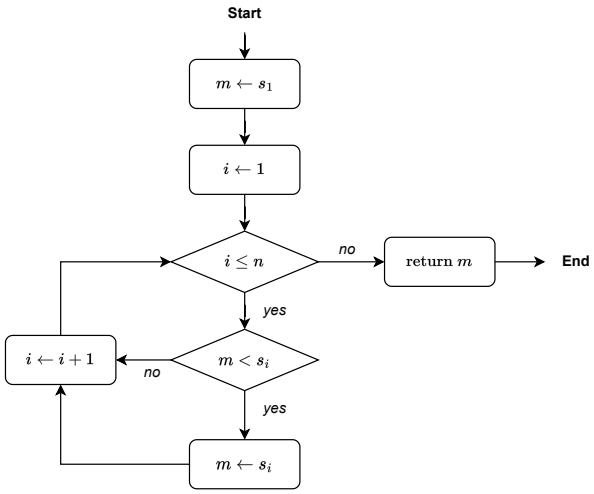
Final Examination

Algorithms & Data structures
NTNU IDATA 2302

December 2023

1 Basic Knowledge (5 points)

Question 1.1 (1 pt.)**.** *Consider the flowchart shown below. It describes an algorithm that accepts a sequence $s = (s_1, \ldots, s_n)$ as an argument and yields one single item. Write the equivalent pseudo-code.*



Solution. The pseudo code below captures the behavior as the flowchart above.

Input: $s = (s_1, \ldots, s_n) \in \mathbb{N}^n$
$m \leftarrow s_1$;
$i \leftarrow 1$;
while $i \leq n$ **do**
  if $m < s_i$ **then**
    $m \leftarrow s_i$;
  end
  $i \leftarrow i + 1$;
end
return m

□

Grading. The description of neither the inputs or the output matter here, the point is to see whether one understand a simple flowchart.

- 0.5 pt. for the loop structure
- 0.5 pt. for the inner conditional

Question 1.2 (1 pt.)**.** *Using the Big-O notation, is it valid to write the following:* $O(n) + O(2n) \in O(n)$*? Explain your reasoning.*

Solution. Yes, this expression is valid. The $f \in O(g)$ captures that fact that g is an upper bound of the function f. Now if one adds a linear upper bound (say $O(n)$) to another (say $O(2n)$), the result is still a linear upper bound.

Another way to look at this is to recall that the big-O notation disregard constant factor. So adding up theses terms we would get:

$$O(2n) + O(n) = O(3n) = O(n) \tag{1}$$

□

Grading.

- 0.5 pt. for a correct answer (valid expression)
- 0.5 pt. for a meaningful explanation.

Question 1.3 (1 pt.)**.** *Consider the graph* G *shown below and the tree* T *aside. Is* T *a "spanning tree" of* G*? Explain your reasoning.*

Solution. T is not a spanning tree, because it does *not* include all the vertices of G. T does not include Vertex E so it is not a spanning tree. □

Grading.

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

Question 1.4 (1 pt.)**.** *Consider the minimum-heap shown below in Figure 1.4. What is the configuration after one extracts the minimum value 12? Explain your reasoning.*

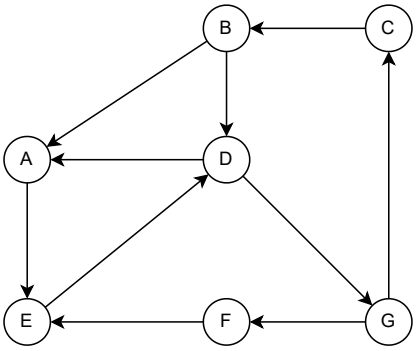


Figure 1: A sample graph G with 7 vertices

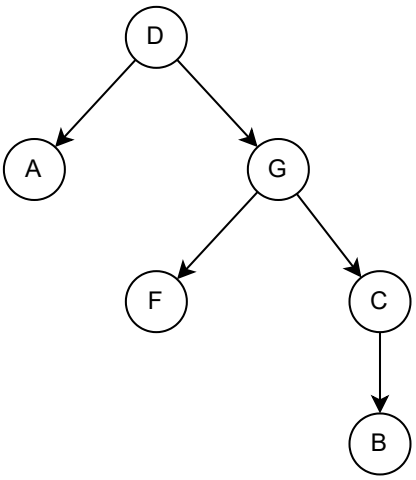


Figure 2: A tree built on G vertices

Figure 3: Graph and spanning Trees

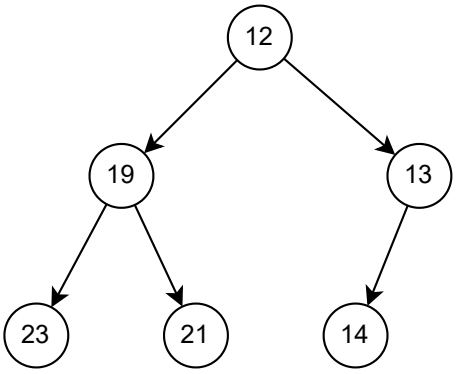


Figure 4: A sample minimum-heap with 6 values

Solution. When we remove the last item in the heap (i.e., 14) is move as a temporary root. Then it moves this new root down the tree to restore the heap invariants. In that case, we swap it with 13, which becomes the new root, and the process stops here. □

Grading.

- 0.5 pt. for a correct configuration after removing 12

- 0.5 pt. for a valid justification, swap with the last item, and "bubbling down" the head to restore the heap property.

Question 1.5 (1 pt.)**.** *In statement "P = NP" is one of the famous open problem in Computer Science. In this context, what do P and NP stand for?*

Solution. *P* and *NP* are two complexity classes. P represents the set of problems that can be solved in polynomial time by a deterministic Turing machine, whereas NP represents the set of problems that can be solve in polynomial time by a non-deterministic Turing machine. □

Grading. Check whether the notion of complexity class is understood

- 0.5 pt. for complexity classes (sets of computational problems)
- 0.5 pt. for accurate description of P and NP characteristics.

2 Reversing a String (5 points)

Let's look at the problem of reversing a given string, that is building the string that contains the same characters but in the opposite order. Here are a few examples:

- "hello" becomes "olleh" ;
- "car" becomes "rac";
- "madam" stays "madam";
- "evil" becomes "live";
- "desserts" becomes "stressed" ;
- etc.

In this exercise, we look at a procedure that accepts a string as input, and returns its reserve (as a new string). Here is what it would look like in Java:

```
1  String reverse(String input) {
2    String reversed = "";
3    int index = input.length - 1;
4    while (index >=0) {
5      reversed += input.charAt(index);
6      index--;
7    }
8    return reversed;
9  }
```

Question 2.1 (2 pt.)**.** *How many operations are performed by this algorithm? Express this number as a function of the length ℓ of the input string.*

Your count of operation should include:

- *comparisons and other logical tests*
- *arithmetic operations such as addition, subtraction, division, multiplication. We consider that adding a character to a string is a valid addition.*
- *variable assignments*

Solution. The operations of interest are the following:

- Line 2, there is one assignment. This runs once.
- Line 3, there are two, an assignment and a subtraction These two run once.
- Line 4, there is one comparison, which will run $\ell + 1$ times
- Line 5, there is an assignment and an addition, these run for every character in the input string, so ℓ times.
- Line 6, there is an assignment and a subtraction, which runs for every character in the input string, so ℓ times.

We can summarize these in the table below, and we see, that in total, we need $5\ell + 4$ operations for a string of length ℓ.

Line	Fragment	Cost	Runs	Total
2	`result = ""`	1	1	1
3	`index = input.length - 1`	2	1	2
4	`index >= 0`	1	$\ell + 1$	$\ell + 1$
5	`reversed += input.chartAt(index)`	2	ℓ	2ℓ
6	`index--`	2	ℓ	2ℓ
			Grand Total	$5\ell + 4$

□

Grading.

- 1 pt. for a linear relationship
- 0.5 pt. for accounting for the correct operations
- 0.5 pt. for the correct grand total

Question 2.2 (2 pt.)**.** *Propose an equivalent recursive algorithm that reverses an input string*

Solution. One way to reverse a string recursively is to pick up the last character of the input string, and to add in front on the reverse of the remainder. The example below shows how that would look like in Java:

```
1     String reverse(String input) {
2       if (input.equals("")) {
3         return "";
4       }
5       return input.chartAt(input.length-1)
6              + reverse(input.subString(0, input.length-1));
7   }
```

□

Grading.

- 1 pt. for a recursive procedure
- 1 pt. for a correct recursive procedure

Question 2.3 (1 pt.)**.** *Considering runtime and memory consumption, which of these two approaches (iterative and recursive) would you favor? Why?*

Solution. I would favor the iterative one, The recursive one consume more memory because of the call stack. In practice, this call stack has a limited capacity, which, in turn, limits the length of the word that can be reversed. By contrast, the iterative version does not requires any additional memory. □

Grading.

- 1 pt. for recommending the iterative version
- 0.5 pt. for raising memory consumption concern and the call stack.

3 Finding the Majority Element

We now turn our attention to the *majority item problem*. Given a sequence $s = (s_1, s_2, \ldots, s_n)$, find the item that occurs more than 50 %.

Consider the following examples:

- The empty sequence $s = ()$ has no majority item.
- The sequence $s = (A, B, C, C, C, B, A)$ does not admit a majority item. The item 'C' occurs only three times (it should occurs at least 4 times to be a majority item).
- The sequence $s = (A, A, A, A, A, B, B)$ has majority item 'A', which occurs 5 times over 7 items (70 %).
- The sequence $s = (A, A, B, B)$ has no majority item. Both A and B occurs exactly 50 %.

This exercise focuses on building an algorithm, which, given a sequence of characters, returns the majority item if any, or null otherwise. In Java, that would look like

```
char findMajorityItem(char[] input) {
  return ...;
}
```

Question 3.1 (2 pt.)**.** *Propose an algorithm to find the majority item.*

Solution. One solution is to use a hash table to map each character to its number of occurrence. We would proceed as follows:

1. Create an empty hash table, H
2. For each character c in the input sequence
   (a) if $c\neg \in H$, we initialize the count to 1.
   (b) Otherwise, we increment the count by 1.
3. We then search in the hashtable for the first entry whose count is greater than half of the given sequence.

In Java, that could look like:

```
char findMajorityItem(char[] input) {
  var counts = new HashMap<Character, Integer>();
  for (var c: input) {
    if (counts.containsKey(c)) {
      counts.put(c, counts.get(c) + 1);
    } else {
      counts.put(c, 1);
    }
  }
  for (var entry: counts.entrySet()) {
    if (entry.getValue() > input.length / 2) {
      return entry.getKey();
    }
  }
  return null;
}
```

□

Grading.

- 1 pt. for a valid idea: naive approach, recursion, Boyer-Moore, hashing, etc.
- 0.5 pt. for handling the empty sequence (see Example 1)
- 0.5 pt. for handling the 50 % scenario (see Example 4)

Question 3.2 (2 pt.)**.** *What is the runtime complexity of your solution, in the worst case? Explain your reasoning.*

Solution. The algorithm shown above works in two steps. First, it traverses the input sequence $s = (s_1, s_2, \ldots, s_n)$ to build a table that maps each character to its number of occurrences. That takes n steps. Then, its traverses this table to search for a majority element. This takes as many steps as there are distinct characters in s, say k. In general it takes $O(n + k)$.

In the worst case, each character in the input sequence occurs only once, and the hashtable will contain as many entries as there are characters in the input sequence, that is $k = n$. In that case, there cannot be a majority item, and the runtime becomes $O(n + k) = O(2n) = O(n)$. □

Grading.

- 1 pt. for a correct answer
- 1 pt. for a valid justification

Question 3.3 (1 pt.)**.** *What is the memory complexity of your solution, in the worst case? Explain your reasoning.*

Solution. As explained above, in the worst case, the given sequence contains only characters that occur once. The hashtable therefore contains as many entries as there are characters in the given sequence, that is n. In the worst-case, this solution requires $O(n)$ memory. □

Grading.

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

4 The 8-puzzle

The *8-puzzle* is a classic sliding puzzle consisting of a 3×3 grid with eight numbered tiles and one empty space. The goal is to rearrange the tiles from a given initial configuration to a specified goal configuration by sliding the tiles into the empty space. Each tile can move to an adjacent empty space (either vertically or horizontally) in a single move. The puzzle is typically initialized with a random arrangement of the tiles, and the challenge lies in finding a sequence of moves that leads to the desired configuration. Figure 5 illustrates a few possible moves from a given configuration.

Players aim to reach the goal state, usually a specific numerical order, by strategically sliding tiles one at a time until the entire grid is ordered accordingly. The puzzle's complexity arises from the vast number of possible configurations and the limited movements allowed.

Question 4.1 (1 pt.)**.** *How many configurations are possible for a 8-puzzle? Explain your reasoning.*

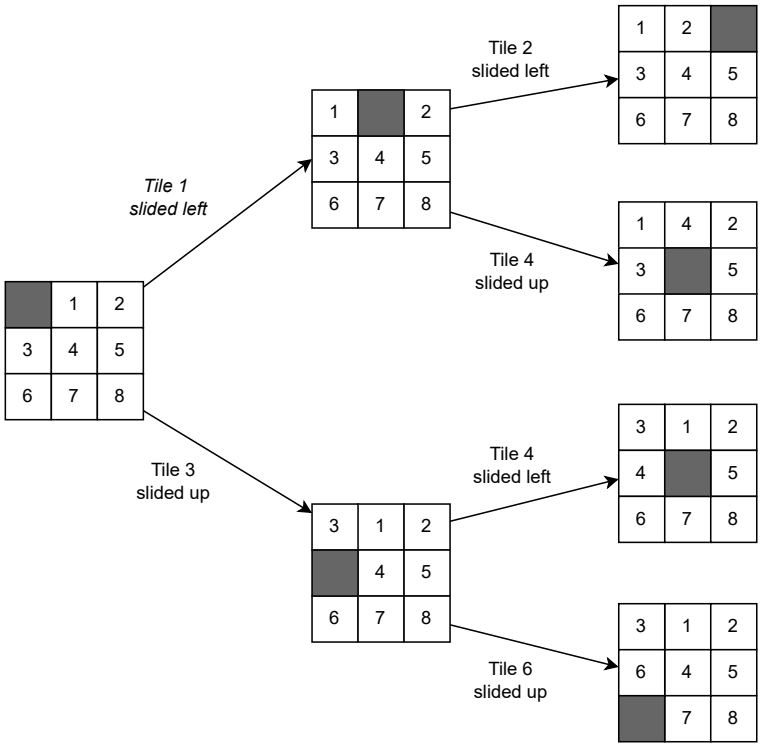


Figure 5: Example of 8-puzzle configurations with transitions from one configuration to the next

Solution. There are nine positions in the 3×3 grid. Each position can take any of the following 9 symbols: 1, 2, 3, 4, 5, 6, 7, 8 and $\varnothing$. It is not possible to repeat any symbols, so there are 9 candidates for the first position, then 8 candidates for the second, 7 for the third, etc.

That gives us $9 \times 8 \times 7 \times \ldots \times 1 = !9$, that is, 362 880 positions. □

Grading.

- 0.5 pt. for the correct answer (!9)
- 0.5 pt. for a valid justification

Question 4.2 (1 pt.)**.** *Consider the initial and target configurations shown below on Figure 6. What is the sequence of move to reach the "target" configuration?*

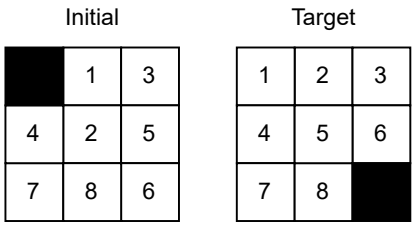


Figure 6: Two 8 puzzle configurations

Solution. The solution is given by the following sequence:

1. 1 moves left ;
2. 2 moves up ;
3. 5 moves left ;
4. 6 moves up.

□

Grading.

- 1 pt. for the correct sequence of moves

Question 4.3 (3 pt.)**.** *Outline an algorithm to check whether a given "target" position is reachable from a given "initial" configuration.*

Assume for instance the following interface for a configuration, with methods to compare configuration and compute what configurations can be reached (in one move) from the current one.

```
interface Configuration {

  /**
   * @return true if the configuration are the same
   */
  boolean equals(Configuration other);

  /**
   * @return the set of configuration that can be reached by moving any
   * tile into the free position.
   */
  Set<Configuration> neighbors();

}
```

Solution. We can use a depth-first search to traverse the graph of configurations that can be reached from the given initial configuration.

The depth-first search proceeds as follows. We need to maintain a list of the configurations we have already visited and a list of configuration yet to explore.

1. Initialize the set of visited configuration as empty
2. Create a stack, that will hold the configuration to visit next, and place the initial configuration on top of it.
3. While there is a configuration in the stack
   (a) Remove the configuration on top of the stack
   (b) Mark it as visited
   (c) If any of its neighbor is the target configuration, then the puzzle is solvable. We are done.
   (d) Otherwise, push onto the stack any neighbor that has not already been marked as visited.
4. When the stack is empty and we have not yet found the target configuration, then the puzzle is not solvable.

In Java, that could look like:

```
1  boolean isReachable(Configuration start, Configuration target) {
2      var visited = new HashSet<Configuration>();
3      var pending = new LinkedList<Configuration>();
4      pending.push(start);
5      while (!pending.isEmpty()) {
6          var current = pending.pop();
7          visited.add(current);
8          for (var next: current.neighbors()) {
```

```
                if (!visited.contains(next)) {
                    pending.push(next);
                }
                if (next.equals(target)) {
                    return true;
                }
            }
        }
        return false;
    }
```

□

Grading.

- 1 pt. for using a known search algorithm (BFS, DFS, branch and bound, etc.)
- 1 pt. for avoiding cycles (e.g., using a set as shown above)
- 0.5 pt for detecting unsolvable puzzles
- 0.5 pt for the overall quality of the explanations, including data structure used, stack, queue, priority, heuristic, etc.