Week 2 Studio Sheet

(To be completed during the Week 2 studio class)

Objectives: The tutorials, 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: Aim to attempt these questions before the tutorial! It will probably not be possible to cover all questions unless the class has prepared them in advance. There are marks allocated towards active participation during the class. You **must** attempt the problems under **Assessed Preparation** section **before** your tutorial class and give your worked out solutions to your tutor at the start of the class – this is a hurdle and failing to attempt these problems before your tutorial will result in 0 mark for that class even if you actively participate in the class.

Instructions to Tutors:

1. The purpose of the tutorials is not to solve the practical exercises!
2. 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 tutorial 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.

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
```

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

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

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.

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

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

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

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

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.

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 $k \geq 0$.

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)]

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?

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.

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.

Supplementary Problems

Problem 12. 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 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.

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 $T(n)$

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.