Algorithms and Data Structures

NTNU IDATA 2302

Session II, May 2023

A few tips before to start. Some questions ask you to argue or explain, other may ask your to prove. When arguing/explaining, we expect a few sentences to justify, but when proving, we expect a detailed step-by-step reasoning.

Add at least a sentence to motivate the answer to every question. No point will be given for a result alone.

You are **not** required to write a program that would actually "compiles". You can—if you feel it helps—but you can also use pseudo-code, or bullet points if you prefer, or any combination there of.

There is a bonus question, that is, you can still have a full score without answering it. If you answer it correctly, the points are however part of your total score.

Good luck!

1 Basic Knowledge

Question 1.1 (1 pt.)**.** *What is the runtime complexity of the Bubble sort algorithm? Why?*

Solution. The Bubble sort algorithm runs in $O(n^2)$, where n is the length of the sequence to sort. □

Grading.

- 0.5 pt. for the correct answer.
- 0.5 pt. for the correct justification.

Question 1.2 (1 pt.)**.** *Consider the algorithm below, which counts the number of time a given letter occurs in the given word. As for the runtime, what is the best case scenario?´*

```
int countOccurrences(String word, char letter) {
  int count = 0;
  for (int i=0 ; i<word.length() ; i++) {
```

```
      if (word.charAt(i) == letter) {
        count++;
      }
    }
    return count;
  }
```

Solution. Best-case and worst-case scenarios are always defined for fixed sized inputs: Here a word of length n. The best-case scenario, which is the one that consumes the least resources, occurs when the given letter never occurs in the given word. This is the "fastest" scenario, because the instruction `count++` is never executed. □

Grading.

- 0.5 pt for the correct scenario.
- 0.5 pt for a correct explanation.

Question 1.3 (1 pt)**.** *Consider an algorithm A whose runtime is described by the function* $f(x) = \frac{3x-1}{x}$*. Is this correct to say that "A runs in* $O(1)$*"? Explain your reasoning?*

Solution. Yes, A does run in $O(1)$. The function $f(x) = \frac{3x-1}{x}$ tends towards 3 has x grows towards infinity. By definition, it is bounded above by a constant, here 3. □

Grading.

- 0.5 pt. for the correct answer (i.e., yes, this is correct).
- 0.5 pt. for the correct justification

Question 1.4 (1 pt.)**.** *Consider the hash-table shown below. It resolves collisions using "separate chaining". Suppose we insert a piece of data with key "Lisa", whose "hash" is 123. Explain where would it be inserted?*

Solution. The item with key "Lisa" would be chained behind the Item "John", which occupies entry 123. This is due to the use of "separate chaining": Each entry is thus a linked-list that contains all the items that collide. □

Grading.

- 0.5 pt. for the correct answer (i.e., linked behind John).
- 0.5 pt. for the correct justification

Question 1.5 (1 pt.)**.** *What are the two main techniques that "dynamic programming" relies on to help improve recursive algorithms?*

Solution. Dynamic programming improves recursive algorithms with two main techniques:

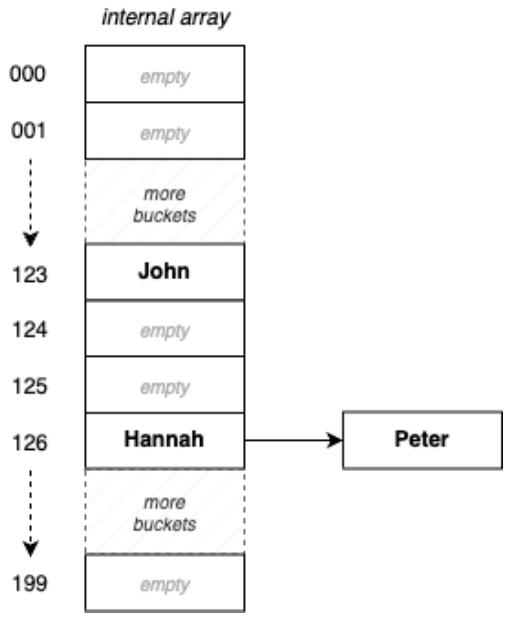

Figure 1: Internal state of the hashtable

- it uses memoïzation to avoid computing again and again similar sub-problems
- it reverses the order of problem resolution so that a problem can be solved instantly, ensuring that all the sub-problems require at any time have already been solved.

□

Grading.

- 0.5 pt. for memoization
- 0.5 pt. for the reverse order of problems

2 Linked-Lists (5 pts.)

Consider the following sketch of a Java implementation of linked-lists.

```
class List<T> {
  private Node<T> head;

  public void delete(T item) { ... }
```

```
}

class Node<T> {
  List next;
  T item;
}
```

Question 2.1 (1.5 pt.)**.** *Provide an iterative algorithm for the* `delete` *operation below. This operation should delete the given item from the list if it exists or left the list unchanged otherwise.*

```
void delete(T item);
```

Solution. To delete a node in such a linked list there are three cases to check;

- The given list is empty
- The element to remove is the first in the list, and we must update the head pointer.
- The element to remove is in the middle of list, and we need to update the previous element, to keep the chain accurate.

One possible implementation is given below. There

```
1       public void delete(T item) {
2         if (head == null) {
3             throw new IllegalStateException("Cannot delete on an empty list");
4         }
5
6         if (head.item.equals(item)) {
7             head = head.next;
8             return;
9         }
10
11        Node<T> current = head;
12        while (current.next != null) {
13            if (current.next.item.equals(item)) {
14                current.next = current.next.next;
15                return;
16            }
17            current = current.next;
18          }
19    }
```

□

Grading.

- 0.5 pt. for each of the three cases.

Question 2.2 (2 pts.)**.** *In the worst case, how many arithmetic and logical operations does your deletion algorithm perform for a linked list containing n nodes? Detail your calculation. Arithmetic and logic operations include:*

- *assignments such as* `x = 23 + y` *;*
- *comparisons, such as* $=$, $<$, $\leq$, *ldots ;*
- *logical operations such as* `&&`, `||`, *or* `!` *;*
- *arithmetic operations such as* $+$, $-$, $\times$, *and so on.*

Solution. The worst case occurs when the list does not contain the given item, because it forces our solution to check every single item and to eventually return false;

The table below summarizes all the operations, their cost, and their number of executions:

Line	Fragment	Cost	Runs	Total
2	`head == null`	1	1	1
5	`head.item.equals(item)`	1	1	1
6	`head = head.first`	1	0	0
11	`current = head`	1	1	1
12	`current.next != null`	1	n	n
13	`current.next.item.equals(item)`	1	n-1	n-1
14	`current.next = current.next.next`	1	n-1	n-1
17	`current = current.next`	1	n-1	n-1
		Grand Total		$4n$

□

Grading.

- 1 pt for accounting for all operations;
- 1 pt for the complete and correct calculation.

No point taken for not including `equals(item)` in the calculation.

Question 2.3. *Give a* recursive *alternative implementation of this* `delete` *operation.*

Solution. We can extract a separate procedure, which will embody the recursion and return the new "head" of the list as follows:

```java
public void delete(T item) {
  head = deleteRecursive(head, item);
}

private Node<T> deleteRecursive(Node<T> current, T item) {
  if (current == null) {
    return null;
  }

  if (current.item.equals(item)) {
    return current.next;
  }

  current.next = deleteRecursive(current.next, item);
  return current;
}
```

□

Grading.

- 0.5 pt. for a recursive algorithm, even a wrong one.
- 1 pt. for a correct implementation

3 Emergency Room Triage

In this exercise, we look at hospitals, and emergency departments in particular. When a new patient arrives, a doctor first quickly decides how urgent is the situation, and then records her in the system, including for instance her name, age, address, and priority (a value from 1 to 10, where 1 is the highest priority). The emergency department can thus retrieve the patient with the highest priority and help them as soon as possible.

Your task is to design a data structure to hold this list of patients.

Question 3.1 (1. pt)**.** *Which data structure would you use to maintain the patient by priority and retrieve them by priority efficiently? Justify your choice.*

Solution. I would use a binary *min-heap*, because it allows to maintain a tree where the root holds the patient with the highest priority (i.e, the smallest value). Extracting this "minimium" is an $O(1)$ operation whereas adding and deleting remains efficient $O(\log n)$, where n would represent the number of patients waiting. □

Grading.

- 1 pt. for choosing a heap with relevant justifications.

- 0.5 pt. for choosing a sorted array or a binary search tree with proper justification.

Question 3.2 (2 pt.)**.** *Explain how the extraction of the patient with the highest priority would work, with the data structure you have selected. Detail the algorithm you would use.*

Solution. To extract the patient with the highest priority level from the min-heap, we would proceed as follows:

1. Remove the root node of the heap, which is always the patient with the highest priority level.
2. Restore the min-heap property by swapping the root node with its smallest child node, and repeating the process until the min-heap property is restored.

□

Question 3.3 (2 pt.)**.** *Explain how the insertion of a patient would work, with your data structure. Detail the algorithm you would use.*

Solution. To insert a new patient into the min-heap, we would proceed as follows:

1. Create a new node for the patient and add it to the bottom-level of the heap as the leftmost node.
2. Compare the priority level of the new node with the priority level of its parent node. If the priority level of the new node is smaller than its parent node, we swap the new node with its parent node to restore the min-heap property.
3. Repeat step 2, until the new node is in the correct position in the heap (i.e., no more swapping is necessary)

□

4 Dependency Management

In this exercise, we look at package management systems, such as Maven in Java, NPM for JavaScript, PIP in Python, Cargo in Rust, NuGet in C#, etc.

Sometimes package installation fails because of so-called cyclic dependencies. Here is an example inspired by the Python ecosystem. The libraries "matplotlib", "numpy", and "scipy" are commonly used together in data science projects. "scipy 1.7.0" depends on "numpy 1.20.3" for array processing, and numpy 1.20.3 depends on "matplotlib 3.4.2", which in turn depends on "numpy". This creates a cycle dependency between the four libraries, which

can cause issues when installing or updating them, and is nicknamed the "dependency hell".

For the sake of simplicity, we imagine the following API for our package management system:

```
1  public class Package {
2    public String getName() { }
3    public String getVersion() { }
4    public List<Package> getDependencies() { }
5    public boolean equals(Object other) { }
6  }
7
8  public class PackageManager {
9    boolean hasCycle(Package package) {}
10 }
```

The interface `Package` represents the packages a developer install in her environment. Each package is identified by a unique name and a version (here a String for the sake of simplicity). Besides, each package may depend on an arbitrary number of other packages that are its "direct" dependencies.

Your task is to propose an algorithm that detects such cycles in the dependencies.

Question 4.1 (1 pt.)**.** *What data structure does the dependencies form?*

Solution. Package dependencies form a *graph*, which can contain cycles. □

Grading.

- 1 pt. Correct answer.

Question 4.2 (2 pts.)**.** *Give an algorithm to detect cycles in the dependencies of a given package. Your algorithm should detect cycles of any length.*

Solution. We can use a graph traversal to check whether starting from a package we can reach. Here I use a "modified" breadth-first search. We keep track of all the paths we are exploring (in a list of lists). Each time we explore the dependencies of a package, we check if that dependency did not already show up in the path that led us here. If that is the case, we have found a cycle, otherwise, we add this dependency to the path. In Java, the algorithm would look something like:

```
1    boolean hasCycle(Package start) {
2      var expandedPaths = new LinkedList<List<Package>>();
3      expandedPaths.add(Arrays.asList(new Package[]{ start }));
4      var expanding = true;
5      while (expanding) {
6        var paths = expandedPaths;
7        expandedPaths = new LinkedList<List<Package>>();
```

```
8           expanding = false;
9           for (var path: paths) {
10              var current = path.getLast();
11              for (var dependency: current.getDependencies()) {
12                  if (path.contains(dependency)) {
13                      return true;
14                  } else {
15                      var newPath = new ArrayList(path);
16                      newPath.add(dependency);
17                      expandedPaths.append(newPath);
18                      expanding = true;
19                  }
20              }
21          }
22      }
23      return false;
24  }
```

□

Grading.

- 1 pt. If the algorithms finds some cycle, for example, fixed-length cycles, or too many cycles.
- 2 pt. If the algorithms detects all cycles.

Question 4.3 (2 pts.)**.** *In the worst case, what is the runtime complexity of your algorithm? Explain your reasoning.*

Solution. Assume that the given start package has p package that it requires, directly or indirectly. These p packages are the p vertices of graph. In the worst case, is graph forms a single long cycle, so it has also p edges (i.e., dependencies), and the algorithm has to explore every node and every edges to detect that cycle. Each node is checked only once (otherwise we found a cycle), so that requires $O(2p)$ □

Grading.

- 1 pt. for the correct answer
- 1 pt. for the correct justification