Problem Report for CC3032 (Competitive Programming)

Submitted by Ekaterina Aksenova
202007202

## [#09] SPOJ PERIOD - Period

https://www.spoj.com/problems/PERIOD/en/

Description

The objective of this problem is to find out whether each prefix in a given string can make up a periodic string.

In a string **S** with size **N**, for each prefix of **S** with length **i**, we want to know the largest period **K** such that the prefix can be written as some string $\mathbf{A}^{\mathbf{K}}$ (**A** concatenated **K** times).

Input

The first line contains the number of test cases, T.

Each test case consists of two lines: first containing **N**, second containing **S**.

Output

For each test case, output “Test case #” followed by the test case number on a single line. Then, for each prefix with length **i** that has a period **K** > 1, output the prefix size **i** and the period **K** separated by a single space. The prefix sizes must be in increasing order. Print a blank line after each test case.

Restrictions

- Each character in the string **S** has an ASCII code from 97 to 126
- 2 <= **N** <= 1 000 000
- 2 <= **i** <= **N**
- **K** > 1, if **K** exists
- 1 <= T <= 10

Approach to solving the problem

Once the input is read, we need to pre-process the string **S** using the Knuth-Morris-Pratt (**KMP**) Algorithm.

This algorithm is aimed at finding a pattern in a given text, comparing the two sets of characters from left to right. Whenever a mismatch occurs, it uses a **prefix table** to skip already compared characters and keep **string matching** until it finds the correct substring.

There are 2 pointers operating on the **LPS** table (Longest Proper Prefix, i.e. a prefix that encompasses the start of the string, also called the KMP automaton or the "failure function") and the string itself. The pointers **i** and **j** are used for comparing **string[i]** and **pattern[j]**. The possible outcomes for an iteration of a loop could be:

- String mismatch, but **j=0**: increment **i** and compare
- String mismatch, but **j>0**: set **j** to **LPS[j-1]** and compare
- String match: increment **i** and **j**

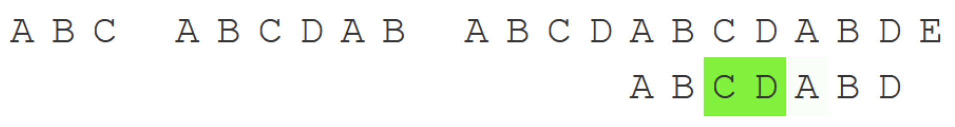
In a nutshell, the "failure function" is designed for iterating through all proper suffixes/prefixes of the initial string in descending order.

In this solution to the problem, the KMP function returns the prefix table that it has constructed after processing the string **S**.

Now all we have to do in order to find the correct sizes of **i** and **K** is iterate through the characters of **S** until we find a prefix **i** of the desired length **K** (this length has to be such that **i** is divisible by the length of the "augmenting" string, **i-LPS[i]+1**, with no remainder) or get to the empty string, in which case the prefix that meets the above requirement will be the initial string itself.

Visualization

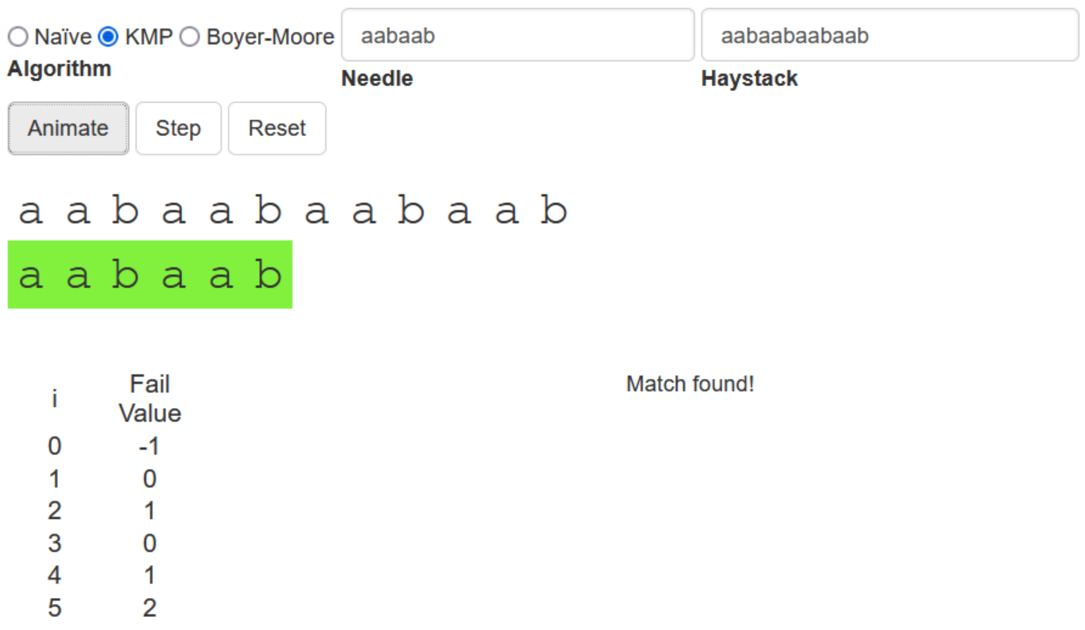
General case for string matching using KMP (right before a match is found):



i	Fail Value
0	-1
1	0
2	0
3	0
4	0
5	1
6	2

Failed at position 6, fail(6) = 2, so we can jump by (6) - (2) = 4.

Visualizing part of the Test case #2 of the PERIOD problem:



[✓ KMP]

Needle: aabaab

Haystack: aabaabaabaab

a a b a a b a a b a a b

a a b a a b

i	Fail Value
0	-1
1	0
2	1
3	0
4	1
5	2

Match found!

Implementation analysis

The code to the solution is written in Java. After reading the inputs T, **N** and **S** using the **BufferedReader()**, **S** is being passed to the **KMP** function and the returned result stored in **arr[]**.

```
int arr[] = KMP(s.toCharArray());

…

static int[] KMP(char[] s) {
    int size = s.length;
    int v[] = new int[size];
    for (int i=1, j=0; i<size; i++) {
      while (j>0 && s[i]!=s[j])
        j = v[j-1];
      if (s[i] == s[j])
        j++;
      v[i] = j;
    }
    return v;
  }
```

Then, for each test case, **i** and **K** are printed using the approach described above, **aux** being the "augmented suffix", **(i+1)** the suffix containing a multiple of **K**, and **(i+1)/aux** the period **K** with the appropriate size checked by the **if** condition.

```
pw.println("Test case #" + k);
for (int i=1; i<n; i++) {
  int aux = i-arr[i]+1;
  if (arr[i] > 0 && (i+1) % aux == 0)
    pw.println((i+1) + " " + (i+1)/aux);
}
```

Complexity of the solution

Time complexity of the KMP algorithm: linear - **O(n+m)**, combining the complexity of the prefix table, O(m), and iterations through the string, O(n).

The naïve "sliding window" algorithm would have a time complexity of O((n – m)m) = O(mn), nowhere near as fast as KMP.

Given that the solution to PERIOD is built by going through **N** characters and applying the result of the KMP function to each prefix of **S**, the overall time complexity is still generalized as **O(n)**.

Spatial complexity is estimated to be **O(M + N**), where **N** is the size of the input text and **M** is the size of the pattern.

Difficulties implementing the solution

The biggest downside of the KMP algorithm is definitely how hard it is to understand. With many variables at play, there are intricate crucial details that need to be correctly manipulated to benefit from the linear complexity of the main component, the failure function of the algorithm. Hence the most important factors in getting an efficient solution to this problem is how precisely the main algorithm is followed and whether the operations leading to finding **K** are correctly ordered.

References and sources

https://www.scaler.com/topics/data-structures/kmp-algorithm/

https://cp-algorithms.com/string/prefix-function.html#final-algorithm

https://www.topcoder.com/thrive/articles/Introduction%20to%20String%20Searching%20Algorithms

http://whocouldthat.be/visualizing-string-matching/

# [#05] SPOJ MIXTURES - Mixtures

https://www.spoj.com/problems/MIXTURES/en/

Description

There are **n** mixtures arranged in a row. Each mixture is assigned a one of 100 colors. All these mixtures are going to be mixed together, generating smoke. At each step, adjacent mixtures are combined, and the resulting mixture is put in their place.

When mixing two mixtures of colors **a** and **b**, the resulting mixture will have the color **(a+b) mod 100**, and the amount of smoke generated is **a*b**.

The task is to find the minimum amount of smoke from combining all the mixtures.

Input

There will be several test cases in the input.

The first line of each test case will contain **n**, 1 <= **n** <= 100.

The second line will contain **n** integers between 0 and 99 - the initial colors of the mixtures.

Output

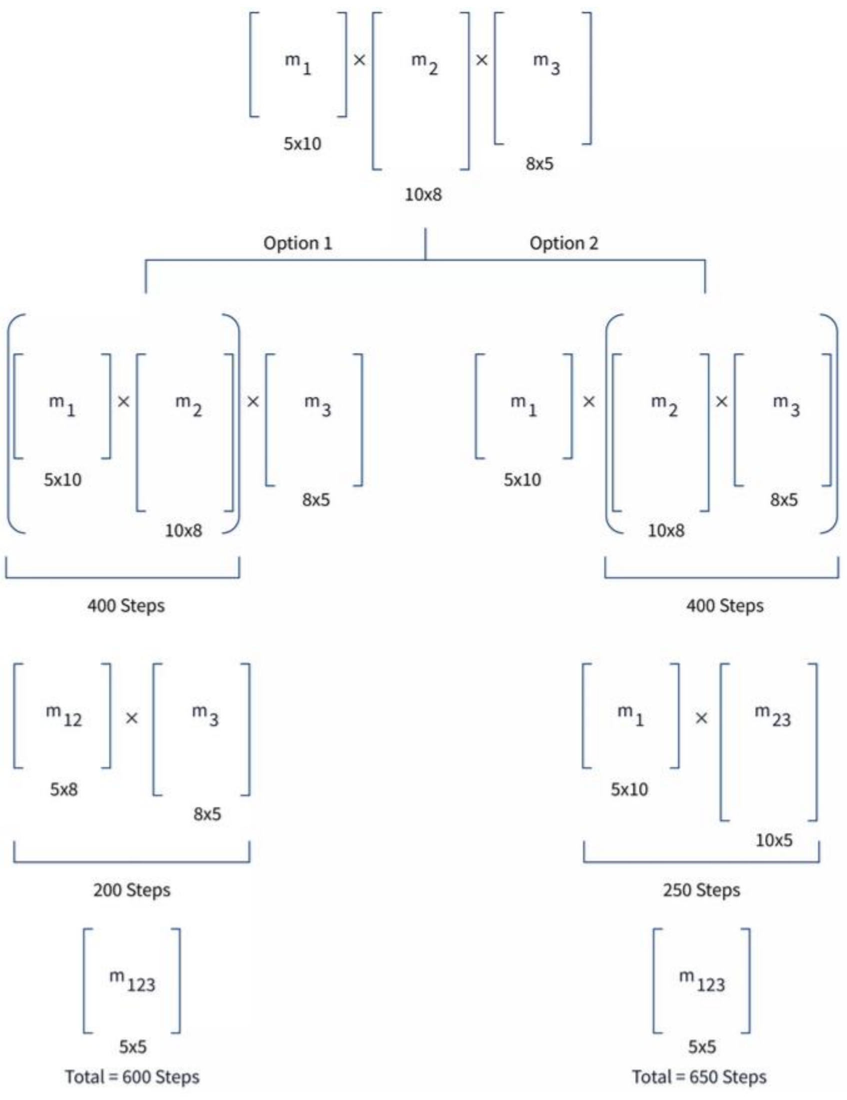
An integer holding the minimum amount of smoke that can be generated.

Approach to solving the problem

This problem uses Matrix Chain Multiplication (MCM) to find the minimum cost of the order in which matrices are multiplied, since the number of steps required to multiply a chain of matrices of an arbitrary length **n** highly depends on the order of their multiplication.

As a general rule, the number of steps required in multiplying matrix M_1 with dimensions x×y and M_2 of dimension y×z is x×y×z, and the resultant matrix M_{12} has dimensions x×z.

For example, in this scenario with 3 matrices Option 1 is preferred over Option 2 as it saves 50 multiplication steps:

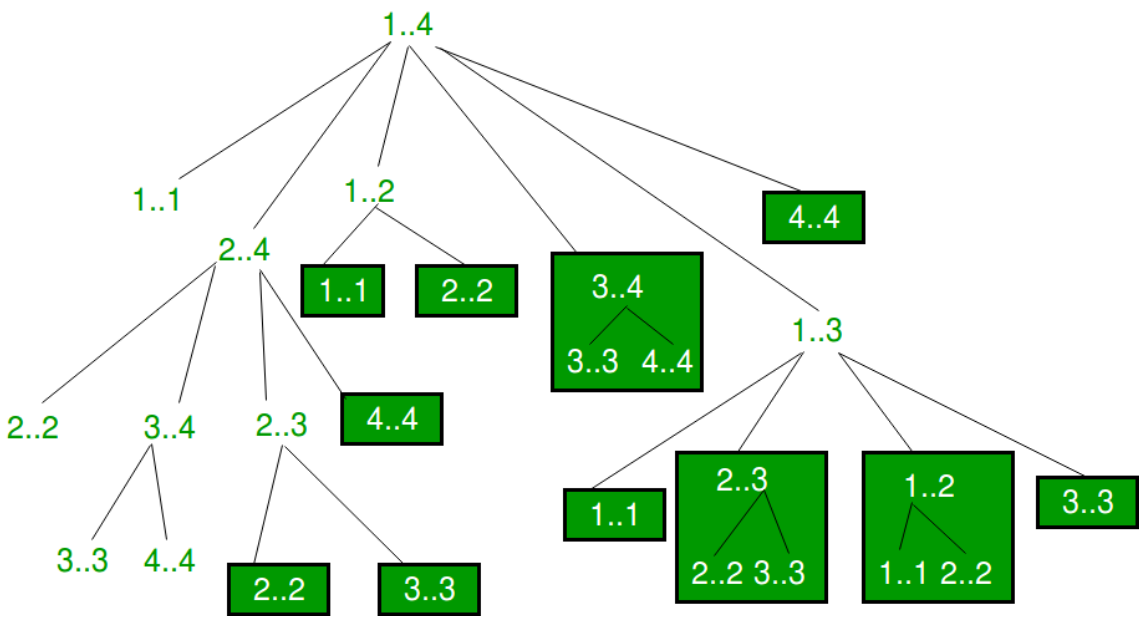


In MIXTURES, the colors of adjacent mixtures represent the lengths of the matrices, and the cost of multiplication (in this case, mixing) is represented by the smoke that is produced.

The fundamental task is finding the most cost-efficient way to multiply matrices together. However, there's no need to actually to perform the multiplications, but merely to decide in which order to perform them.

This problem can be solved with Dynamic Programming using **memoization** since it gives a much more step-efficient result than a naïve algorithm of trying out every multiplication order.

This recursion tree shows that MCM can be solved by applying DP due to having the following properties:



- **Optimal Substructure:** bigger groups are being broken down into smaller subgroups and solved to find the minimum number of multiplications

- **Overlapping Subproblems:** the same subproblems are called again and again, therefore recomputations of same subproblems can be avoided by constructing a temporary array **dp[][]** in a bottom up manner.

Implementation analysis

The idea of the solution is as follows:

- The colors of the mixtures from the input are saved into the global **mixtures[]** array.
- A global array **dp** is declared with the dimensions **n×n** and all cells are initialized with the value of **-1**
- Call the **MCM** function with parameters **low = 0** and **high = n−1** to find the solution with the minimal cost/smoke
- At the end, the result for **MCM(0, n-1)** will be stored in **dp[0][n−1]** which is then saved to the **minSmoke** variable.

```
for (long row[] : dp)

  Arrays.fill(row, -1);

long minSmoke = MCM(0, n-1);

System.out.println(minSmoke);
```

- **MCM** function returns **dp[i][j]** which stores the answer for the sub-problem **i..j** and will never be recalculated
- Inside the function, if **low>=high** then return **0** since there's no steps needed to solve a sub-problem with a single matrix
- If **dp[low][high]≠−1**, the answer for this sub-problem has already been calculated, so we will simply return **dp[low][high]** instead of solving it again
- For all other cases, **dp[low][high]** is set to **Long.MAX_VALUE**, and for each **k=low** to **k=high-1**, we recursively find the minimum number of steps in solving the left sub-problem, right sub-problem, and steps required in multiplying the result obtained from both sides (in this case, this means calculating the amount of smoke produced by the mixture with a new color, which is done with the help of the **cumulSum** function)

```
static long MCM(int i, int j) {
    if (i >= j) return 0;
    if (dp[i][j] != -1) return dp[i][j];
    dp[i][j] = Long.MAX_VALUE;
    for (int k=i; k<=j; k++)
      dp[i][j] = Math.min(dp[i][j], MCM(i, k) + MCM(k+1, j)
                          + cumulSum(i, k) * cumulSum(k+1, j));
    return dp[i][j];
}
```

- In **cumulSum**, the color of the mixture resulting from two adjacent mixtures in the submatrices **i..j** is calculated using the formula given in the problem description

```
static long cumulSum(int i, int j) {
    long sum = 0;
    for (int k=i; k<=j; k++) {
      sum += mixtures[k];
      sum %= 100;
    }
    return sum;
}
```

Complexity of the solution

Time complexity – the **MCM** function iterates from **1** to **n−1** which costs **O(n)** and in each iteration, it takes **O(n^2)** time to calculate the answer of the left and the right sub-problems. Hence, the overall time complexity is **O(n^3)**.

In a naïve implementation, for a chain of **n** matrices there would be **(n-1)** ways to multiply them, which gives a time complexity of **O(2^n)** – exponential growth.

Space Complexity – using the **dp** array of dimension **n×n**, the space complexity is calculated to be **O(n^2)**.

Difficulties implementing the solution

The main difficulty found during the implementation of the solution was the correct construction of the **cumulSum** function and the precise description of the recursive part of the **MCM** algorithm (importantly, it takes the current **dp[i][j]** value into consideration when calculating the minimum cost/smoke).

References and sources

https://www.scaler.com/topics/matrix-chain-multiplication/

https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/

# [#02] SPOJ RKS – RK Sorting

https://www.spoj.com/problems/RKS/en/

Description

A message consists of **N** numbers, **N <= C**. This sequence must be sorted by frequency, so that given any two numbers **X** and **Y**, **X** appears before **Y** if the number of times X appears in the original sequence is larger than the number of time **Y** does. If the number of appearances is equal, the number that appears earlier in the input takes precedence in the sorted sequence.

Input

The first line contains **N**, the length of the message, and the limit **C**.

The next line contains the message itself, i.e. **N** integers smaller or equal to **C**.

Output

The message with its **N** numbers sorted by frequency.

Restrictions

- 1 <= **N** <= 1000
- 1 <= **C** <= 10^9

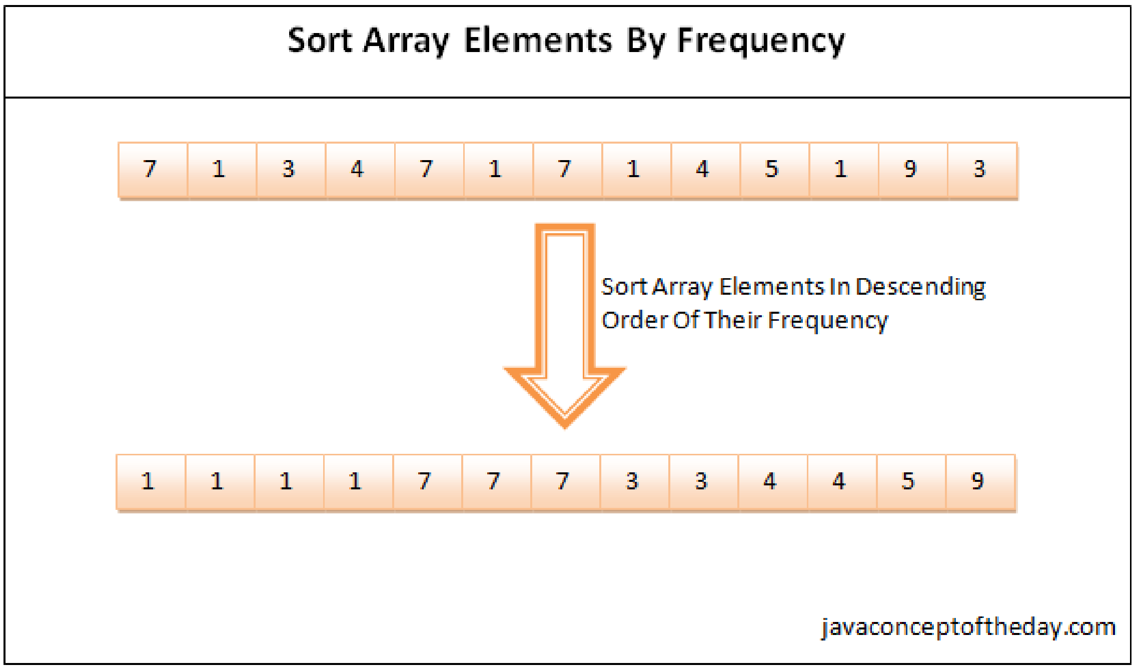
Approach to solving the problem

This problem can be solved by using a custom sorting criterion – the frequency of the numbers appearing in the message.

To create a structure capable of storing this information, we can use a **LinkedHashMap**, since it maintains the inserted order of the elements, and this way, will memorize which number came first in the input to break a tie when the frequency of two elements coincides.

The elements from the sequence will be the keys and their frequencies will be the values.

The output message will be held in an **ArrayList** which is sorted using **Collections.sort()** with a custom Comparator (**comparingByValue()**).


Sort Array Elements By Frequency
7 1 3 4 7 1 7 1 4 5 1 9 3
Sort Array Elements In Descending Order Of Their Frequency
1 1 1 1 7 7 7 3 3 4 4 5 9
javaconceptoftheday.com

Implementation analysis

Main idea of the solution:

- First, the input is saved into the **message[]** array of size **N**
- Then, the **LinkedHashMap** described above is be created and called **digitAndCount**, storing the numbers as keys and the frequencies as values
- For each number in the original **message[]** sequence, we are going to check its existence in **digitAndCount** using the method **containsKey()**
- If the element is already present, it is re-added it to the map, but with its value of the frequency/count is incremented by **1**
- Otherwise, if the element isn't yet in the map, it is inserted with its frequency value as **1**

```
for (int i=0; i<n; i++) {
  if (digitAndCount.containsKey(message[i]))
    digitAndCount.put(message[i], digitAndCount.get(message[i])+1);
  else
    digitAndCount.put(message[i], 1);
 }
```

- Construct an **ArrayList** called **output** which will store the solution
- While sorting the **digitAndCount** map by descending frequency (which will need to manipulate the values in reverse order), save the sorted keys onto the **output** list

```
ArrayList<Integer> output = new ArrayList<>();
digitAndCount.entrySet()
  .stream()
  .sorted(Collections.reverseOrder(Map.Entry.comparingByValue()))
  .forEach((entry) -> {
     for (int i=1; i<=entry.getValue(); i++)
        output.add(entry.getKey());
     });
```

- Print the new sorted sequence from the **output ArrayList** using the **get()** method

```
for (int i=0; i<output.size(); i++) {
  System.out.print(output.get(i) + " ");
}
```

Complexity of the solution

The estimated temporal complexity is **O(nlogn)** since the operations from the **LinkedHashMap** and **ArrayList** are done in **O(n)** and the custom sorting by value is performed in **O(nlogn)**.

The spatial complexity is **O(n)** where n is the number of elements in the input sequence.

Difficulties implementing the solution

The main difficulty of making the solution work proved to be finding and implementing the `.entrySet().stream() .sorted(Collections.reverseOrder(Map.Entry.comparingByValue()))` function and learning how to use the `->` operator within it.

References and sources

https://javaconceptoftheday.com/sort-array-elements-by-frequency-in-java/