# Final Exam 2022

NTNU IDATA 2302

Friday, December 9, 2022

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.)**.** *An algorithm runs in $O(n^2)$. Is it correct to say it also runs in $O(n^3)$?*

*Solution.* Yes this is correct, because the big-O notation denotes an upper bound. Every function $f$ bounded above by another function $g$, would also be bounded by a third function $h$, provided that $g$ is bounded above by $h$. $g$ is just not the tightest bound. □

*Grading.*

- 1 pt. for the notion of "bounded above" or big-O. Tightness is irrelevant for the grading.

**Question 1.2** (1 pt.)**.** *Consider the algorithm below that returns the smallest of its two arguments. Is it correct? Provide a proof.*

```
int minimumOf(int left, int right) {
    if (left < right) {
        return left;
```

```
    } else {
        return left;
    }
}
```

*Solution.* This algorithm is incorrect. A counter example suffices as a proof: Invoking it with left $= 5$ and right $= 3$ yields 5, which is incorrect. □

*Grading.*

- 0.5 pt. for being incorrect.
- 0.5 pt. for the counter example or for other explanation/proof.

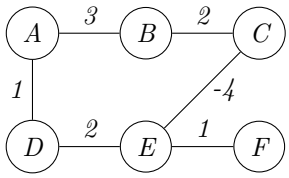
**Question 1.3** (1 pt.)**.** *Consider a sorted sequence, say* $(4, 9, 12, 17, 23)$ *for instance. Which algorithm(s) can help check whether such a sequence contains a given value* $x$*, in less than linear time with respect to the length of the array? What time complexity does the algorithm take?*

*Solution.* Provided that the given sequence is sorted, then using binary search would find an item in $\log_2(n)$. Alternatively a jump search would find the result in $\sqrt{n}$. □

*Grading.*

- 0.5 pt. for mentioning the binary search or the jump search.
- 0.5 pt. for giving the correct complexity class of the selected algorithms.

**Question 1.4** (1 pt.)**.** *Consider the following graph, where edges are labelled with distances between nodes. What is the shortest path between vertices A and F? What is the associated distance?*




*Solution.* The shortest path between nodes $A$ and $F$ goes through $A$, $B$, $C$, $E$, and $F$. Its weight is $3 + 2 - 4 + 1 = 2$. The only alternative is to go through $A$, $D$, $E$, $F$, whose weight is $1 + 2 + 1 = 3$. □

*Grading.*

- 0.5 pt for the correct path;
- 0.5 pt for the correct weight.

**Question 1.5** (1 pt.)**.** *What algorithm can we use to find the shortest path automatically in this graph? Explain your choice.*

*Solution.* As the above graph contains negative weights, which preclude Dijkstra's algorithm. Only the Bellman-Ford algorithm or the Floyd-Warshall algorithm supports such negative weights. □

*Grading.*

- 0.5 pt. for detecting negative weights
- 0.5 pt. for either of the Bellman-Ford algorithm of the Floyd-Warshall algorithms.

## 2 Finding Duplicates

Consider a sequence of integers $s$, such as $s = (4, 61, 3, 10, 17)$. This exercise focuses on finding whether $s$ contains duplicates, that is, the same number showing up several times. Here are a few examples:

- $(1, 3, 4, 3, 2)$ contains twice the value 3
- $(1, 3, 4, 6, 2)$ does not contain duplicate.
- $(1, 3, 4, 3, 4)$ contains duplicates: twice 3, and twice 4.

**Question 2.1** (3 pts.)**.** *Propose an algorithm "hasDuplicate" that detects whether a given sequence contains duplicates and runs in quadratic time in the worst case (i.e., $O(n^2)$).*

*Solution.* A "naive approach" works in quadratic time. It iterates over the given sequence $s$, and, for each element $e$, iterates again over the sequence to check if the element $e$ does not appear somewhere else. That gives the following algorithm, here shown as a Java program:

```
1  boolean hasDuplicate(int[] sequence) {
2      for (int each=0 ; each<sequence.length ; each++) {
3          for (int other=0 ; other<sequence.length ; other++) {
4              if (other != each
5                  && sequence[each] == sequence[other]) {
6                  return true;
7              }
8          }
9      }
10     return false;
11 }
```

□

*Grading.*

- 1 pt. if the algorithm detects duplicates in *some case*.

- 1 pt. if the algorithm detects duplicates in all cases.
- 1 pt. if the algorithm runs in quadratic time.

**Question 2.2** (1. pt)**.** *What is the worst case for this algorithm? What is the best case? Give an example for both.*

*Solution.* This algorithm checks all the pairs of elements $(x, y)$ that can be drawn from the given sequence. In the best case, the first pair is a duplicate and the algorithm stops here. A sequence $s = (1, 1, 2, 10)$ illustrates this case. By contrast, in the worst case, there is no duplicate and the algorithm traverses the whole sequence for each item. A sequence $s = (1, 2, 3, 4)$ illustrates this case. □

*Grading.*

- 0.5 pt. for the correct worst-case and example.
- 0.5 pt. for the correct best-case and example.

**Question 2.3** (2 pts.)**.** *How many comparisons does your algorithm perform in the worst case? Express this number as a function of the length $\ell$ of the given sequence. Detail your calculation.*

*Solution.* The algorithms performs 4 comparisons, one in the outer loop (line 2), one in the inner loop (line 3), and two when checking for duplicates (line 5 and 6). The first is evaluated $\ell + 1$ times (for index from 0 to $\ell$, included). For each valid index (0 to $\ell - 1$), the line 3 is executed, that is $\ell \times (\ell + 1) = \ell^2 + \ell$ times. Finally, lines 5 & 6 , which contain two comparisons, are executed for every valid pairs of indices, that $l^2$ times. That gives us a total of $3\ell^2 + 2^\ell + 1$ comparisons for a sequence of length $l$. The table below summarises the calculations.

| Lines | Instruction | Comparisons | Runs | Total |
|---|---|---|---|---|
| 2 | `each<sequence.length` | 1 | $\ell + 1$ | $\ell + 1$ |
| 3 | `other<sequence.length` | 1 | $\ell \times (\ell + 1)$ | $\ell^2 + \ell$ |
| 5,6 | `other != each && ...` | 2 | $\ell^2$ | $2\ell^2$ |
| | | | Total | $3\ell^2 + 2\ell + 1$ |

□

*Grading.*

- 1 pt. for a correct approach (depending on their algorithm).
- 1 pt. for a correct calculation (all steps are correct).

**Question 2.4** (1. pt)**.** *Where does your algorithm "waste" time performing unnecessary comparisons? Which data structure could you use to avoid that? Why?*

*Solution.* This algorithm iterates many times through the beginning of the array, checking whether it contains a specific element. A *hash table* could avoid such traversals and tells in constant time whether a given item has already been met. □

*Grading.*

- 0.5 pt for where the algorithm does duplicate work. Could also be phrased as checking ordered pairs whereas the order does not matter.
- 0.5 pt for a reference to hash-table or hashing

**Question 2.5** (3 pts.)**.** *Propose another algorithm that detects such duplicates in linear time (i.e., $O(n)$). Why does it run in linear time?*

*Solution.* By using a hash table, we can index all the element we have met so far. We only need to check what has been met so far because the $(x, y)$ will eventually be check when we reach $y$. For instance, given the sequence $s = (1, 2, 3, 4, 1)$, we can detect the duplicate 1 when searching for another 1 after the first one, *or* by searching for another one before the second one. That gives the following algorithm, here shown as a Java program.

```
1  boolean hasDuplicate(int[] sequence) {
2     var metSoFar = new HashSet<Integer>();
3     boolean duplicateFound = false;
4     int i = 0;
5     while (!duplicateFound && i < sequence.length) {
6        duplicateFound = metSoFar.contains(sequence[each]);
7        metSoFar.add(sequence[each]);
8        i++;
9     }
10    return duplicateFound;
11 }
```

Because adding and retrieving from a hash table both take a constant time, we see that this algorithm only iterates once through the given sequence, that is, it runs in $O(\ell)$. □

*Grading.*

- 1 pt. for a faster algorithm.
- 1 pt. for a linear time algorithm
- 1 pt. for a correct justification of the runtime.

## 3 Recursive Sums

Consider the following Java program. It computes the sum of the given array of integers values, for the range delimited by the two indices `start` and `end`. Note the recursive call. You can assume that `start` $\leq$ `end`.

```
int sum(int[]array, int start, int end) {
    if (start >= end) {
        return array[start];
    }
    return array[start] + sum(array, start+1, end);
}
```

**Question 3.1** (2 pts.)**.** *How many operations (i.e, arithmetic and logical operations) would the* `sum` *algorithm perform for a range of size $n$? (Hint: how would you model this number of operations as a recurrence relationship).*

*Solution.* The algorithm contains three operations: One comparison and two additions. We can model the number of operations as the following recurrence relationship:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3 + T(n-1) & \text{otherwise} \end{cases}$$

By expansion, we can see that:

$$\begin{aligned} T(n) &= 3 + T(n-1) \\ &= 3 + (3 + T(n-2)) \\ &= 3 + (3 + (3 + T(n-3))) \\ &= \overbrace{3 + 3 + \ldots + 3}^{n-1 \text{ times}} + T(1) \\ &= 3 \cdot (n-1) + 1 \\ &= 3n - 2 \end{aligned}$$

□

*Grading.*

- 1 pt. for a recurrence relationship
- 0.5 pt. for an accurate recurrence relationship
- 0.5 pt. for a correct calculation

**Question 3.2** (2 pts.)**.** *How much memory would the* `sum` *algorithm require for a problem size $n$? You can assume that each parameter occupies a single memory cell.*

*Solution.* We can model the memory as the following recurrence relationship:

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ 3 + T(n-1) & \text{otherwise} \end{cases}$$

We can reduce it by expansion to $T(n) = 3n$. □

*Grading.*

- 1 pt. for a recurrence relationship
- 0.5 pt. for an accurate recurrence relationship
- 0.5 pt. for a correct calculation

**Question 3.3** (2 pt.)**.** *Sketch an alternative algorithm, using recursion, which computes the same sum. (Hint: try to add terms in a different way).*

*Solution.* There are many possible ways to identify self-similar sub-problems. For example, we can rely on the associativity of the addition over integers and groups additions such as:

$$\begin{aligned} sum(x_1, x_2, \ldots, x_n) &= x_1 + (x_2 + (x_3(+\ldots + (x_n)))) \\ &= (x_1 + \ldots + x_{n/2}) + (x_{1+n/2} + \ldots + x_n) \end{aligned}$$

That yields the following algorithm, here shown as a Java program:

```
int sum2(int[] array, int start, int end) {
   if (end <= start) {
      return array[start];
   }
   var cut = start + (end - start) // 2;
   return sum2(array, start, cut) + sum2(array, cut+1, end);
}
```

□

*Grading.*

- 1 pt. for an algorithm that correctly computes a sum
- 1 pt. for a recursive algorithm

# 4 Coin Change

Consider now the problem that cashiers have when they give change to customers who pay cash. Say for instance the customer must pay 57 kr but gives 70 kr. The cashier must return 13 kr, but using what coins? 1 coin of 10 kr plus 3 coins of 1 kr or maybe another combination thereof. In this exercise, your task is to create an algorithm that finds the minimum number of coins (no note) that adds up to a given amount. We will consider the following four types of coins (so-called denominations): 1 kr, 2 kr, 5 kr, 10 kr.

**Question 4.1** (2 pts.)**.** *Consider we have to give back 13 kr to the customer, and that our cash drawer contains 3 coins for each of the four possible types of coin. One way is to pick one coin of 10 kr and 3 coins of 1 kr. List all the other possible ways to return 13 kr?*

*Solution.* Here are the possible sets of coins whose value add up to 13 kr.

- $3 \times 1\ kr + 1 \times 10\ kr$ (given)
- $3 \times 1\ kr + 2 \times 5\ kr$
- $1 \times 1\ kr + 1 \times 2\ kr + 2 \times 5\ kr$
- $1 \times 1\ kr + 1 \times 2\ kr + 1 \times 10\ kr$
- $2 \times 1\ kr + 3 \times 2\ kr + 1 \times 5\ kr$

□

*Grading.*

- 0.5 pt per correct combination

**Question 4.2** (2 pts)**.** *Provided our drawer contains 3 coins for each denomination, and that we have to return 13 kr. A friend tells us that the best solution to return 3 kr is to use 1 coin of 1 kr and 1 coin of 2 kr. What (similar) problem do we still have to solve in order to use this information for solving the original problem of returning 13 kr?*

*Solution.* We still need to find the minimum number of coins needed to return the 10 kr difference ($10 = 13 - 3$). This is the similar problem: Returning 10 kr, with a drawer that only contains two 1 kr coins, two 2 kr coins, three 5 kr coins, and three 10 kr coins. □

*Grading.*

- 1 pt for a sub problem with 10 kr ;
- 1 pt for a sub problem with the correct drawer;

**Question 4.3** (1 pt)**.** *How would you combine the solution given by our friend with the solution of this remaining problem to solve our initial problem, which is to return 13 kr?*

*Solution.* In this case, since our cash drawer contains a denomination with that exact value, the best solution is to pick one of such 10 kr coin. We can merge the solutions by merging the sets of coins, so the final solution should be one 10 kr coin, one 2 kr coin and one 1 kr coin. □

*Grading.*

- 1 pt for merging the two solutions by making the unions of the set of coins.

Let us model the solution of this coin change problem as a "coin set", that is a set of ordered pairs $(d, n)$, where $d$ is the denomination and $n$ the number of coins of that type. For instance, a set of coins with only two 2 kr coins and one 5 kr coin would be $\{(0, 1), (2, 2), (1, 5), (0, 10)\}$. Should we be programming in Java, our algorithm would have the following signature:

```
CoinSet minimumCoins(int value, CoinSet drawer) {
    // ...
}
```

We assume as well the existence of the following operations to manipulate these `CoinSets`, which we shows here as a Java interface for the sake of simplicity

```
class CoinSet {
    // Empty coin set
    public CoinSet() {}

    // The denomination whose count is greater than 0
    int[] availableDenominations() { ... }

    // Add n coins of the denomination d
    put(int d, int n) { ... }

    // Remove n coins of the denomination d
    take(int d, int n) { ... }

    // Get the set with least coins
    static CoinSet smallest(CoinSet candidates[]) { ... }

}
```

**Question 4.4** (3 pts.)**.** *Sketch a recursive algorithm, which, given a value $v$ (in kroner) and the number of coins available for each denomination (i.e., the state of the cash drawer), finds the minimum number of coins that sum up to this given value $v$.*

*Solution.* One possible recursive design goes as follows. The base cases are:

- when the given value is negative, there is no solution.
- when the price is exactly zero, then we found a solution, which is the empty coin set.

In the recursive case, we explore all the possible denomination, and for each available one, we solve the sub problem with a price reduced by the selected denomination, and a drawer deprived a one coin of the selected denomination.

The Java fragment below implements this design.

```
CoinSet minimumCoins(float value, CoinSet drawer) {
    CoinSet solution;
    if (value < 0) {
        solution = null;
    } else if (value == 0) {
        solution = new CoinSet();
    } else {
        var candidates = new ArrayList<CoinSet>();
        for (var coin: drawer.availableDenominations()) {
            drawer.take(coin, 1);
            candidate = minimumCoins(value - coin, drawer);
            if (candidate != null) {
                candidate.put(coin, 1);
                candidates.append(candidates);
            }
            drawer.put(coin, 1);
        }
        solution = CoinSet.smallest(candidates);
     }
     return solution;
}
```

□

*Grading.*

- 1 pt for a recursive algorithm
- 1 pt for an overall understanding of the idea, for instance, an algorithm misses for instance a base cases, or some adjustment of the drawer in the recursive case.
- 1 pt. for a fully correct algorithm

**Question 4.5** (3 pts.)**.** *What drawback do you see with this recursive algorithm? What technique(s) would you do to improve your solution?*

*Solution.* The problem with such recursive algorithm is that they solve several times the same sub problems, leading to long run time and heavy memory consumption. We could use dynamic programming which combines memoization and iteration to avoid unnecessary recursion and get a solution in linear time. □

*Grading.*

- 1 pt for memory drawback.
- 1 pt for dynamic programming
- 1 pt. for memoization and iteration or bottom-up traversal

**Question 4.6** (2 pts. (Bonus))**.** *Consider now that we have infinitely many coins for each denomination. Would your algorithm still terminate? Explain why?*

*Solution.* Even if we had infinitely many coins in our drawer, each recursion decreases the price, which would eventually becomes zero, and the algorithm would stop. □

*Grading.*

- 1 pt for the fact that the algorithm still terminates
- 1 pt for right justification, the fact the price decreases towards zero.

---

**End of examination**