Problem 1: Array sorting

This assignment should be done individually

- To do so, create a mooshak account with your ualg login (without @ ualg.pt) in mooshak. Example: the student with the number 12345 uses a12345. NB: do not use your name in the login.
- Submit your code to mooshak http://deei-mooshak.ualg.pt/~jvo/ Problem B up to:

  October 13, 2025 – 17h
- A submission will remain *pending* until validated by the instructor during the lab class. Only *final* submissions will be considered for evaluation. Deadline for validation:

  October 31, 2025

Problem

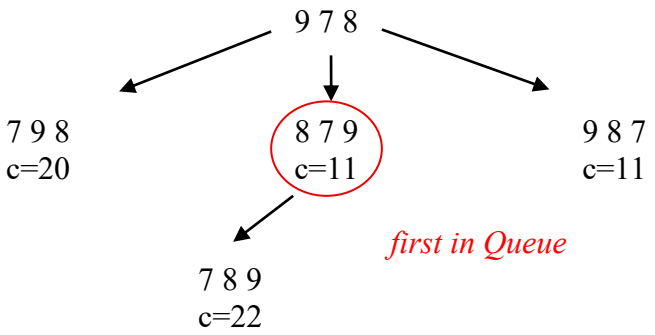
It is intended to sort arrays of integers from an initial state to a given final state. Sorting is performed by swapping two integers at each step. Swapping two even integers has a cost of 2, swapping one even and one odd integer has a cost of 11, and swapping two odd integers has a cost of 20.

The swapping order starts from left to right. The first successor is generated by swapping the leftmost integer with the integer immediately to its right, then with the next integer to the right, up to the rightmost integer. Then the 2nd leftmost integer is swapped with each of the integers to its right, in the same order, and so on.

The successors are inserted into a queue in the order in which they are created and then sorted by the cumulative path cost, which is the sum of the costs from the initial node along the path to the successor. When a new successor arrives, it is inserted at the end of its cost group. If the new successor has a state that already exists in the queue but has a lower cumulative cost, the old successor is removed from the queue, and the new one is inserted at the end of its cost group.
For example, for the array 9 7 8:

To swap	Successor	cost	Queue head		… tail
9 7 8	7 9 8	20	7 9 8 c=20		
9 7 8	8 7 9	11	8 7 9 c=11	7 9 8 c=20	
9 **7 8**	9 8 7	11	8 7 9 c=11	9 8 7 c=11	7 9 8 c=20

The levels expansion for the initial array with configuration 9 7 8, and the associated cost, **c**, is presented below. Note that, though it is not visible in this example, configuration states already expanded are pruned. Note also that when the node with the goal state is reached, it does not need to be put in the queue. In the figure below, it is in the queue only to exemplify the computation of the cumulative cost along the path from the initial state to the successor:



Queue
8 7 9; 9 8 7; 7 9 8
c=11 c=11 c=20

Queue
9 8 7; 7 9 8; 7 8 9
c=11 c=20 c=22
goal

goal reached with cost = 22

Task
There are many approaches to address this problem. However, your task is to write a program that, based on the approach **presented in Tutorial 1 and extended in appendix A**, returns the successive configurations required to achieve the final configuration, starting from the initial one, along the minimum cost path, i.e., along the path requiring the lowest possible cost.

Apply a blind uninformed search algorithm that guarantees obtaining the lowest cost solution for any instance of this problem (within the available memory and processing limits). Note that at this point, no heuristics are allowed.

Please note that any other approach, however meritorious it may be, will be quoted with 0 (zero).

Input
The input consists of two lines: the initial configuration of the array and the final configuration. All integers are separated by a space.

Output
The sequence of configurations from the initial to the final one (both *inclusivé*), each in a new line. At the end of this sequence, a non-negative integer with the minimum cost of the path found should also be presented.

Sample Input 1
```
9 7 8
7 8 9
```

Sample output 1
```
9 7 8
8 7 9
7 8 9
22
```

Sample Input 2
```
6 8 2 5 10
8 10 2 5 6
```

Sample output 2
```
6 8 2 5 10
10 8 2 5 6
8 10 2 5 6
4
```

Sample Input 3
```
14 11 15 13 12
15 14 13 12 11
```

Sample output 3
```
14 11 15 13 12
14 12 15 13 11
12 14 15 13 11
15 14 12 13 11
15 14 13 12 11
35
```

Appendix A

Requirements for validation

There are many possible solutions to this problem; however, the implementation must follow the directives below, and the implementation presented in:

Lab tutorial 1: Small Instances of the 8-puzzle

1) The Main class and function must be the one below.
Note that the only prints to the console are the ones in this class, which print the solution.
No other print calls are allowed in the rest of the code.

```
public class Main {

	public static void main (String [] args) throws Exception {
		Scanner sc = new Scanner(System.in);

		GSolver gs = new GSolver();
		Iterator<GSolver.State> it =
					gs.solve(  new ArrayCfg(sc.nextLine()),
						   new ArrayCfg(sc.nextLine()));
```

```
            if (it==null) System.out.println("no solution found");
            else {
                  while(it.hasNext()) {
                        GSolver.State i = it.next();
                        System.out.println(i);
                        if (!it.hasNext()) System.out.println(i.getK());
                  }
            }
            sc.close();
      }
}
```

2) The array configuration class MUST implement Ilayout interface and define a constructor to create an integer array from a string:

```
      public class ArrayCfg implements Ilayout {

            int data[];

            /**
            @ Create an array of integers from the string,
              where integers are separated by spaces
            */
            ArrayCfg(String s)
            // convert the string to the integer array data declared above
data = Arrays.stream(s.split(" ")).mapToInt(Integer::parseInt).toArray();

            //Other constructors if needed
            // ...

            // implementation of Ilayout methods below:
            // to complete
      }
```

3) The Ilayout interface:
Note that classes that implement this interface MUST also implement toString() method, which is called by the println() function in the Main class:

```
      public interface Ilayout {
            /**
            @ return the children of the receiver.
            */
            List<Ilayout> children();
```

```
        /**
        @  return true if the receiver equals the argument l;
         return false otherwise.
        */
        boolean isGoal(Ilayout l);

        /**
        @ return the cost from the receiver to a successor
        */
        double getK();

        /**
        @ return a string representation of this Ilayout
        */
        String toString();
}
```

4) A graph solver suitable to solve this problem should be implemente, similar to the one presented in the Lab 1 Tutorial, with the necessary modifications:

```
//Graph solver
public class GSolver {

        //…

        /**
        @Implements a graph solver suitable to solve this problem
        based on the version developed in the:

              Lab tutorial 1: Small Instances of the 8-puzzle
        */
        public Iterator<State> solve(Ilayout s, Ilayout goal) {
              // to complete
        }

        // to complete

}
```

5) Note that some of the UnInformed graph solvers use the same code, differing only in the way the successors are sorted in a generalized queue. Remember the lecture slides, where *fringe* is the generalized queue of the successors not yet expanded:

Especificidades

- Largura-primeiro
  - Expande os nós pela ordem que foram gerados
  - *fringe* = fila de espera FIFO, i.e., os novos nós vão para o fim
- Custo-uniforme
  - Expande primeiro os nós de menor custo
  - *fringe* = fila de espera com prioridades, ordenada por custos
- Profundidade-primeiro
  - Expande primeiro os nós mais recentes, i.e., mais profundos
  - *fringe* = Stack LIFO, i.e., nós novos saiem primeiro

http://tiny.cc/ia2024-25