# 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 ith element of a singly linked list containing n items?*

*Solution.* Accessing the i-th item runs in $O(i)$ time, because one must starts at the beginning and navigate from one node to the next, until we reach the i-th node that carries the i-th item. □

*Grading.*

- 1 pt for $O(i)$ or $O(n)$

**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.*

*Solution.* This is true. This is in fact the loop invariant that can help show this program does find the minimum. We can justify it by examining three key points:

- **Before the loop begins (i = 1):** `minimum` is set to `sequence[0]`, which is indeed the minimum of the range `sequence[0..0]`.
- **During the loop:** At each iteration, the algorithm compares `sequence[i]` with `minimum` and updates `minimum` if needed. So, by the time we reach index `i`, `minimum` reflects the smallest value among `sequence[0..i-1]`.
- **After the loop:** When the loop ends, `i == sequence.length`, so the invariant ensures that `minimum` is the minimum value in the entire array.

□

*Grading.*

- 1 pt. For the correct answer. The students do not have to mention it is the loop invariant to get full score, but only to explain why it true across the program.

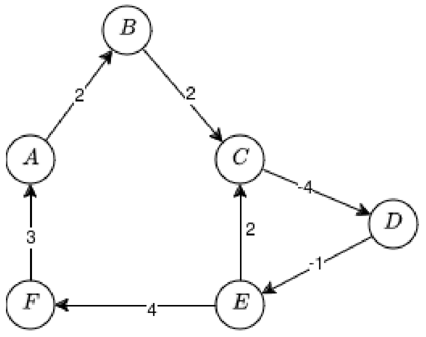
**Question 1.3** (1 pt.)**.** *To be applicable to a given sequence, which property does the* binary search *require on that sequence?*

*Solution.* The sequence must be sorted. This permits the binary search picks an element and then discard either the preceding or the following half, depending on how that element compares to the one we are searching for. □

*Grading.*

- 1 pt. For ordering of the 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.*

*Solution.* There is no shortest path between `A` and `E`. There is a negative-weight cycle in the graph: `C → D → E → C`, with a total weight of $-1$. Since this cycle is reachable from `A`, we can take a path from `A` to `E` that loops through this cycle arbitrarily many times, reducing the total path cost indefinitely. □

*Grading.*

- 0.5 pt for the good answer
- 0.5 pt for a convincing explanation.

**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?*

*Solution.* The best possible worst-case runtime complexity for any comparison-based sorting algorithm is $O(n \log n)$. This is a proven lower bound for the class of algorithms that only use comparisons to decide order. Algorithms like merge sort and heap sort achieve this optimal bound. □

*Grading.*

- 1 pt for stating the correct complexity $O(n \log n)$,
- No penalty for not giving examples of such algorithms

## 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) {
        throw new IllegalArgumentException(
            "Word too long for line width"
        );
    }

    List<String> remaining = words.subList(1, words.size());

    if (paragraph.isEmpty()) {
        return wrapLine(remaining, paragraph + nextWord,
                        nextWord.length(), maxWidth);
    }

    int needed = currentWidth + 1 + nextWord.length();

    if (needed <= maxWidth) {
        // The next word fits on the current line, so we append
        // it after a space
        return wrapLine(remaining,
                        paragraph + " " + nextWord,
                        needed,
                        maxWidth);
    } else {
        // The next word does not fit on the current line, we add
        // a new line and then we append it
        return wrapLine(remaining,
                        paragraph + "\n" + nextWord,
                        nextWord.length(),
                        maxWidth);
    }
}
```

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)
```

*Solution.* The given client call would yield the following calls:

1. paragraph="", currentWidth=0, nextWord="To"
2. paragraph="To", currentWidth=2, nextWord="be"
3. paragraph="To be", currentWidth=5, nextWord="or"

4. paragraph="To be or", currentWidth=8, nextWord="not"
5. paragraph="To be or", currentWidth=3, nextWord="to"
6. paragraph="To be orto", currentWidth=6, nextWord="be"
7. paragraph="To be orto be", currentWidth=9

□

*Grading.*

- 0.5 pt for listing the correct sequence of calls
- 0.5 pt for identifying the correct place where line breaks are inserted

**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.*

*Solution.* Considering only the number of times these two conditions are evaluated, there is no best or worst case, because the *wrapLine* function processes one word per recursive call, and in each call it evaluates the condition `if (needed <= maxWidth)` exactly once.

Therefore, the number of times the condition is evaluated is equal to the number of words: $n$. □

*Grading.*

- 1 pt for correctly identifying that the number of comparisons is $n$
- No partial credit unless justification is clearly on track

**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.*

*Solution.* We can model the number comparison evaluation as follows:

$$C(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 + C(n-1) & \text{otherwise} \end{cases} \tag{1}$$

We can expand the example of Question 2.1 and see how this recurrence behaves:

$$\begin{aligned} C(6) &= 2 + C(5) \\ &= 2 + \big[2 + C(4)\big] \\ &= 2 + \big[2 + \big(2 + C(3)\big)\big] \\ &= 2 + \big[2 + \big(2 + \big[2 + C(2)\big]\big)\big] \\ &= 2 + \big[2 + \big(2 + \big[2 + \big(2 + C(1)\big)\big]\big)\big] \\ &= 2 + \big[2 + \big(2 + \big[2 + \big(2 + \big[2 + C(0)\big]\big)\big]\big)\big] \\ &= 2 + \big[2 + \big(2 + \big[2 + \big(2 + \big[2 + 0\big]\big)\big]\big)\big] \\ &= \overbrace{2 + 2 + 2 + 2 + 2 + 2}^{6 \text{ times}} + 0 \\ &= 6 \times 2 = 12 \end{aligned} \tag{2}$$

So, the number of comparisons is $C(n) = 2n$. □

*Grading.*

- 0.5 pt for writing the recurrence relation correctly
- 0.5 pt for solving it to get $C(n) = 2n$

**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.*

*Solution.* We can iterate over the list of words and accumulate the word in the current line. When the length of the current line exceed the maximum width, we insert a new line.

```java
public static String wrapLine(List<String> words, int maxWidth) {
    String paragraph = "";
    int currentWidth = 0;

    for (String word : words) {
        if (word.length() > maxWidth) {
            throw new IllegalArgumentException(
                "Word too long for line width"
            );
        }

        if (paragraph.isEmpty() || currentWidth == 0) {
            paragraph += word;
            currentWidth = word.length();
        } else if (currentWidth + 1 + word.length() <= maxWidth) {
            paragraph += " " + word;
            currentWidth += 1 + word.length();
        } else {
            paragraph += "\n" + word;
            currentWidth = word.length();
        }
    }

    return paragraph;
}
```

□

*Grading.*

- 1 pt for correctly handling line breaks and spacing
- 1 pt for preserving the same output structure as the recursive version

## 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

| 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

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.

**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.*

*Solution.* We compute the password strength by iterating through its characters and calculating the raw score as shown in this Java example.

```
double strength(String password) {
    if (password.length() < 4) {
       return 0.0;
    }
    int strength = 0;
    for (char anySymbol: password.toCharArray()) {
       if (Character.isLowerCase(anySymbol)) {
           strength += 1;
       } else if (Character.isUpperCase(anySymbol)) {
           strength += 2;
       } else if (Character.isDigit(anySymbol)) {
           strength += 4;
       } else {
           strength += 8;
```

```
        }
    }
    int maxStrength = password.length() * 8;
    return new Double(strength) / maxStrength;
}
```

□

*Grading.* Penalize by 0.5 pt by missing feature, including:

- Checking for the minimal length
- Accounting correctly for lower case characters
- Accounting correctly for upper case characters
- Accounting correctly for digits
- Accounting correctly for non alpha-numeric characters
- Normalizing the score

**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*

*Solution.* For a password containing $n$ characters, the worst case occurs when there is only non alpha numeric characters, such as in "#!@@@#$". In this case, every condition is evaluated before we enter the "else" clause. By contrast, the best-case scenario, is when the password only contains lower-case characters (e.g., "admin"), because only the first conditional gets evaluated. □

*Grading.*

- 0.25 pt for a clear description of when worst-case happens
- 0.25 pt for a correct example that exercises the worst-case
- 0.25 pt for a clear description of when best-case happens
- 0.25 pt for a correct example that exercises the best-case

**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.*

*Solution.* The algorithm examines each character of the password exactly once in a single loop. For each character, it performs a small, fixed number of checks to determine whether it is a lowercase letter, uppercase letter, digit, or symbol. These operations (`Character.isLowerCase`, `isUpperCase`, etc.) are constant-time operations.

Therefore, the total amount of work done is directly proportional to the number of characters in the password. If $n$ is the length of the password, the algorithm performs $n$ iterations, each doing $O(1)$ work.

Hence, the overall time complexity is $O(n)$.

□

*Grading.*

- 0.5 pt for the correct answer, for instance linear/$O(n)$ for the example above.
- 0.5 pt for a convincing justification.

**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?*

*Solution.* To support this feature, we can maintain a list of commonly used or weak passwords (e.g., from public breach data). Before computing the strength score, we check whether the input password appears in this list.

To make this check efficient, we store the list in a hash table (e.g., `HashSet<String>` in Java). This allows us to test for membership in $O(1)$ average-case time.

If the password is found in this list, we can return a strength of 0.0 immediately, regardless of its composition. This penalizes predictable and insecure passwords, even if they have a varied character set.

```java
private Set<String> knownPasswords = new Set<String>();

double strength(String password) {
    if (password.length() < 4) {
       return 0.;
    }
    if (this.knownPasswords.contains(password)) {
       return 0.;
    }
    int strength = 0;
    for (char anySymbol: password.toCharArray()) {
       if (Character.isLowerCase(anySymbol)) {
           strength += 1;
```

```
        } else if (Character.isUpperCase(anySymbol)) {
            strength += 2;
        } else if (Character.isDigit(anySymbol)) {
            strength += 4;
        } else {
            strength += 8;
        }
    }
    int maxStrength = password.length() * 8;
    return new Double(strength) / maxStrength;
}
```

This approach improves both security and efficiency. □

*Grading.*

- 0.5 pt for proposing a valid algorithmic change to reject or penalize common passwords.
  - E.g., checking the password before scoring or reducing score afterward.
- 0.5 pt for identifying a hash-based structure (e.g., `HashSet` / hash table) and justifying its use for fast membership checking ($O(1)$).
  - Full credit for hash table / hash set.
  - Half credit (0.25 pt) for less efficient structures like lists or arrays, if justified.

## 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.*

*Solution.* One possible optimal assignment (with a minimum total dissatisfaction of 1) is:

- Sophie is assigned Max ($d = 0$)
- Simon is assigned Michelle ($d = 0$)
- Sara is assigned Malik ($d = 1$)

One possible worst-possible assignment (with a maximum total dissatisfaction of 11) is:

- Sophie is assigned Malik ($d = 4$)
- Simon is assigned Maria ($d = 3$)
- Sara is assigned Max ($d = 4$)

□

*Grading.*

- 0.5 pt for correct best-case example
- 0.5 pt for correct worst-case example

**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.*

*Solution.* One correct solution is to:

- Generate all possible assignments of mentor to students
- For each assignment, evaluate the total dissatisfaction associated.
- Return one assignment with the lowest dissatisfaction score.

This brute-force approach guarantees an optimal result but may be slow for large $n$.

Alternative answers may propose: - A greedy heuristic (assign preferred mentors to students in order) - A backtracking algorithm that prunes unpromising partial assignments

Partial credit awarded for any sound and well-justified approach. □

*Grading.*

- 1 pt for proposing a complete algorithm that can find an optimal or approximate solution
- 1 pt for describing the algorithm logic clearly and correctly

**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.*

*Solution.* The brute-force algorithm generates all $\frac{m!}{(m-s)!}$ possible assignments and scores each one in $O(s)$, so the total complexity is $O(\frac{m!}{(m-s)!}) \times O(s)$. The numerator, $m!$ grows faster than the denominator $m - s!$ so the complexity boils down to $O(m! \cdot s)$. □

*Grading.*

- 0.5 pt for identifying the correct asymptotic bound (e.g., $O(n! \cdot n)$)
- 0.5 pt for correctly linking the complexity to the algorithm structure

**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.*

*Solution.* The brute-force algorithm is guaranteed to find the optimal assignment but is inefficient for large input sizes. It could be improved using more advanced combinatorial search algorithms, such as maximum bipartite matching, or by applying heuristics.

Greedy methods are often more efficient but do not guarantee an optimal assignment. A better solution might to use pruning or branch-and-bound for instance.

□

*Grading.*

- 1 pt for recognizing limitations of their own algorithm and suggesting possible directions for improvement (e.g., efficiency, optimality)