Final Examination

Algorithms & Data Structures
NTNU IDATA 2302

Dec. 2024

1 Basic Knowledge

Question 1.1 (1 pt.)**.** *In the* binary search tree *(BST) shown below, which node is the successor of* Node L*? Explain your reasoning. The nodes are ordered alphabetically.*

Solution. In a binary search tree, the successor of a node X, is the node that follows X according to the ordering. In this case, nodes are ordered alphabetically. The nodes that directly follows L according to the ordering is thus *Node P.* □

Grading.

- 0.5 pt. for the correct answer, Node P
- 0.5 pt. for the correct justification: the successor is the next node according to ordering.

Question 1.2 (1 pt.)**.** *Consider the following algorithm, shown as a Java program. Does this algorithm always terminate? Explain your reasoning?*

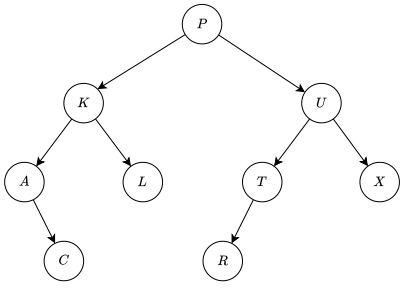

Figure 1: A sample binary search tree, where nodes are ordered by alphabetical order

```
int factorial(int n) {
 if (n == 1) return 1
 else return n * factorial(n+1);
}
```

Solution. This algorithm does not always terminate. It would terminate for any input greater than 1, but will call itself infinitely for any other input. This is due to the recursive case, which moves further away from the base case $n == 1$ at each step. In practice, this will result in a `StackOverflowError` in most programming environments. □

Grading.

- 0.5 pt. for the correct answer: It does not always terminate
- 0.5 pt. for a clear explanation.

Question 1.3 (1 pt.)**.** *Which data structure ensures that every node is greater than its children? What is this data structure used for?*

Solution. A *max-heap* is a data structure that guarantees that every node is greater than all its children. It implements a priority queue abstract data type, and is used to retrieve in logarithmic time the entry with the highest priority. □

Grading.

- 0.5 pt. for the correct data-structure: The heap. OK if it is not explicitly a max-heap.
- 0.5 pt. for the correct use-case: Priority list and retrieval of the item with the highest priority.

Question 1.4 (1 pt.)**.** *Consider an algorithm that has two steps: The first step runs in $O(n)$ (linear time), but the second step runs in $O(n^2)$ (quadratic time). What is the runtime complexity of the complete algorithm? Explain your reasoning.*

Solution. The total duration of this algorithm would be the sum of the duration of its two steps, that is, $O(n) + O(n^2)$. As the big-O notation disregards terms with lesser exponents, we are left with $O(n^2)$. The algorithms therefore also runs in quadratic time. □

Grading.

- 0.5 pt. for the correct answer, $O(n)$
- 0.5 pt. for a relevant justification.

Question 1.5 (1 pt.)**.** *A directed graph G, with four vertices A, B, C and D, is described by the following adjacency matrix. Does G contain a cycle? Explain your reasoning.*

$$G = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \tag{1}$$

Solution. Yes, there is cycle. The adjacency matrix tells us that:

1. Vertex A is connected to vertices B and D
2. Vertex B is connected to vertices A and C
3. Vertex C is connected to vertices B and D
4. Vertex D is connected to vertices A and C

Therefore there is at least one cycle A, B, A □

Grading.

- 0.5 pt. for ...
- 0.5 pt. for ...

2 Longest Common Prefix (5 pts.)

In this section, we look at the *longest common prefix* (LCP) problem. Given two sequences of characters (i.e., strings in Java), the task is to find the longest subsequence of characters that is shared from the start of the two given strings. Consider the following examples:

- The LCP of "apple" and "apply" is "appl";
- The LCP of "apple" and "apricot" is "ap";
- The LCP of "apple" and "apple" is "apple";
- The LCP of "apple" and "banana" is "" (the empty string);
- The LCP of "apple" and "" is "" (the empty string).

Below is the recursive algorithm, which computes the LCP of two strings, namely "left" and "right", here shown as a Java method:

```
1  String longestCommonPrefix(String left, String right) {
2    if (left.isEmpty() || right.isEmpty()) {
3      return "";
4    } else if (left.charAt(0) != right.charAt(0)) {
5      return "";
6    } else {
7      return left.charAt(0) + longestCommonPrefix(left.substring(1),
8                                                  right.substring(1));
9    }
10 }
```

Question 2.1 (1 pt.)**.** *In this recursive LCP algorithm, what is/are the base case(s)? What is/are the recursive case(s)?*

Solution. The *base cases* are the execution paths where the algorithm does *not* call itself. There are two here

- The first one is when either of the given strings is empty.
- The second one is when neither string is empty and their respective first character differ.

By contrast, the *recursive cases* are the execution paths where the algorithm calls itself. There is only one here, which occurs whenever neither of the two strings is empty and their respective first character are the same. □

Grading.

- 0.5 pt. for the correct base cases
- 0.5 pt. for the correct recursive case

Question 2.2 (1 pt.)**.** *Consider the number of string concatenations performed by our LCP algorithm. What is the worst-case scenario? Given an example of the input strings that would trigger that worst-case scenario?*

Note that the LCP algorithm contains only one string concatenation, on line 7. It is denoted by a `+` *sign in Java.*

Solution. The worst-case scenario occurs when the two given string are the same. In that case, the algorithm calling itself all the way through the given string, one character at a time. An example of input string would be `LCP("apple", "apple")` □

Grading.

- 0.5 pt. for the correct number of items, and appropriate explanation
- 0.5 pt. for naming the binary search

Question 2.3 (2 pt.)**.** *In the worst-case scenario, how many* string concatenations *does this algorithm performs. Express this number as a mathematical function f. Explain your reasoning and calculations.*

Solution. There are three executions paths in this recursive algorithms:

1. If either of the given strings is empty, no string concatenation is performed;
2. If neither of the given strings is empty, and their first characters are different, no string concatenation is performed either.
3. Otherwise, we perform 1 string concatenation and we "recurse".

In the worst-case scenario, however, the two given strings are the same and the two base cases boil down to one, where the given string is empty. We can then formalize this with a recurrence relationship:

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

We can expand an example to see how this recurrence behaves:

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

We can therefore conclude that $f(n) = n$.

□

Grading.

- 1 pt. for understanding how the worst-case scenario simplifies the algorithm.
- 0.5 pt. for a correct recurrence relationship
- 0.5 pt. for a correct (or meaningful) calculation, $f(n) = n$

Question 2.4 (1 pt.)**.** *Propose an alternative algorithm solving the LCP, but more efficient than the recursive one used in the previous questions. Explain why your solution is more efficient.*

Solution. We can improve the efficiency by not using recursion and using a loop instead. By avoiding recursion, we reduce the memory consumption, since the call stack will not be used during the execution.

```
String longestCommonPrefix(String left, String right) {
  var lcp = "";
  for(int i = 0 ;
      i < Math.min(left.length(), right.length()) ;
```

```
      i++)
  {
      if (left.charAt(i) == right.charAt(i)) {
        lcp = lcp + left.charAt(i);
      } else {
        break;
      }
  }
  return lcp;
}
```

□

Grading.

- 0.5 pt. for a correct algorithm
- 0.5 pt. for a correct/meaningful as for why this alternative algorithm outperforms the one given earlier.

3 Number Guessing Game

We now look at the "Number Guessing Game", where the first player chooses a number that the other player tries to find. First the two players must agree upon a minimum and maximum value. Then, after each attempt, the first player tells the other whether that was correct, too large or too small. A typical game between a player and a "CPU" could look like:

CPU: I have chosen a number between 0 and 100 (included). Can you find it?
PLAYER: Let's try 97.
CPU: Too large!
PLAYER: Let's try 56.
CPU: Too small!
PLAYER: Let's try 61.
CPU: Too large
PLAYER: Let's try 59.
CPU: Well done! You've found it in 4 tries.

Question 3.1 (1 pt.)**.** *As a player that plays against the CPU, what would be your strategy (i.e., algorithm) to win the game? Explain how you would choose your next try.*

Solution. There are various search algorithms that we can use: Linear search, jump search or binary to name a few. I would use a binary search to quickly narrow down the search interval. The binary search goes I follow:

1. Initially, the search space is the whole agreed-upon interval $[a, b]$, for instance $[0, 100]$ in the example game above. Here, `a` and `b` are variable that we will adjust as we narrow down towards the target value.

2. As long as I have not yet found the target value

   - Try the middle item μ of the search space, such as $\mu = a + \lfloor \frac{b-a}{2} \rfloor$
   - If it is too large, I would restrict the search space to $[a, \mu - 1]$
   - If it is too small, I would restrict the search space to $[\mu + 1, b]$

□

Grading.

- 0.5 pt. for a clearly explained algorithm.
- 0.5 pt. for an algorithm that works. It is not necessary to use the binary search, any search algorithm would do it.

Question 3.2 (1 pt.)**.** *Considering your total number of attempts, what is the best-case scenario of your strategy? When does it occur? In that best-case scenario, how many attempts will you need to find a value chosen in an interval* $[a, b]$ *where* a *and* b *are two integers such that* $a < b$*. Explain your reasoning.*

Solution. The best-case scenario occurs when the CPU has chosen the middle of the whole agreed-upon interval. In that case, the first guess will be correct and only one. Only one attempt will be needed. □

Grading.

- 0.5 pt. for the clear description of the best-case scenario and when does it happen.
- 0.5 pt. for the correct number of attempts, and an appropriate explanation.

Question 3.3 (2 pt.)**.** *Considering again your total number of attempts, what is the* worst-case scenario *for your strategy? When does it occur? In that worst-case scenario, how many attempts will you need?*

Solution. The binary search repeatedly halves the search space in two even parts. Consider for instance searching for a value between $a = 21$ and $b = 36$, there are $b - a + 1 = 16$ candidates. The algorithm would proceed as follows:

1. We choose $\mu_1 = a + \lfloor \frac{b-a}{2} \rfloor = 27$.
2. Depending on the CPU's answer, the next attempt can be either of $\mu_2 \in \{24, 32\}$
3. Depending on the CPU's answer, the next attempt can be any of $\mu_3 \in \{22, 26, 30, 34\}$

4. Depending on the CPU's answer, the final attempt will be any of $\mu_4 \in \{21, 23, 25, 27, 29, 31, 33, 35\}$

In this game setting, the binary search has a complexity of $O(\log_2 b - a + 1)$. In the example, above, there are 16 items to search from, and we will need $\log_2(16) = 4$ attempts. Note that there might be a fifth attempt if the chosen value is 36, because of the floor operator that may create uneven splits. Overall, the worst case occur if the CPU has chosen a value is among the last level. These values are one away from any value chosen as a cut in the previous attempt. □

Grading.

- 1 pt. for the clear description of the best-case scenario and when does it happen.
- 1 pt. for the correct number of attempts, and an appropriate explanation.

Question 3.4 (1 pt.)**.** *Without proving it, do you think there exists (or not) another algorithm, faster than your solution. Explain your reasoning.*

Solution. I do not think so. There is a search algorithm faster than the binary search: The interpolated search, which runs in $(\log_2 \log_2 n)$ Unfortunately, it does not apply because we are searching over a complete range of integers. There is no mapping from indices to value to interpolate over. □

Grading.

- 0.5 pt. for the correct/consistent answer (yes/no) with the previous answers
- 0.5 pt. for naming other algorithms that exist.

4 Course Schedule

We now look at the *Course schedule problem*. We are given a list of university courses, along with their set of prerequisite courses. Students who would like to enroll in a course must have already completed all of its prerequisite courses.

Consider, for example, the following courses:

- *Discrete math* (DM) has no prerequisite: Any student can join.
- *Computer architecture* (CA) has no prerequisite.
- *Algorithms and data structures* (ADS) requires DM and CA
- *Database* (DB) requires ADS
- *Operating systems* (OS) requires ADS

In this scenario, a student must complete these courses in an order that complies with their prerequisites. There may be many valid orderings, such as (DM, CA, ADS, DB, OS) or (CA, DM, ADS, OS, DB), but the order within each prerequisite group must be respected. In contrast, the ordering (DM, CA, OS, ADS, DB) is invalid because one cannot enroll in the operating system (OS) course without having completed ADS first. Note that there may not be any valid ordering whenever the dependencies between courses form a cycle, for instance, if DM had OS as prerequisite.

In this problem, we aim at:

1. Find *one* valid ordering, represented as a sequence of all the courses;
2. Detect cases where there is no valid ordering.

Question 4.1 (1 pt.)**.** *What data structure (or mathematical structure) would you use to reason about courses and their prerequisites? Explain how this data structure would represent the courses and their prerequisites.*

Solution. A directed graph $G = (V, E)$ accurately captures such a set of courses. The set of vertices V represents the courses, that is, $V = \{DM, CA, ADS, DB, OS\}$. The set of edges E represents the dependencies between courses, such as for every course, there exists exactly one edge from that course to each of its prerequisites. □

Grading.

- 0.5 pt. for the notion of *directed graph*;
- 0.5 pt. for a relevant mapping to vertices and edges.

Question 4.2 (1 pt.)**.** *Consider the following API that enables manipulating courses.*

```
class Course {

  /**
   * Returns "true" if and only if "this" and the "other" object
   * represent the very same course.
   * For example:
   *  - dm.equals(dm) is true
   *  - ads.equals(db) is false
   */
  boolean equals(Course other) { ... }

  /**
   * Returns the sequence of direct prerequisites of this course.
   * For example:
   *  - ads.prerequisites() yields [] (i.e., the empty list)
   *  - ads.prerequisites() yields [dm, ca]
```

```
    */
   List<Course> prerequisites() { ... }
}
```

Assuming prerequisites do NOT *form a cycle, propose an algorithm to list the prerequisites (direct or indirect) of a single given course, in a valid order. Below is the signature of such an algorithm:*

```
List<Course> orderPrerequisites(Course target) {
   // Your algorithm here
}
```

Note that you do not *have to implement the Course class. It is there for you to use in your algorithm.*

Solution. We can use a *depth-first post-order* traversal starting from the given target. In a post-order traversal, we process a node only after its children. This reflects the fact that a course must be completed *after* all its prerequisites. In pseudo Java, that would look like:

```
1  List<Course> orderPrerequisites(Course target) {
2     var schedule = new ArrayList<Course>();
3     for (Course prerequisite: target.prerequisites()) {
4         var ordering = orderPrerequisites(prerequisite);
5         for (Course c: ordering) {
6             if (!schedule.contains(c)) {
7                schedule.add(c);
8             }
9         }
10    }
11    schedule.add(target);
12    return schedule;
13 }
```

□

Grading.

- 0.5 pt. for the notion of *depth-first traversal*;
- 0.5 pt. for a correct algorithm that does not return duplicates courses.

Question 4.3 (1 pt.)**.** *We now address the problem of cycles in the prerequisites. How can we modify the* `orderPrerequisites(...)` *algorithm to detect and handle cycles in the prerequisite graph? A cycle occurs whenever a course is directly or indirectly a prerequisite of itself. Propose an updated algorithm.*

Solution. A cycle exists if, while we progress along a path, we meet again a course that we have already met. To detect cycles, we must therefore keep track of the courses we have already seen. As we progress down the path, we accumulate the courses we meet.

We need to modify the signature of the algorithm for it to accept a set "known" courses. In pseudo Java, that would look like:

```
List<Course> orderPrerequisites(Course target,
                                Set<Course> knownCourses)
{
   if (knownCourses.contains(target)) {
      throw new RuntimeError("Cycle detected!")
   }
   knownCourses.add(target);
   var schedule = new ArrayList<Course>();
   for (Course prerequisite: target.prerequisites()) {
       var ordering = orderPrerequisites(prequisite,
                                         new HashSet(knownCourses));
       for (Course c: ordering) {
           if (!schedule.contains(c)) {
              schedule.add(c);
           }
       }
   }
   schedule.add(target);
   return schedule;
}
```

It is necessary to "clone" the set `knownCourses` before processing each prerequisite. Cloning guarantees that we only detect cycles whenever a prerequisite has already been met *upstream* along the path that led to the current course, and not if a course *downstream* is a shared prerequisite.

In terms of implementation, the use of a `Set<Course>` is not strictly necessary: A `List<Course>` would work just fine. Indeed, per construction, the collection of known courses cannot include duplicate, but using a `Set<Course>`, and in particular a `HashSet<Course>`, yields better performance on larger graphs, with a membership test that runs in constant time (i.e., "`set.contains(item)`" runs in $O(1)$). □

Grading.

- 50 % for accumulating known or visited courses along the path;
- 50 % for a correct algorithm, which properly duplicates the collection of "known courses" before processing a prerequisite.

Question 4.4 (2 pts. BONUS)**.** *Finally, we look at finding a valid ordering for a set of courses (as opposed to a single course in the previous questions). Given*

a set of courses and their prerequisite relationships, design an algorithm to find a valid ordering such that all prerequisites of every course are completed before the course itself. Assume you have access to the `orderPrerequisites(Course)` *procedure from the previous question, which returns a valid ordering for a single course and its prerequisites.*

Solution. We can simply aggregate the orderings from separate courses, by concatenating them and filtering out duplicates. If a course is already in the common prerequisites, it necessarily precedes any course later appended, and does not need to be repeated. In pseudo Java that would look like:

```
List<Course> order(Set<Course> courses) {
    var schedule = new ArrayList<Course>();
    for (Course c: courses) {
        var ordering = orderPrerequisites(c, new HashSet());
        for (Course p: ordering) {
            if (!schedule.contains(p)) {
                schedule.add(p);
            }
        }
    }
    return schedule;
}
```

□

Grading.

- 50 % for concatenating "local" orderings
- 50 % for filtering out duplicates courses