Week 2 Applied Sheet

Objectives: The applied sessions, in general, give practice in problem solving, in analysis of algorithms and data structures, and in mathematics and logic useful in the above.

Instructions to the class: You should actively participate in the class.

Instructions to Tutors: The purpose of the applied class is not to solve the practical exercises! The purpose is to check answers, and to discuss particular sticking points, not to simply make answers available.

Supplementary problems: The supplementary problems provide additional practice for you to complete after your applied class, or as pre-exam revision. Problems that are marked as **(Advanced)** difficulty are beyond the difficulty that you would be expected to complete in the exam, but are nonetheless useful practice problems as they will teach you skills and concepts that you can apply to other problems.

Problems

Problem 1. Find a closed form for the following recurrence relation:

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

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)]

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.

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 $n = 2^k$.

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?

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.]

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.

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

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

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.

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.

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.

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.

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.

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

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

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

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.

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 **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?

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 $T(n)$.