Due before 17:00 (EDT) on Tuesday 6 April

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.
- 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**.
- Each problem set may be completed in groups of up to three—**except for Problem Set 0**. If you are working in a group for this problem set, please consult https://github.com/MarkUsProject/Markus/wiki/Student-Guide for a brief explanation of how to create a group on MarkUs.
- 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 one or more partner(s), you must form a group on MarkUs, and make one submission per group.
- Your submitted file(s) should not be larger than **5MB**. You might exceed this limit if you use a word processor like Microsoft Word to create a PDF; in that case, you should look into PDF compression tools to make your PDF smaller, but please make sure that your PDF is still legible!
- Submissions must be made *before* the due date on MarkUs. Please see the Assessment section on the course website for details on how late submissions will be handled.
- MarkUs is known to be slow when many students try to submit right before a deadline. **Aim to submit your work at least one hour before the deadline. It is your responsibility to meet the deadline.** You can submit your work more than once; the most recent version submitted within the deadline (or within the late submission period) is the version that will be marked.
- The work you submit must be that of your group; you may not use or copy from the work of other groups, or external sources like websites or textbooks. Please see the section on Academic Integrity in the course syllabus for further details.

Additional instructions

- All final Big-O, 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 (so $\mathcal{O}(\log n)$, not $\mathcal{O}(\lceil \log n \rceil)$).
- For algorithm analysis questions, you can jump immediately from a closed-form step count expression to an asymptotic bound without proof (e.g., write "the number of steps is $3n+\log n$, which is $\Theta(n)$"). This applies to upper and lower bounds (Big-O and Omega) as well.
- However, you must evaluate all summations *before* turning them into an asymptotic bound.

- Unless specified otherwise in the question, you should use floor/ceiling when you are counting steps exactly as natural numbers.
- Unless specified otherwise in the question, you should count the cost of all lines of code in an algorithm, including constant-time steps. *This does **not** mean that each line must count as a separate step!* Just that each line must be accounted for in your analysis (even if this means multiple lines are counted as a single step).
- Show your work and explicitly cite all facts from the course notes, lecture slides, or worksheets that you use in your solutions. You may not use any other external facts—unless explicitly allowed by the question.

1. [10 marks] Analyzing nested loops.

(a) [**4 marks**] Analyze the running time of function `nested3` below, in terms of its input n, concluding with a Theta bound on the running time.

```
def nested3(n: int) -> None:
    i = 1
    while i < n:            # Loop 1
        j = i
        while j > 1:        # Loop 2
            k = 0
            while k < n:    # Loop 3
                k = k + 2
            j = j // 2
        i = i * 2
```

(b) [**4 marks**] Analyze the running time of function `up_and_down` below, in terms of its input n, concluding with a Theta bound on the running time.

```
def up_and_down(n: int) -> None:
    i = 0
    while i < n:            # Loop 1
        j = i
        if i % 2 == 1:
            while j > 0:    # Loop 2
                j = j - 1
                print(j)
        else:
            while j < n:    # Loop 3
                j = j + 1
                print(j)
        i = i + 1
```

(c) [**2 marks**] Find, with proof, an **exact** expression for the number of `print` statements executed by function `up_and_down` from the previous part, in terms of its input n.
(*Hint*: You may want to introduce cases for n.)

2. [10 marks] Worst-case analysis.

Consider the following function:

```
1 def some(lst: list, s: int) -> bool:
2     """Precondition: lst is a non-empty list of integers."""
3     for i in range(len(lst)):           # Loop 1
4         for j in range(i):              # Loop 2
5             if lst[i] + lst[j] == s:
6                 return True
7     return False
```

(a) **[2 marks]** Find, with proof, an asymptotic upper bound (Big-O) on the worst-case running time of `some`.

(b) **[3 marks]** Find, with proof, an asymptotic lower bound (Omega) on the worst-case running time of `some`, **that matches your upper bound**.

(c) **[5 marks]** Find, with proof, an input family for which the running time of `some` is $\Theta($`len(lst)`$)$.

3. [10 marks] Worst-case and Best-case analysis.

Consider the following function:

```
def loopy(lst: list) -> None:
    """Precondition: lst is a non-empty list of integers."""
    n = len(lst)
    for i in range(n-1):                        # Loop 1
        if lst[i] % 2 == 0:
            d = lst[i+1] - lst[i]
            for j in range(i+1, n):             # Loop 2
                for k in range(i, j):           # Loop 3
                    if lst[j] - lst[k] < d:
                        d = lst[j] - lst[k]
            for j in range(i+1, n):             # Loop 4
                lst[j] = lst[j] + d
        else:
            j = i + 1
            while j < n and lst[j] > 0:         # Loop 5
                lst[j] = lst[j] + 1
                j = j + 1
```

(a) [**5 marks**] Find, with proof, an asymptotic tight bound (Theta) on the worst-case running time of `loopy`. Your analysis should consist of two separate proofs for matching upper and lower bounds on the worst-case running time.

(b) [**5 marks**] In general, we define the **best-case running time** of an algorithm `func` as follows (where $\mathcal{I}_n$ is the set of all inputs of size n):

$$BC_{\texttt{func}}(n) = \min\{\text{running time of executing } \texttt{func(x)} \mid x \in \mathcal{I}_n\}$$

Note that this is analogous to the definition of worst-case running time, except we use **min** instead of max.

Analyse the **best-case** running time of `loopy` to find a Theta bound for it. Your analysis should consist of two separate proofs for matching upper and lower bounds on the best-case running time.

HINT: You should review the definitions of what it means for a function $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ to be an upper bound or lower bound on the worst-case running time of an algorithm, and take the time to write down the corresponding definitions for the best-case running time, to ensure that you know exactly what you are trying to prove. Your definitions are likely to be very similar to the ones for the worst-case, but *they should NOT be identical* (else you are doing it wrong)!

SUB-HINT: You may find it helpful to first translate the simpler statements "M is an upper bound on the minimum of set S" and "M is a lower bound on the minimum of set S" to make sure you have the right idea. Compare this with what it means for a value to be an upper or lower bound on the *maximum* of a set. Finally, coming back to algorithms, remember that upper and lower bounds can be proved separately on both the worst-case and the best-case running times (since these are two different functions).