

# COMPSCI 320 S2 2018

## Assignment 2 (Written)

Due date: August 11, 2018 (11.59pm)

Version 1.1 of 2018-08-02, correct parameter for  $f_1()$  and  $f_2()$  in Q1a and Q1b

Solve the following questions, then submit your answers to Canvas in a file named <YourUPI>\_A2.pdf.

Scanned handwritten assignments are fully acceptable, provided that they are neat.

If you plan to continue to postgraduate work in computer science, please consider learning how to use L<sup>A</sup>T<sub>E</sub>X or AMSTeX. To help you get started with some “learning by example”: you’ll find some of my L<sup>A</sup>T<sub>E</sub>X sourcefiles in the Files area of the Canvas-site for CompSci 320.

1. Let

$$f_1(n) = 3n + 10 \lg n$$

$$f_2(n) = n$$

$$f_3(n) = 2^n + n^2$$

$$f_4(n) = 1000n^2$$

Note that  $\lg n = \log_2 n$ .

- (a) True or false:  $f_1(n) = O(f_2(n))$ . Explain your reasoning. Your reasoning need not be mathematically formal to gain full marks, but it must persuade your marker that you’re not randomly “guessing” at the correct answer. **T** (1 mark)
- (b) True or false:  $f_2(n) = O(f_1(n))$ . Explain your reasoning. **T** (1 mark)
- (c) True or false:  $f_2(n) = O(f_3(n))$ . Explain your reasoning. **T** (1 mark)
- (d) True or false:  $f_3(n) = O(f_4(n))$ . Explain your reasoning. **Not Sure** (1 mark)
- (e) True or false:  $f_4(n) = O(f_3(n))$ . Explain your reasoning. **T** (1 mark)

2. A program `p.py` accepts an  $n$ -character string as input, and produces an  $n$ -character string as output.

- (a) If `p.py` runs in 1 second when  $n = 10$ , 2 seconds when  $n = 20$ , 4 seconds when  $n = 40$ , and 8 seconds when  $n = 80$ , would you conclude that `p.py` is implementing an  $O(n)$ -time algorithm? Explain your reasoning briefly. (2 marks)
- (b) If `p.py` runs in 1 second when  $n = 10$ , 20 seconds when  $n = 20$ , and doesn’t produce an output for  $n = 40$  or for  $n = 80$  even if you wait an hour for it to complete, would you conclude that the runtime of `p.py` is *not*  $O(n)$ ? Explain your reasoning briefly. (2 marks)
- (c) If your inspection of the code for `p.py` reveals that it exhaustively searches through all of the length-3 substrings of its input, would you conclude that its runtime is *not*  $O(n)$ ? Explain your reasoning briefly. (2 marks)

3. “Prior to 10 July 1967, ... [the] main coins in [New Zealand] usage were the halfpenny (1/2d), penny (1d), threepence (3d), sixpence (6d), shilling (1s), florin (2s), and halfcrown (2s 6d).”<sup>1</sup>  
“Under this system, there were 12 pence in a shilling and 20 shillings, or 240 pence, in a pound.”<sup>2</sup>

<sup>1</sup>Coins of the New Zealand Dollar, Wikipedia, 2018-07-01T02:22Z.

<sup>2</sup>£sd, Wikipedia, 2018-07-27T11:10Z.

(a) Using the cashier's algorithm as explained in the lecture slides, make change for 61d. Show your work. Hint: express all coinage values in half-pennies. (1 mark) ✓

(b) If the input value  $x$  to the cashier's algorithm is a 64-bit unsigned integer value (in half-pennies), then the output multiset  $S$  could have an enormous number of coins. Develop a representation for  $S$  as a string that a human could quickly read and understand – even if its value  $x$  is very large. Explain your representation briefly, with a couple of examples. (2 marks)

(c) Put an upper bound on the maximum length of the string representation for  $S$  you developed in your previous question, for any  $x < 2^{64}$ , where  $S$  describes the change as calculated by the cashier's algorithm. Your upper bound need not be exact but it should be fairly tight, i.e. at most a few characters larger than the longest possible encoded string for  $S$ , and at least as long as the largest possible encoded string for  $S$ . (1 mark)

(d) Can you find an input  $x < 200$  half-pennies, for which the cashier's algorithm does *not* find a minimum number of coins for this amount? If not, then list three test-cases you considered during your search, indicating the input and the corresponding output. If so, your answer should show  $x$ , the multiset  $S$  computed by the cashier's algorithm, and a multiset  $S$  with value  $x$  but with fewer coins. (2 marks)

(e) Note that the runtime for the cashier's algorithm, as described in the lecture slides, is at least linear in the number of coins in its answer  $S$  because each iteration of the while loop will take  $\Omega(1)$  time. Devise a more efficient algorithm for the case that  $x$  is very large, then write pseudocode for your more-efficient algorithm in the style of the lecture slides. (1 mark)

4. Consider the (non-weighted) Interval Scheduling Problem that was discussed in class.

(a) Illustrate the greedy algorithm on the input  $((0,3), (1,4), (4,6), (4,7), (7,8), (7,9), (8,11), (10,11))$ . (2 marks) ✓

(b) In a weighted version of the problem, you're required to find a schedule of maximal weight. If the weight of a job is in the third position of its tuple, can you find a weight-8 solution for input  $((0,3,1), (1,4,2), (4,6,1), (4,7,2), (7,8,2), (7,9,1), (8,11,2), (10,11,1))$ ? Explain briefly. (1 mark) ✓

(c) Consider the preceding problem and generalise its input, at least slightly: describe a restricted form of the weighted interval scheduling problem for which a greedy algorithm will find the optimal solution in  $O(n \lg n)$  time. To get full marks, you must also explain how your greedy algorithm makes its optimal choice. (1 mark)

5. Consider the following Python 3 code:

check →

```
def is_prime(k):
    """Return True if k is prime or <= 1, otherwise return False """
    return all(k % d != 0 for d in range(2,k))

def prime_index(k):
    """Return the index of the largest prime p <= k in [0,1,2,3,5,7,11,...] """
    if k < 4:
        return max(k,0)
    else:
        return is_prime(k) + prime_index(k-1)


def prime_indices(n):
```

```

"""Prints a table of prime indices from 1..n"""
for i in range(0,n+1):
    print(i,prime_index(i))

```

Note to students who are not fluent in Python: the `is_prime()` function iterates over all `d` from 2 to `k+1`, stopping (with return value `True`) as soon as it finds a divisor, otherwise returning `False`. The `prime_index()` function uses the fact that, in Python 3, booleans are an enumerated subclass of integers, with `True == 1` and `False == 0`.

(a) What is the asymptotic runtime of `is_prime(n)`? (1 mark) 

(b) Use your answer to the previous problem to compute an upper bound on the asymptotic runtime of `prime_index(n)`. Explain your reasoning. (1 mark)

(c) Given your prior asymptotic analysis, which is the greater inefficiency in an execution of `prime_indices(n)` for very large `n`,

- its re-evaluations of `is_prime(k)` at points `k` which had previously been evaluated, or
- the unnecessary tests of all  $d > \sqrt{k}$  by `is_prime(k)`? Note that these tests are unnecessary because no integer  $k$  has a factor  $d > \sqrt{k}$ .

Explain your answer briefly. To receive full marks, you must indicate how the inefficiency could be avoided by making a small change to one of these functions. If you explain this small change clearly in English, you need not express it in either pseudocode or in Python.

(2 marks)

6. If your solution to Problem 3 of Assignment 1 was not accepted by the automarker because of excessive runtime, please submit it again – the timelimit is now 3 seconds. If this solution is now greenlighted, and if you believe it is running an  $O(n)$ -time algorithm, to gain additional marks on Assignment 1 please send your source code to [Clark](#) as an attachment to an email explaining your asymptotic runtime analysis.