Week 2 Applied Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the applied 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 these 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, pseudocode may be provided where it illustrates a particular useful concept.

Problems

Problem 1. 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

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

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

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

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

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

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

$$\begin{aligned}2T(n-1)+a&=2(2^{n-1}b+(2^{n-1}-1)a)+a,\\ &=2\times2^{n-1}b+2\times(2^{n-1}-1)a+a,\\ &=2^nb+2^na-2a+a,\\ &=2^nb+(2^n-1)a,\\ &=T(n)\end{aligned}$$

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

Problem 2. 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

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

(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

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

as required.

Problem 3. 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 $T(\frac{n}{2})$ 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.

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

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

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

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

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

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

$$\begin{aligned} T(n) &= n\,T(1) + \log(n) \times c\,n, \\ &= n\,b + c\,n\log(n), \end{aligned}$$

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

$$\begin{aligned} 2T\left(\frac{n}{2}\right) + c\,n &= 2\left[\frac{b\,n}{2} + \frac{c\,n\log(\frac{n}{2})}{2}\right] + c\,n \\ &= b\,n + c\,n(\log(n) - \log(2)) + c\,n \\ &= n\,b + c\,n \times \log(n) \\ &= T(n) \end{aligned}$$

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

Problem 4. 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 non-negative integers k and $n = 2^k$.

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 non-negative integers k and $n = 2^k$.

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 non-negative integer k and $m = 2^k$. 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 5. 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
```

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)`.

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

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, and considering the root as level 0, has $2^{h+1}-1$ nodes, we make $\Theta(2^{\log_2(p)}) = \Theta(p)$ function calls, each of which does a constant amount of work. Hence the time complexity is $\Theta(p)$.

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 $\Theta(\log(p))$. 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
```

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 4, we showed that this recurrence relation has the solution $T(p) = b + c\log_2(p)$. Hence, the time complexity is $\Theta(\log(p))$.

Problem 6. Recommender systems are widely employed nowadays to suggest new books, movies, restaurants, etc that a user is likely to enjoy based on his past ratings. One commonly used technique is collaborative filtering, in which the recommender system tries to match your preferences with those of other users, and suggests items that got high ratings from users with similar tastes. A distance measure that can be used to analyse how similar the rankings of different users are is counting the number of inversions. The counting inversions problem is the following:

- **Input:** An array V of n distinct integers.
- **Output:** The number of inversions of V, i.e., the number of pairs of indices (i, j) such that $i < j$ and $V[i] > V[j]$.

The exhaustive search algorithm for solving this problem has time complexity $\Theta(n^2)$. Describe an algorithm with time complexity $O(n\log n)$ for solving this problem.

[Hint: Adapt a divide-and-conquer algorithm that you already studied.]

Solution

Note that potentially there are $\Theta(n^2)$ inversions, so an algorithm for solving the problem with time complexity $O(n \log n)$ cannot look individually at each possible inversion.

The basic idea is to adapt the Merge Sort algorithm to solve this problem. We split the array in the middle and invoke recursive calls on the first and the second halves. Each recursive call will count the number of inversions in that subarray *and also sort the elements of that subarray*. Getting the elements of the subarrays sorted is key to allowing us to count, during the merging procedure, in time $\Theta(n)$ the number of "split inversions", i.e., inversions in which i belongs to the left subarray and j to the right subarray.

When we are performing the merging procedure, at each step the smallest remaining element is selected (and it will be either the first remaining element of the left subarray or of the right subarray, as the subarrays are sorted). If that smallest element comes from the left subarray, then there are no split inversions to be counted (as the index of this element is smaller than the indices of all elements in the right subarray). On the other hand, if that smallest element comes from the right subarray, then the number of split inversions should be increased by the amount of elements still to be processed in the left subarray (as all those elements have smaller indices than the selected one).

The pseudocode of the algorithm is presented below:

```
1:  function SORT-AND-COUNTINV(array[lo..hi])
2:      if lo = hi then
3:          Return (array[lo], 0)
4:      else
5:          mid = ⌊(lo + hi)/2⌋
6:          (array[lo..mid], Inv_L) = SORT-AND-COUNTINV(array[lo..mid])
7:          (array[mid+1..hi], Inv_H) = SORT-AND-COUNTINV(array[mid+1..hi])
8:          (array[lo..hi], Inv_S) = MERGE-AND-COUNTSPLITINV(array[lo..mid], array[mid+1..hi])
9:          Inv = Inv_L + Inv_H + Inv_S
10:         Return (array[lo..hi], Inv)
```

```
1:  function MERGE-AND-COUNTSPLITINV(A[i..n_1], B[j..n_2])
2:      result = empty array
3:      splitInversions = 0
4:      while i ≤ n_1 or j ≤ n_2 do
5:          if j > n_2 or (i ≤ n_1 and A[i] ≤ B[j]) then
6:              result.append(A[i])
7:              i += 1
8:          else
9:              result.append(B[j])
10:             j += 1
11:             splitInversions = splitInversions + n_1 - i + 1
12:     Return (result, splitInversions)
```

Problem 7. You are given as input an n-by-n grid of distinct numbers (represented as a matrix), and want to find a local maximum. For each number, its neighbours are the numbers immediately above it, below it, to its left, and to its right. Note that while most numbers have 4 neighbours, the ones on the edge of the matrix only have 3 neighbours, and the ones in the corners only have 2 neighbours. We will consider a number to be a local maximum if all its neighbours are smaller than it.

Given the matrix `M` your algorithm for finding a local maximum should have a worst-case time complexity of $O(n)$ and should output a single pair of coordinates i and j such that `M`$[i][j]$ is a local maximum. If there are multiple local maxima, your algorithm should output the coordinates of exactly one local maximum, and this can be any

of the existing local maxima.

Two examples of matrices with their local maxima in red are given below.

$$\begin{bmatrix} 1 & 2 & 27 & 28 & 29 & 30 & 49 \\ 3 & 4 & 25 & 26 & 31 & 32 & 48 \\ 5 & 6 & 23 & 24 & 33 & 34 & 47 \\ 7 & 8 & 21 & 22 & 35 & 36 & 46 \\ 9 & 10 & 19 & 20 & 37 & 38 & 45 \\ 11 & 12 & 17 & 18 & 39 & 40 & 44 \\ 13 & 14 & 15 & 16 & 41 & 42 & 43 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 & 6 & 10 & 15 & 21 & 28 & 164 & 201 & 203 & 206 & 210 & 215 & 221 & 228 \\ 2 & 5 & 9 & 14 & 20 & 27 & 34 & 163 & 202 & 205 & 209 & 214 & 220 & 227 & 234 \\ 4 & 8 & 13 & 19 & 26 & 33 & 39 & 162 & 204 & 208 & 213 & 219 & 226 & 233 & 239 \\ 7 & 12 & 18 & 25 & 32 & 38 & 43 & 161 & 207 & 212 & 218 & 225 & 232 & 238 & 290 \\ 11 & 17 & 24 & 31 & 37 & 42 & 46 & 160 & 211 & 217 & 224 & 231 & 909 & 908 & 907 \\ 16 & 23 & 30 & 36 & 41 & 45 & 48 & 159 & 216 & 223 & 230 & 260 & 906 & 904 & 902 \\ 22 & 29 & 35 & 40 & 44 & 47 & 49 & 158 & 222 & 229 & 235 & 340 & 305 & 903 & 901 \\ 51 & 52 & 53 & 54 & 55 & 56 & 57 & 157 & 506 & 505 & 504 & 503 & 502 & 501 & 650 \\ 101 & 102 & 127 & 128 & 129 & 130 & 149 & 156 & 601 & 302 & 327 & 328 & 629 & 630 & 649 \\ 103 & 104 & 125 & 126 & 131 & 132 & 148 & 155 & 603 & 604 & 625 & 626 & 631 & 632 & 648 \\ 105 & 106 & 123 & 124 & 133 & 134 & 147 & 154 & 605 & 606 & 623 & 624 & 633 & 634 & 647 \\ 107 & 108 & 121 & 122 & 135 & 136 & 146 & 153 & 607 & 608 & 621 & 622 & 635 & 636 & 646 \\ 109 & 110 & 119 & 120 & 137 & 138 & 145 & 152 & 609 & 610 & 619 & 620 & 637 & 638 & 645 \\ 111 & 112 & 117 & 118 & 139 & 140 & 144 & 151 & 611 & 612 & 617 & 618 & 639 & 640 & 644 \\ 113 & 114 & 115 & 116 & 141 & 142 & 143 & 150 & 613 & 614 & 615 & 616 & 641 & 642 & 643 \end{bmatrix}$$

Solution

This is a question from a previous assignment of the unit.

First notice that just starting from some element, and moving step-by-step to the biggest neighbour of the current element until a local maximum is found (gradient-style approach) would be $\Theta(n^2)$ in the worst-case (for example, if we start from the top left element in the first matrix, as we would need to completely traverse half of the matrix columns).

To improve on this we can use a divide-and-conquer approach. But notice that just dividing the matrix in the middle column, somehow making a decision on which half of the matrix to proceed with based on the elements in that column (and their neighbours), and then calling this same procedure recursively would not be enough. The reason for that is that in the worst-case there would be $\Theta(\log n)$ recursive calls and in each call we would need time $\Theta(n)$ to decide with which half we should proceed; thus resulting in an overall time complexity of $\Theta(n \log n)$. The same logic holds for doing it based on the middle row. So the size of the matrices we give as input to each recursive call needs to decrease faster.

The initial idea to solve this problem in time $O(n)$ is the following:

1. Consider the cross formed by the middle column and the middle row, and in time $O(n)$ find the maximum element in that cross, lets denote it by x.
2. If x is a local maximum, just finish the execution by returning the coordinates of x.
3. Otherwise, pick a neighbour y of x such that $y > x$. Note that by the definition of x, y needs to be outside the cross and bigger than all elements in the cross.
4. The cross divides the original matrix into 4 sub-matrices, proceed recursively with the sub-matrix (denoted by A) that contains y.

Note that, since y is bigger than all elements in the cross, then there is at least one local maximum of M which is actually in the sub-matrix A (if we were to start at y and keep moving step-by-step to bigger neighbouring elements until finding a local maximum, we could never go back to a element in the cross).

There is one small technical issue that we still need to solve: when we draw the cross on a sub-matrix B of the original instance, it might be the case that the maximum element z in that cross is bigger than all its neighbours in B, but not in the original matrix M. Look, for instance, at what would happen in the second recursion level of the second matrix given as example above: 340 would be the maximum in the cross, and also bigger than its neighbours in the sub-matrix of the recursive call (260, 235 and 260), but not bigger than 503 (which was part of the cross in the previous recursive call). There are a few ways to solve this, one of them is using a "window frame" instead of a cross (i.e., picking the first, middle and last columns as well as the first, middle and last rows, and computing the maximum among all elements on the 3 columns and 3 rows to check if we found a local maximum/or how we should proceed with the recursion). Reflect about why this modification solves the previous issue.

When we are trying to solve the problem on a n-by-n matrix, we spend time $\Theta(n)$ to check how to proceed, and then do a recursive call on a subproblem of size roughly $n/2$-by-$n/2$. Solving the recurrence relation $T(n) = T(n/2) + \Theta(n)$, we get that the worst-case time complexity is $\Theta(n)$.

Supplementary Problems

Problem 8. Write a Python function that computes $F(n)$ (the n^{th} Fibonacci number) by using the recurrences given in Problem 2(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 12 where *master theorem* is applied to obtain the time complexity of this approach.

Problem 9. 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

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

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

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

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

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

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

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

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 11. 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

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

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

$$\begin{aligned} T(n) &= 3^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} \frac{3^i n^2}{(2^i)^2}, \\ &= 3^k T\left(\frac{n}{2^k}\right) + n^2 \sum_{i=0}^{k-1} \left(\frac{3}{4}\right)^i. \end{aligned}$$

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

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

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

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

Using log laws from the Week 1 revision problems' sheet, we can write

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

To three decimal places, $\log_2(3) = 1.585$ and $\log_2\left(\frac{3}{4}\right) = -0.415$. Since $4 - 4n^{-0.415}$ is between 1 to 4 for $n > 2$, the asymptotic behaviour is

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

Problem 12. 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.

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

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

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

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

Problem 1 is not applicable since it is a linear recurrence relation, and does not have a term of the form $T\left(\frac{n}{b}\right)$. For Problem 9, 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 4 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 2 is not applicable since the recurrence is not of the correct form. In Problem 5, 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 4, which we already verified has the solution $\Theta(\log(n))$, ie. $\Theta(\log(p))$ for the power function.

For Problem 8, the recurrence is the same as that of Problem 5, 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 10 is not applicable since the recurrence is not of the right form. We can solve Problem 11 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.

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

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

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 $af\left(\frac{n}{b}\right)\le c_2 f(n)$ for $c_2<1$ for sufficiently large n.

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

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

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 $n\ge n_k$ for some constant n_k. We can write the sum as

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

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

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

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

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

Problem 13. Consider the problem of multiplying two square matrices $\mathbf{X}$ and $\mathbf{Y}$ of order n (i.e., $\mathbf{X}$ and $\mathbf{Y}$ are n-by-n matrices) to obtain the matrix $\mathbf{Z} = \mathbf{X} \cdot \mathbf{Y}$. Assume that the matrices' elements are numbers of fixed size and that the basic operations of addition, subtraction and multiplication of those single numbers can be executed in a single step. Let's assume for simplicity that n is a power of two (but the techniques and analysis of this problem can be generalised for other n).

(a) What is the complexity of an algorithm that simply computes each element $\mathbf{Z}[i][j]$ of the matrix $\mathbf{Z}$ using the formula you learned in high-school, $\mathbf{Z}[i][j] = \sum_{k=1}^{n} \mathbf{X}[i][k] \cdot \mathbf{Y}[k][j]$?

(b) Consider the following alternative approach: Let's split the matrices into four parts by dividing them in the middle both vertically and horizontally, i.e., $\mathbf{X} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$ and $\mathbf{Y} = \begin{bmatrix} \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} \end{bmatrix}$, where $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{G}$ and $\mathbf{H}$ are square matrices of order $n/2$. If follows straightforwardly from the definition of matrix multiplication (as you can check) that

$$\mathbf{Z} = \begin{bmatrix} \mathbf{A} \cdot \mathbf{E} + \mathbf{B} \cdot \mathbf{G} & \mathbf{A} \cdot \mathbf{F} + \mathbf{B} \cdot \mathbf{H} \\ \mathbf{C} \cdot \mathbf{E} + \mathbf{D} \cdot \mathbf{G} & \mathbf{C} \cdot \mathbf{F} + \mathbf{D} \cdot \mathbf{H} \end{bmatrix}.$$

What is the complexity of computing $\mathbf{Z}$ using those eight recursive calls?

(c) (Strassen's matrix multiplication algorithm) The following approach for matrix multiplication was developed by German mathematician Volker Strassen in 1969: The matrices $\mathbf{X}$ and $\mathbf{Y}$ are still divided in four parts as above, but now only seven recursive calls are done:

- $\mathbf{M}_1 = \mathbf{A} \cdot (\mathbf{F} - \mathbf{H})$
- $\mathbf{M}_2 = (\mathbf{A} + \mathbf{B}) \cdot \mathbf{H}$
- $\mathbf{M}_3 = (\mathbf{C} + \mathbf{D}) \cdot \mathbf{E}$
- $\mathbf{M}_4 = \mathbf{D} \cdot (\mathbf{G} - \mathbf{E})$

- $\mathbf{M}_5 = (\mathbf{A}+\mathbf{D})\cdot(\mathbf{E}+\mathbf{H})$
- $\mathbf{M}_6 = (\mathbf{B}-\mathbf{D})\cdot(\mathbf{G}+\mathbf{H})$
- $\mathbf{M}_7 = (\mathbf{A}-\mathbf{C})\cdot(\mathbf{E}+\mathbf{F})$

and the matrix $\mathbf{Z}$ is then obtained as

$$\mathbf{Z} = \begin{bmatrix} \mathbf{M}_5+\mathbf{M}_4-\mathbf{M}_2+\mathbf{M}_6 & \mathbf{M}_1+\mathbf{M}_2 \\ \mathbf{M}_3+\mathbf{M}_4 & \mathbf{M}_1+\mathbf{M}_5-\mathbf{M}_3-\mathbf{M}_7 \end{bmatrix}.$$

Show that this approach would indeed produce the right result.

(d) What is the complexity of Strassen's matrix multiplication algorithm?

Solution

(a) Using the standard method for computing the matrix multiplication, your algorithm simply loops through the n^2 elements of $\mathbf{Z}$, and computes its value using n multiplications and $n-1$ additions. Therefore the complexity of the algorithm is $\Theta(n^3)$.

(b) In this case, there are eight recursive calls to subproblems of size $n/2$ (multiplications of $n/2$-by-$n/2$ matrices), plus n^2 additions of elements (one per element of $\mathbf{Z}$). Therefore the recurrence that we get is

$$T(n) = \begin{cases} 8T\left(\frac{n}{2}\right)+n^2 & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1. \end{cases}$$

Solving this recurrence using the master method (or other techniques), we get that the complexity of this approach is still $\Theta(n^3)$.

(c) Let's show that the algorithm indeed produces the right output. To begin we have that

$$\begin{aligned} \mathbf{M}_5+\mathbf{M}_4-\mathbf{M}_2+\mathbf{M}_6 &= (\mathbf{A}+\mathbf{D})\cdot(\mathbf{E}+\mathbf{H})+\mathbf{D}\cdot(\mathbf{G}-\mathbf{E})-(\mathbf{A}+\mathbf{B})\cdot\mathbf{H}+(\mathbf{B}-\mathbf{D})\cdot(\mathbf{G}+\mathbf{H}) \\ &= \mathbf{A}\cdot\mathbf{E}+\mathbf{A}\cdot\mathbf{H}+\mathbf{D}\cdot\mathbf{E}+\mathbf{D}\cdot\mathbf{H}+\mathbf{D}\cdot\mathbf{G}-\mathbf{D}\cdot\mathbf{E}-\mathbf{A}\cdot\mathbf{H}-\mathbf{B}\cdot\mathbf{H}+\mathbf{B}\cdot\mathbf{G}+\mathbf{B}\cdot\mathbf{H}-\mathbf{D}\cdot\mathbf{G}-\mathbf{D}\cdot\mathbf{H} \\ &= \mathbf{A}\cdot\mathbf{E}+\mathbf{B}\cdot\mathbf{G} \end{aligned}$$

as desired. In the same way,

$$\begin{aligned} \mathbf{M}_1+\mathbf{M}_2 &= \mathbf{A}\cdot(\mathbf{F}-\mathbf{H})+(\mathbf{A}+\mathbf{B})\cdot\mathbf{H} \\ &= \mathbf{A}\cdot\mathbf{F}-\mathbf{A}\cdot\mathbf{H}+\mathbf{A}\cdot\mathbf{H}+\mathbf{B}\cdot\mathbf{H} \\ &= \mathbf{A}\cdot\mathbf{F}+\mathbf{B}\cdot\mathbf{H} \end{aligned}$$

as desired. Similarly,

$$\begin{aligned} \mathbf{M}_3+\mathbf{M}_4 &= (\mathbf{C}+\mathbf{D})\cdot\mathbf{E}+\mathbf{D}\cdot(\mathbf{G}-\mathbf{E}) \\ &= \mathbf{C}\cdot\mathbf{E}+\mathbf{D}\cdot\mathbf{E}+\mathbf{D}\cdot\mathbf{G}-\mathbf{D}\cdot\mathbf{E} \\ &= \mathbf{C}\cdot\mathbf{E}+\mathbf{D}\cdot\mathbf{G} \end{aligned}$$

and so this part is also correct. Finally,

$$\begin{aligned} \mathbf{M}_1+\mathbf{M}_5-\mathbf{M}_3-\mathbf{M}_7 &= \mathbf{A}\cdot(\mathbf{F}-\mathbf{H})+(\mathbf{A}+\mathbf{D})\cdot(\mathbf{E}+\mathbf{H})-(\mathbf{C}+\mathbf{D})\cdot\mathbf{E}-(\mathbf{A}-\mathbf{C})\cdot(\mathbf{E}+\mathbf{F}) \\ &= \mathbf{A}\cdot\mathbf{F}-\mathbf{A}\cdot\mathbf{H}+\mathbf{A}\cdot\mathbf{E}+\mathbf{A}\cdot\mathbf{H}+\mathbf{D}\cdot\mathbf{E}+\mathbf{D}\cdot\mathbf{H}-\mathbf{C}\cdot\mathbf{E}-\mathbf{D}\cdot\mathbf{E}-\mathbf{A}\cdot\mathbf{E}-\mathbf{A}\cdot\mathbf{F}+\mathbf{C}\cdot\mathbf{E}+\mathbf{C}\cdot\mathbf{F} \\ &= \mathbf{D}\cdot\mathbf{H}+\mathbf{C}\cdot\mathbf{F} \end{aligned}$$

and therefore we can conclude that Strassen's approach obtains the correct result.

(d) In this case, there are seven recursive calls to subproblems of size $n/2$, plus $O(n^2)$ additions/subtractions of elements (at most three additions/subtractions per element of $\mathbf{Z}$). Therefore the recurrence that

we get is

$$T(n) = \begin{cases} 7T\left(\frac{n}{2}\right) + O(n^2) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1. \end{cases}$$

Solving this recurrence using the master method (or other techniques), we get that the complexity of this approach is $\Theta(n^{\log_2 7})$. Note that $\log_2 7 < 2.81$.

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 $T(2^m) = \ldots$

(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 9, which we know has the solution $S(m) = \Theta(m\log(m))$. Therefore, we have

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

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