Coding Problems - Study Notes

Problem 1: Generating Valid OTPs

Problem Statement

George is a computer programmer who needs to create a program that generates a list of valid OTPs for a new website. He is given a numerical string `Num`.

Criteria for Valid OTP

1. The OTP must be an integer value between 1 and Num (inclusive)
2. The OTP must not have leading zeroes
3. The OTP must have an even number of digits
4. The first half of the OTP must be the same as the second half of the OTP

Input Specification

- **input1**: A string Num, representing the maximum value for the OTP

Output Specification

- Return a string representing the total number of valid OTPs that can be generated for the given value of Num

Examples

Example 1:

Input: `33` **Output**: `3` **Explanation**:

- Valid OTP 1 is 11
- Valid OTP 2 is 22
- Valid OTP 3 is 33
- Total valid OTPs: 3

Example 2:

Input: `1010` **Output**: `10` **Explanation**:

- Valid OTPs: 11, 22, 33, 44, 55, 66, 77, 88, 99, 1010
- Total valid OTPs: 10

Solution Code

```
import java.io.*;
import java.util.*;

class UserMainCode {
    public String validOtp(String input1) {
        long num = Long.parseLong(input1);
        int count = 0;

        int numDigits = input1.length();

        for (int totalDigits = 2; totalDigits <= numDigits * 2; totalDigits += 2) {
            int halfDigits = totalDigits / 2;

            long firstHalfStart = (long) Math.pow(10, halfDigits - 1);
            long firstHalfEnd = (long) Math.pow(10, halfDigits) - 1;

            for (long firstHalf = firstHalfStart; firstHalf <= firstHalfEnd; firstHalf+
+) {
                String otpStr = String.valueOf(firstHalf) + String.valueOf(firstHalf);
                long otp = Long.parseLong(otpStr);

                if (otp <= num) {
                    count++;
                } else {
                    break;
                }
            }

            String smallestOtpStr = String.valueOf(firstHalfStart) + String.valueOf(fir
stHalfStart);
            long smallestOtp = Long.parseLong(smallestOtpStr);
            if (smallestOtp > num) {
                break;
            }
        }

        return String.valueOf(count);
    }
}
```

Algorithm Explanation

1. For each even number of digits (2, 4, 6, etc.), generate valid OTPs
2. For an OTP with `totalDigits` digits, the first half has `halfDigits = totalDigits/2` digits
3. The first half ranges from `10^(halfDigits-1)` to `10^halfDigits - 1` (to avoid leading zeros)
4. For each first half value, create the OTP by concatenating it with itself
5. Count only those OTPs that are ≤ Num
6. Optimize by breaking early when OTPs exceed Num

Problem 2: Money Conversion

Problem Statement

Sally works at a money exchange center and deals with currencies involving Dollars, Rupees, and Pounds only. She needs to automate the conversion process to convert currencies to rupees.

Conversion Table

Currency	Symbol	Value in Rupees
Dollar	D	70 R
Pound	P	100 R
Rupee	R	1 R

Input Specification

- **input1**: A string representing money (numeric value followed by currency letter)
  - Format: "VALUE CURRENCY" (e.g., "120 D" represents 120 Dollars)

Output Specification

- Return the value of the currency in Rupees
- Note: Divide the converted value by 10000007 (use modulo operation)

Examples

Example 1:

Input: `120 D` **Output**: `8400` **Explanation**:

- 120 D represents 120 Dollars
- 120 × 70 = 8400
- 8400 % 10000007 = 8400
- Therefore, 8400 is returned as the output

Example 2:

Input: `240 P` **Output**: `24000` **Explanation**:

- 240 P represents 240 Pounds
- 240 × 100 = 24000
- 24000 % 10000007 = 24000
- Therefore, 24000 is returned as the output

Solution Code

```
import java.io.*;
import java.util.*;

class UserMainCode{
    public int moneyConversion(String input1){
        String[] parts = input1.trim().split(" ");
        long value = Long.parseLong(parts[0]);
        char currency = parts[1].charAt(0);

        long rupees = 0;

        if(currency == 'D'){
            rupees = value * 70;
        } else if(currency == 'P'){
            rupees = value * 100;
        } else if(currency == 'R'){
            rupees = value;
        }

        int result = (int)(rupees % 10000007);

        return result;
    }
}
```

Algorithm Explanation

1. Split the input string to extract the numeric value and currency letter
2. Parse the numeric value as a long integer
3. Get the currency character (D, P, or R)
4. Apply conversion:
    - If 'D': multiply by 70
    - If 'P': multiply by 100
    - If 'R': keep as is
5. Apply modulo 10000007 to the result
6. Return the final value as integer

Key Takeaways

Problem 1 - Valid OTPs

- Pattern recognition: Valid OTPs are formed by repeating a number (e.g., 11, 22, 1010)
- Use string concatenation to form OTPs
- Optimize with early breaking when exceeding the limit
- Handle large numbers with `long` data type

Problem 2 - Money Conversion

- String parsing and splitting techniques
- Currency conversion with fixed rates
- Modulo operation for large number handling
- Use `long` for intermediate calculations to avoid overflow

Testing Tips

1. Test with boundary values (minimum and maximum)
2. Test with different digit lengths
3. Verify output data types match requirements
4. Check for overflow scenarios with large inputs

Problem 3: Rearrange String

Problem Statement

James has given a string S of length N to his brother. He asked him to create a new string F by rearranging the entire string by following specific conditions.

Conditions for Rearrangement

1. Pick one letter from string S, at a time
2. Only that letter will be picked which has the highest occurrence in the string
3. The letter is removed from S and appended to F
4. If there are more than 1 letter having same occurrence, then the preference will be given to the letter which comes first in the lexicographic order
5. In case of conflict of occurrences, the preferences will be in below order:
    - **First preference**: Numeric value (0-9)
    - **Second preference**: Upper case letter (A-Z)
    - **Third preference**: Lower case letter (a-z)

Input Specification

- **input1**: An integer value N, representing the length of the string
- **input2**: A string value S

Output Specification

- Return a string value representing the final string F

Note

- You must perform the above operation till all the characters are completely removed from string S
- The input string may contain uppercase, lowercase English letters along with numbers from 0-9

Examples

Example 1:

Input1: `10` **Input2**: `abccaAdA11` **Output**: `1Aac1Aabcd`

Explanation:

- S = abccaAdA11
- Step 1: In the initial string '1', 'a', 'c' and 'A' have equal number of occurrences i.e., 2
  - As mentioned above, the preference will be given to '1', 'A', 'a' and 'c' respectively
  - So, F = 1Aac
- Step 2: Similarly, we will remove '1', 'A', 'a', 'b', 'c' and 'd' from S
  - Remaining: bcadA1
  - Now the final string F = 1Aac1Aabcd
- Therefore, 1Aac1Aabcd is returned as the output

Example 2:

Input1: `30` **Input2**: `28ucuBtIAZe3XkYr9edBr2eqk2X0z8` **Output**: (Process according to the rules)

Solution Code

```
import java.io.*;
import java.util.*;

class UserMainCode {
    public String rearrangeString(int input1, String input2) {
        StringBuilder result = new StringBuilder();
        Map<Character, Integer> freqMap = new HashMap<>();

        for (char c : input2.toCharArray()) {
            freqMap.put(c, freqMap.getOrDefault(c, 0) + 1);
        }

        while (!freqMap.isEmpty()) {
            int maxFreq = 0;
            for (int freq : freqMap.values()) {
                maxFreq = Math.max(maxFreq, freq);
            }

            List<Character> candidates = new ArrayList<>();
            for (Map.Entry<Character, Integer> entry : freqMap.entrySet()) {
                if (entry.getValue() == maxFreq) {
                    candidates.add(entry.getKey());
                }
            }

            char selected = selectCharacter(candidates);

            result.append(selected);

            int newFreq = freqMap.get(selected) - 1;
            if (newFreq == 0) {
                freqMap.remove(selected);
            } else {
                freqMap.put(selected, newFreq);
            }
        }

        return result.toString();
    }

    private char selectCharacter(List<Character> candidates) {
        List<Character> digits = new ArrayList<>();
        List<Character> uppercase = new ArrayList<>();
        List<Character> lowercase = new ArrayList<>();

        for (char c : candidates) {
            if (Character.isDigit(c)) {
                digits.add(c);
            } else if (Character.isUpperCase(c)) {
                uppercase.add(c);
            } else {
                lowercase.add(c);
            }
        }

        if (!digits.isEmpty()) {
            Collections.sort(digits);
            return digits.get(0);
        }

        if (!uppercase.isEmpty()) {
            Collections.sort(uppercase);
            return uppercase.get(0);
        }
```

```
            Collections.sort(lowercase);
            return lowercase.get(0);
        }
    }
```

Algorithm Explanation

1. **Build Frequency Map**: Create a HashMap to store the frequency of each character in the string
2. **Main Loop**: Continue until all characters are processed
    - Find the maximum frequency among all remaining characters
    - Identify all characters with the maximum frequency (candidates)
    - Select the appropriate character based on priority rules:
        - First priority: Numeric characters (0-9), pick smallest
        - Second priority: Uppercase letters (A-Z), pick smallest lexicographically
        - Third priority: Lowercase letters (a-z), pick smallest lexicographically
3. **Update**: Append selected character to result and decrease its frequency
4. **Remove**: If frequency becomes 0, remove the character from the map
5. **Repeat**: Continue until the frequency map is empty

Priority Selection Logic

```
If multiple characters have same frequency:
├─ Check for digits (0-9)
|  └─ Select smallest digit
├─ Check for uppercase (A-Z)
|  └─ Select smallest uppercase letter
└─ Check for lowercase (a-z)
   └─ Select smallest lowercase letter
```

Time Complexity

- **O(N × M)** where N is the length of string and M is the number of unique characters
- Each iteration finds max frequency and selects character

Space Complexity

- **O(M)** where M is the number of unique characters in the string

---

Summary of All Problems

Problem	Key Concept	Data Structure	Time Complexity
Valid OTPs	Pattern Recognition	String Manipulation	O(N)

Problem	Key Concept	Data Structure	Time Complexity
Money Conversion	String Parsing	Basic Operations	O(1)
Rearrange String	Frequency & Priority	HashMap, Lists	O(N × M)