CSC165H1: Problem Set 4

Due April 4, 2018 before 10pm

General instructions

Please read the following instructions carefully before starting the problem set. They contain important information about general problem set expectations, problem set submission instructions, and reminders of course policies.

- Your problem sets are graded on both correctness and clarity of communication. Solutions that are technically correct but poorly written will not receive full marks. Please read over your solutions carefully before submitting them.
- Each problem set may be completed in groups of up to three. If you are working in a group for this problem set, please consult `https://github.com/MarkUsProject/Markus/wiki/Student_Groups` for a brief explanation of how to create a group on MarkUs.

  Exception: Problem Set 0 must be completed individually.
- Solutions must be typeset electronically, and submitted as a PDF with the correct filename. **Handwritten submissions will receive a grade of ZERO.**

  The required filename for this problem set is **problem_set4.pdf**.
- Problem sets must be submitted online through MarkUs. If you haven't used MarkUs before, give yourself plenty of time to figure it out, and ask for help if you need it! If you are working with a partner, you must form a group on MarkUs, and make one submission per group. "I didn't know how to use MarkUs" is not a valid excuse for submitting late work.
- Your submitted file should not be larger than 9MB. This may happen if you are using a word processing software like Microsoft Word; if it does, you should look into PDF compression tools to make your PDF smaller, although please make sure that your PDF is still legible before submitting!
- Submissions must be made *before* the due date on MarkUs. You may use *grace tokens* to extend the deadline; please see the Problem Set page for details on using grace tokens.
- The work you submit must be that of your group; you may not refer to or copy from the work of other groups, or external sources like websites or textbooks. You may, however, refer to any text from the Course Notes (or posted lecture notes), except when explicitly asked not to.

Additional instructions

- All final Big-Oh, Omega, and Theta expressions should be fully simplified according to three rules: don't include constant factors (so $\mathcal{O}(n)$, not $\mathcal{O}(3n)$), don't include slower-growing terms (so $\mathcal{O}(n^2)$, not $\mathcal{O}(n^2 + n)$), and don't include floor or ceiling functions.
- You may use common algebraic properties of logarithms (change of base, log of a product, etc.), and the fact that for all $x, y \in \mathbb{R}^+$, $x > y \Leftrightarrow \log x > \log y$.
- You may use (without proving them) all of the **Properties of Big-Oh, Omega, and Theta** from the Course Notes, as long as you clearly state which theorems you are using.
- When proving a bound on $WC(n)$ or $BC(n)$ you must start by writing the statement that you are proving. In addition, use the phrases 'at most' and 'at least' correctly.

1. **[4 marks] Binary representation and algorithm analysis**. Consider the following algorithm, which manually counts up to a given number n, using an array of 0's and 1's to mimic binary notation.[1]

```
from math import floor, log2


def count(n: int) -> None:
    # Precondition: n > 0.
    p = floor(log2(n)) + 1    # The number of bits required to represent n.
    bits = [0] * p            # Initialize an array of length p with all 0's.

    for i in range(n):  # i = 0, 1, ..., n-1
        # Increment the current count in the bits array. This adds 1 to
        # the current number, basically using the loop to act as a "carry" operation.
        j = p - 1
        while bits[j] == 1:
            bits[j] = 0
            j -= 1
        bits[j] = 1
```

For this question, assume each individual line of code in the above algorithm takes constant time, i.e., counts as a single step. (This includes the `[0] * p` line.)

(a) **[3 marks]** Prove that the running time of this algorithm is $\mathcal{O}(n \log n)$.

(b) **[1 mark]** Prove that the running time of this algorithm is $\Omega(n)$.

[1] This is an extremely *inefficient* way of storing binary, and is certainly not how modern hardware does it. But it's useful as an interesting algorithm on which to perform runtime analysis.

2. **[10 marks] Worst-case and best-case algorithm analysis.** Consider the following function, which takes in a list of integers.

```
def myprogram(L: List[int]) -> None:
    n = len(L)
    i = n - 1
    x = 1
    while i > 0:
        if L[i] % 2 == 0:
            i = i // 2  # integer division, rounds down
            x += 1
        else:
            i -= x
```

Let $WC(n)$ and $BC(n)$ be the worst-case and best-case runtime functions of `myprogram`, respectively, where n represents the length of the input list `L`. You may take the runtime of `myprogram` on a given list `L` to be equal to the number of executions of the `while` loop.

(a) **[3 marks]** Prove that $WC(n) \in \mathcal{O}(n)$.

(b) **[2 marks]** Prove that $WC(n) \in \Omega(n)$.

(c) **[2 marks]** Prove that $BC(n) \in \mathcal{O}(\log n)$.

(d) **[3 marks]** Prove that $BC(n) \in \Omega(\log n)$.

Note: this is actually the hardest question of this problem set. A correct proof here needs to argue that the variable `x` cannot be too big, so that the line `i -= x` doesn't cause `i` to decrease too quickly!

3. **[14 marks] Graph algorithm**. Let $G = (V, E)$ be a graph, and let $V = \{0, 1, \ldots, n-1\}$ be the vertices of the graph. One common way to represent graphs in a computer program is with an **adjacency matrix**, a two-dimensional n-by-n array[2] M containing 0's and 1's. The entry $M[i][j]$ equals 1 if $\{i, j\} \in E$, and 0 otherwise; that is, the entries of the adjacency matrix represent the edges of the graph.

   Keep in mind that graphs in our course are *symmetric* (an edge $\{i, j\}$ is equivalent to an edge $\{j, i\}$), and that no vertex can ever be adjacent to itself. This means that for all $i, j \in \{0, 1, \ldots, n-1\}$, $M[i][j] == M[j][i]$, and that $M[i][i] = 0$.

   The following algorithm takes as input an adjacency matrix M and determines whether the graph contains at least one **isolated vertex**, which is a vertex that has no neighbours. If such a vertex is found, it then does a very large amount of printing!

```
1  def has_isolated(M):
2      n = len(M)    # n is the number of vertices of the graph
3      found_isolated = False
4
5      for i in range(n):   # i = 0, 1, ..., n-1
6          count = 0
7          for j in range(n):    # j = 0, 1, ..., n-1
8              count = count + M[i][j]
9          if count == 0:
10           found_isolated = True
11           break
12
13     if found_isolated:
14         for k in range(2 ** n):
15             print('Degree too small')
```

   (a) **[3 marks]** Prove that the worst-case running time of this algorithm is $\Theta(2^n)$.

   (b) **[3 marks]** Prove that the best-case running time of this algorithm is $\Theta(n^2)$.

   (c) **[1 mark]** Let $n \in \mathbb{N}$. Find a formula for the number of adjacency matrices of size n-by-n that represent valid graphs. For example, a graph $G = (V, E)$ with $|V| = 4$ has 64 possible adjacency matrices.

   Note: a graph with the single edge $(1, 2)$ is considered different from a graph with the single edge $(2, 3)$, and should be counted separately. (Even though these graphs have the same "shape", the vertices that are adjacent to each other are different for the two graphs.)

   (d) **[2 marks]** Prove the formula that you derived in Part (c).

   (e) **[2 marks]** Let $n \in \mathbb{N}$. Prove that the number of n-by-n adjacency matrices that represent a graph with at least one isolated vertex is at most $n \cdot 2^{(n-1)(n-2)/2}$.

   (f) **[3 marks]** Finally, let $AC(n)$ be the average-case runtime of the above algorithm, where the set of inputs is simply all valid adjacency matrices (same as what you counted in part (c)).
   Prove that $AC(n) \in \Theta(n^2)$.

[2] In Python, this would be a list of length n, each of whose elements is itself a list of length n.