Week 2 Studio Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the tutorial classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to the tutorial problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, psuedocode may be provided where it illustrates a particular useful concept.

Assessed Preparation

Problem 1. Consider the following algorithm that returns the number of occurrences of *target* in the sequence A. Identify a useful invariant that is true at the beginning of each iteration of the **while** loop. Prove that it holds, and use it to prove that the algorithm is correct.

```
1:  function COUNT(A[1..n], target)
2:      count = 0
3:      i = 1
4:      while i ≤ n do
5:          if A[i] = target then
6:              count = count + 1
7:          end if
8:          i = i + 1
9:      end while
10:     return count
11: end function
```

Solution

A useful invariant is that, at the start of iteration i, `count` is equal to the number of occurrences of `target` in $A[1..i-1]$, where we consider $A[1..0]$ to be an empty list.

Note: To prove this loop invariant, we will use induction. First we show that the invariant holds at initialisation, at the start of the first iteration of the loop. This is our base case. Next we assume that the invariant holds at the start of some iteration of the loop, and show that it still holds at the start of the next iteration. At this point we are done, since we have shown that the invariant holds at the start of the first loop, and that if it holds at the start of loop i, it also holds at the start of loop $i+1$. This means it holds at the start of every loop, and importantly, that it holds at the start of the loop where the loop condition is false, i.e., it holds when the loop ends.

Proof: At the start of the first iteration, $i = 1$. Also, `count` $= 0$, so `count` is equal to the number of occurrences of `target` in $A[1..0]$, since $A[1..0]$ is an empty list. So the invariant is true at initialisation.

Assume that the invariant holds at the start of k-th iteration. So `count` is equal to the number of occurrences of `target` in $A[1..k-1]$. Call this number of occurrences c. During this iteration of the loop, $i = k$. If $A[k] =$ `target`, we will increment `count`, so `count` will equal $c+1$, which is the number of occurrences of `target` in $A[1..k]$. If $A[k] \neq$ `target`, then `count` will not be changed, so `count` will equal c, which is equal to the number of occurrences of `target` in $A[1..k]$. Either way the invariant holds at the start of $k+1^{th}$ iteration, that is, `count` is equal to the number of occurrences of `target` in $A[1..k]$. Since we know the invariant holds at the start, by induction it holds for all values of i, including when $i = n+1$, so the invariant holds.

To prove the algorithm is correct, we need to show that at loop termination, `count` is equal to the number of occurrences of `target` in A. The invariant tells us that `count` is equal to the number of occurrences of target in $A[1..i-1]$, but at loop termination, $i = n+1$, so `count` is equal to the number of occurrences of target in $A[1..n]$ which is all of A. Therefore the algorithm is correct.

Problem 2. Find a closed form solution for the following recurrence relation:

$$T(n) = \begin{cases} T(n-1)+c, & \text{if } n > 0, \\ b, & \text{if } n = 0. \end{cases}$$

Solution

To find a closed form for T, we will use the method of telescoping, i.e. we look for a pattern as we substitute the recurrence relation into itself. For $n > 0$ we have

$$\begin{aligned} T(n) &= T(n-1)+c, \\ T(n) &= (T(n-2)+c)+c = T(n-2)+2c, \\ T(n) &= (T(n-3)+c)+2c = T(n-3)+3c. \end{aligned}$$

Continuing the pattern, we see that the general form for $T(n)$ seems to be

$$T(n) = \begin{cases} T(n-k)+kc, & \text{if } n > 0, \text{for all } k \leq n, \\ b, & \text{if } n = 0. \end{cases}$$

We want a closed-form solution, so we need to eliminate the term $T(n-k)$. To do so, we make use of the fact that we have $T(0) = b$. By setting $k = n$, $T(n-k)$ becomes $T(0)$ and hence we obtain

$$\begin{aligned} T(n) &= T(n-n)+nc, \\ &= T(0)+nc, \\ &= b+nc. \end{aligned}$$

To prove this is correct, we check to see if our equation satisfies the recurrence. Manipulating $T(n)$, we see that

$$T(n) = nc+b = (n-1)c+b+c = T(n-1)+c,$$

as required. We also have $T(0) = b+0 = b$ as required. Hence this is a valid solution.

Studio Problems

Problem 3. Write psuedocode for insertion sort, except instead of sorting the elements into non-decreasing order, sort them into non-increasing order. Identify a useful invariant of this algorithm.

Solution

We write the usual insertion sort algorithm, except that when performing the insertion step, we loop as long as $A[j] <$ `key`, rather than $A[j] >$ `key`, so that larger elements get moved to the left.

```
1: function INSERTION_SORT(A[1..n])
2:     for i = 2 to n do
3:         Set key = A[i]
4:         Set j = i - 1
```

```
5:        while j ≥ 1 and A[j] < key do
6:            A[j+1] = A[j]
7:            j = j - 1
8:        end while
9:        A[j+1] = key
10:   end for
11: end function
```

A useful invariant is that at the end of iteration i, the sub-array $A[1..i]$ is sorted in non-increasing order.

Problem 4. Find a closed form for the following recurrence relation:

$$T(n) = \begin{cases} 3T(n-1), & \text{if } n > 0, \\ c, & \text{if } n = 0. \end{cases}$$

Solution

To find a closed form for T, we will use the method of telescoping. For $n > 0$ we have

$$\begin{aligned} T(n) &= 3T(n-1), \\ T(n) &= 3(3T(n-2)) = 3^2 T(n-2), \\ T(n) &= 3^2(3T(n-3)) = 3^3 T(n-3). \end{aligned}$$

Continuing the pattern, we see that the general form for $T(n)$ seems to be

$$T(n) = \begin{cases} 3^k T(n-k), & \text{if } n > 0, \text{for all } k \leq n, \\ c, & \text{if } n = 0. \end{cases}$$

We want a closed-form solution, so we need to eliminate the term $T(n-k)$. To do so, we make use of the fact that we have $T(0) = c$. By setting $k = n$, $T(n-k)$ becomes $T(0)$ and hence we obtain

$$\begin{aligned} T(n) &= 3^n T(n-n), \\ &= 3^n T(0), \\ &= c3^n. \end{aligned}$$

We should now check that this solution is correct. Substituting our proposed solution into the recurrence, we find

$$3T(n-1) = 3(c3^{n-1}) = c(3 \times 3^{n-1}) = c3^n = T(n),$$

as required. We also have $T(0) = c \times 3^0 = c$ as required.

Problem 5. Find a closed form for the following recurrence relation:

$$T(n) = \begin{cases} 2T(n-1) + a, & \text{if } n > 0, \\ b, & \text{if } n = 0. \end{cases}$$

Solution

To find a closed form for T, we will use the method of telescoping. For $n > 0$ we have

$$\begin{aligned}
T(n) &= 2T(n-1)+a,\\
T(n) &= 2(2T(n-2)+a)+a = 2^2T(n-2)+(1+2)a,\\
T(n) &= 2^2(2T(n-3)+a)+3a = 2^3T(n-3)+(1+2+4)a
\end{aligned}$$

Continuing the pattern, we see that the general form for $T(n)$ seems to be

$$T(n) = \begin{cases} 2^kT(n-k)+(1+2+\ldots+2^{k-1})a, & \text{if } n>0, \text{for all } k \le n,\\ b, & \text{if } n=0. \end{cases}$$

First, we will evaluate the sum $1+2+\ldots+2^{k-1}$. Recall from the Week 1 studio sheet that this sum evaluates to 2^k-1, and hence we have

$$T(n) = \begin{cases} 2^kT(n-k)+(2^k-1)a, & \text{if } n>0, \text{for all } k \le n,\\ b, & \text{if } n=0. \end{cases}$$

We want a closed-form solution, so we need to eliminate the term $T(n-k)$. To do so, we make use of the fact that we have $T(0)=b$. By setting $k=n$, $T(n-k)$ becomes $T(0)$ and hence we obtain

$$\begin{aligned}
T(n) &= 2^nT(n-n)+(2^n-1)a,\\
&= 2^nT(0)+(2^n-1)a,\\
&= 2^nb+(2^n-1)a.
\end{aligned}$$

We now verify that this solution is correct by substitution. The recurrence reads

$$\begin{aligned}
2T(n-1)+a &= 2(2^{n-1}b+(2^{n-1}-1)a)+a,\\
&= 2\times 2^{n-1}b+2\times(2^{n-1}-1)a+a,\\
&= 2^nb+2^na-2a+a,\\
&= 2^nb+(2^n-1)a,\\
&= T(n)
\end{aligned}$$

as required. We also have $T(0) = 2^0b+(2^0-1)a = b$ as required.

Problem 6. Find a closed form for the following recurrence relation:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right)+n & \text{if } n>1,\\ 1 & \text{if } n=1. \end{cases}$$

State the order of growth of the solution using big-O notation.

Solution

To find a closed form for T, we will use the method of telescoping. For $n > 1$ we have

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right)+n,\\
T(n) &= 2\left[2T\left(\frac{n}{4}\right)+\frac{n}{2}\right]+n = 2^2T\left(\frac{n}{4}\right)+2n,\\
T(n) &= 2^2\left[2T\left(\frac{n}{8}\right)+\frac{n}{4}\right]+2n = 2^3T\left(\frac{n}{8}\right)+3n
\end{aligned}$$

Continuing the pattern, we see that the general form for $T(n)$ seems to be

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

We want a closed-form solution, so we need to eliminate the term $T\left(\frac{n}{2^k}\right)$. To do so, we make use of the fact that we have $T(1) = 1$. By setting $k = \log_2(n)$, $T\left(\frac{n}{2^k}\right)$ becomes $T(1)$ and hence we obtain

$$\begin{aligned} T(n) &= 2^{\log_2(n)} T\left(\frac{n}{2^{\log_2(n)}}\right) + n\log_2(n) \\ &= nT(1) + n\log_2(n) \\ &= n + n\log_2(n) \end{aligned}$$

In order to check that our solution is correct, we can substitute it back into the original recurrence.

$$\begin{aligned} 2T\left(\frac{n}{2}\right) + n &= 2\left(\frac{n}{2} + \frac{n}{2}\log_2\left(\frac{n}{2}\right)\right) + n, \\ &= n + n\log_2\left(\frac{n}{2}\right) + n, \\ &= n + n\left(\log_2\left(\frac{n}{2}\right) + 1\right), \\ &= n + n\left(\log_2\left(\frac{n}{2}\right) + \log_2(2)\right), \\ &= n + n\left(\log_2\left(\frac{n}{2} \times 2\right)\right), \\ &= n + n\log_2(n), \\ &= T(n) \end{aligned}$$

as required. We also have that $T(1) = 1 + \log_2(1) = 1$ as required. The asymptotic behaviour of this function is $O(n\log(n))$.

Problem 7. Use mathematical induction to prove that the recurrence relation

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + c, & \text{if } n > 1, \\ b, & \text{if } n = 1, \end{cases}$$

has a solution given by $T(n) = b + c\log_2(n)$ for all $n = 2^k$ for some $k \geq 0$.

Solution

Let's denote the proposed solution by

$$T'(n) = b + c\log_2 n.$$

We want to show that $T(n) = T'(n)$ for all $n = 2^k$, for $k \geq 0$.

Base Case:

Let $n = 1$.

$$T(1) = b = b + c * 0 = b + c\log_2(1) = T'(1)$$

as required.

Inductive Case:

Assume $T(m) = T'(m)$ for some $m = 2^k, k \geq 0$. Since the recurrence only makes sense for powers of two, we will not attempt to show that $T(m+1) = T'(m+1)$, but rather that $T(2m) = T'(2m)$. Beginning with

the left hand side,

$$\begin{aligned} T(2m) &= T\left(\frac{2m}{2}\right) + c, \\ &= T(m) + c. \end{aligned}$$

We invoke the inductive hypothesis that $T(m) = T'(m)$ so that we can write

$$T(m) + c = T'(m) + c.$$

Finally, we obtain the desired result by rearranging and using log laws.

$$\begin{aligned} T'(m) + c &= b + c\log_2(m) + c, \\ &= b + c\log_2(m) + c\log_2(2), \\ &= b + c\log_2(2m), \\ &= T'(2m), \end{aligned}$$

and hence $T(2m) = T'(2m)$ as required. Therefore, by induction on n, it is true that

$$T(n) = T'(n), \qquad \text{for } n = 2^k, k \geq 1.$$

and hence that $b + c\log_2(n)$ is a solution for $T(n)$.

Problem 8. Let $F(n)$ denote the n^{th} Fibonacci number. The Fibonacci sequence is defined by the recurrence relation $F(n) = F(n-1) + F(n-2)$, with $F(0) = 0, F(1) = 1$.

(a) Use mathematical induction to prove the following property for $n \geq 1$:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}$$

(b) Prove that $F(n)$ satisfies the following properties:

$$F(2k) = F(k)[2F(k+1) - F(k)] \tag{1}$$

$$F(2k+1) = F(k+1)^2 + F(k)^2 \tag{2}$$

[Hint: Use part (a)]

Solution

(a) Define the following:

$$L(n) = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n, \qquad R(n) = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}.$$

We want to show that $L(n) = R(n)$ for all $n \geq 1$.

Base Case:

Let $n = 1$. We have

$$L(1) = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F(2) & F(1) \\ F(1) & F(0) \end{bmatrix} = R(1),$$

as required.

Inductive case:

Assume that $L(k) = R(k)$ for some $k \geq 1$, i.e. assume that

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^k = \begin{bmatrix} F(k+1) & F(k) \\ F(k) & F(k-1) \end{bmatrix}.$$

We want to prove that $L(k+1) = R(k+1)$. Beginning with the left hand side, we express $L(k+1)$ in terms of $L(k)$ by writing

$$\begin{aligned} L(k+1) &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{k+1}, \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^k \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1, \\ &= L(k) \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}. \end{aligned}$$

Invoking the inductive hypothesis that $L(k) = R(k)$, we can write

$$L(k) \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = R(k) \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix},$$

and proceed to show that

$$\begin{aligned} R(k) \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} &= \begin{bmatrix} F(k+1) & F(k) \\ F(k) & F(k-1) \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}, \\ &= \begin{bmatrix} F(k+1)+F(k) & F(k+1) \\ F(k)+F(k-1) & F(k) \end{bmatrix}, \\ &= \begin{bmatrix} F(k+2) & F(k+1) \\ F(k+1) & F(k) \end{bmatrix}, \\ &= R(k+1). \end{aligned}$$

Hence $L(k+1) = R(k+1)$ as required. Therefore, by induction on n, it is true that

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix} \qquad \text{for all } n \geq 1.$$

Solution

(b) From part (a) we have

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}.$$

Substituting $n = 2k$ we have

$$\begin{aligned}
\begin{bmatrix} F(2k+1) & F(2k) \\ F(2k) & F(2k-1) \end{bmatrix} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{2k}, \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{k} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{k}, \\
&= \begin{bmatrix} F(k+1) & F(k) \\ F(k) & F(k-1) \end{bmatrix} \begin{bmatrix} F(k+1) & F(k) \\ F(k) & F(k-1) \end{bmatrix}, \\
&= \begin{bmatrix} F(k+1)^2 + F(k)^2 & F(k+1)F(k) + F(k)F(k-1) \\ F(k)F(k+1) + F(k-1)F(k) & F(k)^2 + F(k-1)^2 \end{bmatrix}.
\end{aligned}$$

At this point we have obtained the following equalities:

$$F(2k) = F(k)F(k+1) + F(k-1)F(k) \tag{3a}$$

$$F(2k+1) = F(k+1)^2 + F(k)^2 \tag{3b}$$

Notice that Equation (3b) is the required identity for $F(2k+1)$, but Equation (3a) does not look quite right. The identity we are required to prove did not contain $F(k-1)$. To remove this unwanted term, we make use of the definition of F, namely that

$$F(k+1) = F(k) + F(k-1) \implies F(k-1) = F(k+1) - F(k)$$

Substituting into Equation (3a) gives

$$\begin{aligned}
F(2k) &= F(k)F(k+1) + F(k-1)F(k), \\
&= F(k)F(k+1) + F(k+1)F(k) - F(k)^2, \\
&= 2F(k)F(k+1) - F(k)^2, \\
&= F(k)[2F(k+1) - F(k)].
\end{aligned}$$

as required.

Problem 9. Consider the following recursive function for computing the power function x^p for a non-negative integer p.

```
1: function POWER(x, p)
2:     if p = 0 then return 1
3:     if p = 1 then return x
4:     if p is even then
5:         return POWER(x, p/2) × POWER(x, p/2)
6:     else
7:         return POWER(x, p/2) × POWER(x, p/2) × x
8:     end if
9: end function
```

What is the time complexity of this function (assuming that all multiplications are done in constant time)? How could you improve this function to improve its complexity, and what would that new complexity be?

Solution

The time complexity can be formally obtained by solving the following recurrence relation where $T(p)$ corresponds to the running time of the function for `POWER(x,p)`.

$$T(p) = \begin{cases} 2.T\left(\frac{p}{2}\right) + c, & \text{if } p > 1, \\ b, & \text{if } p = 1, \end{cases}$$

We are not asked to prove the time complexity formally, so we will just make a reasonable argument. The function recurses until $p = 0$. This will take $\log_2(p)$ levels of recursion since we are halving p each time. For each function call, we make two recursive calls, hence the call tree is a binary tree of height $\log_2(p)$. Since a binary tree of height h has $2^h - 1$ nodes, we make $O(2^{\log_2(p)}) = O(p)$ function calls, each of which does a constant amount of work. Hence the time complexity is $O(p)$.

We can improve the time complexity by noticing that the function is actually making the same recursive call twice. Instead of calling POWER($x, p/2$) twice, we should just call it once and then square the answer. With this improvement, the height of the call tree is still $\log_2(p)$, but we only make one function call per level, hence the time complexity will be $O(\log(p))$. The improved implementation might look something like this.

```
1: function POWER(x, p)
2:     if p = 0 then return 1
3:     y = POWER(x, p/2)
4:     if p is even then
5:         return y^2
6:     else
7:         return y^2 × x
8:     end if
9: end function
```

The recurrence relation for the improved function is

$$T(p) = \begin{cases} T\left(\frac{p}{2}\right) + c, & \text{if } p > 1, \\ b, & \text{if } p = 1, \end{cases}$$

and, in our solution to problem 7, we showed that this recurrence relation has the solution $T(p) = b + c\log_2(p)$. Hence, the time complexity is $O(\log(p))$.

Problem 10. Consider the typical Merge sort algorithm. Determine the recurrence for the time complexity of this algorithm, and then solve it to determine the complexity of Merge sort.

Solution

Merge sort divides the list in half down the middle, which takes constant time. It then recursively calls itself on both halves. If the complexity of merge sort is given by a function $T(n)$ where n is the size of the input list, then each of these recursive calls takes $T(n)$ time.

After both calls finish, the two halves still need to be merged. This can be done with a constant number of operations for each element in the two lists. This means merging takes $c\,n$, for some constant c.

The recurrence would therefore be $T(n) = 2T(\frac{n}{2}) + c\,n$. Notice that Merge sort takes constant time when

the input is size 1. Call this constant b.

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + cn, & \text{if } n > 1, \\ b, & \text{if } n = 1. \end{cases}$$

To find a closed form for T, we will use the method of telescoping, as in previous questions.

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + cn, \\ T(n) &= 4T(\frac{n}{4}) + cn + cn \\ T(n) &= 8T(\frac{n}{8}) + cn + cn + cn \end{aligned}$$

Continuing the pattern, we see that the general form for $T(n)$ seems to be

$$T(n) = \begin{cases} 2^k T(\frac{n}{2^k}) + kcn, & \text{if } n > 1, \\ b, & \text{if } n = 1. \end{cases}$$

We want a closed-form solution, so we need to eliminate the term $T(\frac{n}{2^k})$. Setting $k = \log(n)$ gives

$$\begin{aligned} T(n) &= nT(1) + \log(n) \times cn, \\ &= nb + cn\log(n), \end{aligned}$$

To prove this is correct, we check to see if our equation satisfies the recurrence. Manipulating $T(n)$, we see that

$$\begin{aligned} 2T(\frac{n}{2}) + cn &= 2\left[\frac{bn}{2} + \frac{cn\log(\frac{n}{2})}{2}\right] + cn \\ &= bn + cn(\log(n) - \log(2)) + cn \\ &= nb + cn \times \log(n) \\ &= T(n) \end{aligned}$$

as required. We also have $T(1) = b + 0 = b$ as required. Hence this is a valid solution. Notice that is has complexity $O(n\log(n))$, as expected.

Problem 11. Write a Python function that computes $F(n)$ (the n^{th} Fibonacci number) by using the recurrences given in Problem 8(b). What are the time and space complexities of this method of computing Fibonacci numbers? You may assume that all operations on integers take constant time and space.

Solution

We are not asked for a rigorous proof of the complexity, so we provide a reasonable argument. Each time we recurse, n is halved (plus or minus one), hence the depth of the recursion tree will be $\log_2(n)$. Since each function call alone requires constant space, the space complexity of such an implementation is $O(\log(n))$. Note that although the recurrence relation has three recursive terms in the first case, it requires only two recursive calls since the term $F(k)$ is used twice. If the $F(k)$ term is computed once and reused, we recurse twice at each node, and since the number of nodes in a binary tree of height h is $O(2^h)$, the number of nodes in the tree would be

$$O\left(2^{\log_2(n)}\right) = O(n),$$

hence the time complexity is $O(n)$. If we were to naively recurse three times instead of reusing $F(k)$, then

in the worst case, the number of nodes in the tree would be

$$O\left(3^{\log_2(n)}\right) = O\left(n^{\log_2(3)}\right) = O\left(n^{1.585}\right)$$

which is worse than $O(n)$. Also see the solution to Problem 15 where *master theorem* is applied to obtain the time complexity of this approach.

Supplementary Problems

Problem 12. Find a function T that satisfies the following recurrence relation:

$$T(n) = \begin{cases} T(n-1) + a\,n, & \text{if } n > 0, \\ b, & \text{if } n = 0. \end{cases}$$

Write an asymptotic upper bound on the solution in big-O notation.

Solution

To find a closed form for T, we will use the method of telescoping. For $n > 0$ we have

$$T(n) = T(n-1) + an,$$
$$T(n) = (T(n-2) + a(n-1)) + an = T(n-2) + an + a(n-1),$$
$$T(n) = (T(n-3) + a(n-2)) + an + a(n-1) = T(n-3) + an + a(n-1) + a(n-2)$$

Continuing the pattern, we see that the general form for $T(n)$ seems to be

$$T(n) = \begin{cases} T(n-k) + a\sum_{i=n-k+1}^{n} i, & \text{if } n > 0, \text{for all } k \leq n, \\ b, & \text{if } n = 0. \end{cases}$$

We want a closed-form solution, so we need to eliminate the term $T(n-k)$. To do so, we make use of the fact that we have $T(0) = b$. By setting $k = n$, $T(n-k)$ becomes $T(0)$ and hence we obtain

$$\begin{aligned} T(n) &= T(n-n) + a\sum_{i=n-n+1}^{n} i, \\ &= T(0) + a\sum_{i=1}^{n} i, \\ &= b + a\left(\frac{n(n+1)}{2}\right). \end{aligned}$$

Where we have recognised that the sum is $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$. Let's verify that this is correct by substitution. We have

$$\begin{aligned} T(n-1) + an &= b + a\left(\frac{(n-1)n}{2}\right) + an, \\ &= b + a\left(\frac{(n-1)n}{2} + n\right), \\ &= b + a\left(\frac{(n-1)n + 2n}{2}\right), \\ &= b + a\left(\frac{n(n+1)}{2}\right), \\ &= T(n), \end{aligned}$$

as required. We also have $T(0) = b + a \times 0 = b$. Finally, the asymptotic behaviour of the solution is $O(n^2)$.

Problem 13. Find a function T that is a solution of the following recurrence relation

$$T(n) = \begin{cases} 3T\left(\frac{n}{2}\right) + n^2 & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases}$$

Write an asymptotic upper bound on the solution in big-O notation.

Solution

To find a closed form for T, we will use telescoping. For $n > 1$ we have

$$T(n) = 3T\left(\frac{n}{2}\right) + n^2,$$
$$T(n) = 3\left[3T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2\right] + n^2 = 3^2 T\left(\frac{n}{4}\right) + \frac{3n^2}{2^2} + n^2,$$
$$T(n) = 3^2\left[3T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2\right] + \frac{3n}{2^2} + n^2 = 3^3 T\left(\frac{n}{8}\right) + \frac{3^2 n^2}{4^2} + \frac{3n^2}{2^2} + n^2.$$

Continuing the pattern, we see that the general form for $T(n)$seems to be

$$\begin{aligned} T(n) &= 3^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k} \frac{3^i n^2}{(2^i)^2}, \\ &= 3^k T\left(\frac{n}{2^k}\right) + n^2 \sum_{i=0}^{k} \left(\frac{3}{4}\right)^i. \end{aligned}$$

Using the formula for the sum of the first t terms of a geometric series, we obtain

$$\begin{aligned} T(n) &= 3^k T\left(\frac{n}{2^k}\right) + n^2 \left(\frac{\left(\frac{3}{4}\right)^{k+1} - 1}{\frac{-1}{4}}\right), \\ &= 3^k T\left(\frac{n}{2^k}\right) + 4n^2 \left(1 - \left(\frac{3}{4}\right)^{k+1}\right), \\ &= 3^k T\left(\frac{n}{2^k}\right) + 4n^2 - 4n^2 \left(\frac{3}{4}\right)^{k+1}. \end{aligned}$$

We want a closed-form solution, so we need to eliminate the term $T\left(\frac{n}{2^k}\right)$. To do so, we make use of the fact that we have $T(1) = 1$. By setting $k = \log_2(n)$, $T\left(\frac{n}{2^k}\right)$ becomes $T(1)$ and hence we obtain

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

Using log laws from the Week 1 studio sheet, we can write

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

To three decimal places, $\log_2(3) = 1.585$ and $\log_2\left(\frac{3}{4}\right) = -0.415$. Since $4 - 3n^{-0.415}$ is between 1 to 4 for $n \geq 1$, the asymptotic behaviour is

$$\begin{aligned} T(n) &= O(n^{1.585}) + O(n^2), \\ &= O(n^2). \end{aligned}$$

Problem 14. (Advanced) Consider the recurrence relation

$$T(n) = 2T(\sqrt{n}) + \log_2(n).$$

Although rather daunting at first sight, we can solve this recurrence by transforming it onto one that we have seen before!

(a) Make the substitution $m = \log_2(n)$ to obtain a new recurrence relation in terms of m, which should look like $T(2^m) = ...$

(b) Define a new function $S(m) = T(2^m)$, and rewrite your recurrence relation from (a) in terms of $S(m)$

(c) Solve the new recurrence relation for $S(m)$ [Hint: it is one that you have already done on this sheet!]

(d) Use your solution from (c) to write an asymptotic bound in big-O notation for $T(n)$

Solution

Making the substitution $m = \log_2(n)$, since $n = 2^{\log_2(m)}$, we obtain the recurrence relation

$$T(2^m) = 2T(\sqrt{2^m}) + m.$$

Since $\sqrt{n} = n^{1/2}$, we can write this as

$$T(2^m) = 2T(2^{m/2}) + m.$$

Now, we define the suggested function $S(m) = T(2^m)$, and use the fact that $T(2^{m/2}) = S(m/2)$ to obtain

$$S(m) = 2S(m/2) + m.$$

This is just the recurrence relation from Problem 6, which we know has the solution $S(m) = O(m\log(m))$. Therefore, we have

$$T(2^m) = O(m\log(m)),$$

and hence by substituting back $m = \log_2(n)$, we find that $T(n) = O(\log(n)\log(\log(n)))$.

Problem 15. (Advanced) Consider a divide-and-conquer algorithm that splits a problem of size n into $a \geq 1$ sub-problems of size n/b with $b > 1$ and performs $f(n)$ work to split/recombine the results of the subproblems. We can express the running time of such an algorithm by the following general recurrence.

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1. \end{cases}$$

A method, often called the *divide-and-conquer master theorem* can be used to solve many equations of this form, for suitable functions f. The master theorem says

$$T(n) = \begin{cases} \Theta(n^{\log_b(a)}) & \text{if } f(n) = O(n^c) \text{ where } c < \log_b(a), \\ \Theta(n^{\log_b(a)}\log(n)) & \text{if } f(n) = \Theta(n^{\log_b(a)}), \\ \Theta(f(n)) & \text{if } f(n) = \Omega(n^{c_1}) \text{ for } c_1 > \log_b(a) \text{ and } af\left(\frac{n}{b}\right) \leq c_2 f(n) \text{ for } c_2 < 1 \text{ for large } n. \end{cases}$$

In case 3, by large n, we mean for all values of n greater than some threshold n_k. Intuitively, the master theorem is broken into three cases depending on whether the splitting/recombining cost $f(n)$ is smaller, equal to, or bigger than the cost of the recursion.

(a) Verify your solutions to the previous exercises using the master theorem, or explain why the master theorem (as stated above) is not applicable

(b) Use telescoping to show that a solution to the master theorem satisfies the following

$$T(n) = \Theta(n^{\log_b(a)}) + \sum_{i=0}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right).$$

(c) Using Part (b), prove the master theorem. You may assume for simplicity that n is an exact power of b, ie. $n = b^i$ for some integer i, so that the subproblem sizes are always integers. To make it easier, you should prove each of the three cases separately.

Solution

Problems 2, 4, and 5 are not applicable since they are linear recurrence relations, and do not have a term of the form $T\left(\frac{n}{b}\right)$. For Problem 6, we can apply the master theorem with $a = 2$, $b = 2$ and $f(n) = n$. Since $\log_2(2) = 1$, we have that $f(n) = \Theta(n)$, which is case 2 of the master theorem. It then tells us that the

solution is

$$T(n) = \Theta(n^{\log_2(2)} \log(n)) = \Theta(n \log(n)),$$

which agrees with our solution.

Problem 7 can be solved using the master theorem with $a = 1$, $b = 2$ and $f(n) = c$. We have $\log_2(1) = 0$, so $n^{\log_2(1)} = 1$, which means that since $f(n) = \Theta(1)$, case 2 applies again and we get

$$T(n) = \Theta(n^{\log_2(1)} \log(n)) = \Theta(\log(n)),$$

which agrees with our solution.

Problem 8 is not applicable since the recurrence is not of the correct form. In Problem 9, the bad implementation of the power function has a time complexity of the form

$$T(p) = \begin{cases} 2T\left(\frac{p}{2}\right) + c & \text{if } p > 1, \\ \Theta(1) & \text{otherwise.} \end{cases}$$

We can solve this using the master theorem with $a = 2$, $b = 2$ and $f(p) = c$. Since $\log_2(2) = 1$, we see that $f(p)$ is dominated by $p^{\log_2(2)} = p$, hence case 1 applies. It tells us that the solution is

$$T(p) = \Theta(p^{\log_2(2)}) = \Theta(p),$$

which agrees with our solution. The improved version of the power function has the same time complexity as Problem 7, which we already verified has the solution $\Theta(\log(n))$, ie. $\Theta(\log(p))$ for the power function.

For Problem 11, the recurrence is the same as that of Problem 9, hence the solution is also $\Theta(n)$ which agrees with us. For the case where we were to naively recurse three times instead of reusing $F(k)$, the time complexity is of the form

$$T(n) = \begin{cases} 3T\left(\frac{n}{2}\right) + c & \text{if } n > 2, \\ \Theta(1) & \text{otherwise.} \end{cases}$$

We can solve this using the master theorem with $a = 3$, $b = 2$ and $f(n) = c$. Since $\log_2(3) = 1.585$, we see that $f(n)$ is dominated by $n^{\log_2(3)} = n^{1.585}$. Hence, case 1 applies and the solution is

$$T(n) = \Theta(n^{\log_2(3)}) = \Theta(n^{1.585})$$

which agrees with our solution.

Problem 12 is not applicable since the recurrence is not of the right form. We can solve Problem 13 with $a = 3$, $b = 2$, and $f(n) = n^2$. We have $\log_2(3) \approx 1.585$, hence $f(n) = n^2$ dominates $n^{\log_2(3)}$, and since $3\left(\frac{n}{2}\right)^2 = \frac{3}{4}n^2$, case 3 of the master theorem applies with the constant $c_2 = \frac{3}{4}$. Therefore the solution is

$$T(n) = \Theta(n^2),$$

which agrees with us. Finally, Problem 14 is not applicable because the $T(\sqrt{n})$ term does not fit the master theorem.

Solution

Let's prove the master theorem by solving the recurrence using telescoping. We have

$$\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + f(n), \\
&= a\left(aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right) + f(n), \\
&= a\left(a\left(aT\left(\frac{n}{b^3}\right) + f\left(\frac{n}{b^2}\right)\right) + f\left(\frac{n}{b}\right)\right) + f(n), \\
&= \ldots \\
&= a^k T\left(\frac{n}{b^k}\right) + \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right).
\end{aligned}$$

To utilise the base case, we use $k = \log_b(n)$ so that $T\left(\frac{n}{b^k}\right)$ becomes $T(1)$ and we obtain

$$\begin{aligned}
T(n) &= a^{\log_b(n)}\Theta(1) + \sum_{i=0}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right), \\
&= \Theta(n^{\log_b(a)}) + \sum_{i=0}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right).
\end{aligned}$$

If $f(n) = \Theta(n^c)$, we can write the summation as

$$\begin{aligned}
\sum_{i=0}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right) &= \sum_{i=0}^{\log_b(n)-1} a^i \Theta\left(\left(\frac{n}{b^i}\right)^c\right), \\
&= \Theta\left(n^c \sum_{i=0}^{\log_b(n)-1} \left(\frac{a}{b^c}\right)^i\right)
\end{aligned}$$

Now we consider the three different cases given by the master theorem.

Case 1: $f(n) = O(n^c)$ for some $c < \log_b(a)$

Since $c < \log_b(a)$, we have $\frac{a}{b^c} > 1$, so the summation becomes a geometric series. Using the formula for a geometric series and ignoring constant factors, we have

$$\begin{aligned}
\sum_{i=0}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right) &= O\left(n^c \sum_{i=0}^{\log_b(n)-1} \left(\frac{a}{b^c}\right)^i\right), \\
&= O\left(n^c \frac{\left(\frac{a}{b^c}\right)^{\log_b(n)} - 1}{a/b^c - 1}\right), \\
&= O\left(n^c \left(\frac{a^{\log_b(n)}}{b^{c\log_b(n)}} - 1\right)\right), \\
&= O\left(n^c \left(\frac{n^{\log_b(a)}}{n^c} - 1\right)\right), \\
&= O\left(n^{\log_b(a)} - n^c\right).
\end{aligned}$$

Therefore, the behaviour of the recurrence relation is

$$T(n) = \Theta(n^{\log_b(a)}) + O(n^{\log_b(a)} - n^c),$$

and since $c < \log_b(a)$, this is

$$T(n) = \Theta(n^{\log_b(a)}).$$

Case 2: $f(n) = \Theta(n^{\log_b(a)})$

In this case, we have $\frac{a}{b^c} = 1$ since $c = \log_b(a)$, hence every term of the summation is the same, so

$$\begin{aligned}\sum_{i=0}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right) &= \Theta\left(n^{\log_b(a)} \sum_{i=0}^{\log_b(n)-1} 1\right), \\ &= \Theta(n^{\log_b(a)} \log_b(n))\end{aligned}$$

Hence the behaviour of the recurrence relation is

$$T(n) = \Theta(n^{\log_b(a)}) + \Theta(n^{\log_b(a)} \log_b(n)) = \Theta(n^{\log_b(a)} \log(n)).$$

Case 3: $f(n) = \Omega(n^{c_1})$ for $c_1 > \log_b(a)$ and $af\left(\frac{n}{b}\right) \leq c_2 f(n)$ for $c_2 < 1$ for sufficiently large n

We have assumed that $af\left(\frac{n}{b}\right) \leq c_2 f(n)$ for some $c_2 < 1$. Let's apply this fact iteratively to obtain

$$a^i f\left(\frac{n}{b^i}\right) \leq c_2^i f(n).$$

If you are not convinced, try proving this fact by induction on i. We only assume that this relation holds for large n, so suppose that this holds for all $n \geq n_k$ for some constant n_k. We can write the sum as

$$\sum_{i=0}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right) \leq \sum_{i=0}^{n_k} a_i f\left(\frac{n}{b_i}\right) + \sum_{i=n_k+1}^{\log_b(n)-1} c_2^i f(n).$$

Since n_k is a constant, the first summation is a constant, hence we can write

$$\sum_{i=0}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right) = O(1) + O\left(\sum_{i=n_k+1}^{\log_b(n)-1} c_2^i f(n)\right).$$

Since $f(n)$ is nonnegative, we can extend the sum to 0 to ∞ to obtain an upper bound, and use the infinite geometric series formula to obtain

$$\begin{aligned}\sum_{i=0}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right) &= O(1) + O\left(\sum_{i=0}^{\infty} c_2^i f(n)\right), \\ &= O(1) + O\left(f(n) \sum_{i=0}^{\infty} c_2^i\right), \\ &= O(1) + O\left(f(n)\left(\frac{1}{1-c_2}\right)\right), \\ &= O(f(n)).\end{aligned}$$

Therefore the summation is upper bounded by $O(f(n))$. Since the $i = 0$ term of the sum is just $f(n)$ itself, the summation is also lower bounded by $\Omega(f(n))$, which means that the $f(n)$ bound is tight, i.e.

$$\sum_{i=0}^{\log_b(n)-1} a^i f\left(\frac{n}{b^i}\right) = \Theta(f(n)).$$

Therefore the asymptotic behaviour of the recurrence is

$$T(n) = \Theta(n^{\log_b(a)}) + \Theta(f(n)),$$

but since we assume that $f(n) = \Omega(n^{c_1})$ for some $c_1 > \log_b(a)$, the second term dominates and hence we just have

$$T(n) = \Theta(f(n)).$$