I have another piece of code, I just want for you to understand my mental processes and analyse it..

```
/*
Online Java - IDE, Code Editor, Compiler

Online Java is a quick and easy tool that helps you to build, compile, test your programs online.
*/

import java.util.*;

public class Main
{
    static String sequence;
    static int n=1;

    final static String str = "123456789";   //whole block ascending
    //final static String str = "123124125";  //each 3 block ascending and digit in each block ascending
    //final static String str = "123124215";    // each 3 block ascending ONLY   (215 not ascending)

    //final static String str = "101112131415";
    //final static String str = "32332432536";
    //final static String str = "326325324323";
    //final static String str = "666667";


    final static String strBackup=str;
    static StringJoiner sj = new StringJoiner(",");
    static StringBuilder ascNums = new StringBuilder("");

    public static void main(String[] args)
    {
        System.out.println("Welcome to Online IDE!! Happy Coding :)");
        System.out.println("Number to be examined: " + str);
        ascending(str);
    }

    public static String ascending(String str)
```

```
    {
        String blockBefore="";
        System.out.println("This is the block size: " + n);

        if((str.length()%n==0))
        {
            if (!(str.length()<(2*n)))
            {
                System.out.println("There are minimum of two
blocks of size: " + n);
                String block = str.substring(str.length()-n,
str.length());

                try
                {
                    blockBefore = str.substring((str.length() -
(2*n)),(str.length()-n));

                    System.out.println("This is block: " + block);
                    System.out.println("This is block before: " +
blockBefore);

                    if
((Long.valueOf(blockBefore))<Long.valueOf(block))
                    {
                        String blockBeforeBlock =
str.substring((str.length() - (3*n)),(str.length()-(2*n)));

                        System.out.println("Performing recursion,
examining block " + "("+blockBefore+")" +" with block
before it" + "("+blockBeforeBlock+")");
                        sj=new StringJoiner(",");

                        if (ascNums.length()==0)
                        {
                            sj.add(blockBeforeBlock);
                            sj.add(blockBefore);
                            sj.add(block);
                        }
                        else
                        {
                            sj.add(blockBeforeBlock);
                        }
```

```
                    }

                    sequence = sj.toString()+",";

                    ascNums.insert(0,sequence);

                    System.out.println("*****THIS IS THE
SEQUENCE: " + ascNums);

                    return ascending(str.substring(0,
(str.length()-n)));
                }
                else
                {
                    n=n+1;
                    System.out.println("Performing recursion,
restoring original String");
                    sj=new StringJoiner(",");
                    ascNums=new StringBuilder("");

                    return ascending(strBackup);
                }
            }
            catch (StringIndexOutOfBoundsException |
ArrayIndexOutOfBoundsException e)
            {

        if (ascNums.length()==0)
                {
                    sj.add(blockBefore);
                    sj.add(block);
                    ascNums.insert(0,sj.toString());
                }

                System.out.println(blockBefore + " are digit(s)
of first block n"+"("+n+")" + " of " + strBackup);
                System.out.println(strBackup + " consists of
ascending numbers of group size" +"("+n+"): " +
ascNums);

                System.exit(0);
                return null;
            }
```

```
            }
        }

        else
        {
            n++;
            System.out.println("Value of block size has been
increased to: " + n);
            sequence="";
            ascNums=new StringBuilder("");
            sj=new StringJoiner(",");

            str=strBackup;

            System.out.println("String length: " + str.length());
            System.out.println("Block size: " + n);

            if (str.length()%n==0)
            {
                return ascending(str);
            }
        }

        System.out.println(strBackup + " does not consist of
ascending number in any block size " + n + "(n): " +
ascNums);
        System.exit(0);
        return null;

    }
}
```

Thanks for sharing this! Let's carefully unpack your mental process and analyze what this code is doing. I'll go step by step.

High-Level Purpose

From reading your code, your goal seems to be:

1. Take a **string of digits** (like `"123456789"`).
2. Try to split it into **blocks of size** `n`, where each block is interpreted as a number.

3. Check whether the numbers **form an ascending sequence**.
4. If not, **increase** `n` and try again, recursively, until a valid ascending block sequence is found or all possibilities are exhausted.

The program also handles several variants:

- Ascending overall (`123456789`).
- Ascending within 3-digit blocks (`123124125`).
- Cases where some blocks are not ascending (`123124215`).

Key Variables & Logic

Variable	Purpose
`str`	The input string you want to examine.
`n`	Current block size (starts at 1, incremented if sequence fails).
`strBackup`	A backup of the original string (used to restart recursion with new `n`).
`sj`	`StringJoiner` to build a comma-separated list of ascending blocks.
`ascNums`	`StringBuilder` storing the current ascending sequence.
`sequence`	Temporary string holding joined blocks for insertion into `ascNums` .

Core Logic

1. **Check divisibility**:

   java Copy code

   ```java
   if((str.length() % n == 0))
   ```

   Ensures blocks of size `n` fit evenly into the string.

2. **Handle the last block**:

   java Copy code

   ```java
   String block = str.substring(str.length()-n, str.length());
   ```

```
String blockBefore = str.substring(str.length()-(2*n), str.length()-n);
```

- `block` → the last `n` digits
- `blockBefore` → the block immediately before the last

3. **Compare numeric values**:

java Copy code

```
if ((Long.valueOf(blockBefore)) < Long.valueOf(block))
```

Checks if the sequence is ascending for these two blocks.

4. **Recursive call**:

java Copy code

```
return ascending(str.substring(0,(str.length()-n)));
```

- If ascending, recurse on the **remaining string** (removing the last block).
- Builds the sequence in `ascNums` by prepending blocks.

5. **Failure case**:
   - If blocks are **not ascending**, increase `n` and restart with `strBackup`.
6. **Edge handling**:
   - `StringIndexOutOfBoundsException` occurs when recursion reaches the first block.
   - At that point, the sequence is complete, and program prints it.

Observations on Your Mental Process

From this code, I see you're thinking in **these steps**:

1. **Divide-and-conquer**: Examine the string from the end, checking small blocks, then gradually extending block size.
2. **Recursion-first approach**: You rely heavily on recursion to "peel off" blocks from the end.
3. **Dynamic block size**: You realize not all strings can be split with block size 1, so you systematically try larger block sizes.
4. **Robustness**: You handle exceptions (like running out of string) as signals that recursion has finished.
5. **Tracking sequences**: You carefully build `ascNums` with a `StringJoiner`, inserting blocks in correct order.

Strengths

- Very systematic approach to finding ascending blocks.
- Flexible: works with different block sizes and sequences.
- Recursive logic matches the mental model of "check last block, then recurse on remaining."

Potential Improvements / Considerations

1. `static` **variables for state** (`n`, `ascNums`, `sj`) make recursion less clean:
   - Recursion is easier to reason about if all state is **passed as parameters**, rather than modified globally.
2. **Repeated code & redundancy**:
   - Reinitializing `sj`, `ascNums` multiple times is verbose. Could be simplified.
3. **Performance**:
   - Each time the sequence fails, it restarts from scratch with `strBackup`. Could try smarter strategies to avoid rechecking known valid parts.
4. **Long conversion**:
   - You use `Long.valueOf()` to compare blocks. Works for small blocks, but may fail for very long strings (>19 digits). Could compare strings lexicographically instead.

Summary

Your code is essentially implementing a **block-wise ascending sequence detector with recursive backtracking and dynamic block size search**.

- It's a very explicit, step-by-step exploration, matching a **mental model of peeling off last blocks and checking order**.
- There's careful handling of edge cases and systematic adjustment of `n`.

In short: **you're thinking like a recursive detective, examining each block carefully and willing to backtrack when things fail.**

If you want, I can **draw a step-by-step diagram showing how your recursion unfolds for** `"123124125"` — it really makes your thought process visual. It would make your mental flow obvious at a glance.

Do you want me to do that?