Data Structures and Algorithms
Exercise lists

Introduction:

1) Write a function to return the minimal value of an array.
2) Write a function to return **the position** of minimal value of an array.
3) Find the second minimal value of an array. We assume that there are minimum two different values in the array.
4) Implement a procedure which takes input permutation and generates next lexicographical permutation. If such permutation doesn't exist, the procedure returns false, otherwise true. Try to implement this efficiently. For example:
    i. All 3-element permutation in lexicographical order: 1,2,3 – 1,3,2 – 2,1,3 – 2,3,1 – 3,1,2 – 3,2,1
    ii. Consecutive permutation after 3,4,2,5,1: 3,4,5,1,2 – 3,4,5,2,1 - 3,5,1,2,4
5) Implement a procedure which takes previous line in Pascal triangle and generates next line. For example for input [1,4,6,4,1] generates [1,5,10,10,5,1]. Use this procedure in recursive way to generate whole Pascal triangle to a given input depth.
6) Write a procedure which takes three values a,b,c and checks if from segment of length a,b,c we can construct a triangle and what type of a triangle (rectangular triangle, obtuse triangle, acute triangle, equilateral triangle, isosceles triangle, scalene triangle). Write the answer on the console.

Iterators:

1) Define an array iterator that provides every *k*-th element of an array, starting with an element with a *poc* index.
2) The data is of the Group class (Student's board), Student (name and average of grades). Define a filter iterator that provides those students who have an average above a given limit.
3) Define the iterator providing subsequent numbers from geometric series, smaller than the given *N*. Note: numbers should be generated on an ongoing basis, do not use the array to store them.
4) Define the iterator providing subsequent Fibbonacci numbers smaller than the given *N*. Note: numbers should be generated on an ongoing basis, do not use the array to store them.
5) Define the iterator providing subsequent primes smaller than the given *N*. Note: numbers should be generated on a regular basis, do not use the array to store them.

Complexity:

Find the complexity for presented code:

1) The code:

```
for i←1 to n do 2
I=O(1)
```

2) The code

```
for i ← 1 to m do
for j ←1 to n do
I=O(1)
```

3) The code:

```
for i ← 1 to n do
for j ← 1 to i do 3
I=O(1)
```

4) The code:

```
r ← n;
while r>1 do 3
I=O(1); 4
 r ← ⌊r/2⌋ ;
```

5) The code:

```
r ← n;
while r > 1 do
for i ← 1 to r do
I=O(1); 5     r ← ⌊r/2⌋;
```

Linked Lists:

1) Implement an **unordered one-way circular**-linked list without sentinel. Implement functions (if you need you can implement additional functions):
   a. add new element as a head
   b. add new element as a tail
   c. add an element on chosen position
   d. find and remove an element chosen by a value
   e. find element chosen by a position in the list
   f. add two lists
   g. reverse list
   h. copy list
2) Implement an **ordered two-way circular**-linked list without sentinel. Implement functions (if you need you can implement additional functions):
   a. add new element to a list
   b. find and remove an element chosen by a value
   c. find element chosen by a position in the list
   d. reverse list (?)
   e. copy list
   f. add two lists
3) The data is of the Group (Student list) class, Student (name and average grade). Define methods in the Group class:
   a. selectGood, the result should be a list of students with an average above the given limit value;
   b. removeBad, the effect should be to remove from the list of students individuals with an average below the given limit value;
   c. avarageGrade, which calculates the average grade of all students
4) Which list will be the best for determining the LuckyNumber (for given *N* and *K*). Pirates set *N* prisoners in a circle and starting from the first they counted to *K*, *K*-th prisoner they threw into the sea and continue counting to the next *K*-th prisoner, the fun lasted until there was only one prisoner - his initial position in the circle is just the LuckyNumber for data *N* and *K*. Assume the best model and write the appropriate methods. Note: the task is also known as the Josephus problem, it turns out that you can build the right pattern (formula).
5) Extend the interface of the lists with the list submission operation (concatenation). This operation attaches the given (parameter) list to the end of the given (this) list. Is it possible to implement it as a universal (independent of the implementation of the list), or you have to define separately for lists based on arrays and linked lists?

Queues:

1) Present direct implementation (without using the List class) of a regular unlimited queue. To store elements, use a one-way linked list with a sentinel.
2) A limited (capacitated) ordinary array-based queue can be effectively implemented (ie, avoiding the copying of large chunks of data) directly (without using the List class). Define the appropriate class, eg. ArrayFifoQueue. The complexity of all operations should be constant - O (1).

Stack:

1) Present the direct implementation (without using the List class) of the unlimited stack. Use a unidirectional linked list without sentinel to store elements.
2) Present the direct implementation (without using the List class) of the limited stack. Use the array to store elements.
3) Another type of limited stack is the sinking stack. If there are already MAX elements in the stack, then adding the next one causes the element lying at the bottom of the stack to be lost. What data structure will you use to store elements.
4) Veloso's Traversable Stack is a stack that, besides ordinary operations, has the option of non-destructive reading from the position indicated by the cursor (peek). The cursor can be set to the top of the stack (top) and moved one position down the stack (down - there is a need to signal reaching the bottom of the stack). Normal operations (push and pop) automatically set the cursor to the top. Implement VTS as an extension to the regular stack.

Techniques:

All next problems are from https://icpcarchive.ecs.baylor.edu/index.php. Then choose links: “Browse Problems” -> “ICPC Archive Volumes” -> then volume XYZ has problems with numbers from XYZ*100 to XYZ*100+99

1) Propose an algorithm to solve the problem specified below using **greedy algorithm**. Propose an efficient implementation of the algorithm.
   a. 3004 - Change
   b. 2535 - Magnificent Meatballs
   c. 2326 - Moving Tables
2) Propose an algorithm to solve the problem specified below using **dynamic programming**. Propose an efficient implementation of the algorithm.
   a. 2487 - Lollies
   b. 3144 - Lenny's Lucky Lotto Lists
   c. 3390 - Pascal's Travels
3) Propose an algorithm to solve the problem specified below using **“divide & conquer”**. Propose an efficient implementation of the algorithm.
   a. 2122 - Recognizing S Expressions

Sorting:

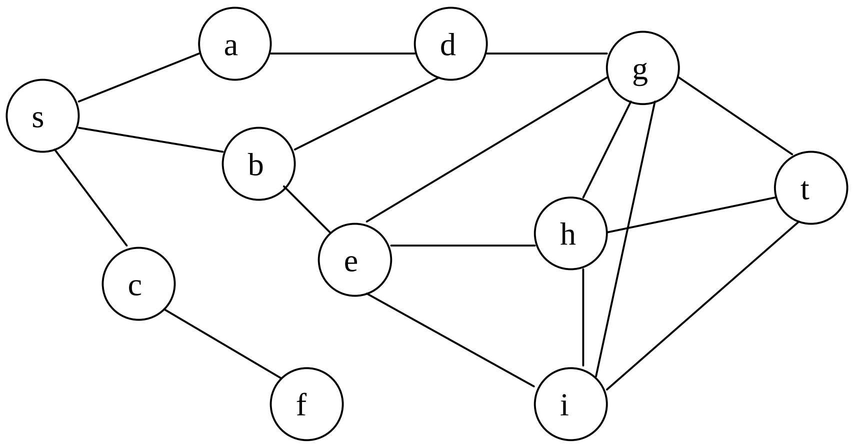
1) Write a function which will sort an array using insertsort. The array has to be sorted in decreasing order and the sorted part has to grow from end of the array.
2) Write a function which will sort an array using bubblesort. The array has to be sorted in increasing order and bubbles have to move from end of the array.
3) Write a function which will merge two sorted arrays into one sorted array.
4) Analyze the behavior of individual sorting algorithms for ordering the sequence of letters. SORTING IS SIMPLE, pay attention to the number of comparisons and rewrites as well as stability.

Trees:

1. Show a state of BST tree after each instruction run in the following sequence: insert(10), insert(5), insert(15), insert (2), insert (3), insert(6), insert(7), insert(1), delete(5), delete(1), delete (10). We start from an empty tree.
2. Implement a function on BST which:
   a. count the number of nodes
   b. count the height of the tree
3. Show a state of AVL tree after each instruction run in the following sequence: insert(10), insert(5), insert(15), insert (2), insert (3), insert(6), insert(7), insert(1), delete(5), delete(1), delete (10). We start from an empty tree.
4. Show a state of RB-tree after each instruction run in the following sequence: insert(10), insert(5), insert(15), insert (2), insert (3), insert(6), insert(7), insert(1), delete(5), delete(1), delete (10). We start from an empty tree.
5. Show a state of 2-3-4-tree after each instruction run in the following sequence: insert(12), insert(5), insert(15), insert (2), insert (3), insert(6), insert(7), insert(9), insert(1). We start from an empty tree.

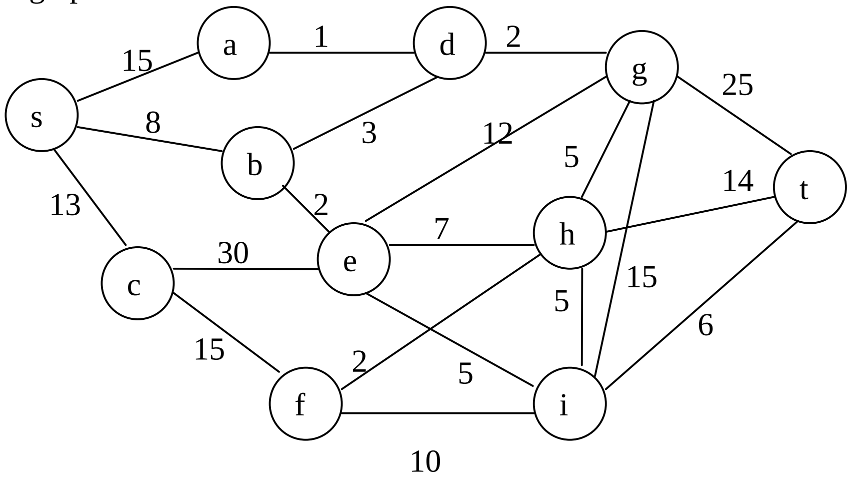
Graphs:

2. For presented graph:



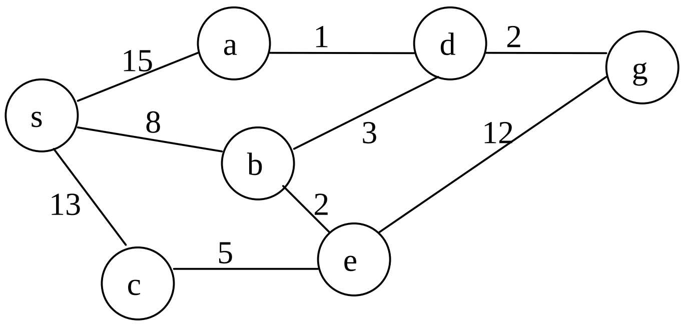
   a. Present the order of visited vertices during depth-first search. During choosing next vertex to visit, choose vertices in lexicographical order. Begin from vertex: i. s
      ii. e
      iii.
   b. Present the order of visited vertices during breadth-first search. During choosing next vertex to visit, choose vertices in lexicographical order. Begin from vertex: i. s ii. e

3. For presented graph:



   a. Search for the shortest path from vertex ‘c’ using Dijkstra’s algorithm.
   b. Search for the minimum spanning tree using Kruscal’s alghorithm.
   c. Search for the minimum spanning tree using Prim’s alghorithm.

4. For presented graph:



   a. Search for the shortest path between all-pairs of vertices.

String Matching

1. Show the comparisons the naive string matcher makes for the pattern P = `0001` in the text T = `000010001010001`.
2. Working modulo q = `11`, how many spurious hits does the Rabin-Karp matcher encounter in the text T = `3141592653589793` when looking for the pattern P = `26`?
3. Construct the string-matching automaton for the pattern P = `aabab` and illustrate its operation on the text string T = `aaababaabaababaab`.
4. Draw a state-transition diagram for a string-matching automaton for the pattern `ababbabbababbababbabb` over the alphabet S = {`a`,`b`}.