GIT DEPARTMENT OF COMPUTER ENGİNEERİNG

CSE 222/505 - SPRİNG 2020

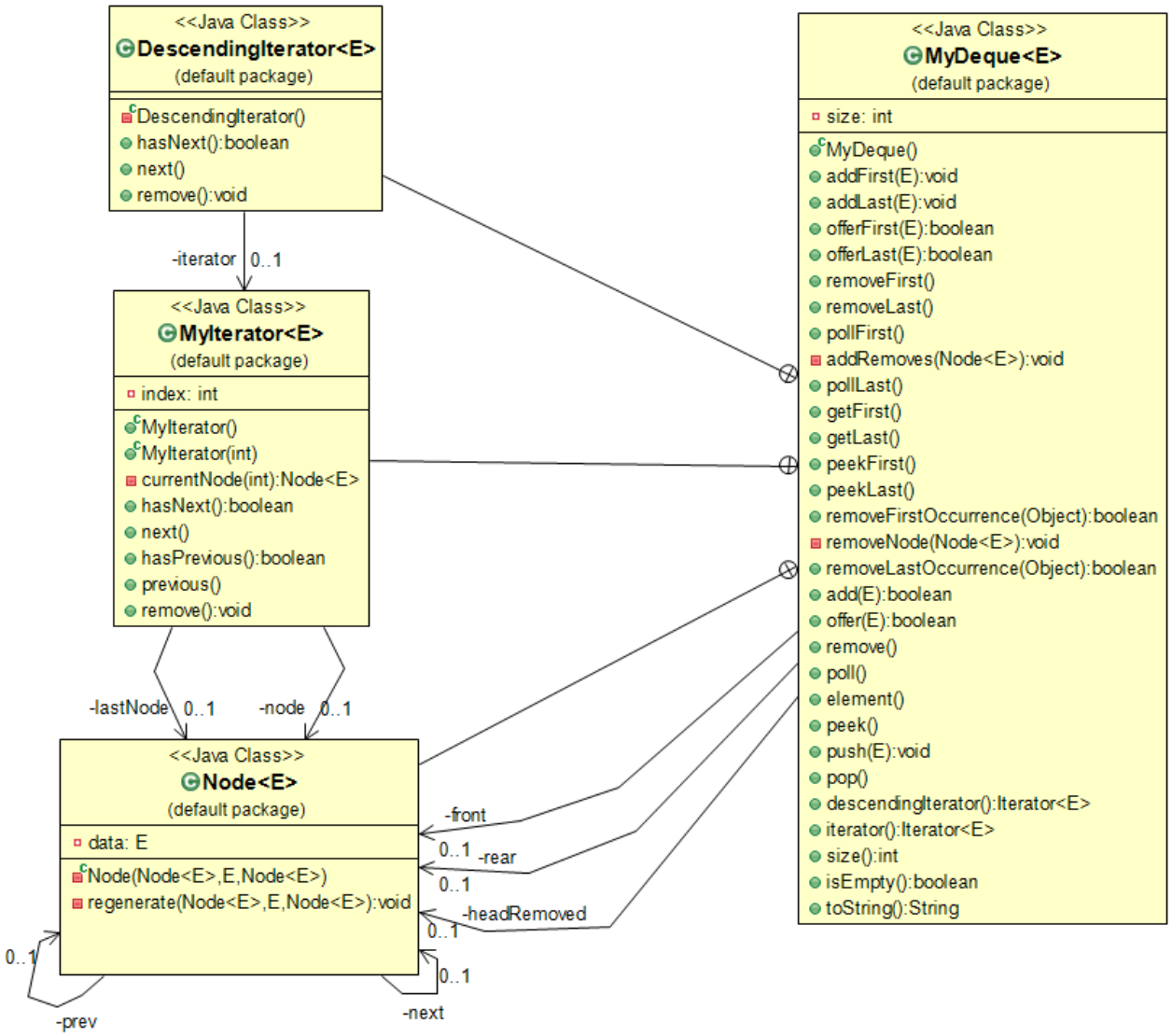
HOMEWORK 4 REPORT

ŞEYDA ÖZER

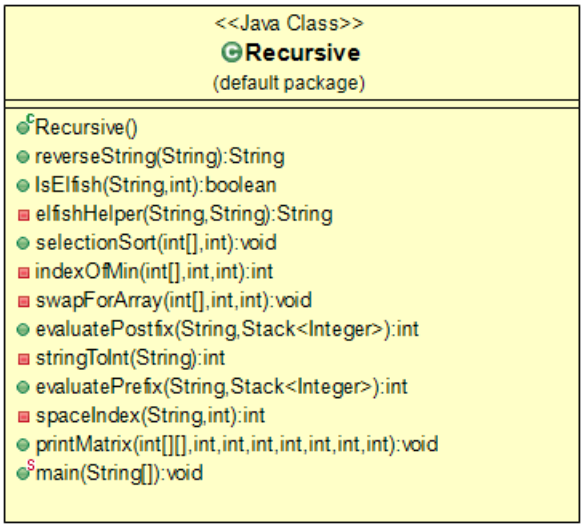
171044023

CLASS DIAGRAMS

Q2:



Q3:



PROBLEM SOLUTION APPROACH

Q2:

Firstly, I created the Node class for the linked lists that my deque class should contain. I added an extra regenerated method for my node class because when I want to reuse a node I deleted, I have to change its data. Then I created two node objects named Front and Rear for the linked list holding the elements of the deque. Front node is the head of the deque and rear node is the tail of the deque. I used the linked list that holds the elements I deleted as a single linked list , so I used one node for the linked list. This node is the head of the elements that are removed. I also created the iterator class for my deque class and I used this class for toString method. After these processes, I override the methods in the deque interface. In all the methods that I deleted a node, I added the node I deleted from the deque list to the list of the elements deleted. In all the methods I added an element to the Deque list, I used it if there was any node in the list of deleted elements, if not, I would create a new node.

Q3:

First, I created a Recursive Class. In this class, I created a main method and all the methods required for the 3rd question.

Part1: reverseString

Base Case: The given string is empty.

Other problems:

- to divide a string word by Word

Solution:

If the string contains the space character, I am dividing the string from the space character into two parts as first part and second part. I am adding a space character at the beginning of the first part. In this case, the first part is understood to be the first word, and the method returns it. Also, the second part should be sent to the method again.

If the string does not contain the space character, I used the string as first part and I accepted the part like a word. Also I accept the second part is empty, so I send an empty string to the method.

Part2: IsElfish

Base Cases:

- The given string is empty.
- The counter equals to 3.

Other Problems:

- Increment of the counter only once for each of the e, l and f characters

Solution:

When I find one of the letters e, l and f in the word, I increase the counter. In this case, if there is more than one of the letter e, the counter will increase each time. So when I find any of these letters, I remove all the letters in the word that are the same as that letter. so the problem of increasing the counter is solved. I then send the string to the method from the next character, so the method stops working when the string is empty.

Part3: selectionSort

Base Cases:

- The index equals to (array.length – 1)
- The array is empty

Solution:

Initially, the index is zero. All elements are compared from this index to the end.

I used an helper method to make this comparison more comfortable. This method makes element-by-element comparison, increasing the index one at a time.

If the index of the minimum element found is not the same as the index that we sent, these elements is swapped.

Part4: evaluatePostfix

Base Case: The postfix expression is empty.

Other problems:

- a number of digits greater than 1
- the character is operator or operand

Solution:

If the string contains a space character, I divide the string from the space character into two parts, the first and second parts. I check the first part, if it is a digit, I send it to the helper method to find the integer value of the string. Then I push the integer value into the stack. If not, this method pops two integer values off the stack and evaluates these values with the operator. It then pushes the result of the evalution into the stack.

After these processes, I will send the second part to the method again. When the last string does not contain a space character, I accept the first part equals to the string and the second part is empty. If this method arrives the base case, it returns the element of the stack.

Part5: evaluatePrefix

Base Case: The prefix expression is empty.

Other problems:

- a number of digits greater than 1
- the character is operator or operand

Solution:

I divide the expression from the space character into two part, the first and second parts. If the expression does not contain a space character, I accept the second part equals to expression and the first part is empty string, so I send the empty string to this method and this method stops.

If the expresion contains a space character, firstly I find the index of the first space character in the string. Later I send the index to the helper method. This method finds the index of the last space character in the string. So I divide the string according to the index. The part after the last space is my second part. If the first character of the second part is digit, I send the string to helper method. This method calculates integer value of the string. Then I push this integer value into the stack. If this character is not digit, I pop two values off the stack and evalute the two values with the character(operand). Then I push the result of the evalution into stack. If this method arrives the base case, it returns the element of the stack.

Part6 : printMatrix

Base Cases:

- array is empty
- counter is equals to m*n

Other problems:

- index changes
- change of bounds

Solution:

In each element of the matrix, the counter increases by one and the method stops when the counter is m*n. The direction is determined when the bound indexes are reached in the matrix. Also bounds are changed before the matrix reaches the first element.

TEST CASES

Q1:

Test Case 1: when deque is empty, using peek method.

```
System.out.println(deque.peek());
```

Test Case 2: when deque is empty, using peekFirst method.

```
System.out.println(deque.peekFirst());
```

Test Case 3: when deque is empty, using peekLast method.

```
System.out.println(deque.peekLast());
```

Test Case 4: using peekFirst method.

```
System.out.println("The front element of the deque: " + deque.peekFirst());
```

Test Case 5: using peekLast method.

```
System.out.println("The rear element of the deque: " + deque.peekLast());
```

Test Case 6: using peek method.

```
System.out.println("The element of the deque: " + deque.peek());
```

Test Case 7: when deque is empty, using element method.

```
try {
    deque.element();
} catch(NoSuchElementException e) {
    System.out.println(e);
}
```

Test Case 8: when deque is empty, using getFirst method.

```
try {
    deque.getFirst();
} catch(NoSuchElementException e) {
    System.out.println(e);
}
```

Test Case 9: when deque is empty, using getLast method.

```
try {
    deque.getLast();
} catch(NoSuchElementException e) {
    System.out.println(e);
}
```

Test Case 10: using getFirst method.

```
System.out.println("The front element of the deque: " + deque.getFirst());
```

Test Case 11: using getLast method.

```
System.out.println("The rear element of the deque: " + deque.getLast());
```

Test Case 12: using element method.

```
System.out.println("The element of the deque: " + deque.element());
```

Test Case 13: when deque is empty, using poll method.

```
System.out.println(deque.poll());
```

Test Case 14: when deque is empty, using pollFirst method.

```
System.out.println(deque.pollFirst());
```

Test Case 15: when deque is empty, using pollLast method.

```
System.out.println(deque.pollLast());
```

Test Case 16: using pollFirst method.

```
deque.pollFirst();
System.out.println(deque);
```

Test Case 17: using pollLast method.

```
deque.pollLast();
System.out.println(deque);
```

Test Case 18: using poll method.

```
deque.poll();
System.out.println(deque);
```

Test Case 19: when deque is empty, using remove method.

```
try {
    deque.remove();
} catch(NoSuchElementException e) {
    System.out.println(e);
}
```

Test Case 20: when deque is empty, using removeFirst method.

```
try {
    deque.removeFirst();
} catch(NoSuchElementException e) {
    System.out.println(e);
}
```

Test Case 21: when deque is empty, using removeLast method.

```
try {
    deque.removeLast();
} catch(NoSuchElementException e) {
    System.out.println(e);
}
```

Test Case 22: using removeFirst method.

```
deque.removeFirst();
System.out.println(deque);
```

Test Case 23: using removeLast method.

```
deque.removeLast();
System.out.println(deque);
```

Test Case 24: using remove method.

```
deque.remove();
System.out.println(deque);
```

Test Case 25: using offerFirst method with valid parameter and with null.

```
deque.offerFirst(1);
System.out.println(deque);

deque.offerFirst(null);
System.out.println(deque);

deque.offerFirst(2);
System.out.println(deque);
```

Test Case 26: using offerLast method with valid parameter and with null.

```
deque.offerLast(5);
System.out.println(deque);

deque.offerLast(null);
System.out.println(deque);

deque.offerLast(6);
System.out.println(deque);
```

Test Case 27: using addFirst method with valid parameter and with null.

```
try {
deque.addFirst(3);
System.out.println(deque);

deque.addFirst(null);
System.out.println(deque);

deque.addFirst(4);
System.out.println(deque);
} catch(NullPointerException e) {
    System.out.println(e);
}
```

Test Case 28: using addLast method with valid parameter and with null.

```
try {
deque.addLast(7);
System.out.println(deque);

deque.addLast(null);
System.out.println(deque);
deque.addLast(8);

System.out.println(deque);
} catch(NullPointerException e) {
    System.out.println(e);
}
```

Test Case 29: using removeFirstOccurence method.

```
if(deque.removeFirstOccurrence(10))
    System.out.println("The element 10 is removed");
else
    System.out.println("The element 10 is not removed");
System.out.println(deque);

if(deque.removeFirstOccurrence(1))
    System.out.println("The element 1 is removed");
else
    System.out.println("The element 1 is not removed");
System.out.println(deque);
```

Test Case 30: using removeLastOccurence method.

```
if(deque.removeLastOccurrence(11))
    System.out.println("The element 11 is removed");
else
    System.out.println("The element 11 is not removed");
System.out.println(deque);

if(deque.removeFirstOccurrence(2))
    System.out.println("The element 2 is removed");
else
    System.out.println("The element 2 is not removed");
System.out.println(deque);
```

Test Case 31: using iterator method.

```
Iterator<Integer> iter = deque.iterator();
System.out.print("My deque: ");
while(iter.hasNext())
    System.out.print(iter.next() + " ");
```

Test Case 32: using descendingIterator method.

```
Iterator<Integer> descIter = deque.descendingIterator();
System.out.print("\nMy deque: ");
while(descIter.hasNext())
    System.out.print(descIter.next() + " ");
```

Q3:

PART 1:

Test Case 1: strings containing different numbers of words

```
System.out.println("Seyda");
System.out.println(rec.reverseString("Seyda"));

System.out.println("this function writes the sentence in reverse");
System.out.println(rec.reverseString("this function writes the sentence in reverse"));

System.out.println("Deque is a queue that supports adding or removing items from both ends of the data structure");
System.out.println(rec.reverseString("Deque is a queue that supports adding or removing items from both ends of the data structure"));
```

Test Case 2: empty string

```
System.out.println("Testing empty string");
System.out.println(rec.reverseString(""));
```

Test Case 3: string is null

```
System.out.println("Testing null");
System.out.println(rec.reverseString(null));
```

PART 2:

Test Case 1: String constains e, l, f

```
System.out.println("result (tasteful): " + rec.IsElfish("tasteful", 0));
```

Test Case 2: String constains e, l, f, f

```
System.out.println("result (waffles): " + rec.IsElfish("waffles", 0));
```

Test Case 3: String constains e

```
System.out.println("result (Seyda): " + rec.IsElfish("Seyda", 0));
```

Test Case 4: String constains e, l

```
System.out.println("result (translate): " + rec.IsElfish("translate", 0));
```

Test Case 5: String constains e, f

```
System.out.println("result (safe): " + rec.IsElfish("safe", 0));
```

Test Case 6: String contains e, l , l

```
System.out.println("result (cell): " + rec.IsElfish("cell", 0));
```

Test Case 7: String is empty

```
System.out.println("result (empty string): " + rec.IsElfish("", 0));
```

Test Case 8: null

```
System.out.println("result null: " + rec.IsElfish(null, 0));
```

PART 3:

Test Case 1: arrays of different sizes

```
int [] array1 = {4, 6, 7, 2, 5};

rec.selectionSort(array1, 0);

int [] array2 = {10, 15, 70, 40, 22, 9, 2, 25, 46, 0};

rec.selectionSort(array2, 0);
```

Test Case 2: empty array

```
int [] array3 = {};
```

```
rec.selectionSort(array3, 0);
```

PART 4:

Test Case 1: strings of different sizes

```
rec.evaluatePostfix("10 23 32 + *", stack1)
```

```
rec.evaluatePostfix("12 2 * 7 - 135 * 12 +", stack1)
```

```
rec.evaluatePostfix("35 1000 * 2000 - 33 / 19000 16 + 26 -", stack1)
```

Test Case 2: empty string

```
rec.evaluatePostfix("", stack1)
```

Test Case 3: null

```
rec.evaluatePostfix(null, stack1)
```

PART 5:

Test Case 1: strings of different sizes

```
rec.evaluatePrefix("* + 32 23 10", stack2)
```

```
rec.evaluatePrefix("+ 12 * 135 - 7 * 2 12", stack2)
```

```
rec.evaluatePrefix("- 26 + 16 19000 / 33 - 2000 * 1000 35", stack2)
```

Test Case 2: empty string

```
rec.evaluatePrefix("", stack1)
```

Test Case 3: null

```
rec.evaluatePrefix(null, stack1)
```

PART 6:

Test Case 1: Matrices of dirrefent sizes

```
int [][] arr = {{0,1,2,3}, {13,14,15,4}, {12,19,16,5}, {11,18,17,6}, {10,9,8,7} };
```

```
rec.printMatrix(arr, 0, 0, 1, 0, arr.length-1, 0, arr[0].length-1);
```

```
int [][] arr2 = {{1, 2, 3, 4, 5, 6, 7},
                 { 8, 9,10,11,12,13,14},
                 {15,16,17,18,19,20,21},
                 {22,23,24,25,26,27,28},
                 {29,30,31,32,33,34,35},
                 {36,37,38,39,40,41,42}};

rec.printMatrix(arr2, 0, 0, 1, 0, arr2.length-1, 0, arr2[0].length-1);

int [][] arr3 = {{1, 2, 3, 4, 5, 6, 7, 8},
                 { 9,10,11,12,13,14,15,16},
                 {17,18,19,20,21,22,23,24},
                 {25,26,27,28,29,30,31,32},
                 {33,34,35,36,37,38,39,40},
                 {41,42,43,44,45,46,47,48},
                 {49,50,51,52,53,54,55,56},
                 {57,58,59,60,61,62,63,64}};

rec.printMatrix(arr3, 0, 0, 1, 0, arr3.length-1, 0, arr3[0].length-1);
```

RUNNING AND RESULTS

Q2:

Test Case 1: when deque is empty, using peek method.

```
This deque is empty.

null
```

Test Case 2: when deque is empty, using peekFirst method.

```
This deque is empty.

null
```

Test Case 3: when deque is empty, using peekLast method.

```
This deque is empty.

null
```

Test Case 4: using peekFirst method.

```
[ 3 2 1 5 6 7 ]

The front element of the deque: 3
```

Test Case 5: using peekLast method.

```
[ 3 2 1 5 6 7 ]

The rear element of the deque: 7
```

Test Case 6: using peek method.

```
[ 3 2 1 5 6 7 ]

The element of the deque: 3
```

Test Case 7: when deque is empty, using element method.

```
This deque is empty.

null
```

Test Case 8: when deque is empty, using getFirst method.

```
This deque is empty.

null
```

Test Case 9: when deque is empty, using getLast method.

```
This deque is empty.

null
```

Test Case 10: using getFirst method.

```
 java.util.NoSuchElementException: This deque is empty.
```

Test Case 11: using getLast method.

```
java.util.NoSuchElementException: This deque is empty.
```

Test Case 12: using element method.

```
java.util.NoSuchElementException: This deque is empty.
```

Test Case 13: when deque is empty, using poll method.

```
This deque is empty.

null
```

Test Case 14: when deque is empty, using pollFirst method.

```
This deque is empty.

null
```

Test Case 15: when deque is empty, using pollLast method.

```
This deque is empty.

null
```

Test Case 16: using pollFirst method.

```
[ 3 2 1 5 6 7 ]

[ 2 1 5 6 7 ]
```

Test Case 17: using pollLast method.

```
[ 2 1 5 6 7 ]

[ 2 1 5 6 ]
```

Test Case 18: using poll method.

```
[ 2 1 5 6 ]

[ 1 5 6 ]
```

Test Case 19: when deque is empty, using remove method.

```
java.util.NoSuchElementException: This deque is empty.
```

Test Case 20: when deque is empty, using removeFirst method.

```
java.util.NoSuchElementException: This deque is empty.
```

Test Case 21: when deque is empty, using removeLast method.

```
java.util.NoSuchElementException: This deque is empty.
```

Test Case 22: using removeFirst method.

```
[ 1 5 6 ]
[ 5 6 ]
```

Test Case 23: using removeLast method.

```
[ 5 6 ]
[ 5 ]
```

Test Case 24: using remove method.

```
[ 5 9 ]
[ 9 ]
```

Test Case 25: using offerFirst method with valid parameter and with null.

Null is not inserted.

```
[ 1 ]
[ 1 ]
[ 2 1 ]
```

Test Case 26: using offerLast method with valid parameter and with null.

Null is not inserted.

```
[ 3 2 1 5 ]
[ 3 2 1 5 6 ]
```

Test Case 27: using addFirst method with valid parameter and with null.

```
[ 3 2 1 ]
java.lang.NullPointerException: This deque does not insert null elements.
```

Test Case 28: using addLast method with valid parameter and with null.

```
[ 3 2 1 5 6 7 ]
java.lang.NullPointerException: This deque does not insert null elements.
```

Test Case 29: using removeFirstOccurence method.

```
[ 9 11 10 9 11 10 10 ]
The element 10 is removed
[ 9 11 9 11 10 10 ]
The element 1 is not removed
[ 9 11 9 11 10 10 ]
```

Test Case 30: using removeLastOccurence method.

```
[ 9 11 9 11 10 10 ]
The element 11 is removed
[ 9 11 9 10 10 ]
The element 2 is not removed
[ 9 11 9 10 10 ]
```

Test Case 31: using iterator method.

```
My deque: 9 11 9 10 10
```

Test Case 32: using descendingIterator method.

```
My deque: 10 10 9 11 9
```

Q3:

PART 1:

Test Case 1: strings containing different numbers of words

```
Seyda
Seyda

this function writes the sentence in reverse
reverse in sentence the writes function this

Deque is a queue that supports adding or removing items from both ends of the data structure
structure data the of ends both from items removing or adding supports that queue a is Deque
```

Test Case 2: empty string

```
Testing empty string

```

Test Case 3: string is null

```
Testing null

```

PART 2:

Test Case 1: String constains e, l, f

```
result (tasteful): true
```

Test Case 2: String constains e, l, f, f

```
result (waffles): true
```

Test Case 3: String constains e

```
result (Seyda): false
```

Test Case 4: String constains e, l

```
result (translate): false
```

Test Case 5: String constains e, f

```
result (safe): false
```

Test Case 6: String contains e, l , l

```
result (cell): false
```

Test Case 7: String is empty

```
result (empty string): false
```

Test Case 8: null

```
result null: false
```

PART 3:

Test Case 1: arrays of different sizes

```
4 6 7 2 5
The sorting array:
2 4 5 6 7

rec.selectionSort(array1, 0);

10 15 70 40 22 9 2 25 46 0
The sorting array:
0 2 9 10 15 22 25 40 46 70

rec.selectionSort(array2, 0);
```

Test Case 2: empty array

```
The sorting array:

rec.selectionSort(array3, 0);
```

PART 4:

Test Case 1: strings of different sizes

```
Result of this postfix expression ( 10 23 32 + * ): 550

Result of this postfix expression ( 12 2 * 7 - 135 * 12 + ): 2307

Result of this postfix expression ( 35 1000 * 2000 - 33 / 19000 16 + 26 - ): 18990
```

Test Case 2: empty string

```
Result of this postfix expression ( empty string ): 0
```

Test Case 3: null

```
Result of this postfix expression ( null ): 0
```

PART 5:

Test Case 1: strings of different sizes

```
Result of this prefix expression ( * + 32 23 10 ): 550

Result of this prefix expression ( + 12 * 135 - 7 * 2 12 ): 2307

Result of this prefix expression ( - 26 + 16 19000 / 33 - 2000 * 1000 35 ): 18990
```

Test Case 2: empty string

```
Result of this postfix expression ( empty string ): 0
```

Test Case 3: null

```
Result of this postfix expression ( null ): 0
```

PART 6:

Test Case 1: matrices of different sizes

First array:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Second array:

```
1 2 3 4 5 6 7 14 21 28 35 42 41 40 39 38 37 36 29 22 15 8 9 10 11 12 13 20 27 34 33 32 31 30 23 16 17 18 19 26 25 24
```

Third array:

```
1 2 3 4 5 6 7 8 16 24 32 40 48 56 64 63 62 61 60 59 58 57 49 41 33 25 17 9 10 11 12 13 14 15 23 31 39 47 55

54 53 52 51 50 42 34 26 18 19 20 21 22 30 38 46 45 44 43 35 27 28 29 37 36
```