EXAM TOPICS

The guidelines for the exams:

- Midterm is 90 minutes; Final exam is 180 minutes (3 hours).

- Midterm and final are worth 20% of your total grade each.

- Books, notes, communication with other people, calculators or the Internet access is not allowed. You can take one page of size A4 with any reference information you may need.

- Students who had extenuating reasons why they could not write the midterm (or final) during the regular time, are given one extra opportunity to rewrite it. The make-up exam has different questions covering similar topics at a similar level of detail.

Note: Before the midterm graph algorithms and hashing-related algorithms (Parts 6 and 7 from the list below) were covered during the lectures. Some easier questions on these topics may be asked during the midterm as well. More exercises on these topics will follow in the practice sessions after the midterm, and will be included in the final.

0.1 Question Types

[MathStatements] Prove or disprove mathematical statements about algorithms or create your own. Most often they are about asymptotic bounds for time complexity, some probabilistic reasoning or verifying that some invariant stays true.

[DrawDiagrams] Draw memory states of some data structure (such as array, linked list, heap, tree, graph) as some operations are performed.

[UnderstandAlgorithms] Given a pseudocode of some algorithm, draw trees of recursive calls, local variables or other processed data as the algorithm executes.

[CompareImplementations] Given an ADT with several implementation alternatives or an algorithmic task with one or more ways to implement it, compare the implementations by writing and analyzing relevant pseudocode and finding its time complexity.

[DesignByParadigm] Write a pseudocode for an algorithmic task observing some limitations or using one or more paradigms for algorithm creation (see chapter titles in *Anany Levitin. Introduction to the Design and Analysis of Algorithms 3rd Edition. Pearson, 2011.*).

[ProblemAnalysis] Given a real-world algorithmic problem, formalize it by listing your assumptions, breaking it into subproblems and suggesting their solutions in human language.

Please note the following:

- Questions related to algorithm implementations in C++ are not part of the exam. Reasoning at the pseudocode level, Python and hand-drawn memory states is typically sufficient.

- The latter two question types ((DesignByParadigm) and (ProblemAnalysis)) are more suitable for the Final exam as they take more time.

- A single question may contain related subtasks (A), (B), (C). If you do only some subtasks correctly, they will get positive grades even when they use incorrect input from earlier subtasks.

0.2 List of Topics

Part 1: Asymptotic Growth Rate

1. Analyze the growth rate of functions and sequences:

    A. Given an closed expression of a function, express its growth rate $\Theta(g(n))$ writing $g(n)$ in the simplest form possible. *(WS1.Problem1)*.

    B. Compare two or more functions in terms of their asymptotic growth rate. *(WS1.Problem5)*, *(WS1.Problem6)*, *(WS1.Problem7)*.

2. Analyze simple programs and pseudocode:

    A. Express time complexity for a code snippet from the inside out *(WS1.Problem2)*. *(WS1.Problem3)*.

    B. Apply assumptions on how fast are certain dependencies or built-in data structures (e.g. lists, sets, dictionaries in Python or other language) to analyze simple algorithms *(WS1.Problem8)*.

    C. Write a recurrence to express time complexity of an algorithm and solve it using the Masters theorem or other means *(WS2.Problem2)*, *(WS2.Problem3)*, *(WS2.Problem4)*, *(WS2.Problem7)*,

3. Analyze other complexity measures besides the worst-case time complexity:

    A. Evaluate the space complexity for an algorithm and its asymptotic growth rate.

    B. Evaluate the amortized time complexity, if some operation is applied many times. Evelute the expected time complexity, if inputs have known probability distribution *(WS3.Problem1)*, *(WS7.Problem4)*.

    C. Count the number of comparisons (sorting, searching or ranking algorithms), multiplications or other specific operations *(WS2.Problem6)*.

Part 2: Lists, Stacks, Queues

1. Typical implementations for Lists, Stacks, Queues:

    A. Given an implementation, draw the memory state at a certain moment, e.g. an array or a linked list *(WS3.Problem7)*, *(WS3.Problem8)*.

    B. Create a singly linked list implementation of some ADT method. *(WS3.Problem4)*.

    C. Create a doubly linked list implementation of some ADT method. *(WS3.Problem6)*.

2. Implement a data structure in pseudocode, compare the implementation alternatives:

    A. Express dependent ADT operations in terms of simpler ADT operations.

    B. Given a list/stack/queue algorithm pseudocode, find its time complexity. *(WS3.Problem5)*.

    C. Given a problem description and an ADT, implement some algorithm at ADT Level.

3. Write algorithms using Lists, Stacks or Queues. Algorithms can call list-like data structures using their ADT functions.

    A. Write algorithms and estimate the time complexity of algorithms processing expressions or using stack to navigate a tree-like structure *(WS5.Problem5)*, *(WS5.Problem6)*.

    B. Use stacks for any other purposes. *(WS3.Problem2)*, *(WS3.Problem3)*.

Part 3: Tree-like Structrues

1. Tree concepts.

    A. Use the concepts of non-rooted trees (plain graph level), rooted trees, ordered trees.

    B. Use the concepts of binary and n-ary trees. For binary trees distinguish full, complete and perfect trees.

    C. Use the concept of binary search tree (labels/keys compare according to the in-order traversal order).

    D. Encode multiway trees with binary trees (and binary trees into multiway trees) *(WS6.Problem1)*, *(WS6.Problem2)*.

2. Priority Queues and Heaps.

    A. Define the priority queue ADT, analyze various non-heap ways to implement it.

    B. Define a heap data structure, compute parents and children, peform insert and delete-min (or delete-max) *(WS5.Problem1)*, *(WS5.Problem2)*, *(WS5.Problem3)*, *(WS5.Problem4)*.

    C. Use priority queues to build Huffman prefix code given the alphabet of messages and their probabilities. *(WS5.Problem7)*.

3. Tree traversals and Backtracking.

    A. Use BFS traversal and DFS traversal with possible pre-order, in-order, post-order callbacks.

    B. Solve algorithmic tasks using backtracking *(WS3.Problem9)*.

Part 4: N-ary Search Trees

1. Regular BSTs

    A. Insert, delete and find keys in a binary search tree. *(WS6.Problem4)*.

    B. Answer the questions about their properties. *(WS6.Problem3)*.

    C. Perform various flavors of DFS traversals (in-order, pre-order, post-order), find in-order predecessors and successors.

    D. Reason about the expected height of a BST, if you insert keys in certain order. *(WS6.Problem8)*.

2. Self-balancing Search Trees.

    A. Draw AVL Trees, answer questions about their properties (worst-case depth etc.), insert and delete keys *(WS6.Problem5)*, *(WS6.Problem6)*. *(WS6.Problem7)*.

    B. Insert, delete and find keys in multiway search trees. In particular, draw 2-4 Trees, answer questions about their properties, insert and delete keys *(WS7.Problem1)*, *(WS7.Problem2)*, *(WS7.Problem3)*.

3. Create and Use Augmented Trees:

    A. Augment trees appropriately for the task where extra information for any node is computed from the node itself and its children.

    B. Computing RANK(v) – how many nodes w in the given tree satisfy the inequality $w.key \leq v.key$. *(WS7.Problem5)*.

    C. Computing COUNT(a, b) – how many keys are between a and b.

Part 5: Sorting

1. Time-complexity for sorting algorithms:

    A. Use Stirlings formula to evaluate factorials and binomial coefficients. *(WS8.Problem6)*.

    B. Count comparisons in a decision tree to find the lower bound of comparisons needed. *(WS8.Problem1)*

    C. Analyze some inefficient algorithms such as Bubblesort. *(WS8.Problem4)*

2. Various sorting algorithms:

    A. Use Mergesort, draw memory states, analyze complexity, count comparisons. *(WS8.Problem5)*.

    B. Use Heapsort, draw memory states, analyze complexity, count comparisons. *(WS8.Problem2)*.

    C. Use Quicksort, draw memory states, analyze complexity, count comparisons. *(WS8.Problem3)*.

Part 6: Graph algorithms

1. Run graph traversal algorithms:

    A. Run Breadth-first-search (BFS) on undirected and directed graphs, classify edges as forward edges, back edges or cross edges.

    B. Run Depth-first-search (DFS) on undirected and directed graphs, add start and finish timestamps, classify edges as forward edges, back edges, cross edges or forward edges.

    C. Topologically sort vertices in a directed graph or establish that it is impossible.

    D. Find strongly connected components using Kosarajus algorithm.

2. Run graph optimization algorithms:

    A. Run single-source shortest paths algorithms such as Dijkstras and Bellman-Ford.

    B. Run all-pairs shortest paths algorithms such as Floyd–Warshall.

    C. Run MST algorithms such as Prims and Kruskals.

3. Run flow-related algorithms:

    A. Reason with augmenting paths regarding maximum flow or maximum matching problems.

    B. Run maximum flow algorithms such as Ford-Fulkerson or Edmonds–Karp.

    C. Run maximum matching algorithms such as Hopcroft-Karp algorithm.

Part 7: Sets, dictionaries and hashing

1. Use hashing data structure:

    A. Describe and compute some typical implementations for hashing functions based on modular arithmetic.

    B. Resolve hash collisions by chaining and analyze the expected time complexity for such hashtables.

    C. Resolve hash collisions using various open addressing methods – linear probing, quadratic probing or double hashing.

2. Implement and use sets, multisets or maps:

    A. Compare hashing-based vs. tree-based implementations of sets and maps.

    B. Describe polynomial-based rolling hash algorithm, Rabin-Karp string search algorithm and its uses in checking plagiarism.

    C. Use and reason about secure hashing algorithms (such as SHA-256 or MD5), how they are used in password caching or communication algorithms.

Data Structures: Exam Topics

WORKSHEET 01: ASYMPTOTIC BOUNDS

Why study the asymptotic bounds or the Big-O notation? Efficiency of algorithms is largely determined by their behavior for large inputs. It is not easy to describe the speed of an algorithm for every single input, so it is described by some smooth function $f(n)$ – an estimate from above of how fast the algorithm is for inputs of length n.

1.1 Concepts and Facts

Informally, *asymptotic* means the behavior as some parameter goes to infinity; upper and lower *bounds* are inequalities satisfied by something.

Definition: The *runtime upper bound* (also called the *worst-case running time*) for the given algorithm is a function $T : \mathbf{N} \mapsto \mathbf{N}$ that equals to the maximum possible number of elementary steps needed to complete the algorithm for any input of length n.

(Here $\mathbf{N}$ is the set of all nonnegative integers.)

Introductory Questions:

- What is the worst running time to multiply two square matrices of size $n \times n$? What is the size of input in this case?

- Let $T(n)$ be a numeric algorithm receiving single natural numbers as input. Does the runtime upper bound function change, if the input numbers are provided in binary (instead of decimal) notation?

- Is $T(n)$ a non-decreasing function (i.e. do longer inputs always imply a longer running time)?

- What counts as an elementary step in the runtime upper bound definition? (Any CPU instruction? One line in a pseudocode? One comparison in a sorting algorithm?)

Definition (Big-O): Let $g \colon \mathbb{N} \to \mathbb{R}_{0+}$ be a function from natural numbers (non-negative integers) to non-negative real numbers. Then $O(g)$ is the set of all functions $f \colon \mathbb{N} \to \mathbb{R}^{0+}$ such that there exist real constants $c > 0$ and $n_0 \in \mathbb{N}$ satisfying

$$0 \leq f(n) \leq c \cdot g(n) \ \text{ for all } \ n \geq n_0.$$

Definition (Big-Omega): Let $g \colon \mathbb{N} \to \mathbb{R}_{0+}$ be a function from natural numbers to non-negative real numbers. Then $\Omega(g(n))$ is the set of all functions $f \colon \mathbb{N} \to \mathbb{R}$ such that there exist real constants $c > 0$ and $n_0 \in \mathbb{N}$ satisfying $\forall n \in \mathbb{N} \ \big(n \geq n_0 \to f(n) \geq c \cdot g(n)\big)$.

Definition (Big-Theta): Let $g \colon \mathbb{N} \to \mathbb{R}_{0+}$ be a function from natural numbers to non-negative real numbers. Then $\Theta(g)$ is the set of all functions $f \colon \mathbb{N} \to \mathbb{R}$ such that there exist positive constants $c_1, c_2 > 0$ and $n_0 \in \mathbb{N}$ satisfying

$$\forall n \in \mathbb{N} \ \big(n \geq n_0 \to 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\big).$$

Definition (Asymptotic bounds):

- If $f(n) \in O(g(n))$, $g(n)$ is also called *asymptotic upper bound* of $f(n)$.

- If $f(n) \in \Omega(g(n))$, then $g(n)$ is called *asymptotic lower bound* of $f(n)$.

- If $f(n) \in \Theta(g(n))$, then $g(n)$ is called *asymptotic growth order* of $f(n)$.

Some bounds can be established without using the definitions of the Big-O, Big-Omega, and Big-Theta concepts directly.

Properties of the asymptotic bounds:

Dominant term: If $f(n) = f_1(n) + f_2(n)$, where $f_2(n) < f_1(n)$ for all sufficiently large n, then $O(f(n))$ and $O(f_1(n))$ are the same.

Consequently, if $f(n) = f_1(n) + f_2(n) + \ldots + f_k(n)$ can be written as a finite sum of other functions, then the fastest growing one determines the asymptotic growth order of the entire sum $f(n)$

Additivity: If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $g_2(n)$, then $f_1(n) + f_2(n)$ is in $O(\max(g_1(n), g_2(n)))$. Typically, one of the $g_1(n)$ or $g_2(n)$ is asymptotically larger than the other (say, $g_1(n) > g_2(n)$ for all sufficiently large n), and instead of $O(\max(g_1(n), g_2(n)))$ we can simply take $O(g_1(n))$.

Multiplicativity: If $f(n)$ is in $O(g(n))$ and $c > 0$ is a positive constant, then $c \cdot f(n)$ is also in $O(g(n))$.

Transitivity: If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is also in $O(h(n))$.

Polynomial Dominance: For any positive integer k, n^k is dominated by cn^k for all $n \geq n0$ and some constant c.

Big-O and the Limit of the Ratio:

If the following limit exists and is finite:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = C < +\infty,$$

then $f(n)$ is in $O(g(n))$.

Theorem (Changing the Base of Logarithm): If $a, b > 1$ are real numbers, then $\log_a n$ is in $\Theta(log_b n)$.

Proof: The last result directly follows from the formula to change the base of a logarithm:

$$\forall a, b, m > 1 \left(\log_a b = \frac{\log_m b}{\log_m a} \right).$$

Note: It is common to just one base (usually, it is base 2 or base e of the natural logarithm), and write just $O(\log n)$ without specifying base at all. Formally, $\log n$ in our course denotes $\log_2 n$.

In other contexts (where constant factors matter) the base of logarithm cannot be omitted.

1.2 Problems

Problem 1: Are the following statement true or false? Prove or disprove them using the definitions of $O(g(n))$, $\Omega(g(n))$ or $\Theta(g(n))$:

(A) $f(n) = 13n + 7$ is in $\Theta(n)$.

(B) $f(n) = 3n^2 - 100n + 6$ is in $O(n^2)$. (Verify the definition that $f(n) \in O(g(n))$, where $g(n) = n^2$.)

(C) $f(n) = 3n^2 + 100n + 6000$ is in $O(n^2)$.

(D) $f(n) = 3n^2 - 100n + 6$ is in $O(n\sqrt{n})$.

Answer:

(A) True. Select $c_1 = 13$, $c_2 = 14$, $n_0 = 7$. We should verify

$$0 \le c_1 \cdot n \le f(n) \le c_2 \cdot n.$$

Indeed, $13n \le 13n + 7$, since $0 \le 7$.

Also, $13n + 7 \le 14n$, since $7 \le n$ (whenever $n \ge n_0 = 7$).

(B) True. Select $c = 3$ and $n_0 = 34$. Let us verify the inequalities:

$$0 \le 3n^2 - 100n + 6 \le 3n^2.$$

For $n \ge 34$, then

$$3n^2 - 100n + 6 \ge 3 \cdot 34 \cdot n - 100n + 6 = 2n + 6 \ge 0.$$

So, the expression is non-negative for sufficiently large n ($n \ge 34$).

The other inequality $3n^2 - 100n + 6 \le 3n^2$, since $-100n + 6 \le 0$ and $6 \le 100n$.

(C) True. Select $c = 5$ and $n_0 = 100$. At this point (when $n \ge 100$) you can prove that $100n \le n^2$ and $6000 \le n^2$ and also $3n^2 \le 3n^2$. Add all three inequalities to get $3n^2 + 100n + 6000 \le 5n^2$.

(D) False. To disprove that $f(n) = 3n^2 - 100n + 6$ is not in $O(g(n))$ where $g(n) = n\sqrt{n}$, we show how to find an example value $n \ge n_0$ such that $3n^2 - 100n + 6 > c \cdot n\sqrt{n}$ for any positive constants n_0, c.

First, notice that $3n^2 - 100n + 6$ is in $\Omega(n^2)$, for example $3n^2 - 100n + 6 > n^2$ whenever $n \ge 34$.

Let us consider some positive constant c. Then $n^2 \ge cn\sqrt{n}$ can be rewritten as $\sqrt{n} \ge c$ or $n \ge c^2$. Once you consider natural numbers $n \ge \max(34, c^2)$, you will have **both** inequalities $3n^2 - 100n + 6 \ge n^2$ and $n^2 > cn\sqrt{n}$; combining them gives $3n^2 - 100n + 6 > cn\sqrt{n}$.

Problem 2: Let us have a zero-based dictionary D with n items from $D[0]$ to $D[n-1]$.

LINEARSEARCH(D, w)
1.   **for** i **in** RANGE$(0, n)$:
2.   **if** $w == D[i]$:
3.   **return** FOUND w at location i
4.   **return** NOT FOUND

Let $T(n)$ be the worst-case running time for this algorithm. Find some asymptotic upper bound for $T(n)$ – the smallest set $O(g(n))$ such that $T(n)$ is in $O(g(n))$.

Answer:

You can pick $g(n) = n$ and argue that $T(n)$ is in $O(n)$, i.e. the search time is linear in size of the array n.

If you can assume that the dictionary D contains entries in a sorted order, you could use binary search instead. It has much better runtime complexity: $T(n) = \log_2 n$.

Problem 3: What is the worst running time to find, if the given input m is a prime number. Assume that the input m is written in decimal notation using n digits.

Primality testing is done by the following algorithm testing divisibility by all numbers $d \in \{2, 3, \ldots, \lfloor \sqrt{m} \rfloor\}$:

ISPRIME(m)

1. **for** d **in** RANGE($2, \sqrt{m}+1$):
2.   **if** m % $d == 0$:
3.   **return** FALSE
4. **return** TRUE

Answer:

Consider the number m containing, say, $n = 100$ digits. In this case $m \leq 10^{100}$ and $\sqrt{m} \leq 10^{50}$. To check, if such number is a prime number (in the worst case), we need to make about 10^{50} operations.

In general, the time complexity of this algorithm is $O\left(10^{\frac{n}{2}}\right)$. We see that the algorithm is extremely inefficient for (moderately) long inputs. On the other hand, checking primality of 100-digit numbers can be done very fast using an efficient, but probabilistic algorithm by Rabin-Miller. Here is Python code to find the largest 100-digit prime number:

```python
import sympy

for i in range(1, 1000):
    if sympy.isprime(10**100 - i):
        print('Prime number 10**100-{}'.format(i))
```

Output looks like this:

```
Prime number 10**100-797
Prime number 10**100-911
```

Problem 4: Answer the following Yes/No questions:

(A) For any $g(n)$, is the set of functions $\Theta(g(n))$ the intersection of $O(g(n))$ and $\Omega(g(n))$?

(B) Does every function $f(n)$ defined for all natural numbers and taking positive values belong to the set $Omega(1)$?

(C) Let $f(n), g(n)$ be two functions from natural numbers to non-negative real numbers. Is it true that we have either $f(n)$ in $O(g(n))$ or $g(n)$ in $f(n)$ (or both)?

(D) Does the definition of $f(n)$ in $O(g(n))$ make sense, if $f(n)$ and $g(n)$ can take negative values?

(E) Are these two sets of functions $O(\log_2 n)$ and $O(\log_{10} n)$ the same? If not, find which one is larger (contains more functions)?

(F) Let $f(n)$ be a function from natural numbers to non-negative real numbers. Do we always have that $f(n)$ is in $O(f(n))$, and $f(n)$ is in $\Omega(f(n))$ and $f(n)$ is in $\Theta(f(n))$? (In other words, is being in Big-O, in Big-Omega and in Big-Theta a reflexive relation?)

(G) Let $f(n), g(n), h(n)$ be functions from natural numbers to non-negative real numbers. It is known that $f(n)$ is in $O(g(n))$ and also $g(n)$ is in $h(n)$. Can we always imply that $f(n)$ is in $O(h(n))$. (In other words, is being in Big-O, in Big-Omega and in Big-Theta a transitive relation?)

(H) Let $f(n), g(n)$ be functions from natural numbers to non-negative real numbers. It is known that $f(n)$ is in $\Theta(g(n))$. Can we always imply that $g(n)$ is in $\Theta(f(n))$? (In other words, is being in Big-Theta an equivalence relation?)

(I) A function $f(n)$ is defined for natural arguments and takes natural values. It is known that $f(n)$ is in $O(1)$. Is it true that $f(n)$ is a constant function: $f(n) = C$ for all $n \in \mathbf{N}$.

Answer:

(A) True. We need to prove in two directions.

(1) If $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, then also $f(n) \in \Theta(g(n))$.

Indeed, if we have $f(n)$ bound from above by $c_1 \cdot g(n)$ for all $n \geq n_1$, and also $f(n)$ bound from below by $c_2 \cdot g(n)$ for all $n \geq n_2$, then we also have $c_1 g(n) \leq f(n) \leq c_2 g(n)$ as soon as $n \geq \max(n_1, n_2)$.

This means that for all sufficiently large n all the values $f(n)$ will be bound from both sides which is same as $f(n) \in \Theta(g(n))$.

(2) If $f(n) \in \Theta(g(n))$ then both $f(n) \in (g(n))$ and $f(n) \in \Omega(g(n))$ must hold.

Indeed, if there are constants c_1, c_2 such that for all $n \geq n_0$ we have $c_1 g(n) \leq f(n) \leq c_2 g(n)$, then the function $f(n)$ is bound from above and from below. I.e. it belongs to $O(g(n))$ and $\Omega(g(n))$ where the same constants can be used.

(B) False. You could take any function $f(n) = \frac{1}{n}$. In this case all the values are non-negative, but there does not exist a positive constant c such that all the values $\frac{1}{n} \geq c \cdot 1$ (in fact these values converge to 0).

Class $\Omega(1)$ includes only those functions that have a positive (or infinite) lower limit. See Lower Limit.

(C) False. Most functions encountered in algorithm analysis are comparable, using Big-O notation: namely, either $g(n)$ bounds $f(n)$ from above (for all sufficiently large n) or vice versa.

But it is not difficult to build functions that are not comparable. For example,

$$\left\{ \begin{array}{l} f(n) = n \\ g(n) = n^{(}1 + \sin n) \end{array} \right\}$$

Here the function $g(n)$ is not monotonous (it takes random values between n^0 and n^2). It is also possible to create two functions that are monotonous (non-decreasing) and still incomparable so that $f(n) \notin O(g(n))$ and also $g(n) \notin O(f(n))$.

(D) True. All the definitions still make sense, if functions $f(n)$ and $g(n)$ can take negative values. The important requirement is that functions $f(n)$ and $g(n)$ be *asymptotically non-negative* – i.e. they only take finitely many negative values. In this case n_0 can be selected sufficiently large, so that $0 \geq f(n)$ and $0 < g(n)$ whenever $n > n_0$. (Which means that negative values $f(n) < 0$ etc. can be simply ignored as the arguments n are not sufficiently large.)

Problem 5: Order these functions in increasing order regarding Big-O complexity (f_i is considered not larger than f_j iff $f_i \in O(f_j)$).

- $f_1(n) = n^{0.9999} \log_2 n$
- $f_2(n) = 10000n$
- $f_3(n) = 1.0001^n$
- $f_4(n) = n^2$

Problem 6: Order these functions in increasing order regarding Big-O complexity:

- $f_1(n) = 2^{2^{10000}}$
- $f_2(n) = 2^{10000n}$
- $f_3(n) = \binom{n}{2} = C_n^2$
- $f_4(n) = \binom{n}{\lfloor n/2 \rfloor}$
- $f_5(n) = \binom{n}{n-2}$
- $f_6(n) = n!$
- $f_7(n) = n\sqrt{n}$

Problem 7: Order these functions in increasing order regarding Big-O complexity:

- $f_1(n) = n^{\sqrt{n}}$
- $f_2(n) = 2^n$
- $f_3(n) = n^{10} \cdot 2^{n/2}$
- $\sum_{i=1}^{n}(i+1).$

Problem 8: A black box $\mathcal{B}$ receives two numbers $k_1, k_2 \in \{1, \ldots, n\}$ as inputs and returns a value $v = \mathcal{B}(k_1, k_2)$ after $O(1)$ time. What is the worst-case time complexity to find the maximum possible value $v = \mathcal{B}(k_1, k_2)$ for any two inputs.

What if the black box receives permutations of n elements as its inputs?

WORKSHEET 02: RECURSION

There is a proverb: *To understand recursion, you must first understand recursion.* Recursion allows to analyze algorithms, to find their runtime even in complex cases.

2.1 Concepts and Facts

Definition: A Fibonacci sequence is defined by the following recurrent formula:

$$F(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

Statement: For every nonnegative integer n the Fibonacci number $F(n)$ can be computed by the following expression:

$$F(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right).$$

Such expressions are named *closed formulas*, since they can be evaluated directly, without repeating recurrent formula many times. See its proof – <https://brilliant.org/wiki/linear-recurrence-relations/>`_. The expression shows that (apart from an expression $((1 - \sqrt{5})/2)^n$ that goes to 0) Fibonacci numbers grow as a geometric series.

Definition: A factorial for any non-negative integer is defined by the following recurrent formula:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

Statement: Factorials for large n satisfy Stirlings approximation:

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e} \right)^n.$$

Note: The asymptotic equality $f(n) \sim g(n)$ for two positive functions $f(n), g(n)$ denotes that the ratio $f(n)/g(n)$ has limit 1 as $n \to \infty$. Such relation also would imply that $f(n)$ and $g(n)$ are in Big-Theta of each other. In fact, $f(n) \sim g(n)$ is stronger than Big-Theta, since $f(n) \in \Theta(g(n))$ does allow any finite and bounded ratios $f(n)/g(n)$.

Definition: Function code that invokes itself (with different arguments to avoid infinite loops) is called a *function implemented with recursion*.

Example: This is a straightforward recursive implementation of factorial:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

For large `n` this implementation is problematic, since it keeps too many function calls on the activation stack. Therefore in practice factorial is usually implemented without recursion (one can multiply the numbers in a loop), or using a tail recursion.

Definition: Function code that contain recursive call in just one place and the recursive call is the entire return expression of the function is called *function implemented with tail-recursion.*

Example: Here is tail-recursive factorial:

```
def factorial_tail(n, result):
    if n == 0:
        return result
    else:
        return factorial_tail(n - 1, n * result)
```

Algorithms using divide-and-conquer paradigm (such as MergeSort, Quicksort) may call themselves multiple times. In these cases we must be very careful regarding the depth of recursion tree as the number of function calls grows exponentially. Here is an extremely inefficient algorithm to compute Fibonacci numbers: any call to `fibonacci_bad(n)` (where $n \geq 2$) leads to two more calls. Such implementations are called *excessive recursion.*

Example:

```
def fibonacci_bad(n):
    if n <= 1:
        return n
    else:
        return fibonacci_bad(n - 1) + fibonacci_bad(n - 2)
```

Master Theorem: Let $f(n)$ be an increasing function that satisfies the recurrence relation:

$$f(n) = a \cdot f\left(\frac{n}{b}\right) + cn^d$$

Here we assume that $n = b^k$, where k is a positive integer, $a \geq 1$, $b > 1$ is an integer, c, d are real numbers (where $c > 0$ and $d \geq 0$). Then the asymptotic growth for $f(n)$ can be found like this:

$$f(n) \text{ is in } \begin{cases} O(n^d), & \text{if } a < b^d, \\ O(n^d \log n), & \text{if } a = b^d, \\ O(n^{\log_b a}), & \text{if } a > b^d. \end{cases}$$

This theorem can be used to find the asymptotic runtime for recursive algorithms.

Example (Binary Search): Binary search searches items in a sorted array of n elements: $A[0] < A[1] < \ldots < A[n-2] < A[n-1]$. At every point it maintains a search interval $[\ell, r]$ so that the searchable item w satisfies inequalities $D[\ell] < w < D[r]$. The initial call is BINARYSEARCH$(A, 0, n-1, w)$. After that the binary search calls itself recursively on shorter intervals:

BINARYSEARCH(D, ℓ, r, w)
1.   **if** $\ell > r$:
2.   **return** NOT FOUND w

---

3. $m = \lfloor (\ell + r)/2 \rfloor$

4.   **if** $w == D[m]$:

5.   **return** FOUND w at location m

6.   **else if** $w < D[m]$:

7.   **return** BINARYSEARCH$(D, \ell, m - 1, w)$

8.   **else**:

9.   **return** BINARYSEARCH$(D, m + 1, r, w)$

Denote the runtime of this algorithm on an array of length n by $T(n)$. Denote by a constant K the upper bound of the time necessary to compute the middlepoint m on Line 3 and to do all the comparisons. Use the Masters theorem to find the time complexity for $T(n)$.

Answer:

The runtime $T(n)$ satisfies the following recurrence:

$$T(n) = \begin{cases} K, & \text{if } n = 1, \\ K + T(\lfloor n/2 \rfloor), & \text{if } n > 1. \end{cases}$$

In the Masters theorem $a = 1$, $b = 2$, $c = K$, $d = 0$. Since $a = b^d$, we have $T(n) \in O(n^d \log n) = O(\log n)$. We conclude that the BINARYSEARCH$(\ldots)$ algorithm runs in logarithmic time $O(\log n)$, where n is the length of the array.

Example (Hanoi Tower): You need to move a set of disks (enumerated $1, 2, \ldots, n$ from smallest to largest) from one peg to another, one disk at a time, while obeying the rule that a larger disk cannot be placed on top of a smaller disk. You have altogether three pegs: `from_peg` is the peg, where all the disks are placed originally (smallest disk 1 at the top); `to_peg` is the peg, where these disks must end up at the very end. And there is also `aux_peg` – auxiliary peg that can be used during the movements, but should be freed at the end.

```python
def tower_of_hanoi(n, from_peg, to_peg, aux_peg):
    if n == 1:
        print("Move disk 1 from peg {} to peg {}".format(from_peg, to_peg))
        return

    tower_of_hanoi(n-1, from_peg, aux_peg, to_peg)
    print("Move disk {} from peg {} to peg {}".format(n, from_peg, to_peg))
    tower_of_hanoi(n-1, aux_peg, to_peg, from_peg)
```

The input of this algorithm is n (the number of disks), its output is a valid schedule describing valid movements of the disks. Let $H(n)$ denote the running time of this algorithm expressed as the number of `print()` statements. Express $H(n)$ (the number of disk movements in the algorithm) in terms of previous values $H(m)$, where $m < n$. Solve the recursion and find a closed formula for $H(n)$.

Answer:

The runtime $H(n)$ satisfies the following recurrence:

$$H(n) = \begin{cases} 1, & \text{if } n = 1, \\ 1 + 2 \cdot H(n - 1), & \text{if } n > 1. \end{cases}$$

In this case Masters theorem cannot be applied, since $H(n)$ is expressed via $H(n-1)$ rather than in terms of $H(n/b)$ for some constant b. Fortunately, $H(n)$ can be evaluated directly.

Observe that $H(1) = 1$, $H(1) = 1 + 2 \cdot H(1) = 3$, $H(3) = 1 + 2 \cdot H(2) = 7$, and so on. We observe that $H(n) = 2^n - 1$. This can be proven by induction.

Base: $n = 1$. In this case $H(1) = 1$, and also $H(1) = 2^1 - 1$; so the formula $H(n) = 2^n - 1$ is true in this case.

Inductive Hypothesis: Assume that for $n = k$ disks the number of print statements is indeed $H(k) = 2^k - 1$.

Induction Step: We now prove that the statement is also true for $n = k + 1$. In this case $H(k + 1) = 1 + 2 \cdot H(k)$ by the given recurrent formula. On the other hand, by inductive hypothesis,

$$H(k + 1) = 1 + 2 \cdot (2^k - 1) = 1 + 2 \cdot 2^k - 2 = 2^{k+1} + 1 - 2 = 2^{k+1} - 1.$$

We now see that $H(k + 1) = 2^{k+1} - 1$, which means that the formula $H(n) = 2^n - 1$ is also true for $n = k + 1$. Inductive step is completed.

Example (Karatsuba Multiplication Algorithm): Given two non-negative integer numbers of the same length n (written in binary), write an algorithm to multiply these numbers. Consider an algorithm that is faster than the school algorithm (it would multiply two numbers of length n in $O(n^2)$ time):

KARATSUBA(n_1, n_2)
1.   if $(n_1 < 10)$ **or** $(n_2 < 10)$:
2.   **return** $n_1 \cdot n_2$ (*fall back to traditional multiplication*)
3. $m = \max(\text{SIZE}(n_1), \text{SIZE}(n_2))$
4. $m_2 = \lfloor m/2 \rfloor$
5. $h_1, \ell_1 = \text{SPLITAT}(n_1, m_2)$
6. $h_2, \ell_2 = \text{SPLITAT}(n_2, m_2)$
       (*Three recursive calls of Karatsubas algorithm.*)
7. $z_0 = \text{KARATSUBA}(\ell_1, \ell_2)$
8. $z_1 = \text{KARATSUBA}(\ell_1 + h_1, \ell_2 + h_2)$
9. $z_2 = \text{KARATSUBA}(h_1, h_2)$
12. $y = z_1 - z_2 - z_0$
13.   **return** $(z_2 \cdot 10^{2 \cdot m_2}) + (y \cdot 10^{m_2}) + z_0$

By $T(n)$ denote the runtime of Karatsubas algorithm on two numbers having length n each. (Assume that non-recursive parts take some constant time K.) Provide the asymptotic bound extimate for $K(n)$.

Note: We typically assume that addition and multiplication take $\Theta(1)$ time as they are CPU operations. But multiplication of very long numbers cannot be done in constant time. Instead assume that operations on individual bits are done in constant time. Things like Boolean operations, bit arithmetic, checking conditional statements.

Answer:

In this algorithm one call causes three recursive calls; each call has arguments that are half the size. It means that $T(n) = 3 \cdot T(n/2) + K$.

In Masters theorem we would have $a = 3$, $b = 2$, $c = K$, $d = 0$. In this case $a > b^d$, so $T(n) \in O\left(n^{\log_b a}\right) = O\left(n^{\log_2 3}\right)$. So the time complexity of this is $O(n^{1.58})$ which is significantly better than $O(n^2)$.

2.2 Problems

Problem 1: Answer the following questions regarding the asymptotic behavior of functions.

    (A) Have students generate 10 functions and order them based on asymptotic growth.

    (B) Find a tight asymptotic bound for $\binom{n^2}{3168}$, and write it using the simplest notation possible.

    (C) Find a simple, tight asymptotic bound for $f(n) = \log_2\left(\sqrt{n}^{\sqrt{n}}\right) - \log_{10}\left(\sqrt[3]{n}^{\sqrt[3]{n}}\right)$.

    (D) Is 2^n in $\Theta\left(3^n\right)$? Is $2^{2^{n+1}}$ in $\Theta\left(2^{2^n}\right)$?

    (E) Show that $(\log n)^a$ is in $O(n^b)$ for all positive constants a and b.

    (F) Let $f(n) = (\log_2 n)^{\sqrt{n}}$ and $g(n) = (\log_{10} n)^{\sqrt{n}}$. Is $f(n)$ in $\Theta(g(n))$?

    (G) Show that $(\log n)^{\log n}$ is in $\Omega(n)$.

    (H) Is $(2n)!$ in $O(n!)$? Is $\sqrt{(2n)!}$ in $O(\sqrt{n!})$? Is $\sqrt{\log_2((2n)!)}$ in $\sqrt{\log_2(n!)}$

Problem 2: Consider Euclid algorithm to find the greatest common divisor (written around 300 B.C. in *Elements*):

EUCLIDGCD(a, b)
1.   **if** $b == 0$:
2.   **return** a
3.   **else**:
4.   **return** EUCLIDGCD$(b, a \bmod b)$

It is known that for a given input length n the worst-case running time is to run the algorithm on subsequent Fibonacci numbers: F_m and F_{m-1}, where F_m is the largest Fibonacci number of length not exceeding n.

Write a precise estimate (without using unknown constant factors as in Big-O notation) on how many calls of EUCLIDGCD(a, b) are needed, if both inputs have length not exceeding n.

Note: Imagine that both arguments to the Euclid algorithm are two natural numbers a, b containing up to 100 digits each. Estimate the maximum number of recursive calls until the grater common divisor is found.

Problem 3: Given a sequence a_i $(i = 0, \ldots, n-1)$ we call its element a_i a *peak* iff it is a local maximum (at least as big as any of its neighbors):

$$a_i \geq a_{i-1} \quad \text{and} \quad a_i \geq a_{i+1}$$

(In case if $i = 0$ or $i = n - 1$, one of these neighbors does not exist; and in such cases we only compare a_i with neighbors that do exist.)

    (A) Suggest an algorithm to find some peak in the given array $A[0], \ldots, A[n-1]$ and find its worst-case running time.

    (B) Suggest an algorithm that is faster than linear time to find peaks in an array. Namely, its worst-case running time should satisfy the limit:

$$\lim_{n \to \infty} \frac{T(n)}{n} = 0.$$

Question 4: Select the correct asymptotic complexity of an algorithm with runtime $T(n, n)$ where

$$\begin{cases} T(x, c) = \Theta(x) \text{ for } c \le 2, \\ T(c, y) = \Theta(y) \text{ for } c \le 2, \text{ and,} \\ T(x, y) = \Theta(x + y) + T(\lfloor x/2 \rfloor, \lfloor y/2 \rfloor) \text{ otherwise.} \end{cases}$$

    a. $\Theta(\log n)$.

    b. $\Theta(n)$.

    c. $\Theta(n \log n)$.

    d. $\Theta(n log^2 n)$.

    e. $\Theta(n^2)$.

    f. $\Theta(2^n)$.

Question 5: Just like the tail-recursive factorial, write a tail-recursive Fibonacci program. This way you will also avoid excessive recursion – exponential increase of the number of recursive calls.

To achive this, you may need to pass multiple parameters in the recursive call to the recursive Fibonacci function.

Answer:

```python
# Tail Recursive function to calculate nth Fibonacci number
def fibonacci_tail(n, a, b) -> int:
    if n == 0:
        return a
    else:
        return fibonacci_tail_recursive(n - 1, b, a + b)

# Shows how to initialize the function's fibonacci_tail(...) arguments:
def fibonacci_tail(n: int) -> int:
    return fibonacci_tail_recursive(n, 0, 1)
```

Question 6: It is known that Taylor series for $y = \sin x$) is given by formula:

$$\lim_{n \to \infty} S(x, n) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots = \sin x.$$

The series converges for every $x \in \mathbb{R}$. Write a tail-recursive function that for any argument x computes the approximation for $\sin x$ by adding up the first 50 terms of the Taylor series. The use of global variables is not allowed – all data manipulation should be done with local variables function calls and their return values. Your solution should use as few multiplications and divisions as possible.

Answer:

As the global variables are not allowed, we can accumulate the partial sum as one of the arguments passed to the recursive method calls.

$$S(n, x) = \begin{cases} x & \text{if } n = 0, \\ S(n - 1, x) + \frac{(-1)^n \cdot x^{2n-1}}{(2n-1)!} & \text{if } n > 0. \end{cases}$$

For a constant x build the sequence $S(0, x), S(1, x), S(2, x), \ldots$ using the recursive calls. You can write a recursive function in pseudocode that computes $S(50, x)$.

This answer is incomplete as the number of multiplications and divisions is not minimized. In fact, computing x^{2n-1} would require many multiplications; same as $(2n - 1)!$. So you can try to optimize this further.

Problem 7: Koch snowflake consists of three sides. Each side connects two vertices of an equilateral triangle ABC. Consider, for example, two points A and B and the edge connecting them e.

If the length of e is 1 unit or shorter, then AB is connected by a straight line segment. Otherwise, the segment AB is subdivided into three equal parts: e_1, e_2, e_3. The middle part is complemented with two more line segments f_1 and g_1 to make another equilateral triangle (with side length three times smaller than the ABC). Finally, the Koch snoflakes edge algorithm is called on each of the segments e_1, f_2, g_2, e_3 recursively.

The initial call for $e = AB$ is SNOWFLAKEEDGE$(e, 0)$, where $d = 0$ is the initial depth in the recursion tree. Here is the pseudocode for the algorithm:

SNOWFLAKEEDGE(e, d):
    if $|e| \leq 1$:
        draw a straight edge e
    else:
        Split e into three equal parts e_1, e_2, e_3
        Construct a regular triangle out of edges e_2, f_2, g_2 to the outside
        SNOWFLAKEEDGE$(e_1, d + 1)$
        SNOWFLAKEEDGE$(f_2, d + 1)$
        SNOWFLAKEEDGE$(g_2, d + 1)$
        SNOWFLAKEEDGE$(e_3, d + 1)$

In this algorithm we assume that the Koch snowflake is drawn as vector graphics on a device with infinite resolution.

(A) How many levels does the depth parameter d reach, if the initial size of the edge is $|e| = n$.

(B) Estimate the number of recursive calls of SNOWFLAKEEDGE(e, d), if the initial size of the edge is $|e| = n$.

(C) Write a recursive time complexity of this algorithm $T(n)$ and estimate it with Masters theorem.

Data Structures: Exam Topics

WORKSHEET 03: ADTS AND IMPLEMENTATIONS

An abstract data type (ADTs) provide standard names for operations to interact with a certain data structure. ADTs should be distinguished from the implementation of the data structure – the same ADT operations can be supported in multiple ways with programming constructs like arrays, pointers, custom objects etc. Abstract Data Types (just as any other API interfaces) let us ignore implementation details when using them in other algorithms.

3.1 Concepts and Facts

Definition: A data container C that stores items imposes *extrinsic order* iff this container ensures predictable order of how they can be visited or retrieved. Namely, for any two elements $a, b \in C$, either that *a preceeds b* or *a succeeds b* in this container.

Example: Items stored in a linked list or in an array can be visited in a predictable order such as $A[0], A[1], \ldots$. This extrinsic order does not need to be related with possible ordering of items themselves (items in an array are not always stored in increasing or decreasing order).

Definition: Items a, b stored in a data container have *intrinsic order* iff they can be compared by themselves. Such as $a = b$, $a < b$, or $a > b$. (Some data structures such as Binary Search Trees (BSTs) store keys according to their intrinsic order).

There may be data structures (such as sets stored in a hashtable) that do not use either intrinsic or extrinsic order. We can iterate over such containers, but will get the elements in some unpredictable order.

Definition: A data structure is *immutable*, if it cannot change its state after its creation Other data structures that can be changed are called *mutable*.

Example: A `string` data structure is immutable in many programming languages – a string object stays unchanged throughout its lifetime. Appending new characters or modifying the string is possible, but the result is always a new string object.

Immutable data structures can be easily passed as parameters to functions (just by copying its address or reference, but without cloning the data). Immutable data structures also can serve as keys in larger data structures.

Definition: Some property for a mutable data structure is a *representation invariant*, if it is necessary for any consistent state of this data structure and it is preserved during the ADT operations. Namely, if the property was true before the operation (insertion, deletion etc.), then it must be also true after that operation. It is thus *unchangeable* or *invariant*.

Violating the representation invariant makes the data structure inconsistent and unusable. If an API method is run on an inconsistent data structure, there are no expectations (program can crash, loop forever or produce any result).

Example: Some mutable data structures (such as `java.util.HashSet`) allow to insert other mutable objects such as `ArrayList` objects into the hashtable. If some object is modified after being inserted, it is not rehashed and its location becomes invalid. At that time the representation invariant of the hashtable is broken – the object cannot be located in its appropriate hashing bucket anymore.

Definition A *stack* is a data structure with extrinsic order – it allows to access (to read or to delete) the element which was the latest to be inserted (LIFO policy). A stack supports the following operations:

S.INITIALIZE()   *(create a new empty stack or clear existing contents)*
S.PUSH(x)   *(add item x to the top of the stack)*
S.POP()   *(remove and return an item from the top of the stack)*
S.ISEMPTY()   *(true iff the stack is empty)*

Optionally stacks can support some non-essential operations (can be expressed with the above operations considered essential):

S.TOP()   *(return, but do not remove the top element)*
S.SIZE()   *(return the number of elements in the stack)*

In practice stacks may also need S.ISFULL() operation to find, if the container has place for one more item. Abstract stacks may assume that the stack is never full.

Definition A *queue* is a data structure with extrinsic order – it allows to access (seeing or deleting) the single element which was the first to be inserted (FIFO policy). A queue supports the following operations:

Q.INITIALIZE()   *(create a new empty queue or clear existing contents)*
Q.ENQUEUE(x)   *(add item x to the end of the queue)*
Q.DEQUEUE()   *(remove and return an item from the front of the queue)*
Q.ISEMPTY()   *(true iff the queue is empty)*

Optionally queues can support other operations, but many queue applications do not need them:

Q.FRONT()   *(return the front element in the queue without dequeuing)*
S.SIZE()   *(return the number of elements in the queue)*

Definition: A *double-ended queue* or a *deque* is a data structure with extrinsic order allowing elements to be added to the front (as in a stack) and also to the back (as in a queue). It supports the following operations:

D.INITIALIZE()   *(create a new empty deque or clear existing contents)*
D.PUSHFRONT(x)   *(add item x to the front of the deque)*
D.POPFRONT()   *(remove and return an item from the front of the deque)*
D.PUSHBACK(x)   *(add item x to the back of the deque)*
D.POPBACK()   *(remove and return an item from the back of the deque)*
D.FRONT()   *(return the front element in the deque without removing it)*
D.BACK()   *(return the back element in the deque without removing it)*
D.ISEMPTY()   *(true iff the deque is empty)*
D.SIZE()   *(return the number of elements in the deque)*

Optionally, a deque can be used to search for an item x with a given key $x.k$ or to find the predecessor or successor of an item x by its pointer/reference.

$D.\text{SEARCH}(k)$ *(return an item x with key k or "nil")*
$D.\text{PREDECESSOR}(x)$ *(return the predecessor of x)*
$D.\text{SUCCESSOR}(x)$ *(return the successor of x)*

Deques are often implemented as doubly linked lists with front and back pointers (and the current size maintained in a separate variable). In this case pushing and popping items happens in constant time, searching requires scanning through the list in $O(n)$ time. And predecessors/successors can be found by following the `next` and `prev` pointers.

STL class `vector` in C++ supports the deque operations, but also allows random access just as in array (returning an element by its index). Similar things are supported by Python lists or by `java.util.List` interface. Such data structures are more powerful than either stacks, queues or deques – they behave like arrays (without strict limit on size).

3.2 Problems

Problem 1: In some Pythons implementations, the dynamic array is grown by $n/8$ whenever the list overflows. Assume that r is the ratio between inserts and reads for this dynamic array. Find the value r for which this growth factor is the optimal one.

Problem 2: Reverse the order of elements in stack S in the following ways:

(A) Use two additional stacks, but no auxiliary variables.

(B) Use one additional stack and some additional non-array variables (i.e. cannot use lists, sequences, stacks or queues).

Problem 3: Read elements from an iterator and place them on a stack S in ascending order using one additional stack and some additional non-array variables.

Problem 4: Given a data structure implementing the Sequence ADT, show how to use it to implement the Set interface. (Your implementation does not need to be efficient.)

Problem 5: What are the costs for each ADT operation, if a queue is implemented as dynamic array?

Problem 6: Which operations become asymptotically faster, if list ADT is implemented as a doubly linked list (instead of a singly linked list)?

Problem 7: (Stack Implementation as Array): Stack is implemented as an array. In our case the array has size $n = 5$. Stack contains integer numbers; initially the array has the following content.

size	5
length	2
array[]	11 12 13 14 15

Stack has the physical representation with `length` $= 2$ (the number of elements in the stack), `size` $= 5$ (maximal number of elements contained in the stack). We have the following fragment:

```
pop();
push(21);
push(22);
pop();
push(23);
push(24);
pop();
push(25);
```

Draw the state of the array after every command. (Every `push(elt)` command assigns a new element into the element `array[length]`, then increments `length` by 1. The command `pop()` does not modify the array, but decreases `length` by 1.

If the command cannot be executed (`pop()` on an empty stack; `push(elt)` on a full stack), then the stack structure does not change at all (`array` or `length` are not modified). To help imagine the state of this stack, you can shade those cells that do not belong to the array.

Answer:

The figure below shows stack memory states after each of the 8 operations above are completed:

	array[ ]					length
at start	11	12	13	14	15	2
after line 1	11	12	13	14	15	1
after line 2	11	21	13	14	15	2
after line 3	11	21	22	14	15	3
after line 4	11	21	22	14	15	2
after line 5	11	21	23	14	15	3
after line 6	11	21	23	24	15	4
after line 7	11	21	23	24	15	3
after line 8	11	21	23	25	15	4

Problem 8 (Queue Implementation as a Circular Array): A queue is implemented as an array with `size` elements; it has two extra variables `front` (pointer to the first element) and `length` (the current number of elements in the queue). Current state is shown in the figure:

size	6
front	2
length	4
array[ ]	1 3 5 7 9 11

Enumeration of array elements starts with 0. The array is filled in a circular fashion. The command `enqueue(elt)` inserts a new element at

$$(\text{front} + \text{length}) \bmod \text{size},$$

where mod means the remainder when dividing by `size`. It also increments the `length` element.

The command `dequeue()` does not change anything in the array, but increments `front` by 1 and decreases $length$ by 1. Thus the queue becomes shorter by 1.

```
dequeue();
enqueue(21);
dequeue();
enqueue(22);
enqueue(23);
enqueue(24);
dequeue();
```

Show the state of the array after every command – `array, length, front` variables after every line. (Shade the unused cells.)

Answer:

The figure below shows queue memory states after each of the operations above are completed:

Problem 9: We suggest a specialized Abstract Data Type (ADT) named `Backtracker` which can be used to solve backtracking tasks such as N-Queens problem, Sudoku and other combinatorial tasks.

A backtracker object stores some (partially solved) instance of a backtracking problem, and an external driver can use it to visit all nodes rooted tree in the DFS order. As soon as it sees a complete valid solution, the solution can be output. Sometimes we want to find all solutions, if there exist more than one. *Backtracking is typically an inefficient exponential time algorithm, but it can be improved, if it can establish early that in the given subtree there are no more solutions.*

Here is the ADT:

(Initialize Backtracker with its initial state, e.g. an empty chessboard)
$B = \text{BACKTRACKER}(s : State)$
(Get available moves at the given level, e.g. all queen positions for some vertical)
$B.\text{MOVES}(level : int) : \text{LIST} < \text{Move} >$
(Is the move valid in the given state, e.g. is the chosen position attacked by earlier queens)
$B.\text{VALID}(level : int, move : Move) : \text{BOOLEAN}$
(Record the move to the backtracker, e.g. add one more chosen queen position)
$B.\text{RECORD}(level : int, move : Move)$
(Opposite to record – undo the move, e.g. remove the latest queen position)
$B.\text{UNDO}(level : int, move : Move)$
(Is the search successfully completed, e.g. all N queens already placed?)
$B.\text{DONE}(level : int) : \text{BOOLEAN}$

(Output the successful solution for the Backtracker object, e.g. print the chessboard)

$B.\text{OUTPUT}()$

(A) Write the pseudocode for a function $\text{ATTEMPT}(b : \text{BACKTRACKER}, level : \text{INT})$ so as to find the first solution starting with the backtracker object on level *level*.

(B) Modify the function $\text{ATTEMPT}(b : \text{BACKTRACKER}, level : \text{INT})$ so that it does not stop until it outputs all valid solutions.

Answer:

(A) Here is the code to find just one solution for a backtracker:

```
ATTEMPT(b : BACKTRACKER, level : Int) : Bool
    successful = FALSE
    moves = b.MOVES(level)
    for move in moves:
        if b.VALID(level, move):
            b.RECORD(level, move):
            if b.DONE(level):
                successful = TRUE:
            else:
                successful = ATTEMPT(b, level)
                if not successful:
                    b.UNDO(level, move)
        if SUCCESSFUL:
    return successful
```

See Robert E. Noonan. An Object-Oriented View on Backtracking

WORKSHEET 04: BUILD AND DEBUG ON LINUX

This worksheet covers build tools (Makefile, CMake), unit testing (Catch2) and memory leak detection (Valgrind). Since the exams do not include practice part with computing devices or C++, this is the only worksheet that does not contribute any problems to the midterm or final exams.

4.1 Concepts and Facts

4.1.1 Building with Makefile

This is a minimalistic Makefile that builds `myprogram` from a single source file `myprogram.cpp`.

```
CC=g++
CFLAGS=-std=c++17

all: myprogram

myprogram: myprogram.cpp
        $(CC) $(CFLAGS) -o myprogram myprogram.cpp

.PHONY: clean
clean:
        rm -f myprogram
```

Utility *make* was invented in 1976. Analyzes dependencies (which file was updated when), target names are typically same as file names. In the above example

- In a *Makefile* each line should be either non-indented (a variable definition or a target) or be indented by exactly one *TAB* character. Preceding that *TAB* by an (invisible) whitespace would break it.

- *.PHONY clean* declaration says that the target name has nothing to do with any filename. (If we do not use it, then *clean* would not run provided there is a file *clean*.

A more complicated testfile using multiple sources (and even multiple executables – one for software itself, another for unit-tests) is shown in the Catch2 subsection (see below).

Use Makefile to Run Testfiles:

```
# Define variables
PROGRAM = myprogram
INPUT_FILES = $(wildcard test*.txt)
OUTPUT_FILES = $(patsubst test%.txt,output.test%.txt,$(INPUT_FILES))

.PHONY: test
```

```
test: $(OUTPUT_FILES)


output.test%.txt: $(PROGRAM) test%.txt
        ./$(PROGRAM) < $< > $@ || true


# Clean up target
clean:
        rm -f $(OUTPUT_FILES)
```

The snippet || *true* prevents stopping the make task, if some of the program execution returns non-zero return code or crashes.

4.1.2 Building with CMake

Currently C++ developers use mostly *CMake* – a meta-build tool that creates makefiles or other build artefacts from a description of the project and its dependencies described in the CMakeLists.txt file. CMake can participate in other build chains – such as Gradle build tasks, where Android app written in Java or Kotlin needs some native code in C++.

In other cases custom Bash shell, Python or Groovy can be used instead of Makefile or CMakeLists.txt. Build tools are often part of larger build infrastructures using crontab time-scheduling, Continuous Integration tools such as Jenkins.

CMakeLists.txt Example:

```
cmake_minimum_required(VERSION 3.10)
project(myprogram)
add_executable(myprogram myprogram.cpp)
```

Here is how to use it:

```
cmake .
make
```

4.1.3 Unit-tests with Catch2

To run a project with Catch2 tests we need two different build goals in *Makefile*. One of them is builds the executable you can run; another one builds the test harness (executable that can be used to run the unit tests).

The following things are commonly used with Catch2 tests:

- No dependencies on additional libraries; but tests should include header file catch.hpp containing various assert definitions and macros.

- Testcases can check normal execution – for example, REQUIRE(...) verifies that the expression is true.

- Testcases can check abnormal cases when the expected behaviour is throwing an exception. For example, REQUIRE_THROWS_AS(...) means that the expression throws an exception of the specified type.

- It is possible to have common initialization section in a testcase, which is then used by multiple sections (each section receives the same initial state, but does something different).

- Using Catch2 means that you produce one more executable (see the next subsection for an example of a Makefile to build two different executables in the same directory).

```cpp
#define CATCH_CONFIG_MAIN

#include "catch.hpp"
#include "Stack.h"

TEST_CASE("Exceptions on empty stack", "[stack]")
{
  Stack stack(3);
  REQUIRE_THROWS_AS(stack.top(), std::out_of_range);
  REQUIRE_THROWS_AS(stack.pop(), std::out_of_range);
  stack.push(17);
  stack.pop();
  REQUIRE_THROWS_AS(stack.top(), std::out_of_range);
}

TEST_CASE("Lifo order", "[stack]")
{
  Stack stack(3);
  stack.push(1);
  stack.push(2);
  REQUIRE(stack.top() == 2);
  REQUIRE(stack.pop() == 2);
  REQUIRE(stack.top() == 1);
  REQUIRE(stack.pop() == 1);
  REQUIRE_THROWS_AS(stack.top(), std::out_of_range);
}

TEST_CASE("3-element stack", "[stack]")
{
  // common initialization part
  Stack stack(3);
  stack.push(11);
  stack.push(12);
  stack.push(13);

  SECTION("Stack is full") {
    REQUIRE_THROWS_AS(stack.push(14), std::out_of_range);
  };

  SECTION("Multiple top calls") {
    REQUIRE(stack.top() == 13);
    REQUIRE(stack.top() == 13);
  };
}
```

4.1.4 A Makefile to build Catch2 test executable

In this example we assume that the unit-testing executable is built from these sources:

TestStack.cpp: Catch2 testcases; its source is shown above.

catch.hpp: Catch2 header file, which you do not need to change.

Stack.h: Stack ADT methods.

Stack.cpp: Stack implementation.

Meanwhile, there is also the source file `StackMain.cpp` (a program doing something useful and using our stack). In this case the main program can be built with `make all`, but the testcases can be built with `make test`.

The Makefile to compile such project is shown below:

```
CC=g++
CFLAGS=-std=c++17 -g
SRCDIR=.
OBJDIR=.
SRC=$(wildcard $(SCRCDIR)/*.cpp)
OBJ1=$(OBJDIR)/Stack.o $(OBJDIR)/StackMain.o
OBJ2=$(OBJDIR)/Stack.o $(OBJDIR)/TestStack.o
EXECMAIN=$(SRCDIR)/stack-main
EXECTEST=$(SRCDIR)/stack-test

all: $(EXECMAIN)
test: $(EXECTEST)

$(EXECMAIN): $(OBJ1)
  $(CC) $(CFLAGS) -o $@ $^

%.o: %.cpp
  $(CC) $(CFLAGS) -c -o $@ $<

$(EXECTEST): $(OBJ2)
  $(CC) $(CFLAGS) -o $@ $^


.PHONY: clean
clean:
  rm -f $(SRCDIR)/*.o $(EXEC1) $(EXEC2)
```
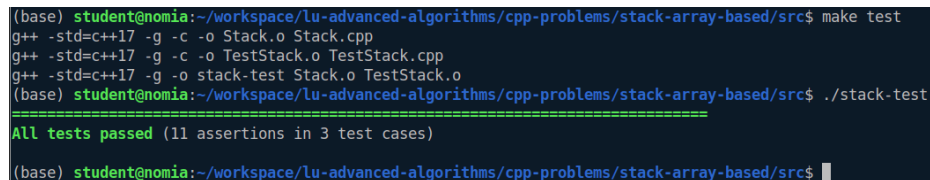

Fig. 1: Sample Output from Catch2 testcases.

4.1.5 Debugging with gdb

`gdb myprogram`: Start gdb and load the myprogram executable.

`run`: Start the program.

`break <line_number>`: Set a breakpoint at the specified line number.

`info break`: Show all defined breakpoints.

`delete <breakpoint_number>`: Delete the specified breakpoint.

`next`: Step over the current line.

`step`: Step into the function called on the current line.

`finish`: Continue execution until the current function returns.

`backtrace`: Show the current call stack.

list: Show the current source code around the current line.

print <variable_name>: Print the value of the specified variable.

display <variable_name>: Display the value of the specified variable after each step.

watch <variable_name>: Set a watchpoint on the specified variable.

info registers: Show the current state of all CPU registers.

x/<length><format><address>: Examine memory at the specified address, with the specified format and length.

layout src: Display the source code and assembly code in separate windows.

layout regs: Display the CPU registers and the source code in separate windows.

layout split: Display the source code and the program output in separate windows.

layout next: Switch to the next layout.

4.1.6 Valgrind

Memory leak detection: Valgrind can detect memory leaks by identifying when memory is allocated but not freed. Use `--leak-check` option.

```
valgrind --leak-check=yes ./myprogram
# (or write directly to a file)
valgrind --leak-check=yes --log-file=leak_report.txt ./myprogram
```

```
1 2 0 3 0 0 4 0 5
AAA
1 2 0 3 0 0 4 0 5
BBB
1 2 3 4 5
==9520==
==9520== HEAP SUMMARY:
==9520==     in use at exit: 144 bytes in 9 blocks
==9520==   total heap usage: 12 allocs, 3 frees, 77,968 bytes allocated
==9520==
==9520== 144 (64 direct, 80 indirect) bytes in 4 blocks are definitely lost in loss record 2 of 2
==9520==    at 0x4849013: operator new(unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==9520==    by 0x1092BE: insert(Node*&, Node*&, int) (edit-linked-lists.cpp:23)
==9520==    by 0x10953E: main (edit-linked-lists.cpp:103)
==9520==
==9520== LEAK SUMMARY:
==9520==    definitely lost: 64 bytes in 4 blocks
==9520==    indirectly lost: 80 bytes in 5 blocks
==9520==      possibly lost: 0 bytes in 0 blocks
==9520==    still reachable: 0 bytes in 0 blocks
==9520==         suppressed: 0 bytes in 0 blocks
==9520==
==9520== For lists of detected and suppressed errors, rerun with: -s
==9520== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
(base) student@nomia:~/workspace/lu-advanced-algorithms/cpp-problems/edit-linked-lists/src$
```

Fig. 2: Sample Output from Valgrind for memory leak check.

Memory error detection: Valgrind can detect memory errors: accessing memory that has already been freed, accessing uninitialized memory, and writing to read-only memory. Use `--tool=memcheck` option.

```
valgrind --tool=memcheck ./myprogram
```

Performance profiling: Valgrind can help identify performance bottlenecks by profiling CPU usage, memory usage, and other metrics. Use `--tool=callgrind` option

```
valgrind --tool=callgrind ./myprogram
```

This generates a file called callgrind.out.<pid>. Can use a tool like `kcachegrind` to visualize the profiling data.

To address typical reasons of memory leaks, one should pay attention to the data structures (including built-in ones from STL). Every time a data structure is created, passed as a parameter, copied as a class object or deep-copied (cloned), can be related to a leak or a crash if memory is released in a wrong place.

- What constructors create empty data structures or structures with initializer lists?
- What are copy constructors doing during assignments or function calls?
- When are the destructors called?
- When is a proper time to release memory?

4.2 Problems

Some questions here are open-ended; they are interview-style questions for C++ developers on Linux platforms.

Problem 1: Answer some questions about `Makefile` builds:

(A) What is a dependency in a Makefile, and how is it specified?

(B) How does Makefile determine whether a target needs to be rebuilt or not?

(C) What is the purpose of the `.PHONY` target in a Makefile, and when should it be used?

(D) What is a pattern rule in a Makefile, and how is it used?

(E) What is the meaning of variables `$@` and `$<` in a Makefile ?

(F) How can you specify conditional dependencies in a Makefile, and why would you want to do this? (stuff like `ifeq, else, endif`)

Problem 2: Answer some questions about `gdb` build.

(A) How would you compile a C++ program (source files `A.cpp, B.cpp, B.h`). on Linux using the g++ compiler? What flags ensure that it is debuggable?

(B) Which command can be used to set a breakpoint in the program?

(C) Which command can be used to see a value of a variable (or an expression?) in a C++ program while its running?

(D) How can we use the core file generated by gdb? How would you use it debug a program that has crashed?

(E) What is the purpose of the core file generated by gdb? How to use it to debug a program that has crashed?

(F) Can you explain the difference between a stack overflow error and a segmentation fault error? How would you debug each of these types of errors using gdb?

(G) Which command can examine the contents of memory at a particular address in a C++ program?

(H) Which command can show the call stack of a C++ program during debugging?

(I) Can you explain what the watch command in gdb does? How to use it to monitor a variable in your C++ program?

(J) How would you use `gdb` to examine the assembly code generated by the `g++` compiler?

Problem 3: Consider the following C++ code to store custom objects of type `Pair` in an STL `vector` data structure. Explain which constructors and destructors are invoked – how many and at which locations of the code.

```cpp
#include <vector>
#include <iostream>

using namespace std;

class Pair {
  public:
    int nX,nY;
  private:

};

int main(int argc, char** argv) {

  vector<Pair> myVector;
  for(int i=0 ; i<10 ; i++) {
    int x, y;
    cin >> x >> y;
    Pair p;
    p.nX = x; p.nY = y;
    myVector.push_back(p);
  }

  for (auto it = myVector.begin(); it != myVector.end(); ++it) {
    cout << "(" << (*it).nX << "," << (*it).nY << ")" << endl;
  }
  return 0;
}
```

Data Structures: Exam Topics

WORKSHEET 05: PRIORITY QUEUES

Priority queues are data structures that do not require full sorting, but provide the benefit of sorted lists – at every moment we can find the minimum (or maximum) of the items.

5.1 Concepts and Facts

Definition: A priority queue is an abstract data type supporting the following operations:

PRIORITYQUEUE() – create empty priority queue.

Q.INSERT($item$) – insert an item (with any key).

Q.EXTRACTMIN() – remove and return the element with the minimum key.

Priority queues can be implemented in various ways:

- As an ordered list (easy to find the minimum, but inserting new items may be expensive).

- As an unordered list (easy to add new items, but expensive to find the minimum).

- Create a *heap* (described below; insertion and finding the minimum are both fast).

Definition: A binary tree is called a *complete tree* if it has all layers filled in, except, perhaps, the last layer, which is filled from left to right (and may contain some empty slots on the right side of the last layer).

Complete trees were introduced just because they can be easily stored in arrays (all nodes are listed in the array – layer by layer).

Definition: A binary tree storing items with ordered keys is called a *minimum heap*, if it satisfies the following representation invariant:

- The binary tree is a complete binary tree.

- Each parent node in the tree has key that cannot be larger than either of its children.

Similarly we define a *maximum heap* (no parent can be smaller than either of its children). Stacks, queues, deques and their implementations can preserve any *extrinsic order* of the items – typically the order they were inserted. In contrast, heap is a data structure that respects intrinsic ordering of item keys.

Definition: A *prefix encoding* for the given alphabet of messages $\mathcal{A} = \{m_1, m_2, \ldots, m_k\}$ is a function $e : \mathcal{A} \to \{0, 1\}^*$ that maps each message to a codeword (a sequence of 0s and 1s) so that there are no two messages m_i, m_j such that $e(m_i)$ is a prefix of $e(m_j)$.

Any prefix encoding can be represented as a rooted tree with edges labeled with 0s and 1s – and every message is represented by a leaf in that tree.

Statement: Given any sequence of 0s and 1s and any prefix code for the alphabet of messages $\mathcal{A}$, the sequence of bits can be decoded into messages in no more than one way (i.e. there is no ambiguity).

Example: Consider the following Prefix Tree to encode letters in alphabet $\mathcal{A} = \{S, I, E, N, T, A\}$.
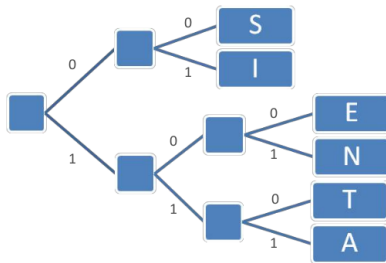
Every letter is encoded as a sequence of 0s and 1s (the path from the root to the respective letter).

- `11100110100` decodes as `111.00.110.100` or `A.S.T.E`.
- `0001100101111` decodes as `00.01.100.101.111` or `S.I.E.N.A`.

It may happen that some codeword has not been received completely (e.g. the codeword is expected to have three bits, but we only got two bits). These are the only scenarios when decoding cannot be completed.

Prefix trees that are optimal for encoding some alphabet of messages (with known message probabilities) uses priority queues. It is named Huffman algorithm.

Huffman Algorithm: Let C be the collection of letters to be encoded; each letter has its frequency $c.freq$ (frequencies are numbers describing the probability of each letter).

> HUFFMAN(C):
> $n = |C|$
> $Q = $ PRIORITYQUEUE(C)   (*Minimum heap by c.freq*)
>   **for** $i = 1$ to $n - 1$ (*Repeat n-1 times*)
> $z = $ NODE()
> $z.left = x = $ EXTRACTMIN(Q)
> $z.right = y = $ EXTRACTMIN(Q)
> $z.freq = x.freq + y.freq$
>   INSERT(Q, z)
>   **return** EXTRACTMIN(Q)   (*Return the root of the tree*)

Intuitively, if some message occurs with probability at least $1/2$, an optimal prefix tree (as obtained by Huffman algorithm) will encode it with one bit. If it occurs with probability at least $1/4$, an optimal prefix tree will spend up to two bits. If some message occurs with probability at least $1/8$, an optimal prefix tree will spend up to three bits. This can be formalized as follows:

Definition: For the message source (with messages $c \in \mathcal{A}$ define *Shannon entropy* with this formula:

$$H(\mathcal{A}) = \sum_{c \in \mathcal{A}} (-\log_2 P(c)) \cdot P(c),$$

where $P(c)$ denotes the probability of the character c in the alphabet.

Statement: No prefix tree can encode the alphabet $\mathcal{A}$ more efficiently than the Shannons entropy. Formally speaking, the expected number (the probabilistic weighted mean) of bits sent per one message $c \in \mathcal{A}$ will be at least $H(\mathcal{A})$.

5.2 Problems

Problem 1:

(A) Assume that heap is implemented as a 0-based array (the root element is `H[0]`), and the heap supports DELETEMIN(H) operation that removes the minimum element (and returns the heap into consistent state).

Find, if the heap property holds in the following array:

$$H[0] = 6, 17, 25, 20, 15, 26, 30, 22, 33, 31, 20.$$

If it is not satisfied, find, which two keys you could swap in this array so that the heap property is satisfied again. Write the correct sequence of array H.

Note: A *consistent state* in a minimum heap means that the key in parent does not exceed keys in left and right child.

(B) Assume that heap is implemented as a 0-based array (the root element is `H[0]`), and the heap supports DELETEMAX(H) operation that removes the maximum element.

If the heap does not satisfy invariant (in a consistent max-heap, every parent should always be at least as big as both children), then show how to swap two nodes to make it correct.

$$96, 67, 94, 10, 67, 68, 69, 9, 10, 11, 50, 67.$$

Problem 2 (Insert into a min-heap): Show what is the final state of a heap after you insert number 6 into the following minimum-heap (represented as a zero-based array):

$$9, 18, 28, 23, 20, 29, 33, 25, 36, 34, 23.$$

Problem 3 (Delete maximum from a Max-Heap): Show what is the final state of a heap after you remove the maximum from the following heap (represented as a zero-based array):

$$96, 67, 94, 10, 67, 68, 69, 9, 10, 11, 50, 67.$$

Problem 4 (Removing from Maximum Heap): Here is an array for a Max-Heap:

16	14	10	8	7	9	3	2	4	1

The image shows array used to store Maximum Heap (a data structure allowing inserts and removal of the maximum element). The array starts with the 0-th element (and any parent node in such tree should always be at least as big as any of its children).

(A) Draw the initial heap based on this array. Heap should be drawn as a complete binary tree.

(B) Run the command DELETEMAX(H) on this initial heap. Draw the resulting binary tree (after the heap invariant is restored – any parent node is at least as big as its children). Draw the binary tree image you get.

(C) On the tree that you got in the previous step (B) run the command INSERT(H, x), where $x = a + b + c$ is the sum of the last three digits of your student ID. Draw the binary tree image you get.

(D) Show the array for the binary tree you got in the previous step (C) (i.e. right after the DELETEMAX(H) and INSERT(H, x) commands have been executed).

Problem 5: Let us remind the *postfix notation* for arithmetic expressions. The following postfix expression:

```
2  17  1  -  *  3  4  *  +
```

represents the same expression as the infix expression $2 * (17 - 1) + 3 * 4$. Consider the following algorithm to evaluate postfix expressions:

POSTORDEREVALUATE($E : array[0..n - 1]$): Int
    $stack = emptyStack()$
    for i **from** 1 **to** n:
        if ISNUMBER($E[i]$):
            $stack$.PUSH($E[i]$)
        else:
            $x1 = stack$.POP()
            $x2 = stack$.POP()
            $res = $ APPLYOP($E[i], x1, x2$)
            $stack$.PUSH(res)

Assume that `stack` in this pseudocode is implemented as an array-based stack. Write the current state of `stack` right after the number 4 is inserted from the input tokens `2 17 1 - * 3 4 * +`.

Problem 6: Consider the task to identify correctly matched vs. incorrectly matched sequences of parentheses. There are three kinds of parentheses (round parentheses `()`, square brackets `[]` and curly braces `{}`); they are correctly matched iff opening and closing parentheses of the same kind (round, square or curly) can be put in pairs so that any two pairs either do not intersect at all or one pair is entirely inside another pair. Here are some examples:

```
correct: ( ) ( ( ) ) { ( [ ( ) ] ) }
correct: ( ( ( ) ( ( ) ) { ( [ ( ) ] ) }
incorrect: ) ( ( ) ) { ( [ ( ) ] ) }
incorrect: ( { [ ] ) }
incorrect: (
```

Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number.

Output: True if and only if all the grouping symbols in X match

Answer:

PARENMATCH($X[0..n - 1]$):
    $S = $ EMPTYSTACK()
    for i **in** range(n):
        if $X[i]$ is an opening parenthesis:
            S.PUSH($X[i]$)
        else if $X[i]$ is a closing parenthesis:
            if S.EMPTY():
                return FALSE *(nothing to match with)*
            if $S.pop()$ does not match the type of $X[i]$:
                return FALSE *(wrong type of parenthesis)*
    if S.EMPTY():
        return TRUE *(every symbol matched)*
    else

> **return** FALSE *(some symbols were never matched)*

Problem 7: Let the alphabet have 6 characters $\mathcal{A} = \{A, B, C, D, E, F\}$ and their probabilities are shown in the table:

c	A	B	C	D	E	F
$P(c)$	45%	13%	12%	16%	9%	5%

(A) Use the Huffman algorithm to create a Prefix Tree to encode these characters.

(B) Compute the Shannon entropy of this information source (sending the messages with the probabilities shown).

(C) Also compute the expected number of bits needed to encode one random letter by the Huffman code you created in (A). (Assume that letters arrive with the probabilities shown in the table.)

Data Structures: Exam Topics

WORKSHEET 06: SEARCH TREES

6.1 Concepts and Facts

Definition: A tree is named *Binary Search Tree* (BST) if the nodes satisfy the *order invariant*: Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If z is a node in the right subtree of x, then $z.key \geq x.key$.

Definition: In a binary tree, the *inorder predecessor* of a node v is a node u iff v directly follows u in the inorder traversal of the nodes. Similarly, the *inorder successor* of a node v is w iff w directly follows v in the inorder traversal.

To delete an internal node from a BST (having both left and right children), you can replace it either by the inorder predecessor or the inorder successor.

Definition: The *height* of a node in a tree is defined by induction:

- Null trees (empty trees) have height -1

- Leaves (single node trees) have height 0

- Any node v has height $h(v) = \max(h(v_{\text{left}}), h(v_{\text{right}})) + 1$

Definition: A binary search tree is called an *AVL tree* iff each node v is balanced. Namely, the heights of both its subtrees do not differ by more than 1.

$$|h(v_{\text{left}}) - h(v_{\text{right}})| \leq 1$$

To see, if a tree is an AVL tree (the representation invariant must be preserved even after we insert or delete something!) we need to store the balance inside the tree node. Such additional information is called graph/tree node augmentation. (Augmentations are used by many algorithms and data structures. AVL trees needing the height is just one example.)

6.2 Problems

Problem 1: Even binary trees that are not complete can be represented as arrays – their nodes written out layer by layer just like a complete tree in a heap. If any node in the tree is missing, it is replaced by Λ. The last non-empty node in the tree is the last element of the array. Draw the trees corresponding to the following arrays (here a, b, c are variable letters stored in the nodes). Distinguish the left and right children clearly.

**(A)** `int a[]` $= \{1, 2, 3, 7, \Lambda, 5\}$;

(B) `int a[]` $= \{1, 2, 4, a, \Lambda, \Lambda, 6, b, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda, c\}$;

(C) Write a pseudocode to check, if the input array represents a binary tree (return True), or it is inconsistent (for example, there are non-empty children under some Λ).

(D) Write a pseudocode to count the internal nodes and the leaves, if you receive an array as an input.

(E) Write a pseudocode to list the vertices of this tree in the post-order traversal order.

Answer:

(A)
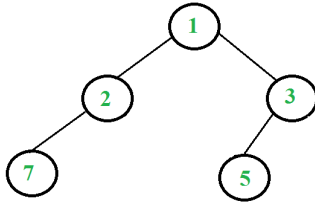
Problem 2: Non-binary ordered trees can be encoded as binary trees (using a bijective encoding function). See https://bit.ly/3khnC0p for details. (If in a general tree the node w is the first child of v, then in the corresponding binary tree w becomes the *left child* of v. If w is the sibling to the right of v, then in the corresponding binary tree w is the *right child* of v.) In this problem we use *bracket representations* for all trees (see https://bit.ly/425tzVa).

(A) Consider the following general tree: `A(B(E)(F)(G))(C)(D(H)(I)(J))`.

   Draw this multiway/general tree. Encode it as the binary tree and draw it.

(B) `A(B(E()(F(J()(K))()))(C()(D(G()(H(L(N)(M))(I))()))()))()`.

   Given the binary tree, restore the original general tree.

(C) Consider the binary tree from (B); list its nodes in their in-order DFS traversal order.

(D) What is the depth of the node N (defined above) in the new (general) tree?

Answer:

(A) The general tree is given on Figure *Multiway Tree*.
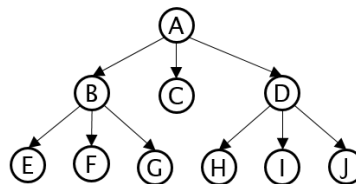
Fig. 1: Multiway Tree

Encoding Step 1 Redraw edges (only connect each node with its first child and also to the sibling to the right). To see clearly which edges will be left-going, and which are right-going, can color them differently. See Figure *Tree with Horizontal Edges*.
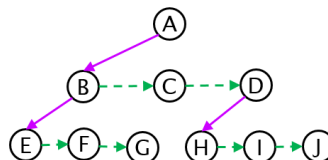

Fig. 2: Tree with Horizontal Edges

Encoding Step 2 Adjust the levels in the new binary tree so that it takes a more conventional look (left children to the left, right children to the right). See Figure *Encoded Binary Tree*.
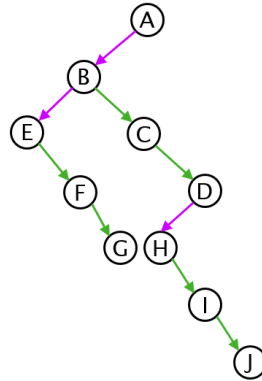
Fig. 3: Encoded Binary Tree
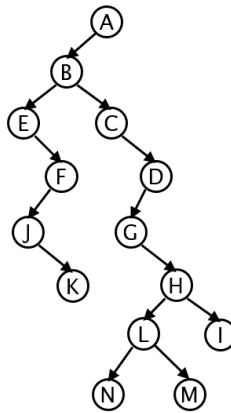
(B) The binary tree looks like this:

Fig. 4: Binary tree for Question 6.1.1

Restored tree can be obtained, if one colors the edges and turns the left children into first children, and the right children to siblings.


Note: See Encoding general trees as binary trees or https://bit.ly/3kdyg8n.

Problem 3: Let B_n denote how many different BSTs for n different keys there exist (all the trees should have correct order invariant). We have $B_1 = 1$ (one node only makes one tree). And $B_2 = 2$.

Draw all the binary search trees to store numbers $\{1, 2, 3\}$ and also the numbers $\{1, 2, 3, 4\}$.

Find the values B_3 and B_4 (the number of binary search trees).

Problem 4: Consider the following Binary Search Tree (BST).
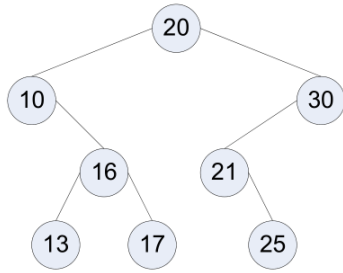
Let a, b be the first two digits of your Student ID. Compute the following numbers:

$$X = 2a,$$
$$Y = 20 + b,$$
$$Z = 3b,$$
$$S = b,$$
$$T = 2(a + b) \bmod 40$$
$$U = (a + b) \bmod 10$$

Run the following commands on this BST (and draw the intermediate trees whenever there is the show command):

BST.INSERT(X)
BST.INSERT(Y)
BST.DELETE(20)
BST.SHOW$()$
BST.INSERT(Z)
BST.INSERT(S)
BST.DELETE(13)
BST.SHOW$()$
BST.INSERT(T)
BST.INSERT(U)
BST.DELETE(X)
BST.SHOW$()$

Ignore a command, if it asks to insert a key that already exists or deletes a key that does not exist.

Question 5: Let T_n be an AVL tree of height n with the smallest possible number of nodes. For example $|T_0| = 1$ (just one node is an AVL tree of height 0); $|T_1| = 2$ (a root with one child only is an AVL tree of height 1) and so on.

(A) Draw AVL trees T_2, T_3, T_4 and T_5.

(B) Write a recurrence to find the number of nodes $|T_n|$ (recurrent formula expresses the number $|T_n|$ using the previous numbers $|T_k|$ with $k < n$).

Problem 6: Let T be some (unknown) BST tree that also satisfied the AVL balancing requirement. After k nodes were inserted (without any re-balancing actions) the tree T' now looks as in the image below.
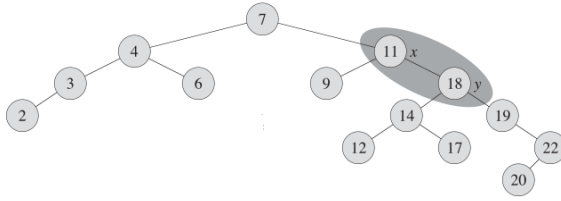
(A) Find the smallest value of k – the nodes that were inserted into the original T to get T'.

(B) Show the tree after LEFTROTATE(T', x) – the left rotation around the node x. Is the resulting tree an AVL tree now?

Problem 7: Assume that a Binary Search Tree T is created by inserting the following keys into an empty tree: $[39, 20, 65, 11, 29, 50, 26]$ (in the given order).

(A) Do the following actions on this tree one after another: $T.\text{INSERT}(22)$, $T.\text{INSERT}(60)$, $T.\text{DELETE}(11)$.

(B) Suggest a sequence of inserts/deletes for the original tree T (with 7 nodes) so that the last delete operation in that sequence causes two rotations.

Problem 8: Consider the binary tree shown below.
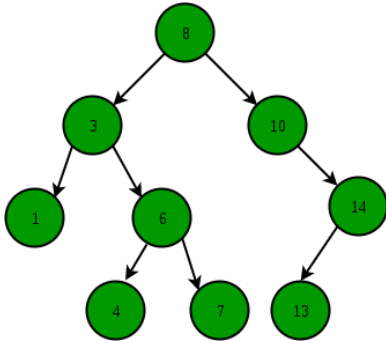
Every key in this tree is being searched with the same probability. Find the expected number of pointers that are followed as we search for a random key in this tree. (For example, searching the key at the root means following 1 pointer, searching the key that is a child of the root means following 2 pointers and so on.)

Data Structures: Exam Topics

WORKSHEET 07: AUGMENTED STRUCTURES

Vanilla binary search trees are easy to reason about, but in practice one may need to make variations for trees and other classical data structures. Common variation is to augment tree nodes with parameters (such as height used for AVL trees), node colors, successor counts and similar parameters. Another common way to generalize binary search trees is *multiway search trees*, where nodes can have more than two children. They may help with processing speed, if nodes match the size of memory pages loaded from the disk.

7.1 Concepts and Facts

Definition: $(2, 4)$-trees are search trees where each node stores one/two/three keys and each non-leaf node has respectively two/three/four child nodes. For example, if $k_1 < k_2 < k_3$ are keys stored in some node, then one of the child pointers is to the left of k_1, another pointer is between k_1 and k_2, one more pointer is between k_2 and k_3, and the last pointer is to the right of k_3. (All the leaf nodes have null-children.)

Moreover, the nodes in $(2, 4)$-tree should be in searchable order (e.g. each subtree stores keys that fall in-between the two keys in the parent node), and all null-leaves are at the same depth. See 2-3-4 Tree in Wikipedia.

- In order to insert a key in such a tree, you find the last non-null node at the bottom where the key fits, and insert it among the existing keys. It might happen that a node overflows (has 4 keys and 5 null children). In this case it is split into two nodes - with 2 and 1 keys respectively (and one more key is promoted one level up). If ithe promoted key causes another overflow, you split again, and so on.

- In order to delete a key from such a tree, do the following steps:

    - If the key to be deleted is not a leaf, replace it by in-order successor (so that you only need to know how to delete leaves).

    - Otherwise proceed as in 2-3-4 Tree Deletion.

Definition: A tree is named a *Red-Black Tree*, if it is a Binary Search Tree, every node is either red or black (a flag stores its color) and it satisfies *red-black invariants*:

Root property: The root is black.

External property: Every leaf (a node with NULL key) is also black.

Internal property: If a node is red, then both its children are black.

Depth property: All simple paths from some node to its descendant leaves have the same number of black nodes.


Note: We can compute the black-height $h_{\text{black}}(v)$: it is 0 for all leaves, and for any other node v, it is the maximum of all $h_{\text{black}}(v_i)$ of its descendants v_i plus one.)


Statement: Each red-black tree can be uniquely represented as a (2,4)-tree. They are just differently drawn represen-
tations of the same concept.

Example: The picture below shows how a red-black tree can be converted into a (2,4) tree. Every black node (and
optionally one or two of its red children) become keys in the new (2,4)-tree. The representation invariants of
red-black trees ensures that the properties of (2,4)-tree are also satisfied.
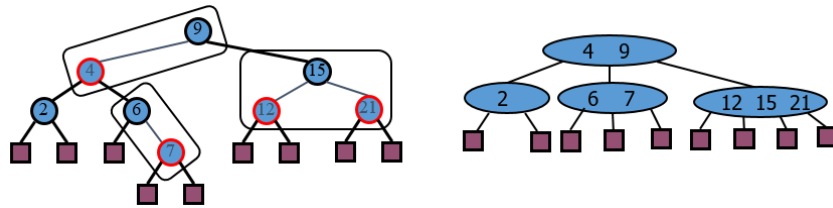
7.2 Problems

Problem 1: Show how to insert a new node 31 into this $(2, 4)$-trees:
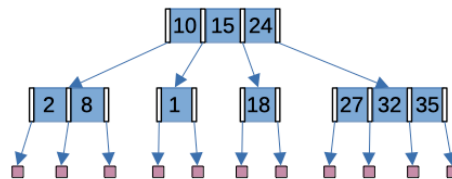
Problem 2: Build an example of (2,4)-tree, where the root has height equal to 3 and where deleting some key would
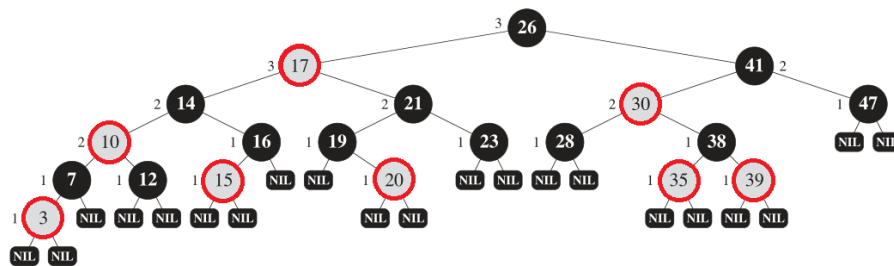cause the height to decrease.

Problem 3:

Fig. 1: Sample Red-Black Tree

(A) Compute the following three key values (u, v, and w):

$$\begin{cases} u = 3(a + b) + 2 \\ v = 3(b + c) + 1 \\ w = 3(c + a) \end{cases}$$

Here a, b, c are the last 3 digits of your Student ID.

Verify the black height of every node in the graph – all NULL leaves have black height equal to zero.
Any other node has black height equal to the number of black nodes that are on some descendant
path. (According to the depth property – the black height of any node should not depend on the path
to the leaf we chose.)

(B) Show how the tree looks after the nodes u, v and w (in this order) are inserted in the Red-Black Tree shown in Figure *Sample Red-Black Tree*.

If any of the values u, v, w coincide with existing nodes, they should not be inserted. (Red-Black trees and BSTs in general can handle duplicates; but here we assume that it stores a map/set with unique keys.)

Show the intermediate steps – the tree after each successive inserted node. Clearly show, which are the red/black vertices in the submitted answers.

Note: Check that your inserts preserve the BST order invariant (along with all the Red-Black tree invariants). Secondly, try to follow the standard algorithm when inserting new nodes (still, preserving the invariants is more important).

Problem 4: Compare the following two implementations: (1) heaps, (2) AVL trees to implement the following ADTs. For each method find the worst-case (or amortized) time complexity $\Theta(g(n))$.

(A) Priority Queue ADT:

$Q = \text{newEmptyQueue}()$
$Q.\text{INSERT}(x)$
$x = Q.\text{DELETEMIN}()$
$x = Q.\text{FINDMIN}()$

(B) Predecessor/Successor ADT:

$S = \text{NEWEMPTYCONTAINER}()$
$S.\text{INSERT}(x)$
$S.\text{DELETE}(x)$
$y = S.\text{PREDECESSOR}(x)$ – return the reference to the next-smaller than x
$y = S.\text{SUCCESSOR}(x)$ – return the reference to the next-larger than x

Problem 5: Assume that you need to build a *range index* data structure R – this data structure is a database-like container where we can insert items x_i (such as page requests for Google Analytics). Each item x has some numeric key $x.k$ (the timestamp of the request or, perhaps, the milliseconds it took to compute the HTTP response). We need to query this data to draw nice graphs – e.g. display barcharts counting the number of requests in each value range or to find the total time spent for requests received during some time period or anything else.

(A) Pick some data-structure to support *range index* described above. Give the time estimate for $R.\text{INSERT}(x)$, $R.\text{DELETE}(x)$, $x = R.\text{FINDBYKEY}(k)$.

(B) Give the time estimate for $R.\text{FINDMIN}()$ and $R.\text{FINDMAX}()$ to find the minimum and the maximum key in the whole data structure R.

(C) Assume that the data structure also supports operation $R.rank(k)$ defined as the number of keys in the index that are smaller or equal to the given value k. Write a pseudocode to compute another operation – $R.\text{COUNT}(k_1, k_2)$ that returns the number of keys k in-between, i.e. satisfying $k_1 \leq k \leq k_2$. (The operation $R.\text{COUNT}(k_1, k_2)$ would be very useful to display barcharts for Google Analytics or similar aggregated data.)

(D) In order to implement $R.rank(k)$ from the previous task, you can augment the items x stored in R by storing additional numerical information (denoted by $x.\gamma$). Which kind of augmented information would you use? Consider the following options (plus any others you might need):

- the minimum key in the subtree rooted at node

- the maximum key in the subtree rooted at node

- the height of the subtree rooted at node

- the number of nodes in the subtree rooted at node

- the rank of node

- the sum of keys in the subtree roted at node

(E) Provide a way to compute $R.rank(k)$ from the values $x.\gamma$ you selected in the previous item.

Data Structures: Exam Topics

WORKSHEET 08: SORTING

Various sorting algorithms are introduced here, because they illustrate algorithm building paradigms (*Anany Levitin. Introduction to the design & analysis of algorithms*). Depending on the context different sorting algorithms may be needed.

8.1 Concepts and Facts

Statement (lower bound for sorting): Any general sorting algorithm receiving an input of n distinct items and producing a sorted output of these items (without assuming anything additional about the input) needs at least $\lceil \log_2 n! \rceil$ comparisons.

Proof: Any decision tree with k levels (i.e. a sorting algorithm making k comparisons, can distinguish at most 2^k different cases. On the other hand, there are $n!$ different ways how n sortable items can be arranged in the input. Thus we must have $2^k \geq n!$ or $k \geq \lceil \log_2 n! \rceil$. (See Comparison Sort.)

Definition: A sorting algorithm is called *stable* iff any two items with equal keys are not swapped: (and thus preserve their initial order). Algorithms that are not guaranteed to preserve such order are *unstable*.

Definition: A sorting algorithm that does not need to know the number of sortable items ahead of the time is called *online*. Those algorithms that receive full sortable array right at the start are called *offline*.

Definition: A sorting algorithm is called in-place, iff it uses just the original array to store the sortable items (plus some local variables). Sorting algorithms that need to allocate new memory for the sortable items are *outplace*.

MergeSort algorithm: This algorithm is a typical illustration of the Divide and Conquer paradigm.

MERGESORT(A, p, r):
1 **if** $p < r$
2      $q = \lfloor (p + r)/2 \rfloor$
3   MERGESORT(A, p, q)
4   MERGESORT($A, q + 1, r$)
5   MERGE(A, p, q, r)

MergeSort is outplace algorithm – it may waste memory, if run on large arrays. On the other hand, MergeSort is nearly optimal regarding the number of comparisons needed, so it may be helpful, if comparing two items takes much time.

Quicksort algorithm: QuickSort has several flavors; this is one of the easiest, but it sometimes needs one extra swap – see Line 13. This variant of Quicksort always uses the leftmost element of the input area as a pivot – it is easy to understand, but may be inefficient, if the input array is nearly sorted already. More advanced Quicksort flavors are randomized or choose the pivot differently.

QUICKSORT($A[\ell \ldots r]$):
1.   **if** $\ell < r$:
2. $i = \ell$ *(i increases from the left, searches elements $\geq$ than pivot)*
3. $j = r + 1$ *(j decreases from the right, searches elements $\leq$ than pivot.)*
4. $v = A[\ell]$ *(v is the pivot.)*
5.   **while** $i < j$:
6. $i = i + 1$
7.   **while** $i < r$ **and** $A[i] < v$:
8. $i = i + 1$
9. $j = j - 1$
10.   **while** $j > \ell$ **and** $A[j] > v$:
11. $j = j - 1$
12. $A[i] \leftrightarrow A[j]$
13. $A[i] \leftrightarrow A[j]$ *(Undo the extra swap at the end)*
14. $A[j] \leftrightarrow A[\ell]$ *(Move pivot to its proper place)*
15.   QUICKSORT($A[\ell \ldots j - 1]$)
16.   QUICKSORT($A[j + 1 \ldots r]$)

Quicksort algorithm has good characteristics for nearly all inputs – it sorts in place (does not need much memory), it usually is quite optimal and also easy to implement. It may behave badly for certain special inputs (for example, if the input array is nearly sorted already).

BUBBLESORT($A[0 \ldots n - 1]$):
1.   **do**:
2. $swapped = $ FALSE
3.   **for** i **in** RANGE($1, n$):   *(from 1 to $n - 1$ inclusive)*
4.   **if** $A[i - 1] > A[i]$:
5. $A[i - 1] \leftrightarrow A[i]$
6. $swapped = $ TRUE
7.   **while** $swapped$:

8.2 Problems

Problem 1:

(A) Find the $O(g(n))$ for the following function: $\log_2 n!$.

(B) What is the lower bound of comparisons needed to sort an array of 5 elements (assume they are all different)?

Problem 2: An array of 10 elements is used to initialize a minimum heap (as the first stage of the Heap sort algorithm):

$$\{5, 3, 7, 10, 1, 2, 9, 8, 6, 4\}$$

Assume that the minimum heap is initialized in the most efficient way (inserting elements level by level – starting from the bottom levels). All slots are filled in with the elements of the 10-element array in the order they arrive.

(A) How many levels will the heap tree have? (The root of the heap is considered L_0 – level zero. the last level is denoted by L_{k-1}. Just find the number k for this array.)

(B) Draw the intermediate states of the heap after each level is filled in. Represent the heap as a binary tree. (If some level L_k is only partially filled and contains less than 2^k nodes, please draw all the nodes as little circles, but leave the unused nodes empty.)

(C) What is the total count of comparisons ($a < b$) that is necessary to build the final minimum heap? (In this part you can assume the worst case time complexity – it is not necessarily achieved for the array given above.)

Answer:

(A) 10 elements need a tree with four levels (complete tree with 10 nodes). The last level L_3 will have just three nodes filled in.

(B) See the picture with all four stages of adding elements (unused slots are gray; the nodes that swap their places during the downheap operations are shown in pink).

Stage 1

Stage 2

Stage 3

Stage 4
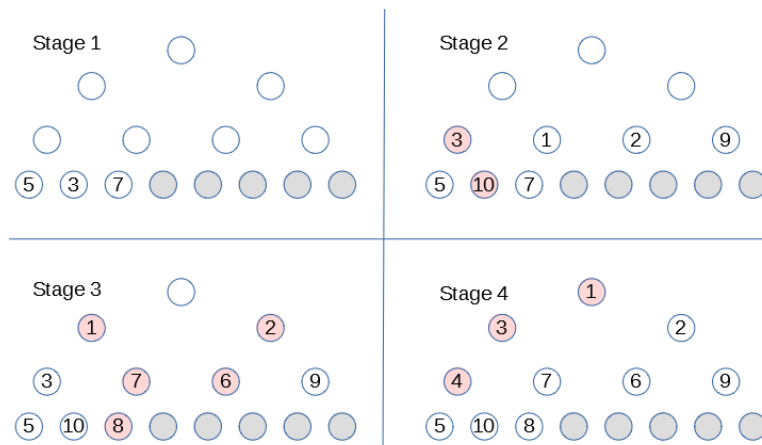
(C) In a downheap operation (when you add a new node on top of two other nodes), you first need to compare the two siblings, then compare their parent with the smallest of the two siblings (and if it is larger than its child, then swap). So every time some node moves one level down, you need to spend at most two comparisons.
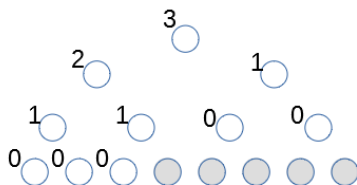
For our complete tree (with five grayed out slots in the last level), the worst case happens, if every node inserted at height h needs to spend $2h$ comparisons to travel to the very bottom (if we assume the worst case – that it is larger than everything that has been inserted so far). So the total number of comparisons is $2 \cdot (1+1+1+2+3) = 16$. In general, this time should grow as $O(n)$, where n is the number of items in the heap being built.

Problem 3:

(A) Run this pseudocode for one invocation QUICKSORT($A[0..11]$), where the table to sort is the following:

$$13, 0, 23, 1, 8, 9, 29, 16, 8, 24, 6, 11.$$

Draw the state of the array every time you swap two elements (i.e. execute $A[k_1] \leftrightarrow A[k_2]$ for any k_1, k_2).

(B) Continue with the first recursive call of QUICKSORT() (the original call QUICKSORT($A[0..11]$) is assumed to be the 0th call of this function). Draw the state of the array every time you swap two elements.

(C) Decide which is the second recursive call of QUICKSORT() and draw the state of the array every time you swap two elements. Show the end-result after this second recursive call at the very end.

Answer:

Your answer can be simple lists of numbers (without any grid lines or additional markings). Just try to keep the lists of numbers aligned.

(A) Swaps during the 0th call:

	0	1	2	3	4	5	6	7	8	9	10	11
	13	0	23	1	8_2	9	29	16	8_1	24	6	11
after Line 12	13	0	**11**	1	8_2	9	29	16	8_1	24	6	**23**
	13	0	11	1	8_2	9	**6**	16	8_1	24	**29**	23
	13	0	11	1	8_2	9	6	**8_1**	**16**	24	29	23
	13	0	11	1	8_2	9	6	**16**	**8_1**	24	29	23
Line 13	13	0	11	1	8_2	9	6	**8_1**	**16**	24	29	23
Line 14	**8_1**	0	11	1	8_2	9	6	**13**	16	24	29	23

(B) Since this example contains two elements equal to 8, we added subscripts to them (to show clearly, where every one is being swapped). As integer numbers they are fully identical to the Quicksort algorithm. (Still, the Quicksort algorithm does redundant swaps on them.)

Swaps during the first recursive call.

	0	1	2	3	4	5	6	7	8	9	10	11
	8_1	0	11	1	8_2	9	6	13	16	24	29	23
	8_1	0	**6**	1	8_2	9	**11**	13	16	24	29	23
	8_1	0	6	1	**8_2**	9	11	13	16	24	29	23
Line 13	8_1	0	6	1	**8_2**	9	11	13	16	24	29	23
Line 14	**8_2**	0	6	1	**8_1**	9	11	13	16	24	29	23

(C) Notice that the second recursive call happens within the first recursive call (sorting the left side of the left half).

Swaps during the second recursive call:

8_2	0	6	1	8_1	9	11	13	16	24	29	23

	8_2	0	6	**1**	8_1	9	11	13	16	24	29	23
Line 13	8_2	0	6	**1**	8_1	9	11	13	16	24	29	23
Line 14	**1**	0	6	8_2	8_1	9	11	13	16	24	29	23

Problem 4:

Consider the BubbleSort algorithm (see the beginning of the worksheet) for a 0-based array $A[0] \ldots A[n-1]$ of n elements.

(A) How many comparisons (`A[i-1] > A[i]`) in this algorithm are used to sort the given array. Show the state of the array after each `for` loop in the pseudocode is finished.

$$A[0] = 9,\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ A[9] = 8.$$

(B) How many comparisons (`A[i-1] > A[i]`) in this algorithm are used to sort the following array:

$$A[0] = 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ A[9] = 0.$$

Answer:

(A) 18 comparisons, 2 executions of the **for** loop:

After the first **for** loop the array is sorted:

$$A[0] = 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ A[9] = 9.$$

After the second **for** loop and 9 more comparisons no further swaps occur and the algorithm stops. The array is still the same:

$$A[0] = 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ A[9] = 9.$$

(B)

90 comparisons, 10 executions of the **for** loop:

After the first **for** loop:

$$A[0] = 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 0,\ A[9] = 9.$$

After the second **for** loop:

$$A[0] = 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 0,\ 8,\ A[9] = 9.$$

After the ninth **for** loop:

$$A[0] = 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ \ 8,\ A[9] = 9.$$

After the tenth **for** loop the array stays the same and the algorithm stops:

$$A[0] = 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ \ 8,\ A[9] = 9.$$

Note: Small values near the end of the list will slow down the Bubble sort considerably. The authors of an accelerated Bubble-sort variant (Comb sort) call such values *turtles*. See https://bit.ly/3mmS6C4.


Problem 5:

We have a 1-based array with 11 elements: $A[1], \ldots, A[11]$. We want to sort it efficiently. Run the MergeSort on this array (see the beginning of the worksheet).

Assume that initially you call this function as $\text{MERGESORT}(A,1,11)$, where $p = 1$ and $r = 11$ are the left and the right endpoint of the array being sorted (it includes both ends).

(A) What is the total number of calls to MERGESORT for this array (this includes the initial call as well as the recursive calls on lines 3 and 4 of this pseudocode).

(B) How many comparisons are needed (in the worst case) to sort an array of 11 items by the MergeSort algorithm?

(C) Evaluate $\log_2 11!$ using Stirlings formula or a direct computation. What is the theoretical lower bound on the number of comparisons to sort 11 items?

Answer:
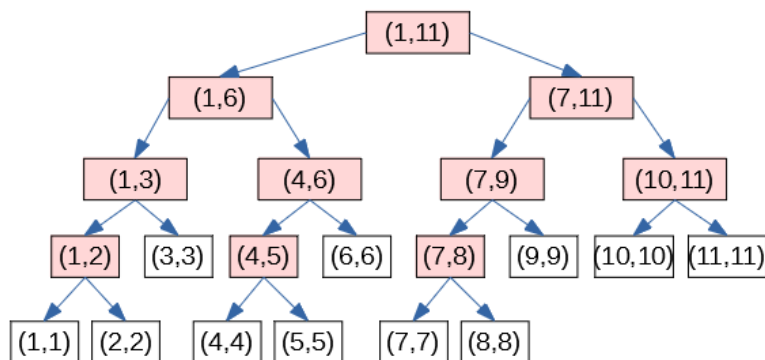
The recursive calls of MERGESORT are shown in the figure – just the parameters p, r for each call. For example, $\text{MERGESORT}(A, 1, 11)$ computes $q = \lfloor (1 + 11)/2 \rfloor = 6$, and causes two more calls to $\text{MERGESORT}(A, 1, 6)$ and $\text{MERGESORT}(A, 7, 11)$ respectively. On the other hand, if $p = r$, then the recursive calls do not happen (one-element list is already sorted). So there are exactly 11 external nodes (leaves) in the recursion tree.

Since the tree of calls is full, it also has 10 internal nodes (shown pink in the picture). The total number of these nodes is $10 + 11 = 21$.

Problem 6:

(A) Some algorithm receives n items as its input and then calls function $f(x_1, x_2, x_3, x_4)$ for any ordered quadruplet x_1, x_2, x_3, x_4 received in the input. Assume that $f(\ldots)$ runs in constant time. Find the time complexity of the whole algorithm.

(B) Some algorithm receives n items as its input and then calls a function f on all subsets of the received items having size $\lfloor n/4 \rfloor$. Assume that $f(\ldots)$ runs in constant time. Find the time complexity of the whole algorithm.

WORKSHEET 09: GRAPH TRAVERSALS

9.1 DFS in Oriented Graphs

Discovery/Finishing times DFS traversal algorithm can mark each vertex with two numbers `d`/`f`, where the first number `d` is the discovery time, and the second number `f` is the finishing time. All these numbers should be different and all of them belong to the interval $[1, 2N]$, where $N = |V|$ is the number of vertices.

Discovery/Finishing times can be stored in the graph nodes (augmented nodes); they are useful in various derived algorithms.

Edges after a DFS traversal DFS traversal turns all edges into four groups:

Tree edges Edges in the depth-first forest G_{DFS}. Edge (u, v) is a tree edge iff (u, v) was first discovered when u was gray (visited, not finished) and v was white (not yet visited).

Back edges Edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Edge is a back edge iff it was first discovered when u was gray and also v was gray (in process of DFS processing, not finished). Also self-loops, which may occur in directed graphs are considered to be back edges.

Forward edges Edges (u, v) that connect a vertex u to a descendant v in a depth-first tree (but they did not become tree edges – since v was first discovered through another path.

Cross edges All other edges – they can go between vertices in the same DFS tree as long as one vertex is not an ancestor of the other. Or they can go between vertices in different depth-first trees.

9.2 Topological Sorting

Definition Given an acyclic directed graph, a sequence of all its vertices $v_1, v_2, \ldots, v_N$ is called a *topological sorting* if for each edge (v_i, v_j) in this graph, vertex v_i precedes the vertex v_j.

Problem 1 (Topological Sorting) Consider the following graph:

(A) Run the DFS traversal algorithm on the graph shown in the figure, mark each vertex with two numbers `d`/`f`, where the first number `d` is the discovery time, and the second number `f` is the finishing time. Separate both numbers with a slash. All these numbers should be different and all of them belong to the interval $[1, 28]$, since the graph has 14 vertices.

If there are multiple ways how to pick a vertex to visit next in the DFS order, always pick the vertex with the alphabetically smallest label. Namely, your DFS traversal should start from the vertex A; every time there is a choice where to go deeper – pick the alphabetically first label not visited. Whenever the DFS traversal runs out of vertices to visit in a given discovery tree (but some nodes are still unvisited), pick the alphabetically smallest node as the root for the next DFS tree, and so on.
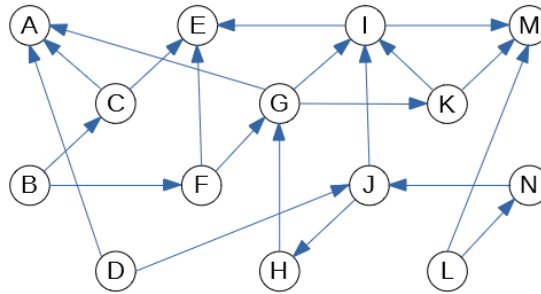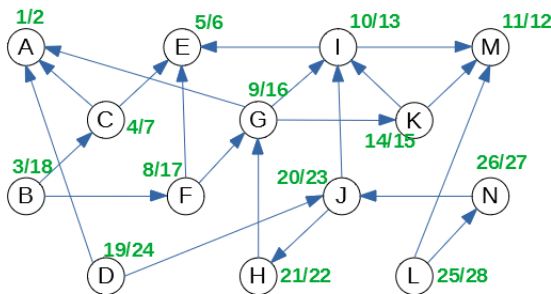
Fig. 1: Directed graph to run DFS and Topological Sorting

(B) In case if the graph shown in the Figure is not a DAG (directed acyclic graph), explain why it is not a DAG and remove some edge so that it becomes a DAG. On the other hand, if the graph in the Figure is already a DAG, explain why it is the case and do not remove any edges.

(C) Produce a topological sorting of the graph obtained in **(B)** – list the vertices in their topological sorting order.

Answer:

(A) The image shows the directed graph with DFS discovery/finishing times.

(B) DFS traversal turns all edges into four groups:

Tree edges Edges in the depth-first forest G_{DFS}. Edge (u, v) is a tree edge iff (u, v) was first discovered when u was gray (visited, not finished) and v was white (not yet visited).

Back edges Edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Edge is a back edge iff it was first discovered when u was gray and also v was gray (in process of DFS processing, not finished). Also self-loops, which may occur in directed graphs are considered to be back edges.

Forward edges Edges (u, v) that connect a vertex u to a descendant v in a depth-first tree (but they did not become tree edges – since v was first discovered through another path.

Cross edges All other edges – they can go between vertices in the same DFS tree as long as one vertex is not an ancestor of the other. Or they can go between vertices in different depth-first trees.

The image shows the same directed graph with edges classified into these four groups (tree edges are shown in bold; all the other edges are dashed – all of them turn out to be cross edges: Some go from one subtree to another; some others go from one tree to another tree).
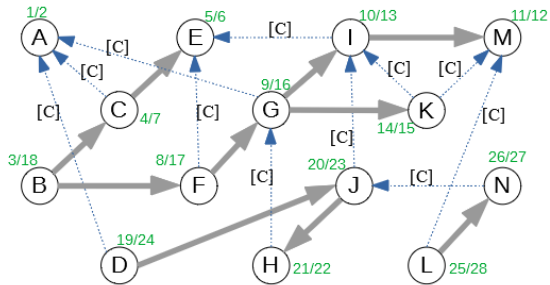
Statement: If some graph contains a cycle, then it must contain some back edge.

Proof: Let $v_0, v_1, \ldots, v_n = v_0$ be a cycle in the graph. DFS traversal eventually visits some vertex in this cycle (say, v_i where $i \in [0, \ldots, n]$). Since this vertex is the first vertex in this cycle to be visited, it starts the DFS discovery – and it will not finish processing v_i until it has visited all the vertices reachable from it. They include all the other vertices in the cycle. At some point (maybe, when discovering v_{i-1} or, maybe, even earlier) there will be an edge pointing back to v_i. This is a back edge by its definition (as it points back from the subtree to the root of this subtree v_i).

Our graph has no back edges, so it does not have any loops. Such graphs are called *directed acyclic graphs* (DAGs) and their vertices can be topologically sorted.

(C) The topological sorting can be obtained by listing the vertices in the reverse order of their finishing times. The largest finishing time is for the vertex L ($f = 28$), the smallest one is for vertex A ($f = 2$). Here is the full list topologically sorted:

$$L, N, D, J, H, B, F, G, K, I, M, C, E, A.$$

9.3 Strongly Connected Components

DFS traversal of a directed graph can be used to find strongly connected components – Kosarajus algorithm. https://bit.ly/3lI20ec, https://bit.ly/3mNU2la.

Definition: A subset of vertices in a directed graph $S \subseteq G.V$ makes a strongly connected component, iff for any two distinct vertices u, v there is a path $u \rightsquigarrow v$ (one or more and also another path $v \rightsquigarrow u$ that goes back from v to u.

If you can travel only in one direction (say, from u to v), but cannot return, then u, v should be in different strongly connected components. (Same thing, if u and v are mutually unreachable.) Every vertex is strongly connected to itself – in a graph with n vertices there are at most n strongly connected components.

Figure shows an example of a graph with $n = 5$ vertices having 3 strongly connected components. Next to that graph is the *transposed graph* G^T where all the edges are reversed.
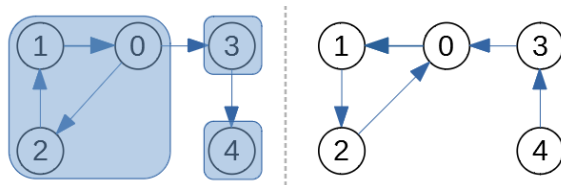
Kosarajus algorithm to find strongly connected components in an arbitrary graph by running DFS twice (i.e. it works

in linear time $O(n + m)$).

$$\text{STRONGLY\_CONNECTED}(G)$$

(compute all finishing times $u.f$)

1   call $\text{DFS}(G)$

(G^T is transposed G, all edges reversed)

2   compute G^T

(visit vertices in decreasing $u.f$ order)

3   call $\text{DFS}(G^T)$

4   **for each** tree T in the forest $\text{DFS}(G^T)$

5   Output T as a component

To see how this works, we can run it on the example graph shown earlier. After the DFS on graph G is run, we get the finishing times for the vertices $0, 1, 2, 3, 4$ (all shown in red on the left side of Figure below). After that we replace G by G^T (to the right side of the same figure), and assign priorities in the decreasing sequence of $u.f$ (the finishing times when running $\text{DFS}(G)$).
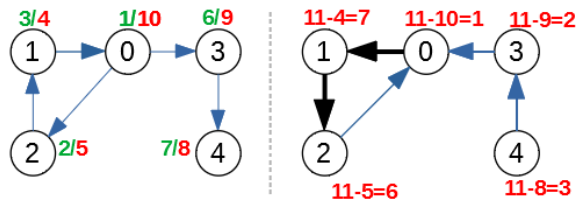
To make this reverse order obvious, we assign new priorities to the vertices in G^T. The new priorities in G^T are the following:

- Vertex 0 has priority $11 - 10 = 1$.

- Vertex 1 has priority $11 - 4 = 7$.

- Vertex 2 has priority $11 - 5 = 6$.

- Vertex 3 has priority $11 - 9 = 2$.

- Vertex 4 has priority $11 - 8 = 3$.

Now run $\text{DFS}(G^T)$. It turns out that the DFS algorithm starts in the vertex `"0"` once again (since it was finished last in $\text{DFS}(G)$). But unlike the DFS algorithm in G itself (it produced just one DFS tree), we get a DFS forest with 3 components (tree/discovery edges shown bold and black in the previous Figure).

- $\{0, 1, 2\}$ (DFS tree has root "0").

- $\{3\}$ (DFS tree has root "3").

- $\{4\}$ (DFS tree has root "4").

They represent the strongly connected components in G (they are also strongly connected in G^T).

Problem 2(Kosarajus algorithm) We start with the graph shown in Figure below.
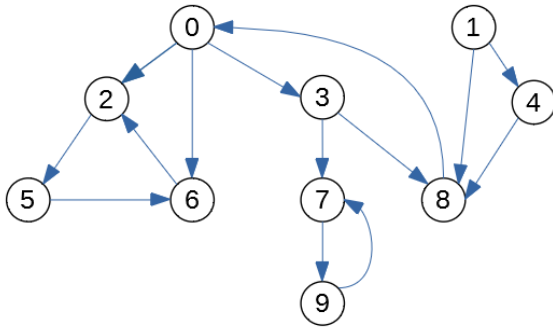
(A) Run the DFS traversal algorithm on the graph G. Mark each vertex with the pair of numbers d/f, where the first number d is the discovery time, and the second number f is the finishing time.

(B) Draw the transposed directed graph (same vertices, but each arrow points in the opposite direction). Run the DFS traversal algorithm on G^T. Make sure that the DFS outer loop visits the vertices in the reverse order by $u.f$ (the finishing time for the DFS algorithm in step (A)). In this case you do not produce the discovery/finishing times once again, just draw the discovery edges used by the DFS on G^T – you can highlight them (show them in bold or use a different color).

(C) List all the strongly connected components (they are the separate pieces in the forest obtained by running DFS on G^T).

9.4 Single-Source Shortest Paths

Definition Let $G(V, E)$ be an (undirected or directed) graph, where each edge is assigned a weight – some real number. One of the vertices $v \in V$ is selected as the source. The Single-Source Shortest Path problem finds the shortest paths between the given vertex v and all other vertices in the graph.

Examples: There are some well-known solutions to the single-source shortest paths problem in special cases:

- BFS (Breath-First-Search) by itself finds the shortest distances from the root to all the other vertices, if every edge has weight 1. Every time we discover a new vertex w (not visited by the BFS earlier) we assign its distance to the root $d_v(w)$ to be $d_v(u) + 1$, where $d_v(u)$ denotes the distance of its parent u.

- Dijkstras algorithm can be used to find the shortest distances from the root to all the other vertices, if every edge has a positive weight.

9.4.1 Positive Edge Weights

Dijkstras algorithm requires $O((m + n) \log_2 n)$ time, if we use priority queues; here $m = |E|$ is the number of edges and $n = |V|$ is the number of vertices in a graph.

In this example we do not implement a priority queue; assume that you can always pick the vertex with the smallest distance and add it to the set S of visited vertexes (those having distances already computed).

Example (Dijkstras Algorithm): We start with the graph shown in Figure below:

Vertex A will be your source vertex. (You can assume that the distance from A to itself is 0; initially all the other distances are infinite, but then Dijkstras algorithm relaxes them).

(A) Run the Dijkstras algorithm: At every phase write the current vertex v; the set of finished vertices and also a table showing the new distances to all A, B, C, D, E (and their parents) after the relaxations from v are performed. At the end of every phase highlight which vertex (among those not yet finished) has the minimum distance. This will become the current vertex in the next phase.
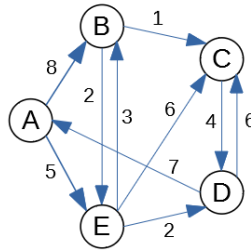
Fig. 2: Graph Diagram for Dijkstras Algorithm

(B) After the algorithm finishes, summarize the answer: For each of the five vertices tell what is its minimum distance from the source. Also show what is the shortest path how to achieve that minimum distance.

Solution Draw Dijkstras algorithm step by step; show results in tables.

(A) At every phase we select the minimum-distance vertex in the priority queue of vertices (not yet added to the set of finished vertices S). This becomes the current vertex v. After that we relax all the edges that go out from the current vertex v (if some distance decreases, we change the parent of this new vertex to become v).
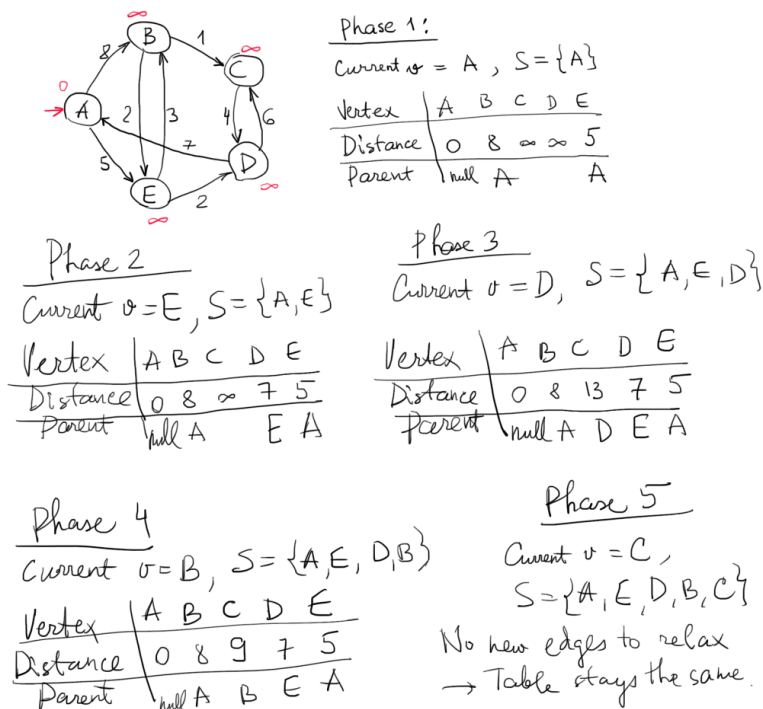

Fig. 3: **(A)** – Five Phases of Dijkstras Algorithm

(B) The result of Dijkstras algorithm can be summarized as shown below. For each vertex we specify the distance from A to that vertex (and also what is the shortest path to achieve it).

Vertex	Distance	Path
A	$d(A, A) = 0$	A
B	$d(A, B) = 8$	$A \to B$
C	$d(A, C) = 9$	$A \to B \to C$
D	$d(A, D) = 7$	$A \to E \to D$
E	$d(A, E) = 5$	$A \to E$

9.4.2 Negative Edge Weights

The Bellman-Ford algorithm solves the single source shortest paths problem in the case in which edge weights may be negative. It can work with directed graphs (and also undirected graphs; not discussed in this exercise). The algorithm initializes the distances to all the vertices u by $u.d = +\infty$. The only exception is the *source vertex*) which gets distance $s.d = 0$ (the distance to itself is 0).

After this initialization in a graph with n vertices it will perform $n-1$ identical iterations. In every iteration it considers all the edges in some order, and relaxes all the edges. After that you can perform one last iteration with Bellman-Ford algorithm: If there are still relaxations that reduce distances even after n steps, this means that there is a negative loop in the original graph (and the shortest paths are not possible to compute as the distances can be reduced infinitely).

Let $G(V, E)$ be a directed graph. Let $w : E \to \mathbf{Z}$ be a function assigning integer weights to all the graphs edges and let $s \in V$ be the source vertex. Every vertex $v \in V$ stores $v.d$ – the current estimate of the distance from the source. A vertex also stores $v.p$ – its parent (the last vertex on the shortest path before reaching v). Bellman-Ford algorithm to find the minimum distance from s to all the other vertices is given by the following pseudocode:

$\text{BELLMANFORD}(G, w, s)$:
    for each vertex $v \in V$:   *(initialize vertices to run shortest paths)*
        $v.d = \infty$
        $v.p = \text{NULL}$
    $s.d = 0$ *(the distance from source vertex to itself is 0)*
    for $i = 1$ **to** $|V| - 1$ *(repeat $|V| - 1$ times)*
        for each edge $(u, v) \in E$
            if $v.d > u.d + w(u, v)$:   *(relax an edge, if necessary)*
                $v.d = u.d + w(u, v)$
                $v.p = u$

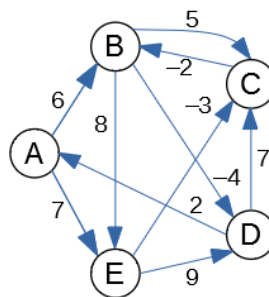**Example (Bellman-Ford):** Consider the graph in Figure:

Fig. 4: Graph Diagram for Bellman Ford Algorithm

Let us pick vertex B as the *source vertex* for Bellman-Ford algorithm. (You could pick the source vertex differently, but then all the distance computations would be different as well.)

(A) Create a table showing all the changes to all the distances to A, B, C, D, E as the relaxations are performed. In a single iteration the same distance can be relaxed/improved multiple times (and you can use distances computed in the current phase to relax further edges). The table should display all $n-1$ iterations (where $n = 5$ is the number of vertices). (*Sometimes it is worth running one more iteration to find possible negative loops*).

Note: Please make sure to release the edges in the alphabetical/lexicographical order: Regardless of which is your source, in every iteration the edges are always relaxed in this order:

$$AB, AE, BD, BE, CB, DA, DC, EC, ED.$$

In fact, any order can work; the only thing that matters is that you consider all the edges. But alphabetical ordering of edges makes the solution deterministic.

(B) Summarize the result: For each of the 5 vertices tell what is its minimum distance from the source. Also tell what is the shortest path how to get there. For example, if your source is E then you could claim that the shortest path $E \rightsquigarrow B$ is of length -5 and it consists of two edges $(E, C), (C, B)$.

Solution Show the Bellman-Ford algorithm in stages; results are shown in tables.

(A) In this case we only need to run three phases (not $n-1 = 4$ phases), since all the distances become stable and do not change anymore after Phase 3. The tables show only those relaxed edges that lead to decreased distances.
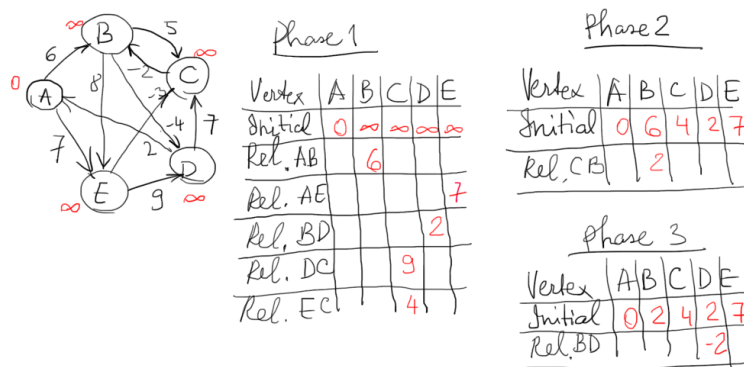

Fig. 5: **(A)** – Phases of Bellman-Fords Algorithm

(B) The result of Bellman-Fords algorithm can be summarized as shown below. For each vertex we specify the distance from A to that vertex (and also what is the shortest path to achieve it).

Vertex	Distance	Path
A	$d(A, A) = 0$	A
B	$d(A, B) = 2$	$A \to E \to C \to B$
C	$d(A, C) = 4$	$A \to E \to C$
D	$d(A, D) = -2$	$A \to E \to C \to B \to D$
E	$d(A, E) = 7$	$A \to E$

Problem 3 (Bellman-Ford) Consider the input graph shown in Fig.1.
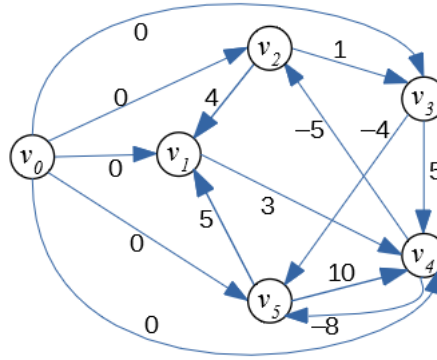

Fig. 6: A directed graph for Bellman-Ford Algorithm

(A) In your graph use the vertex $s = v_0$ as the *source vertex* for Bellman-Ford algorithm. Create a table showing the changes to all the distances to the vertices of the given graph every time a successful edge relaxing happens and some distance is reduced. You should run $n-1$ phases of the Bellman-Ford algorithm (where n is the number of vertices). You can also stop earlier, if no further edge relaxations can happen.

Note: Please make sure to release the edges in the lexicographical order. For example, in a single phase the edge (v_1, v_4) is relaxed before the edge (v_2, v_1), since v_1 precedes v_2.

(B) Summarize the result: For each vertex tell what is its minimum distance from the source. Also tell what is the shortest path how to get there.

(C) Does the input graph contain negative cycles? Justify your answer.

Answer:

(A) Phase 1:

Vertices	v_0	v_1	v_2	v_3	v_4	v_5
Initial distances	0	∞	∞	∞	∞	∞
Relax (v_0, v_1)	0	0	∞	∞	∞	∞
Relax (v_0, v_2)	0	0	0	∞	∞	∞
Relax (v_0, v_3)	0	0	0	0	∞	∞
Relax (v_0, v_4)	0	0	0	0	0	∞
Relax (v_0, v_5)	0	0	0	0	0	0
Relax (v_3, v_5)	0	0	0	0	0	-4
Relax (v_4, v_2)	0	0	-5	0	0	-4
Relax (v_4, v_5)	0	0	-5	0	0	-8
Relax (v_5, v_1)	0	-3	-5	0	0	-8

Phase 2:

Vertices	v_0	v_1	v_2	v_3	v_4	v_5
Relax (v_2, v_3)	0	-3	-5	-4	0	-8

Further phases cannot relax any new edges, so these distances are considered final.

(B) We list the shortest paths from v_0 to all the vertices.

- Distance $d(v_0, v_0) = 0$, path (v_0) has 0 edges and weight 0.

- Distance $d(v_0, v_1) = -3$, path $(v_0 \to v_4 \to v_5 \to v_1)$ has 3 edges and weight $0 + (-8) + 5 = -3$.

- Distance $d(v_0, v_2) = -5$, path $(v_0 \to v_4 \to v_2)$ has 2 edges and weight $0 + (-5) = 5$.

- Distance $d(v_0, v_3) = -4$, path $(v_0 \to v_4 \to v_2 \to v_3)$ has 3 edges and weight $0 + (-5) + 1 = -4$.

- Distance $d(v_0, v_4) = 0$, path $(v_0 \to v_4)$ has 1 edge and weight 0.

- Distance $d(v_0, v_5) = -8$, path $(v_0 \to v_4 \to v_5)$ has 2 edges and weight $0 + (-8) = -8$.

(C) Graph G does not contain negative cycles – otherwise the edge relaxation would continue in Phases 2, 3, and so on.

Note: Just a little modification: $w(v_4, v_2) = -6$ (instead of -5) yields a negative loop: $v_4 \to v_2 \to v_3 \to v_5 \to v_1 \to v_4$ or $(-6) + (1) + (-4) + (5) + (3)$. If we run Bellman-Ford algorithm on such a graph, then relaxing edges does not end after $|V| - 1$ iterations, the minimum distances decrease further and can become negative numbers with arbitrarily large absolute values.

9.5 Minimum Spanning Trees

Definition Let $G(V, E)$ be a connected undirected graph where each edge is assigned a non-negative weight. A *minimum spanning tree* is a subset of edges $MST \subseteq E$ that keeps graph connected, and the total weight of all the edges in MST is the smallest possible.

Prims Algorithm Let $G(V, E)$ be an *undirected* graph. Let $w : E \to \mathbf{Z}$ be a function assigning integer weights to all the graphs edges and let r be the root vertex that will start to grow the minimum spanning tree (MST). Every vertex $v \in V$ stores $v.key$ – the key for a priority queue (initially containing all the vertices). A vertex also stores $v.p$ – its parent (the parent vertex in the ultimate MST; it is assigned only once). Prims algorithm to find the minimum spanning tree in G is given by the following pseudocode:

$\text{MSTPRIM}(G, w, r)$:
    for each vertex $u \in V$:
        $u.key = \infty$
        $u.p = \text{NULL}$
    $r.d = 0$
    $Q = \text{MINIMUMHEAP}(V)$ *(Insert all vertices in a priority queue)*
    while $Q \neq \emptyset$:
        $u = \text{EXTRACTMIN}(Q)$ *(pick a vertex closest to the MST built so far)*
        for each $v \in \text{ADJ}(G, u)$:
            if $v \in Q$ **and** $w(u, v) < v.key$
                $v.p = u$
                $v.key = w(u, v)$

It is an efficient algorithm; it requires $O((m + n) \log_2 n)$ time, if we use priority queues as heaps.

Kruskals Algorithm You start out with a bunch of one-node isolated components. At each step you pick the cheapest edge between any two components and join them together. Let F denote the *forest* containing the little trees used to build the MST. Here is the pseudocode:

KRUSKAL(G)
$\quad F = \emptyset$
$\quad$ **for each** $v \in G.V$:
$\quad\quad$ makeSet(v)
$\quad$ **for each** $(u, v) \in G.E$ ordered by $weight(u, v)$ increasing:
$\quad\quad$ **if** FINDSET$(u) \neq$ FINDSET(v):
$\quad\quad\quad F = F \cup \{(u, v)\} \cup \{(v, u)\}$
$\quad\quad\quad$ UNION(FINDSET(u), FINDSET(v))
$\quad$ **return** F

Example for MSTs We start with the graph shown in Figure:
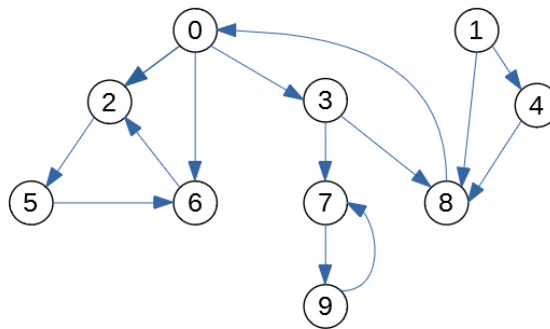
Fig. 7: Graph Diagram for Prims Algorithm.

(A) Vertex A will be your source vertex. It is the first vertex added to the MST vertice set S. At every step you find the lightest edge that connects some vertex in S to some vertex not in S. Add this new vertex to a graph and remember the edge you added. Show how the Prims MST (Minimum Spanning Tree grows) one edge at a time.

Note: In cases when there is a choice between multiple lightest edges of the same weight, pick the edge (v, w) with $v \in S$ and $w \notin S$ such that (v, w) lexicographically precedes any other lightest edge.

(B) Redraw the graph, highlight the edges selected for MST (make them bold or color them differently). Add up the total weight of the obtained MST and write this in your answer (it should be the minimum value among all the possible spanning trees in this graph).

Solution We show the subsequent steps of Prims algorithm.

(A) At each step we show the current set of vertices in MST (denoted by S) and which edge is being added.

1. $S = \{A\}$, adding edge AB

2. $S = \{A, B\}$, adding edge BH

3. $S = \{A, B, H\}$, adding edge HG

4. $S = \{A, B, G, H\}$, adding edge GF

5. $S = \{A, B, F, G, H\}$, adding edge FC

6. $S = \{A, B, C, F, G, H\}$, adding edge CI

7. $S = \{A, B, C, F, G, H, I\}$, adding edge FE

8. $S = \{A, B, C, E, F, G, H, I\}$, adding edge CD

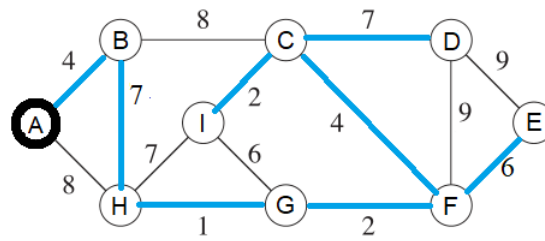**(B)** Solution shows the MST edges added in previous step colored blue:

Fig. 8: MST obtained by Prims Algorithm.

The total weight of this MST is $4 + 7 + 1 + 2 + 4 + 2 + 6 + 7 = 33$. (*In this case the MST is unique. In general case there is no guarantee that there are no other MSTs of the same weight, but the one we found with Prims algorithm is among the lightest ones.*)

Example Continued Run Kruskals algorithm on the same graph as before.

(A) After each step when there is an edge connecting two sets of vertices, write that edge and show the partition where that edge connects two previously disjoined pieces in the forest of trees.

Note:

If there are multiple lightest edges that can be used to connect two disjoined pieces, pick edge (v, w) which lexicographically precedes any other.

(B) Redraw the given graph (show the order how you added the edges in parentheses). Also compute the total weight of this MST.

Solution Here is the Kruskals algorithm showing node clusters:

(A) We list the steps that add edges and join two previously disconnected pieces:

1. Add edge GH, the partition becomes $\{A, B, C, D, E, F, GH, I\}$.

2. Add edge CI, the partition becomes $\{A, B, CI, D, E, F, GH\}$.

3. Add edge FG, the partition becomes $\{A, B, CI, D, E, FGH\}$.

4. Add edge AB, the partition becomes $\{AB, CI, D, E, FGH\}$.

5. Add edge CF, the partition becomes $\{AB, CFGHI, D, E\}$.

6. Add edge FE, the partition becomes $\{AB, CEFGHI, D\}$.

7. Add edge BH, the partition becomes $\{ABCEFGHI, D\}$.

8. Add edge CD, the partition becomes $\{ABCEFGHID\}$.

(B) Solution shows the MST edges added in previous step colored blue. The total weight is 33. The order of their addition is shown in red in parentheses.

Note: In some cases Prims and Kruskals algorithm can yield different MSTs even for the same input graph, but they are both optimal in such cases.
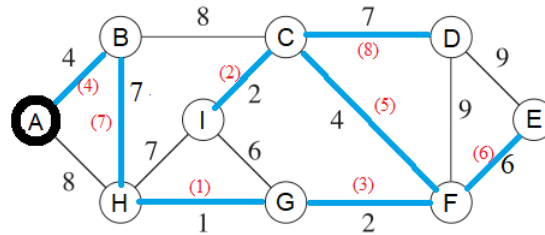

Fig. 9: MST obtained by Kruskals Algorithm.

Problem 4 (Prims Algorithm): Denote the last three digits of your Student ID by a, b, c. Student ID often looks like this: 201RDBabc, where a, b, c are digits. Compute three more digits x, y, z:
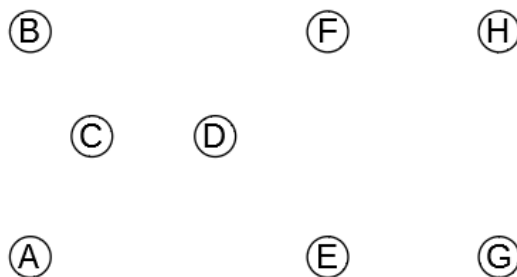
$$\begin{cases} x = (b + 4) \bmod 10 \\ y = (c + 4) \bmod 10 \\ z = (a + b + c) \bmod 10 \end{cases}$$

In this task the input graph $G = (V, E)$ is given by its adjacency matrix:

$$M_G = \begin{pmatrix} 0 & 0 & 5 & 8 & y & 0 & 0 & 0 \\ 0 & 0 & 3 & 7 & 0 & z & 0 & 0 \\ 5 & 3 & 0 & 3 & 0 & 0 & 0 & 0 \\ 8 & 7 & 3 & 0 & 1 & 7 & 0 & 0 \\ y & 0 & 0 & 1 & 0 & 6 & 9 & 6 \\ 0 & z & 0 & 7 & 6 & 0 & x & 2 \\ 0 & 0 & 0 & 0 & 9 & x & 0 & 7 \\ 0 & 0 & 0 & 0 & 6 & 2 & 7 & 0 \end{pmatrix}.$$

(A) Draw the graph as a diagram with nodes and edges. Replace x, y, z with values calculated from your Student ID. Label the vertices with letters A, B, C, D, E, F, G, H (they correspond to the consecutive rows and columns in the matrix).

If you wish, you can use the following layout (edges are not shown, but the vertice positions allow to draw the edges without much intersection). But you can use any other layout as well.

(B) Run Prims algorithm to find MST using $r = A$ as the root. If you do not have time to redraw the graph many times, just show the table with $v.key$ values after each phase. (No need to show $v.p$, as the parents do not change and they are easy to find once you have the final rooted tree drawn.) The top of the table would look like this (it shows Phase 0 – the initial state before any edges have been added).

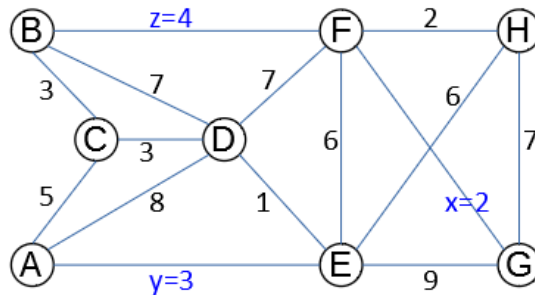| Phase	A	B	C	D	E	F	G	H
0 (initial state)	0	∞	∞	∞	∞	∞	∞	∞

(C) Summarize the result: Draw the MST obtained as the result of Prims algorithm, find its total weight.

Answer:

(A) As an example, consider Student ID with these last 3 digits: $(a, b, c) = (7, 8, 9)$. Compute the values of x, y, z:

$$\begin{cases} x = (b + 4) \bmod 10 = 2, \\ y = (c + 4) \bmod 10 = 3, \\ z = (a + b + c) \bmod 10 = 4. \end{cases}$$

Let us draw the graph with edges (including those labeled by $x = 2$, $y = 3$, $z = 4$).


(B) We show the values of vertices in the priority queue (the value shows the minimum distance to some vertex in the tree built so far using Prims algorithm). The vertices that have been extracted (removed) from the queue show $-$ instead of the key value.

Phase	A	B	C	D	E	F	G	H
0 (initial state)	0	∞	∞	∞	∞	∞	∞	∞
1 (extract A)	–	∞	5	8	3	∞	∞	∞
2 (extract E)	–	∞	5	1	–	6	9	6
3 (extract D)	–	7	3	–	–	6	9	6
4 (extract C)	–	3	–	–	–	6	9	6
5 (extract B)	–	–	–	–	–	4	9	6
6 (extract F)	–	–	–	–	–	–	2	2
7 (extract G)	–	–	–	–	–	–	–	2
8 (extract H)	–	–	–	–	–	–	–	–

Note: During Phase 7 there are two vertices of the same priority value (G and H). We select G which is alphabetically earlier and add the edge (F, G) to the MST. In this situation the order how edges (F, G) and (F, H) does not matter much. In some other graphs the choice of edges (for equal keys in the priority queue) can lead to major differences in subsequent steps – and there may exist multiple MSTs.

(C) Every time we extract a vertex from the priority queue in (B), we set its parent to the other end of the edge that was safe to add at this step. Below is the picture of the resulting MST (tree edges shown bold and light-gray, all the other edges are shown as dashed).
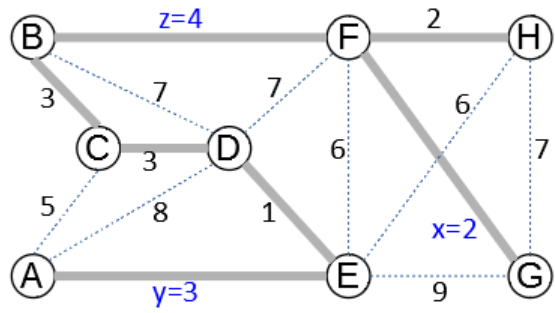
WORKSHEET 10: SHORTEST PATHS AND MSTS

10.1 Shortest Paths: Dijkstra

(Drozdek2013, p.400) and (Goodrich2011, p.640) both define Dijkstras algorithm. See also https://bit.ly/2JSXqMU. It is an efficient algorithm; it requires $O((m + n) \log_2 n)$ time, if we use priority queues; here $m = |E|$ is the number of edges and $n = |V|$ is the number of vertices in a graph.

In this exercise you do not need to implement a priority queue; assume that you can always pick the vertex with the smallest distance and add it to the set S of visited vertexes (those having distances already computed).

10.1.1 Problem

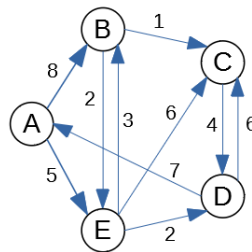We start with the graph shown in Figure below:

Fig. 1: Graph Diagram for Dijkstras Algorithm

Vertex A will be your source vertex. (You can assume that the distance from A to itself is 0; initially all the other distances are infinite, but then Dijkstras algorithm relaxes them).

(A) Run the Dijkstras algorithm: At every phase write the current vertex v; the set of finished vertices and also a table showing the new distances to all A, B, C, D, E (and their parents) after the relaxations from v are performed. At the end of every phase highlight which vertex (among those not yet finished) has the minimum distance. This will become the current vertex in the next phase.

(B) After the algorithm finishes, summarize the answer: For each of the five vertices tell what is its minimum distance from the source. Also show what is the shortest path how to achieve that minimum distance.

10.2 Shortest Paths: Bellman-Ford

Let $G(V, E)$ be a directed graph. Let $w : E \to \mathbf{Z}$ be a function assigning integer weights to all the graphs edges and let $s \in V$ be the source vertex. Every vertex $v \in V$ stores $v.d$ – the current estimate of the distance from the source. A vertex also stores $v.p$ – its parent (the last vertex on the shortest path before reaching v). Bellman-Ford algorithm to find the minimum distance from s to all the other vertices is given by the following pseudocode:

BELLMANFORD(G, w, s):
    for each vertex $v \in V$:   *(initialize vertices to run shortest paths)*
        $v.d = \infty$
        $v.p = \text{NULL}$
    $s.d = 0$ *(the distance from source vertex to itself is 0)*
    for $i = 1$ **to** $|V| - 1$ *(repeat $|V| - 1$ times)*
        for each edge $(u, v) \in E$
            if $v.d > u.d + w(u, v)$:   *(relax an edge, if necessary)*
                $v.d = u.d + w(u, v)$
                $v.p = u$

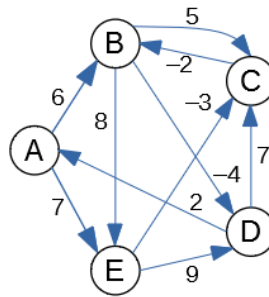**Question 2:** Consider the graph in Figure:

Fig. 2: Graph Diagram for Bellman Ford Algorithm

Let us pick vertex B as the *source vertex* for Bellman-Ford algorithm. (You could pick the source vertex differently, but then all the distance computations would be different as well.)

(A) Create a table showing all the changes to all the distances to A, B, C, D, E as the relaxations are performed. In a single iteration the same distance can be relaxed/improved multiple times (and you can use distances computed in the current phase to relax further edges). The table should display all $n - 1$ iterations (where $n = 5$ is the number of vertices). (*Sometimes it is worth running one more iteration to find possible negative loops*).

Note: Please make sure to release the edges in the alphabetical/lexicographical order: Regardless of which is your source, in every iteration the edges are always relaxed in this order:

$$AB, AE, BD, BE, CB, DA, DC, EC, ED.$$

In fact, any order can work; the only thing that matters is that you consider all the edges. But alphabetical ordering of edges makes the solution deterministic.

(B) Summarize the result: For each of the 5 vertices tell what is its minimum distance from the source. Also tell what is the shortest path how to get there. For example, if your source is E then you could claim that the shortest path $E \rightsquigarrow B$ is of length -5 and it consists of two edges $(E, C), (C, B)$.

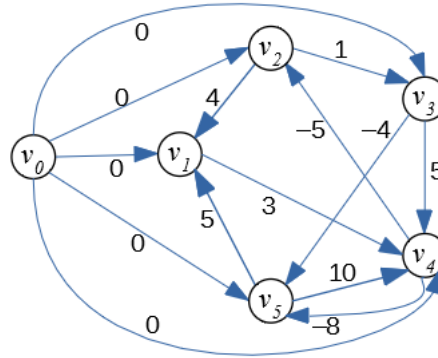**Question 3:** In this task the input graph is shown in the figure below.

Fig. 3: A directed graph for Bellman-Ford Algorithm

(A) In your graph use the vertex $s = v_0$ as the *source vertex* for Bellman-Ford algorithm. Create a table showing the changes to all the distances to the vertices of the given graph every time a successful edge relaxing happens and some distance is reduced. You should run $n-1$ phases of the Bellman-Ford algorithm (where n is the number of vertices). You can also stop earlier, if no further edge relaxations can happen.

Note: Please make sure to release the edges in the lexicographical order. For example, in a single phase the edge (v_1, v_4) is relaxed before the edge (v_2, v_1), since v_1 precedes v_2.

(B) Summarize the result: For each vertex tell what is its minimum distance from the source. Also tell what is the shortest path how to get there.

(C) Does the input graph contain negative cycles? Justify your answer.

10.3 Minimum Spanning Trees

(Goodrich2011, p.651) defines Prims algorithm. It finds a minimum spanning tree in an undirected graph with given edge weights. See also https://bit.ly/2VLz3DK. It is an efficient algorithm; it requires $O((m + n) \log_2 n)$ time, if we use priority queues. In this exercise you do not need to implement a priority queue; assume that you can always compute the minimums in your head and grow the MST accordingly.

10.3.1 Problem

Question 4 (Prims algorithm): Prims algorithm for the graph shown in Figure:

(A) Vertex A will be your source vertex. It is the first vertex added to the MST vertice set S. At every step you find the lightest edge that connects some vertex in S to some vertex not in S. Add this new vertex to a graph and remember the edge you added. Show how the Prims MST (Minimum Spanning Tree grows) one edge at a time.
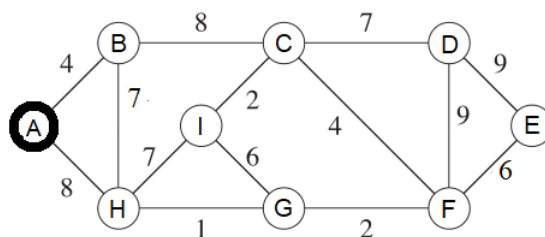
Fig. 4: Graph Diagram for Prims Algorithm.

Note: In cases when there is a choice between multiple lightest edges of the same weight, pick the edge (v, w) with $v \in S$ and $w \notin S$ such that (v, w) lexicographically precedes any other lightest edge.

(B) Redraw the graph, highlight the edges selected for MST (make them bold or color them differently). Add up the total weight of the obtained MST and write this in your answer (it should be the minimum value among all the possible spanning trees in this graph).

Question 5 (Prims algorithm): Denote the last three digits of your Student ID by a, b, c. Student ID often looks like this: 201RDBabc, where a, b, c are digits. Compute three more digits x, y, z:
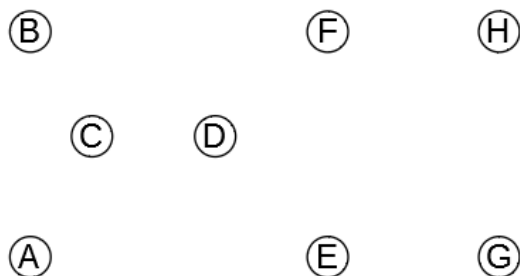
$$
\begin{cases}
x = (b + 4) \bmod 10 \\
y = (c + 4) \bmod 10 \\
z = (a + b + c) \bmod 10
\end{cases}
$$

In this task the input graph $G = (V, E)$ is given by its adjacency matrix:

$$
M_G =
\begin{pmatrix}
0 & 0 & 5 & 8 & y & 0 & 0 & 0 \\
0 & 0 & 3 & 7 & 0 & z & 0 & 0 \\
5 & 3 & 0 & 3 & 0 & 0 & 0 & 0 \\
8 & 7 & 3 & 0 & 1 & 7 & 0 & 0 \\
y & 0 & 0 & 1 & 0 & 6 & 9 & 6 \\
0 & z & 0 & 7 & 6 & 0 & x & 2 \\
0 & 0 & 0 & 0 & 9 & x & 0 & 7 \\
0 & 0 & 0 & 0 & 6 & 2 & 7 & 0
\end{pmatrix}.
$$

(A) Draw the graph as a diagram with nodes and edges. Replace x, y, z with values calculated from your Student ID. Label the vertices with letters A, B, C, D, E, F, G, H (they correspond to the consecutive rows and columns in the matrix).

If you wish, you can use the following layout (edges are not shown, but the vertice positions allow to draw the edges without much intersection). But you can use any other layout as well.

(B) Run Prims algorithm to find MST using $r = A$ as the root. If you do not have time to redraw the graph many times, just show the table with $v.key$ values after each phase. (No need to show $v.p$, as the parents do not change and they are easy to find once you have the final rooted tree drawn.) The top of the table would look like this (it shows Phase 0 – the initial state before any edges have been added).

Phase	A	B	C	D	E	F	G	H
0 (initial state)	0	∞	∞	∞	∞	∞	∞	∞

(C) Summarize the result: Draw the MST obtained as the result of Prims algorithm, find its total weight.

Question 6: Run Kruskals algorithm on the same graph as in the Question 4.

(A) After each step when there is an edge connecting two sets of vertices, write that edge and show the partition where that edge connects two previously disjoined pieces in the forest of trees.

Note: If there are multiple lightest edges that can be used to connect two disjoined pieces, pick edge (v, w) which lexicographically precedes any other.

(B) Redraw the given graph (show the order how you added the edges in parentheses). Also compute the total weight of this MST.

WORKSHEET 11: HASHTABLES

A hash function $\mathcal{U} \to \{0, 1, \ldots, m-1\}$ maps the universe $\mathcal{U}$ of keys into m non-negative integers (the size of the hashtable). It is used for map ADT (insert, delete and exact search). They are used for efficient storing maps and sets.

11.1 Concepts and Facts

Hash function should follow the following guidelines:

- Keys are discrete objects having a discrete representation (large integers, strings, lists and other data structures). Hash function could be defined on float variables as well, but it is **not** a real-valued function: it depends on the way the real values are encoded.

- The hash of an object must not change during its lifetime (either the hashable object is immutable or the hash function depends only on immutable attributes).

- `a == b` implies `hash(a) == hash(b)`. (The reverse might fail during a hash collision).

- A hash function should be fast. It is often assumed to be constant-time $O(1)$; sometimes it is linear-time $O(|k|)$ for longer input keys k.

Simple Uniform Hashing Assumption: Under this assumption a key maps to any slot $\{0, \ldots, m-1\}$ with the same probability, independent from the hashes of all the other keys.

This can be approximated by randomized seed to the hashing algorithm; hash function should map similar, but non-identical objects far from each other. (People sharing the same lastname or birth year should have considerably different hashes.)

Note: Hashing for datastructures (hashtables) does not use *cryptographic hashing*. Cryptographic hashes should create a unique *fingerprint* for the object being hashed. Examples include SHA-256 and other algorithms in SHA-2 family (MD5 no longer considered safe).

In practice most hash functions are computed as a composition $h(obj) = h_2(h_1(obj))$:

- $N = h_1(obj)$ is a *prehash function* which takes objects of various types and returns large integers.

- $h_2(N)$ is usually a modular division to get a nonnegative integer within the desired range.

In Python the prehash function is named simply `hash(...)`; it returns signed 64-bit integers. It can take different argument types. For small integer arguments it returns the integer itself; larger integer objects are hashed to fit into 64-bits. For example, you can run the Python environment on Linux like this (tested on Ubuntu command-line):

```
export PYTHONHASHSEED=0
python
```

On Windows Powershell (such as Anaconda terminal) run Python environment like this:

```
$Env:PYTHONHASHSEED=0
```

On Windows regular terminal run Python environment like this:

```
set PYTHONHASHSEED=0
python
```

```
>>> hash(10**18)
1000000000000000000
>>> hash(10**19)
776627963145224196
>>> hash('\x61\x62\x63')
5573379127532958270
>>> hash('abc')  # 'abc' is same string as '\x61\x62\x63'
5573379127532958270
>>> hash(3.14)
322818021289917443
>>> hash((1,2))
-3550055125485641917
```

Hash Collisions: Finding hash collisions can sometimes be used to slow down Python-related data structures such as dictionaries. Here is advice how to find hash collisions efficiently: https://bit.ly/3nf4bdk. Instead of looping until two hash values will collide, the approach uses some reverse enginering and also Meet in the Middle which reduces the number of checks to be made. In an unsophisticated Python implementation of a Web application one could cause massive hash collisions to stage denial of service attacks. Since Python 3.2 the hash computations are randomized with a seed (PYTHONHASHSEED environment variable); such attacks are now much harder. See https://bit.ly/3CeAVYi on how these collisions affect other programming languages.

11.1.1 String Matching Problem

Definition: We have a text T and a pattern P. We need to find all the offsets where the pattern P occurs in the text.

Naive String Matching: Here is a straightforward solution that finds all offsets:

NAIVESTRINGMATCHING(T, P):
    $n = len(T)$
    $m = len(P)$
    for $s = 0$ **to** $n - m$:
        if $P[0 : m - 1] == T[s : (s + m - 1)]$:
            output Pattern found at offset, s

We can find worst-case behavior. Consider the following pattern and text:

$$\begin{cases} P = \texttt{aaab} \\ T = \texttt{aaaaaaaaaaaaaa} \end{cases}$$

You will end up comparing all four times for any offset $s = 0, \ldots, 10$. The total number of comparisons is $4 \cdot 11 = 44$. In general the number of comparisons is $O(m \cdot (n - m + 1)) = O(m \cdot n)$.

Rabin-Karp Algorithm Hashing algorithms can be used for faster string searching:

RABINKARPSTRINGMATCHING(T, P, a, q)
 $rP = $ HASH($[]$)
 $rT = $ HASH($[]$)
 for i **in** RANGE($0, len(P)$):
  rP.APPEND($P[i]$)
  rT.APPEND($T[i]$)
 for i **in** RANGE($len(P), len(T)$):
  if rP.HASH() == rT.HASH():
   (*Here we need to double-check as collisions are possible*)
   if P == $T[i - len(P) + 1 : i + 1]$
    output Pattern found at offset, $i - len(P) + 1$
  rT.SKIP($T[i - len(P)]$)
  rT.APPEND($T[i]$)

Can we ensure that false matches (hash collisions) do not happen more frequently than with the probability $1/len(P)$?

Definition: Multi-String Matching Problem. We have a text T of length n as before. But now we have not just one pattern to search, but a set of k patterns $\mathcal{P} = \{P_0, P_1, \ldots, P_{k-1}\}$; each pattern has the same length m.

Rabin-Karp Multistring Algorithm Hashing algorithms can be used for faster string searching:

RABINKARPMULTISTRING($T, \mathcal{P}, m$):
 $hashes$:= Set.EMPTY()
 foreach $P_i \in \mathcal{P}$:
  $rP_i = $ ROLLINGHASH($[]$)
  for j **in** RANGE($0, m$):
   rP_i.APPEND($P[j]$)
  $hashes$.INSERT(rP_i.HASH())
 $rT = $ ROLLINGHASH($[]$)
 for j **in** RANGE($0, m$):
  rT.APPEND($T[i]$)
 for j **in** RANGE($1, n - m + 1$)
  if rT.HASH() $\in hashes$ **and** $T[j : j + m - 1] \in \mathcal{P}$
   output Pattern found at offset, j
  rT.SKIP($T[j]$)
  rT.APPEND($T[j + m]$)

This algorithm would take $O(n + km)$ running time. Naive string matching could take $O(nmk)$ running time, if we probe all the k patterns one by one.

11.1.2 Rolling Hash

Definition: Rolling hash is an ADT: It is a data structure that accumulates some input fragment as a sort of list/queue and supports the following operations:

- RH := ROLLINGHASH($[]$) – initialize a rolling hash to an empty list of symbols.

- RH.HASH() – return the hash value from the current list.

- RH.APPEND(val) – adds symbol val to the end of the list (like enqueue(val) for a queue)

- RH.SKIP(val) – removes the front element from the list (like dequeue(val) for a queue). Parameter val is often implicit as it is stored at the front of the list stored in the rolling hash.

In the case of strings, the list is a list of characters. It can be a list of anything, but elements on that list are represented as integers in some encoding. For example if we interpret characters as ASCII codes, then character 'A' is stored as 65 and 'B' is stored as 66.

We want to treat a list of items as a multidigit number $u \in \mathcal{U}$ in base a (the list in the rolling hash is interpreted as a big number). For example, we can choose $a = 256$, the alphabet size for ASCII code.

Polynomial Rolling hash: This is one of the most popular implementations for rolling hashes; it uses modular arithmetic. Pick some prime number q such that the number base a is not divisible by q. Define the three ADT functions as follows:

- $hash() = (u \bmod q)$

- $append(val) = ((u \cdot a) + val) \bmod q = ((u \bmod q) \cdot a + val) \bmod q$

- $skip(val) = (u - val \cdot (a^{|u|-1} \bmod q)) \bmod q = ((u \bmod q) - val \cdot (a^{|u|-1} \bmod q)) \bmod q$

Example: Pick $a = 100$ and $q = 23$. Let RH be a rolling hash storing [61, 8, 19, 91, 37]. We can compute hash value:

$$hash([61, 8, 19, 91, 37]) = (6108199137 \bmod 23) = 12.$$

In general

$$hash([d_3, d_2, d_1, d_1]) = (d_3 \cdot a^3 + d_2 \cdot a^2 + d_1 \cdot a^1 + d_0 \cdot a^0) \bmod q$$

To make it easier to compute, consider computation with a Hamming code:

$$hash([d_3, d_2, d_1, d_1]) = (((d_3 \cdot a + d_2) \cdot a + d_1) \cdot a + d_0) \bmod q$$

Making this faster:

- Cache the result $(u \bmod p)$ (memorize it in the rolling hash data structure).

- Avoid exponentiation in skip: cache $a^{|u|-1} \bmod p$.

- To append: multiply the cashed $(u \bmod p)$ by base a.

- To skip: divide $(u \bmod p)$ by base (division is expensive, can use multicative inverse).

Rolling Hash with a Cyclic Polynomial (Buzhash): First introduce an arbitrary hash function h(c) mapping single characters to integers from the interval $[0, 2^L)$. Assume that there are not too many characters and the function can be defined by a lookup table. Define the cyclical shift (bit rotation) to the left. For example, $s(1011) = 0111$. The rolling hash function for a list of characters $[c_1, \ldots, c_k]$ is defined by this equality:

$$H = s^{k-1}(h(c_1)) \oplus s^{k-2}(h(c_2)) \oplus \ldots \oplus s(h(c_{k-1})) \oplus h(c_k).$$

It looks like a polynomial, but the powers are replaced by rotating binary shifts. The result of this hash function is also a number in $[0, 2^L)$.

11.2 Problems

Problem 1 (Hash Table with Chaining): The hash function used in this exercise is computed as Pythons `hash()`, and then the remainder `hash(x)` mod 11 is found.

(A) Draw a hashtable with exactly 11 slots (enumerated as `T[0]`, `T[1]`, and so on, up to `T[10]`). Insert the following items into this hashtable (keys are country names, but values are float numbers showing their population in millions):

```
('Austria',8.9), ('Azerbaijan',10.1), ('Belgium',11.6), ('Bulgaria',6.9),
('Estonia',1.3), ('Italy',59.6), ('Latvia',1.9), ('Lithuania',2.8)
```

If there are any collisions between the hash values, add the additional hash values to a linked list using *chaining*. Each item in the linked list contains a key-value pair and also a link to the next item.

(B) What is the expected lookup time, if we randomly search any of the 8 countries to look up its population. Finding an item in a hash table takes 1 time unit; following a link in a linked list also takes 1 time unit.

(C) What is the expected lookup time, if the 11-slot hashtable is randomly filled with 8 keys (each key has equal probability to be in any of the slots). You can find this lookup time rounded up to one tenth (one decimal digit precision) – analytic methods as well as a computer simulation is fine.

Answer:

(A) This Python code snippet computes hash values for strings `'a'`, `'ab'`, `'abc'`. First set up the Python environment to compute (repeatable) hash function values:

```
$Env:PYTHONHASHSEED=0
python
```

In the interactive Python environment compute the values of the hash function:

```
list(map(lambda x: hash(x) % 11, ['Austria', 'Azerbaijan', 'Belgium',
'Bulgaria', 'Estonia', 'Italy', 'Latvia', 'Lithuania']))

[7, 7, 9, 7, 9, 6, 7, 6]
```
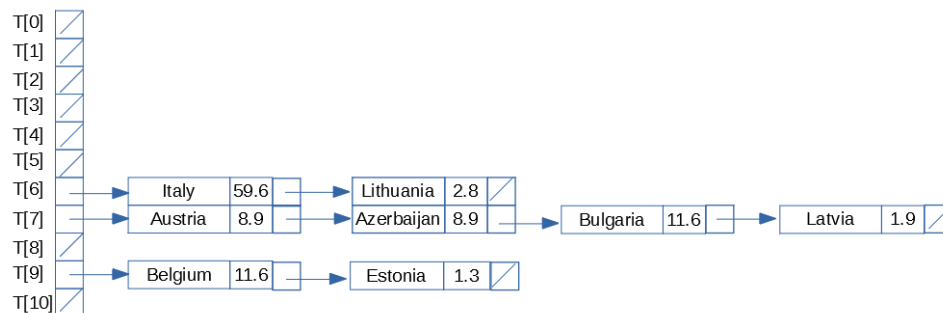

(B) The lookup time can take values $T = 1$, $T = 2$, $T = 3$, and $T = 4$, but with different probabilities. The expected lookup time for a random value is given by the expression:

$$E(T) = 1 \cdot \frac{3}{8} + 2 \cdot \frac{3}{8} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{8} = \frac{16}{8} = 2.$$

(C) The *load factor* in this case is $8/11$, so the average linked list has length $8/11$. The total space of this data structure is $O(m + n)$ (first store a table with m entries, then store n items belonging to our map). The total time for a lookup can be computed as $1 + 8/11$ (one plus the load factor of the hashtable).

Problem 2 (Computing Rolling Hash – Polynomial Method): Assume that we have an alphabet of 100 symbols. They are denoted by pairs of digits: $\{00, 01, \ldots, 99\}$. Select the sliding window size $m = 5$ and the prime number for modulo $q = 23$.

Let the input be `[3, 14, 15, 92, 65, 35, 89, 79, 31]`.

- Initialize an empty rolling hash `rH` and add the first five numbers using `append()` function defined as follows:

$$append(val) = ((u \cdot a) + val) \bmod q = ((u \bmod q) \cdot a + val) \bmod q.$$

- Show how the rolling hash can roll from the list `[3, 14, 15, 92, 65]` to `[14, 15, 92, 65, 35]` (first skip value `3`, then append value `35`.

Problem 3 (Computing Rolling Hash – Cyclic Polynomial): Consider 16 Latin letters with randomly assigned 4-bit codes (i.e. $L = 4$):

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
h(x)	1100	0100	0010	0110	0111	0000	1001	1111	0101	1101	1110	1011	0011	1010	1000	0001

Also assume that the sliding window has size $k = 5$. Find the hash value for $ABIDE$ and then rotate it over to $BIDEN$. Please recall that the rolling hash uses the following formula:

$$H = s^{k-1}(h(c_1)) \oplus s^{k-2}(h(c_2)) \oplus \ldots \oplus s(h(c_{k-1})) \oplus h(c_k),$$

Problem 4 (Rolling Hash ADT with Cyclic Polynomial): Write formulas for the Rolling Hash functions (when using Cyclic Polynomial or Buzhash):

- Formula to initialize `RH` to an empty list.

- Formula to append a new character c_i to the existing list (without skipping anything).

- Formula to skip the head of the existing list c_j (without appending anything).

Problem 5 (Average Complexity of Naive String Matching) Suppose that pattern P and text T are randomly chosen strings of length m and n, respectively, from an alaphabet with a letters ($a \geq 2$). What is the expected character to character comparisons in the naive algorithm, if any single comparison has a chance $1/a$ to succeed?

Problem 6: Assume that we need to create an set containing short phrases written in some human language. Use the following hash function: Compute the sum of character values (modulo the size of our hash table). Will the performance of such hash function implementation be as good as Pythons *hash()* function (also modulo the size of the hash table)? If there are risks using the character value sum, explain these risks.