Final Examination

Algorithms & Data Structures
NTNU IDATA 2302

May. 2025

1 Basic Knowledge

Question 1.1 (1 pt.). *What is the runtime time complexity of accessing the* i*th element of a singly linked list containing* n *items?*

Question 1.2 (1 pt.). *Consider the program below, which finds the smallest number in the given sequence. We assume that the given sequence is never empty nor null.*

```
int findMinimum(int[] sequence) {
    var minimum = sequence[0];
    int i= 1;
    while (i<sequence.length) {
        if (sequence[i] < minimum) {
            minimum = sequence[i];
        }
        i++;
    }
    return minimum;
}
```

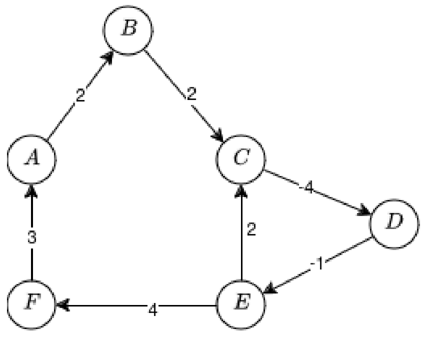
Consider the following statement:

"The variable `minimum` *always contains the smallest value among* `sequence[0]` *to* `sequence[i - 1]`*."*

Is this statement true or false? Justify briefly your reasoning.

Question 1.3 (1 pt.). *To be applicable to a given sequence, which property does the* binary search *require on that sequence?*

Question 1.4. *Consider the undirected, weighted graph below. Note that it contains negative edge weights between C and D (-4) and D and E (-1).*



Does there exist a well-defined shortest path from vertex `A` *to vertex* `E`*? Briefly justify your answer.*

Question 1.5 (1 pt)**.** *For sorting algorithms that are based solely on pairwise comparisons between elements, what is the best possible runtime complexity in the worst-case?*

2 Recursive Line Wrapping (5 pts)

Text editors and note-taking applications automatically "wrap" text to fit within a certain maximum line width. In this exercise, we implement a recursive procedure that does this wrapping one word at a time, ensuring that no line exceeds a given limit.

The function operates on a list of words and returns a formatted paragraph as a string. A new line is started whenever the next word would cause a line to exceed the maximum width.

To simplify the problem, we make the following assumptions:

- Words must be kept whole (no splitting or hyphenation).
- If a word is longer than the maximum width, the function throws an error.
- The function assumes it starts with an empty paragraph.

Here is a recursive implementation in Java:

```java
String wrapLine(List<String> words,
                String paragraph,
                int currentWidth,
                int maxWidth) {
    if (words.isEmpty()) {
        return paragraph;
    }

    String nextWord = words.get(0);
    if (nextWord.length() > maxWidth) {
```

```java
11          throw new IllegalArgumentException(
12              "Word too long for line width"
13          );
14      }
15
16      List<String> remaining = words.subList(1, words.size());
17
18      if (paragraph.isEmpty()) {
19          return wrapLine(remaining, paragraph + nextWord,
20                          nextWord.length(), maxWidth);
21      }
22
23      int needed = currentWidth + 1 + nextWord.length();
24
25      if (needed <= maxWidth) {
26          // The next word fits on the current line, so we append
27          // it after a space
28          return wrapLine(remaining,
29                          paragraph + " " + nextWord,
30                          needed,
31                          maxWidth);
32      } else {
33          // The next word does not fit on the current line, we add
34          // a new line and then we append it
35          return wrapLine(remaining,
36                          paragraph + "\n" + nextWord,
37                          nextWord.length(),
38                          maxWidth);
39      }
40  }
```

Assume the initial call is: `wrapLine(words, "", 0, maxWidth)`

Question 2.1 (1 pt)**.** *Consider the following client code. Trace the execution of `wrapLine` on the input below. For each recursive call, list the values of the arguments passed to the function: `paragraph`, `currentWidth`, and `nextWord`.*

```
var words = ["To", "be", "or", "not", "to", "be"]
var paragraph = wrapLine(words, "", 0, 10)
```

Question 2.2 (1 pt)**.** *We are now looking at the runtime complexity of this `wrapLine` procedure. For the sake of simplicity, we will **only** account for explicit comparisons in the rest of this exercise. There are two comparisons:*

- *Line 10:* `if (nextWord.length() > maxWidth)`
- *Line 25:* `if (needed <= maxWidth)`

Considering that the given list contains n words, which scenario:

- *maximize the number of these comparisons (worst-case), and*
- *minimize the number (best-case)?*

Give concrete examples of input lists that trigger both cases.

Question 2.3 (1 pt)**.** *Considering that the given list contains n words, how many times are these conditions evaluated? Express your answer as a function $C(n)$. Justify your response.*

Question 2.4 (2 pt)**.** *Write an iterative version of `wrapLine`. Your version should produce the same paragraph formatting. You may assume that all input words are shorter than or equal to the max width. At the minimum, your algorithm must accept two parameters: the list of words and the maximum width. Feel free to add any parameters you deem useful.*

3 Checking Password Strength (5. pts)

A password strength checker is a common tool used in websites and applications to evaluate whether a user's password is strong enough to protect their account. These checkers help prevent weak passwords that can be easily guessed or cracked, especially in the face of automated attacks like brute-force or dictionary attacks.

A strength checker does not try to break the password; instead, it analyzes its structure: length, use of different character types, and whether it appears in known lists of weak passwords. By quantifying these aspects, the system can warn users or reject unsafe passwords.

In this exercise, we define password strength using a weighted score based on character composition. We compute a total ”strength” as follows:

- +1 ”strength” point for each lowercase letter (a–z)
- +2 ”strength” point for each uppercase letter (A-Z)
- +4 ”strength” point for each digit (0-9)
- +8 ”strength” point for any other non alphanumeric character

We can then normalize the strength as a value within $[0, 1]$, by dividing its ”strength score by the maximum strength, which is 8 * the length of that password. The table below gives some examples.

The password xyz get a score of 0, because it is too short (it has less than four characters). Note that 123456 gets a high score, althought it is a very simple password. This is due to our rules which are naive, but they capture nonetheless the general idea.

Text	Length	Lower	Upper	Digit	Other	Strength	Normalized
`123456`	6	0	0	6	0	24	0.500
`P@ssw0rd!`	9	5	1	1	2	27	0.388
`T0pSecr3T`	9	5	2	2	0	17	0.25
`trustno1`	8	7	0	1	0	11	0.172
`admin`	5	5	0	0	0	5	0.125
`iloveyou`	8	8	0	0	0	8	0.125
`letmein`	7	7	0	0	0	7	0.125
`xyz`	3	3	0	0	0	3	0.000

Table 1: Computing the strength of a few common password

Question 3.1 (2 pt)**.** *Propose an algorithm to calculate password strength based on the above rules, including a minimum length check. The algorithm should take a text input (String, character array, etc.) and return a real number.*

You may assume the existence of helper functions such as `Character.isLowerCase`*,* `Character.isUpperCase`*,* `Character.isDigit` *in Java or equivalent, to determine the type of each character. You do NOT need to implement these checks manually.*

Question 3.2 (1 pt)**.** *For a password of a given length, give examples of passwords that exercise the worst-case and best-case runtime behavior of your algorithm*

Question 3.3 (1 pt)**.** *In the worst-case, what is the runtime complexity of your algorithm, in Big-O notation, in terms of the password length? Explain your reasoning.*

Assume that any helper function such as `Character.isDigit` *runs in $O(1)$, this is, constant time.*

Question 3.4 (1 pt)**.** *You are asked to improve your checker so that it detects and penalizes common passwords such as "123456" or "secr3t".*

- *How would change your algorithm?*
- *What data structure would you choose to store the list of common passwords, and why?*

4 Assigning Students to Mentors (5 pts)

You are tasked with designing a system that pairs students with mentors for semester-long supervision. Each student must be assigned exactly one mentor, and each mentor can supervise only one student.

Each student ranks **exactly three** different mentors as their preferred choices, in order of preference (1st, 2nd, and 3rd choice). Your goal is to produce an assignment of students to mentors that minimizes the total cost of dissatisfaction.

The dissatisfaction of a student is scored as:

- 0 if the student receives their 1st choice,
- 1 if they receive their 2nd choice,
- 2 if they receive their 3rd choice,
- 3 if they are assigned a mentor not in their preference list.

The overall objective is to minimize the **total dissatisfaction score** across all students.

Below is a sample input for 3 students (Sophie, Simon, and Sara), and 4 mentors (Max, Maria, Michelle, and Malik):

Student	**1st Choice**	**2nd Choice**	**3rd Choice**
Sophie	Max	Maria	Michelle
Simon	Michelle	Max	Maria
Sara	Michelle	Malik	Maria

Question 4.1 (1 pt). *Given the student preferences above, propose two assignments of mentors to students, such that:*

- *One assignment minimizes the total dissatisfaction across students;*
- *One assignment maximizes the total dissatisfaction across students.*

Justify your answers.

Question 4.2 (2 pt). *Design an algorithm that finds a the best assignment of mentor to students, that is an assignment that minimizes the total dissatisfaction. Describe the algorithm in plain language or pseudocode.*

Question 4.3 (1 pt). *Estimate the runtime complexity of your algorithm in terms of m, the number of mentors and s the number of students. Assume there is always more mentors available than students. Explain your reasoning.*

Question 4.4 (1 pt). *Reflect on your proposed solution. Does it always find the optimal assignment? Could it be improved in terms of efficiency or result quality? Explain your reasoning.*