**Faculty of computers and AI,   
Cairo University**

**CS213: Programming II  
Year 2022-2023**

**First Semester**

**Problem Sheet 3 – Version 1.0**

**Course Instructors:**

Dr. Mohammad El-Ramly

**Revision History**

**Version 1.0** By Dr Mohammed El-Ramly 28 Nov. 2022 Main Doc

**Objectives**

This sheet includes programming problems for training on C++ advanced concepts.

**Preparation**

You need to study recursion and backtracking well. You may like to try the problems sheet of Asuit University for training on these topics. You will need also to study templates and STL and exceptions.

**Problems**

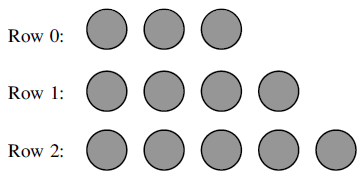
1. **Backtracking.** Youssef Elzayat has his SPACE key on the keyboard not working. He is sending messages to his friends but they do not have spaces like "Iwillmeetyoutomrrow" or "letusplayfootballtonight". His friends have difficulty understanding the message. To help them understand the messages, we need to write a **recursive backtracking** C++ program that divides the given message string to individual meaningful words. A meaningful word is one that exists in the English dictionary. The program should try to form words starting from the first letter. If it finds the word in the dictionary, it calls itself on the remaining text. The program stops as soon as it can divide the message to a useful sentence with all its words in the dictionary. If no solution is found the program will do exhaustive recursion of all possible solutions and then fail. For example, if the message is "Iwearmyshoes", it will try to divide as follows:

* "I" is a word in dictionary, then repeat on the remaining letters.
* Try "w" which is not a word, then try "we" which is a word and repeat on the rest.
* Try "a" which is a word, then repeat on the rest. (So far we have "I we a")
* Then we try "r" which is not a word and "rm" which is not a word. We can keep trying bigger words like "rmy" and "rmys" or if we find no word starts with "rm" we can stop.
* We backtrack and try "ar" which is word and then repeat on the rest "myshoes"
* And so on. The first valid solution will be "I we ar my shoes".

1. **Recursion / Backtracking.** Abdelrahman Mohamed is bored of the simple Nim game we developed in the lecture. He decided to make new variants of it but not sure how. His friend Salaheldeen Salah suggested these two different games with different rules. Help them implement them with a computer player.(If you can, make the code generic and structured into functions or modules using OOP techniques. If possible, make a parent abstract class that defines the required behavior of the game and then its children are different variants of the game.)

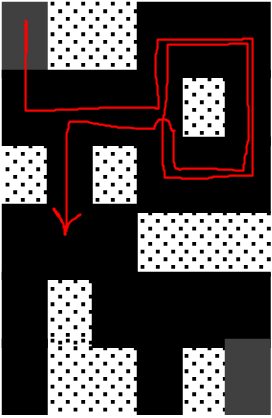


* **Variant one.** In the simple Nim game we developed, the player who takes the last coin loses and the coins the players took away were simply ignored. In this game, the coins go into a pile for each player. The player whose pile contains an **even number** of coins after the last one is taken wins the game.



* **Variant two.** In the simple Nim game we developed, all the coins where put in one pile. In this variant the money is divided to 3 piles as in picture. Each player takes 1, 2 or 3 coins from any pile he likes. The player that takes the last coin loses.

1. **Recursion / Backtracking.** Ziad Mohamed needs your help enhancing the maze solving problem included in the class code and attached with the assignment. After a brain storming session, he decided that he needs to add these features:



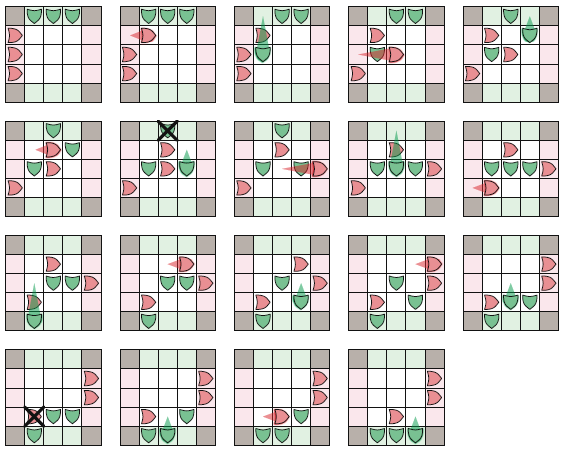
* The mouse should be allowed to try to move in any of the four directions (up, down, right and left). But he should avoid going in circles, i.e., coming to the same point in the path as show in the figure, otherwise the program might have infinite recursion.
* He will add a function that returns all possible solutions in case more than one is available.
* He will add a function that returns the shortest solution in case more than one is available.

1. **Backtracking.** Sohila Abelazeem bought a new car from her savings her new job at Google. She passes by a number of traffic lights every day. Each light could be RED or GREEN. If she is lucky, all lights are green. And if she is very unlucky, all lights are red. She wants to see all the possibilities that she can face in the road. So for a given number of intersections n, e.g. 1, 2, 3, …, n, she wants to develop an **exhaustive recursion** algorithm that generates all possible combinations of traffic lights that she may meet. For example, if she has three traffic lights: 1, 2, 3. Then she may face any of these possibilities:

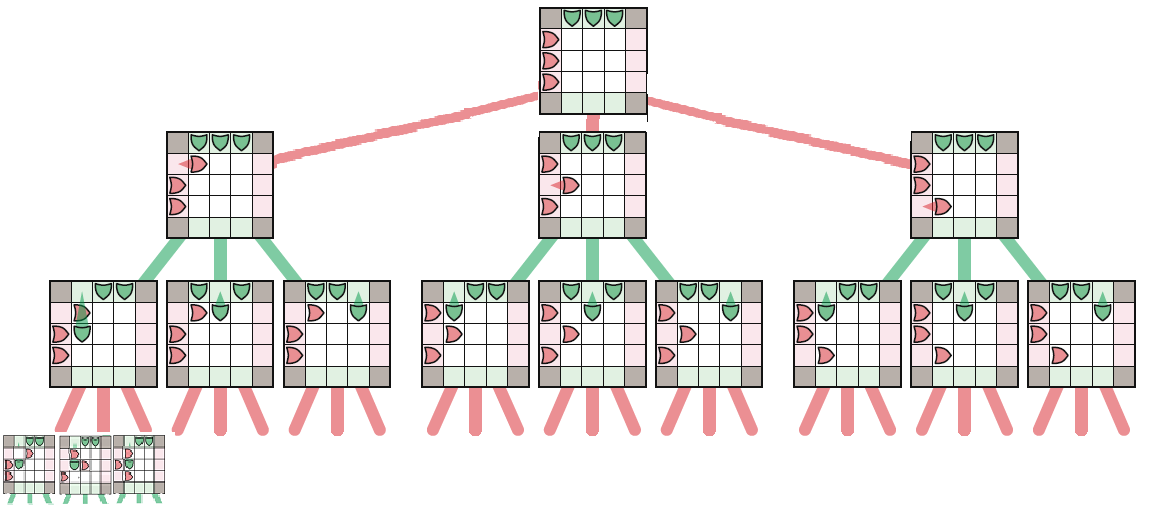
* 1- RED, 2- RED, 3- RED
* 1- RED, 2- GREEN, 3- RED
* 1- RED, 2-RED, 3- GREEN
* 1- RED, 2-GREEN, 3- GREEN
* 1- GREEN, 2- RED, 3- RED
* 1- GREEN, 2- GREEN, 3- RED
* 1- GREEN, 2-RED, 3- GREEN
* 1- GREEN, 2-GREEN, 3- GREEN

Repeat the same problem, if he has also the possibility of having a YELLOW light in countries that use it as a warning before RED.

**5 & 6** **(**This counts as two problems) **Backtracking.** Abdelrahman Wael and Michael Youssery invented A&M game in which is played on a board of size *n*, assume 5 as in the shown figure. Each player starts with *n* – 2 rockets, one player has rockets in the left side facing right and one player has rockets on the top side facing down. Rockets can ONLY move forward. An exam example of game play is shown on the side.

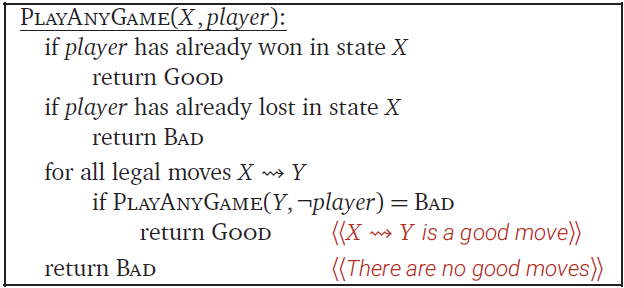


You are required to develop the game with two players. Then add to it a computer player that uses backtracking to decide the best move. At start, game should offer the choice to play against another human or against computer. Similar to what we did for a Nim game in the lecture, a game tree represents all the possible moves from a given state. For example, below is the first two levels of a game tree for the game **state** at the start of the game.



To search this tree for a solution, we recursively define a game **state** to be **good** or **bad** as follows:

* A **good** game state is when (1) a player wins the game or if he can move the opponent to a bad state.
* A **bad** game state is when (1) the current player already lost or (2) if every possible move leads to a good state to the opponent.



In other words, a non-leaf node in the game tree is good if it has at least one bad child, and a non-leaf node is bad if all its children are good. By induction, any player that finds the game in a good state on their turn can win the game, even if their opponent plays perfectly; on the other hand, starting from a bad state, a player can win only if their opponent makes a mistake. This recursive algorithm is defined in the side box.

1. **STL – Map.** Farah Mohamed works remotely with a foreign company that is writing a search engine. One of the criteria for ordering the retrieved documents is how many times a word exists in the document. To find that out, she needs to develop a frequency table for a given text file. A frequency table lists words and the number of times each word appears in a document or a file. Help her write a program that creates a frequency table for a file whose name is entered by the user. Use STL **map** that stores <Key, Value> pairs of **string** and **int**. First, read the input file and clean it of all punctuation and non-alphanumeric characters except "-" which can be part of a word. You may find functions **ispunct(), isalnum(), tolower(),** etc (see **cctype** library) useful to use. Test your program on multiple different files.
2. **Exceptions / Templates.** Kirolos Usama is trying to learn how stacks work by building one. He needs your help to develop a template stack class with methods: push, pop, top, size and empty. An object of our stack class has a fixed size given as a parameter to the constructor and it does not expand. Define also the big 3 methods for it. When used in a program, an object of the stack class will throw exceptions in the following situations:

* Throw a **StackOverflowException** if the application program tries to push data onto a stack that is already full
* Throw a **StackEmptyException** if the application program tries to pop data off an empty stack

Defining the classes **StackOverflowException** and **StackEmptyException**.

Write a function called **manageStack** that displays a menu and allows the user to create a stack of the size e likes and give him the options to check top element, push or pop elements from the stack. The function should catch these exceptions and produce a suitable message and then redisplay the menu.

1. **Template – Set.** Abdullah Mohamed uses a C++ version that does not support STL and he needs a set class for his programming job. Help him by writing a template-based class that implements a set of items. A set is a collection of items in which no item occurs more than once. Internally, you may represent the set using the data structure of your choice (for example, list, vector, arrays, etc.). However, the class should externally support the following functions:

* Add a new item to the set. If the item is already in the set then nothing happens.
* Remove an item from the set.
* Return the number of items in the set.
* Determine if an item is a member of the set.
* Return a pointer to a dynamically created array containing each item in the set. The caller of this function is responsible for de-allocating the memory.

Test your class by creating different sets of different data types (for example, strings, integers, or other classes). If you add objects to your set, then you may need to overload the == and != operators for the object’s class so your template-based set class can properly determine membership.

1. **Exceptions.** Ihab Ashraf is trying to write a program that converts dates from numerical month/day format to alphabetic month/day (for example, 1/31 or 01/31 corresponds to January 31). You will define two exception classes, one called **MonthError** and another called **DayError**. If the user enters anything other than a legal month number (integers from 1 to 12), then your program will throw and catch a **MonthError**. Similarly, if the user enters anything other than a valid day number (integers from 1 to either 29, 30, or 31, depending on the month), then your program will throw and catch a **DayError**. To keep things simple, always allow 29 days for February.