4.1 Selfgrowing Array

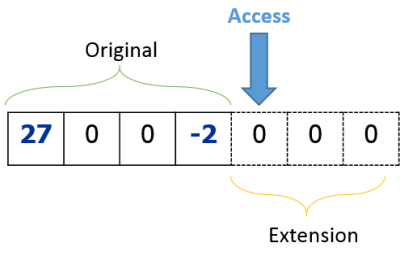
By now you've already gained some experience with arrays: once the size of an array is fixed, it cannot be changed. Any attempt to write to an array index bigger then the size of the allocated array will cause a java exception. (Error and termination of the program if not handled).

The aim of this assignment is to implement an array of int-numbers that provides adaptive growth.

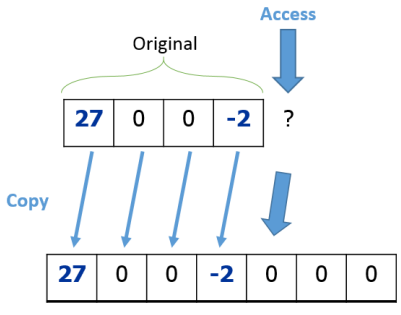
The public interface for the growing array is as follows:

```
class GrowingArray {
    // constructor
    public GrowingArray()

    // pre: any index ind >= 0
    // post: return value at position ind
    // If no value has been set to ind previously, return 0
    public int get(int ind)

    // pre: any index ind >= 0, any value val
    // post: store value val at position ind
    public void set(int ind, int val)

    // post: returns size of currently allocated storage
    public int size()
}
```

By giving the user access via this object's interface (instead of direct access to an array) it is possible to hide the actual implementation. Actually, there would be many different possibilities to implement such a dynamic array. For this exercise, it should be implemented as indicated by the following illustration:



Impression from the outside



Realisation behind the scenes

On the left you can see how the GrowingArray should appear from the outside. Originally it has a size of 4. Then the user tries to access index 4, the array grows automatically to the next power of two which would be 8. On the right you can see how this functionality is implemented internally. As you know an array can not grow. So when the array has to be extended, a new array is allocated with the new size (next power of two). Then all values from the old (smaller) array are copied into this new array and finally the old array is replaced with the new one.

Next power of two

Growing the array exactly to the size required to host a newly requested element can result in a huge number of allocations when sequentially traversing and resizing the array. In order to avoid this, the array should be grown to the next size which is a power of two. For instance: if the user requests Element 931 we grow the array to size 1024 which corresponds to 2^{10}.

Your first task is to implement the subroutine that returns the next power of two given any input number. Download a skeleton for this assignment from http://lec.inf.ethz.ch/baug/informatik2/2016/ex/ex04/GrowingArray/Main.java

Resize

Finally your task is to implement the routine

```
private void resize(int size)
```

which is called within `set`. As motivated earlier it is supposed to

- create a new array using the NextPowerOfTwo function,
- copy the old (smaller) array into the new array,
- replace the internal storage (called "data" in the skeleton code) with the new storage.

You can validate your solution here: https://challenge.inf.ethz.ch/team/websubmit.php?problem=IB16041

4.2 Sorted NameList

Imagine you want to celebrate your birthday with a lot of people. As a first step, you want to have a little database that contains a sorted list of all people you have already thought of. Later, we will complement this idea with more information like adding telephone numbers and addresses and favorites of the invited people, but for now the following is enough: You want just to be able to enter names, check for existence (check if a name is already in the list) and output all names in a sorted order.

The goals of this exercise are:

- Apply the results of the previous exercise in a little more advanced setup. → Using a growing array to build a sorted list.
- Understand the complexity of sorted insertion when using an array.
- Think about possibilities to find an element in an array
- By understanding the complexities of this approach, you will later in the course see how hashing can be used to do this more efficiently.

Your task is the following: implement the class NameList. It contains a sorted list of names. The function `add` can be used to add new names to the sorted list, `contains` checks if a name is already in the list and the function `print` outputs all names in the list in sorted order.

```
class NameList {
    // constructor
    public NameList()
```

```
    // pre: any String
    // post: if the string was not contained in the NameList, add it in a sort
    public void add(String s)

    // pre: any String s
    // post: returns true if string s is contained in the name list, i.e. if
    public boolean contains(String s)


    // post: output all strings of the NameList to system.out.
    public void print()
}
```

Remarks:

- For sorted insertion, compare Strings `l` and `r` by calling `l.compareTo(r)`.
  `l.compareTo(r)` returns one of the following values 0, 1 or -1. 0 means strings l and r are equal, +1 means l is before r and -1 means l is after r with respect to lexicographic ordering.
- To insert a string find the index where the new string s should be inserted such that the list stays lexicographically ordered. It is possible that you have to move other strings to insert the new string at this index.
  Think about required actions using pen and pencil before you start implementing.
- Start with simple implementations for add, contains and print. Only when everything works fine, think about ways to improve your implementation.

As always, we have provided a skeleton that you want to complement here: http://lec.inf.ethz.ch/baug/informatik2/2016/ex/ex04/NameList/Main.java

You can validate your solution here: https://challenge.inf.ethz.ch/team/websubmit.php?problem=IB16042