}\right) + m.$$

So S verifies the much friendlier recurrence $S(m) = 2\,S\left(\frac{m}{2}\right) + m$! We may solve it immediately using the Master Theorem, with $a = 2$, $b = 2$ and $c = 1$, which gives us $S(m) = \Theta(m \log m)$. Using the second equation above, we get

$$T(n) = S(\log n) = \Theta(\log n \log\log n),$$

which solves the original recurrence.

Exercise 1: solving recurrences

Give the best asymptotic upper bound you can to the following recurrences (where, in all cases, $T(0) = 1$):

1. $T(n) = 2\,T\left(\frac{n}{4}\right) + n^{0.51}$
2. $T(n) = 4\,T\left(\frac{n}{2}\right) + 7n$
3. $T(n) = 3\,T\left(\frac{n}{4}\right) + n \log n$
4. $T(n) = 3\,T\left(\frac{n}{3}\right) + \frac{n}{2}$
5. $T(n) = 7\,T\left(\frac{n}{3}\right) + n^2$
6. $T(n) = 6\,T\left(\frac{n}{3}\right) + n^2 \log n$
7. $T(n) = 5\,T\left(\frac{n}{2}\right) + 10n + 3n^2$
8. $T(n) = 4\,T\left(\sqrt{n}\right) + \log^2 n$
9. $T(n) = 2\,T\left(\frac{n}{8}\right) + \frac{1}{n+1} + \sqrt[3]{n}$
10. $T(n) = 65\,T(n-3) + 4^n$

Solution.

1. The Master Theorem with $a = 2$, $b = 4$ and $c = 0.51$ gives us $T(n) = \Theta(n^{0.51})$.

2. Notice that $7n = \Theta(n)$, so we apply the Master Theorem with $a = 4$, $b = 2$ and $c = 1$, which gives us $T(n) = \Theta(n^2)$.

3. The Master Theorem does not apply, because $n \log n$ is not $\Theta(n^c)$ for any c. There is no obvious way of changing variables to get something of the right form, so we try the recursion tree method. The tree is very regular: it is balanced, every node has 3 children. The input size is divided by 4 at each level, so the input size at level k is $4^{-k}n$. This means that the cost of each individual node at level k is $(4^{-k}n)\log(4^{-k}n) = 4^{-k}n(\log n - 2k)$. Since there are 3^k nodes at level k, the sum of the costs at level k is

$$3^k 4^{-k} n(\log n - 2k) = \left(\frac{3}{4}\right)^k n(\log n - 2k) \leq \left(\frac{3}{4}\right)^k n \log n$$

(because $k \geq 0$). Regardless of how many levels there are, the total sum will be bounded by

$$T(n) \leq \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k n \log n = \frac{n \log n}{1 - \frac{3}{4}} = 4n \log n.$$

Let us try to use directly the constant 4, *i.e.*, we guess that, for all $n \geq m$ for some m to be determined, we have $T(n) \leq 4n \log n$. We prove this by induction, starting with the inductive case:

$$\begin{aligned} T(n) &= 3\,T\left(\frac{n}{4}\right) + n \log n \leq 3 \cdot 4\frac{n}{4} \log \frac{n}{4} + n \log n \\ &= 3n(\log n - 2) + n \log n = 4n \log n - 6n \leq 4n \log n \end{aligned}$$

because $n \geq 0$. For the base case, we see that $T(2) = 5 \leq 8 = 4 \cdot 2 \log 2$, so we may conclude by taking $m = 2$ that $T(n) = O(n \log n)$. We actually have $T(n) = \Theta(n \log n)$ because that's also the root of the recursion tree.

4. Since $\frac{n}{2} = \Theta(n)$, we may apply the Master Theorem with $a = 3$, $b = 3$ and $c = 1$, which gives us $T(n) = \Theta(n \log n)$.

5. Straightforward application of the Master Theorem with $a = 7$, $b = 3$ and $c = 2$, which gives us $T(n) = \Theta(n^2)$.

6. This is very similar to point 3. The recursion tree is again extremely regular, each node has 6 children, the input size is $3^{-k}n$ at level k and each node at level k has cost

$$(3^{-k}n)^2 \log(3^{-k}n) = 3^{-2k}n^2(\log n - k \log 3).$$

Since there are 6^k nodes at level k, the cost at level k is

$$6^k 3^{-2k} n^2(\log n - k \log 3) = 2^k 3^{-k} n^2(\log n - k \log 3) = \left(\frac{2}{3}\right)^k n^2(\log n - k \log 3) \leq \left(\frac{2}{3}\right)^k n^2 \log n$$

because $k \geq 0$. Regardless of how many levels there are, we have

$$T(n) \leq \sum_{k=0}^{\infty} \left(\frac{2}{3}\right)^k n^2 \log n = \frac{n^2 \log n}{1 - \frac{2}{3}} = 3n^2 \log n.$$

We guess directly $T = O(n^2 \log n)$ with constant $c = 3$, but this time we start with attempting to determine m. We try out the first few values of T: $T(0) = 1$, $T(1) = 6$, $T(2) = 10$; on the other hand, the first few values of $g(n) := 3n^2 \log n$ are $g(0) =$ undefined, $g(1) = 0$, $g(2) = 12 > T(2)$, so $m = 2$ works. Now we try the inductive step:

$$\begin{aligned} T(n) &= 6\,T\left(\frac{n}{3}\right) + n^2 \log n \leq 6 \cdot 3 \left(\frac{n}{3}\right)^2 \log \frac{n}{3} + n^2 \log n \\ &= 2n^2(\log n - \log 3) + n^2 \log n = 3n^2 \log n - (2\log 3)n^2 \leq g(n), \end{aligned}$$

it works! We conclude $T(n) = \Theta(n^2 \log n)$ (the lower bound is given by the root of the recursion tree).

7. Notice that $10n + 3n^2 = \Theta(n^2)$, so we apply the Master Theorem with $a = 5$, $b = 2$ and $n = 2$, obtaining $T(n) = \Theta(n^{\log 5})$.

8. This looks a lot like a case in which changing variable helps: if we let $S(m) := T(2^m)$, we obtain $S(m) = 4\,T\left(\sqrt{2^m}\right) + (\log 2^m)^2 = 4\,T\left(2^{\frac{m}{2}}\right) + m^2$. Observing that, by definition, $S(\frac{m}{2}) = T\left(2^{\frac{m}{2}}\right)$, we have that S satisfies the recurrence

$$S(m) = 4\,S\left(\frac{m}{2}\right) + m^2,$$

which may be solved immediately as an application of the Master Theorem with $a = 4$, $b = 2$ and $c = 2$, yielding $S(m) = \Theta(m^2 \log m)$. Since, by definition $S(\log n) = T(n)$, we have $T(n) = \Theta(\log^2 n \log\log n)$.

9. Again, $\frac{1}{1+n} + \sqrt[3]{n} = \Theta(\sqrt[3]{n})$, so we apply the Master Theorem with $a = 2$, $b = 8$ and $c = \frac{1}{3}$, obtaining $T(n) = \Theta(\sqrt[3]{n} \log n)$.

10. The odd shape of this recurrence hints to a change of variable. Let us define $S(m) := T(\log m)$. We have

$$S(m) = T(\log m) = 65\,T(\log m - 3) + 4^{\log m} = 65\,T(\log m - \log 8) + 2^{\log m^2} = 65\,T\left(\log \frac{m}{8}\right) + m^2.$$

Since, by definition, we have $S\left(\frac{m}{8}\right) = T\left(\log \frac{m}{8}\right)$, we have that S satisfies the recurrence

$$S(m) = 65\,S\left(\frac{m}{8}\right) + m^2,$$

which may be solved immediately with the Master Theorem, with $a = 65$, $b = 8$ and $c = 2$, giving us $S(m) = \Theta(m^{\log_8 65})$. Since, by definition, $S(2^n) = T(n)$, we have $T(n) = \Theta(2^{(\log_8 65)n}) = O(2^{2.0075n}) = O(4.021^n)$.

Exercise 2: complexity analysis of (nested) loops

Give the asymptotic complexity of the following Python definitions of `f(n)`, taking `n` itself as the input size (all comparisons, arithmetic operations and `math.sqrt` are assumed to cost $O(1)$):

1.
```python
def f(n):
    for i in range(n):
        j = i
        while j > 0:
            # some instructions of cost O(1)
            j -= 1
    for j in range(n):
        # some instructions of cost O(1)
```

2.
```python
import math
def f(n):
    for i in range(int(math.sqrt(n))//2):
        # some instructions of cost O(1)
    for j in range(int(math.sqrt(n))//4):
        # some instructions of cost O(1)
    for k in range(j+7):
        # some instructions of cost O(1)
```

3.
```python
import math
def f(n):
    for i in range(int(math.sqrt(n))//2):
        # some instructions of cost O(1)
        for j in range(int(math.sqrt(n))//4):
            # some instructions of cost O(1)
            for k in range(j, j+7):
                # some instructions of cost O(1)
```

4.
```python
import math
def f(n):
    for i in range(int(math.sqrt(n))//2):
        # some instructions of cost O(1)
        for j in range(i, i+7):
            # some instructions of cost O(1)
            for k in range(j, j+7):
                # some instructions of cost O(1)
```

5.
```python
import math
def f(n):
    h = int(math.sqrt(n)) // 2
    for i in range(h):
        # some instructions of cost O(1)
        for j in range(i * i):
            # some instructions of cost O(1)
            for k in range(1, j):
                if j % h == 0:
                    # some instructions of cost O(1)
```

6.
```python
import math
def f(n):
    h = int(math.sqrt(n)) // 2
    for i in range(h):
        # some instructions of cost O(1)
        for j in range(i * i):
            # some instructions of cost O(1)
            for k in range(1, j):
                if j % h != 0:
                    # some instructions of cost O(1)
```

7.
```python
import math
def f(n):
    h = int(math.sqrt(n)) // 2
```

```python
for i in range(h):
    # some instructions of cost O(1)
    for j in range(i * i):
        # some instructions of cost O(1)
        if j % h == 0:
            for k in range(1, j):
                # some instructions of cost O(1)
```

Solution.

1. Since $\texttt{j} \leq \texttt{n}$, we may bound the inner `while` loop with $O(n)$, so the total complexity of the first `for` loop is $O(n^2)$. The second `for` loop is obviously $O(n)$, which is absorbed by the first in the big O notation, so the complexity of `f` is $O(n^2)$.

2. We have three successive loops, so the complexity of `f` is given by the most complex one. The first two obviously have complexity $O(\sqrt{n})$. For the third one, the value of `j` is the one attained at the end of the second loop, which is `int(math.sqrt(n))//4 - 1`, so the third loop is executed `int(math.sqrt(n))//4 + 6` times, which is still $O(\sqrt{n})$. So the complexity of `f` is $O(\sqrt{n})$.

3. There are three nested `for` loops, the first two obviously have complexity $O(\sqrt{n})$. The innermost loop is always executed 7 times, independently of the value of `j`, so its complexity is $O(1)$. Therefore, the total complexity of `f` is $O((\sqrt{n})^2) = O(n)$.

4. This time, both inner loops have complexity $O(1)$, so the total complexity of `f` is $O(\sqrt{n})$.

5. The outer loop is executed $O(\texttt{h})$ times. For the middle loop, we may bound `i` with `h`, so we may bound the complexity of the loop with $O(\texttt{h}^2)$. For what concerns the innermost loop, we may bound `j` with $O(\texttt{h}^2)$. The fact that the instructions after the `if` are not always executed has no impact on asymptotic complexity, because the test `j % h == 0`, which has complexity $O(1)$, is always executed, so the loop costs $O(\texttt{h}^2)$. Therefore, the complexity of `f` is $O(\texttt{h} \cdot \texttt{h}^2 \cdot \texttt{h}^2) = O(\texttt{h}^5)$. Since $\texttt{h} = O(\sqrt{n})$, this is equal to $O(n^{2.5})$.

6. The fact that the `if` condition in the innermost loop is the negation of the one of the previous exercise changes nothing: every loop is executed the same number of times, and the `if` condition is tested the same number of times. The complexity of `f` is therefore still $O(n^{2.5})$.

7. In this case, the fact of having moved the `if` to the middle loop has an impact on the asymptotic complexity, because now the inner loop, which does *not* have constant cost, is executed only once every `h` times. So, the outer loop is executed `h` times; the middle loop is executed $O(\texttt{h}^2)$ times for each execution of the outer loop, but the inner loop will be executed only $O(\texttt{h})$ times out of those. Therefore, although the `if` test is executed $O(\texttt{h} \cdot \texttt{h}^2) = O(\texttt{h}^3)$ times, the inner loop, which costs $O(\texttt{h}^2)$, is executed only $O(\texttt{h} \cdot \texttt{h}) = O(\texttt{h}^2)$ times. This gives a complexity of $O(\texttt{h}^3 + \texttt{h}^2 \cdot \texttt{h}^2) = O(\texttt{h}^4)$: the first component of the sum is the number of times the `if` is executed, the second component is the number of times the $O(1)$ instructions inside the inner loop are executed. Asymptotically, the cost of the latter dominates on the cost of the former. Since $\texttt{h} = O(\sqrt{n})$, we have that the complexity of `f` is $O(n^2)$.

Exercise 3: an old acquaintance

The very first exercise of TD1 asked you to find, by exhaustive search, the integer square root of a non-negative integer. It looked something like this:

```
def sqrt(n):
    r = 0
    while r * r < n:
        r += 1
    if r * r != n:
        r -= 1
    return r
```

Prove that the above algorithm is correct by showing that the Hoare triple

$$\{0 \le \mathtt{n}\}\ \mathtt{sqrt(n)}\ \{\mathtt{r}^2 \le \mathtt{n} \wedge (\mathtt{r}+1)^2 > n\}$$

is valid. That is, start with the assertion $0 \le \mathtt{n}$ just before the instruction `r = 0` and show how it may be propagated down, using the rules of Floyd-Hoare logic, until you arrive at the assertion $\mathtt{r}^2 \le \mathtt{n} \wedge (\mathtt{r}+1)^2 > n$ just before the instruction `return r`. (*Hint: a possible loop invariant is* $\mathrm{pred}(\mathtt{r})^2 \le \mathtt{n}$, *where* pred *is* truncated subtraction, *defined by* $\mathrm{pred}(x) = x - 1$ *for all* $x > 0$, *and* $\mathrm{pred}(x) = 0$ *for all* $x \le 0$. *Remember that we treat* `if` *statements without* `else` *by adding an* `else` *branch of the form* `else: pass`).

Solution. Here is the program annotated with the assertions:

```
def sqrt(n):
    #! 0 ≤ n
    #! 0 ≤ n ∧ pred(0)^2 ≤ n  (because pred(0) = 0)
    r = 0
    #! 0 ≤ n ∧ pred(r)^2 ≤ n
    while r * r < n:
        #! 0 ≤ n ∧ pred(r)^2 ≤ n ∧ r^2 < n
        #! 0 ≤ n ∧ r^2 ≤ n
        #! 0 ≤ n ∧ pred(r + 1)^2 ≤ n  (because pred(r + 1) = r)
        r = r + 1
        #! 0 ≤ n ∧ pred(r)^2 ≤ n
    #! 0 ≤ n ∧ pred(r)^2 ≤ n ∧ r^2 ≥ n
    if r * r != n:
        #! 0 ≤ n ∧ pred(r)^2 ≤ n ∧ r^2 ≥ n ∧ r^2 ≠ n
        #! 0 ≤ n ∧ pred(r)^2 ≤ n ∧ r^2 > n
        #! 0 ≤ n ∧ pred(r)^2 ≤ n ∧ r^2 > n ∧ r > 0  (because r^2 > n ≥ 0)
        #! 0 ≤ n ∧ (r − 1)^2 ≤ n ∧ (r − 1 + 1)^2 > n  (because pred(r) = r − 1 when r > 0)
        r = r - 1
        #! 0 ≤ n ∧ r^2 ≤ n ∧ (r + 1)^2 > n
        #! r^2 ≤ n ∧ (r + 1)^2 > n
    else:
        #! 0 ≤ n ∧ pred(r)^2 ≤ n ∧ r^2 ≥ n ∧ r^2 = n
        #! r^2 ≤ n ∧ (r + 1)^2 > n  (because r^2 = n obviously implies all this)
        pass
        #! r^2 ≤ n ∧ (r + 1)^2 > n
    #! r^2 ≤ n ∧ (r + 1)^2 > n
    return r
```