July 24, 2025

New Zealand Informatics Competition 2025

Round Two

Editorial by Zalan V

Contents

Introduction

The solutions to this round of NZIC problems are discussed in detail below. In a few questions we may refer to the Big O complexity of a solution, e.g. $O(N)$. There is an explanation of Big O complexity at the end of this document.

Resources

Ever wondered what the error messages mean?

https://www.nzoi.org.nz/nzic/resources/understanding-judge-feedback.pdf

Read about how the server marking works:

https://www.nzoi.org.nz/nzic/resources/how-judging-works-python3.pdf

Ever wondered why your submission scored zero?

https://www.nzoi.org.nz/nzic/resources/why-did-i-score-zero.pdf

See our list of other useful resources here:

https://www.nzoi.org.nz/nzic/resources

Teacups

Problem authored by Zalan V
https://train.nzoi.org.nz/problems/1546

Subtask 1

In this subtask there are at most $1,000$ teacups. To solve this subtask we can represent the two shelves using two arrays (known as a list in Python). Then, for each teacup we check if its pair is already on shelf 2 by simply scanning the list. If the teacup has no pair then it is placed on shelf 2, otherwise the matching teacup is removed from shelf 2 and both are added to shelf 1. Checking whether an array contains an element has a time complexity of $O(N)$ so the overall time complexity of this solution is $O(N^2)$.

Python Solution

```
N = int(input())
shelf1 = []
shelf2 = []
for _ in range(N):
    teacup = int(input())
    if teacup in shelf2:
        shelf2.remove(teacup)
        shelf1.append(teacup)
        shelf1.append(teacup)
    else:
        shelf2.append(teacup)

shelf1.sort()
shelf2.sort()
print("Shelf 1:", *shelf1)
print("Shelf 2:", *shelf2)
```

Subtask 2

In this subtask every teacup is guaranteed to have a pair. Thus, all teacups will be placed on shelf 1, so we can simply sort all the teacups and output them on shelf 1.

Python Solution

```
N = int(input())
teacups = [int(input()) for _ in range(N)]
teacups.sort()
print("Shelf 1:", *teacups)
print("Shelf 2:")
```

Full Solution

In the Subtask 1 solution the slowest part of the algorithm is checking to see if we have seen a teacup before. A better approach is to sort all the teacups that we are given and then work out which shelf to put each teacup on. Once we have sorted the input, matching teacups will be next to each other in our array. This means that we can simply check if the next teacup has the same integer as the current one to find the pairs. The time complexity of this solution is $O(N \log(N))$.

Python Solution

The code for this solution can be found at the end of this document .

Spreadsheet Columns

Problem authored by Joseph G
https://train.nzoi.org.nz/problems/1339

Subtask 1

In this subtask the column name consists of a single letter. We can use the fact that the letters `A` through `Z` have ASCII codes 65 through 90 to calculate the answer.

Python Solution

```
print(ord(input()) - 64)
```

Subtask 2

In this subtask the column name consists of either one or two letters. If the column name is two characters long then we can multiply the value of the first column by 26 and add the two values together.

Python Solution

```
column = input()
if len(column) == 1:
    print(ord(column) - 64)
else:
    print((ord(column[0]) - 64) * 26 + ord(column[1]) - 64)
```

Subtask 3

In this subtask the column is at most column `KIVH` which is column number $200,000$. To solve this subtask we can simply generate all the column names up to `KIVH` and then find the index of the given column.

Python Solution

```
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
columns = list(ALPHABET)
for i in range(200_000 // 26):
    for c in ALPHABET:
        columns.append(columns[i] + c)

print(columns.index(input()) + 1)
```

Full Solution

We can extend the idea from Subtask 2 to solve the full problem. We can read the characters in the column name from left to right and at each step multiply the total by 26.

Python Solution

```
ans = 0
column = input()
for c in column:
    ans = ans * 26 + ord(c) - 64
print(ans)
```

Ribbons

Problem authored by Zalan V
https://train.nzoi.org.nz/problems/1553

Subtask 1

In this subtask $K = 1$, which means that we want to cut a ribbon that is at most M segments long and consists of a single colour. Thus, we need to find the lengths of all sections consisting of the same colour. We need to take care to treat sections longer than M specially, as a section with length $L > M$ could be cut into $L - M + 1$ different sections of length M.

Python Solution

```
N, M, K = map(int, input().split())
colours = list(map(int, input().split()))

best_length = 0
best_count = 0
colours.append(-1)
first = 0
for i in range(N):
    if colours[i] != colours[i + 1]:
        length = i - first + 1
        count = 1
        if M < length:
            count += length - M
            length = M

        if best_length < length:
            best_length = length
            best_count = count
        elif best_length == length:
            best_count += count

        first = i + 1

print(best_length, best_count)
```

Subtask 2

In this subtask $N \leq 100$. We can try every possible subarray of length at most M to see if it has at most K distinct elements. We can find the number of elements by constructing a hash set and checking the number of elements in the set. Constructing the hash set has a time complexity of $O(M)$ and there are roughly $N \times M$ possible subarrays, so the overall time complexity of this solution is $O(NM^2)$.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 3

In this subtask $N \leq 1,000$. We can observe that constructing the set of unique elements in the Subtask 2 solution adds a factor of $O(M)$ to the time complexity. For each starting position, we can instead extend the current ribbon segment from the until we have M segments, and stop when we have more than K unique elements. This works because a ribbon section that can be further extended will not

be optimal, so we can just extend each section as far as we can. The overall time complexity of this solution is $O(NM)$.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 4

In this subtask every colour is between 1 and 25. We can use a two pointers approach to solve this subtask. We maintain a lower_bound and upper_bound which describes a valid section, that is, upper_bound $-$ lower_bound $< M$ and there are at most K unique colours in the range (lower_bound, upper_bound). We can use an array to keep track of the amount of each colour in the current range. The time complexity of this solution is $O(N) \times C$ where C is the maximum colour value.

Python Solution

```
N, M, K = map(int, input().split())
colours = list(map(int, input().split()))

best_length = 0
best_count = 0

colour_counts = [0] * 30
lower_bound = 0
for upper_bound in range(N):
    colour_counts[colours[upper_bound]] += 1

    # Adjust lower_bound until (lower_bound, upper_bound) is a valid section
    while sum(map(bool, colour_counts)) > K or lower_bound == upper_bound - M:
        colour_counts[colours[lower_bound]] -= 1
        lower_bound += 1

    length = upper_bound - lower_bound + 1
    if best_length < length:
        best_length = length
        best_count = 1
    elif best_length == length:
        best_count += 1

print(best_length, best_count)
```

Full Solution

We will improve upon the solution to Subtask 4 for the full solution. Using an array to store the colour counts will not work for the full solution, as the colours can be up to 10^9, so we can instead use a hash map (dictionary in Python). We also need a more efficient way to keep track of how many unique colours there are. We can keep a count of the unique colours, which we increment when a colour count goes from 0 to 1, and similarly decrement when it goes from 1 to 0. This allows us to maintain the number of unique colours with no additional overhead. The overall time complexity of this solution is $O(N)$.

Python Solution

The code for this solution can be found at the end of this document.

WordFlip

Problem authored by Zalan V
https://train.nzoi.org.nz/problems/1533

Subtask 1

In this subtask we are given that $N \leq 1,000$. We can simply find all flips of the word and sort them to find the lexicographically minimal flip. One trick that we can use to simplify our code is to repeat the input word twice as this allows us to find any rotation of the string without any modular arithmetic. The time complexity of this solution is $O(N^2 \log(N))$.

Python Solution

```
N = int(input())
S = input() * 2
flips = [S[i:i+N] for i in range(N)]
flips.sort()
print(flips[0])
```

Subtask 2

In this subtask we are given that each character appears at most $1,000$ times. This also means that $N \leq 26,000$. We can observe that the strongest flip of any string must start with the minimal (first in the alphabet) character in the string. Thus, we can find the minimal character in the string, and then apply the Subtask 1 solution but only consider flips which start with the minimal character. The time complexity of this is $O(N \times C)$ where C is the number of times the minimal character occurs.

An alternative solution to this subtask is to consider all flips one at a time to find the minimum. That is, we keep track of the minimal flip that we have found so far and replace it if we find a better one. This solution has a time complexity of $O(N^2)$ but will still pass this subtask if implemented well.

Python Solution

```
N = int(input())
S = input() * 2
low = min(S)
flips = [S[i:i+N] for i in range(N) if S[i] == low]
flips.sort()
print(flips[0])
```

Python Solution (Alternative)

```
N = int(input())
S = input() * 2
min(S[i:i+N] for i in range(N))
```

Full Solution

The finding the lexicographically minimal rotation of a string is a well studied problem which can be efficiently solved with many different algorithms. There are many complicated algorithms which can be used to solve this problem in $O(N)$ such as Duval's Lyndon Factorization Algorithm or Booth's Algorithm (a modification of the Knuth–Morris–Pratt algorithm). We will describe a less optimal but very simple algorithm here.

We will consider extending the Subtask 2 solution. We start off by constructing an array $x_0, x_1, \ldots$ of the indices at which positions S contains the minimal character in S. We can also think of this array as containing the starting indices of the minimal substrings of S of length 1. Then, in the i-th step

we will find the minimal character in the i-th position of each of the minimal substrings of length $i-1$, and only keep the indices which still describe a minimal substring of length i. We can apply an optimisation to this algorithm by removing overlapping indices, since if two of these indices overlap, that is, $x_j + i = x_{j+1}$ during step i then we can eliminate x_{j+1} since x_j must be optimal since it has the minimum possible suffix.

The overall time complexity of this algorithm is $O(N \log(N))$, but proving this is not exactly simple. At the i-th step of this algorithm we will have $\frac{N}{i}$ indices left, so the runtime of this algorithm will be roughly:

$$\frac{N}{1} + \frac{N}{2} + ... + \frac{N}{N} = N \times \left(\frac{1}{1} + \frac{1}{2} + ... + \frac{1}{N}\right)$$

The sum $\frac{1}{1} + \frac{1}{2} + ...$ is known as the Harmonic Series and the sum of the first N terms is roughly equal to $\log(N)$ so this sum will be roughly equal to $N \log(N)$.

Python Solution

```python
N = int(input())
S = input() * 2

low = min(S)
indices = [i for i in range(N) if S[i] == low]

for offset in range(1, N):
    # We stop when we either have only one remaining index, or the word
    # consists of a repeated string of size offset
    if len(indices) == 1 or len(indices) * offset == N: break

    new_indices = []
    for i in range(len(indices)):
        # We can index negative indices in Python to access the list from the end
        if (indices[i - 1] + offset) % N != indices[i]:
            new_indices.append(indices[i])
    indices = new_indices

    low = min(S[x + offset] for x in indices)
    indices = [x for x in indices if S[x + offset] == low]

print(S[indices[0]:indices[0]+N])
```

Pablo's Homework

Problem authored by Anatol C
https://train.nzoi.org.nz/problems/1475

Subtask 1

In this subtask we are given that each a_i will consist of at most 15 digits. To solve this subtask we can find every possible subsequence of a_i and check if it is divisible by x_i and has no leading zeros (unless it is a single zero). Then, we simply find the longest such subsequence.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 2

In this subtask we are given that $x_i = 2$ or $x_i = 5$. A number is divisible by 2 iff (if and only if) it ends in one of $0, 2, 4, 6, 8$ and similarly a number is divisible by 5 iff it ends in 0 or 5. Thus, to solve this subtask we just need to remove digits from the end of the number until it ends in one of the valid endings.

Python Solution

The code for this solution can be found at the end of this document.

C++ Solution

The code for this solution can be found at the end of this document.

Subtask 3

In this subtask we are given that x_i is a factor of 100. Extending upon the solution to Subtask 2, in the case that $x_i = 10$ the answer simply needs to end with a zero. For the other cases $(4, 20, 25, 50, 100)$ we can first check if an answer with a single zero is possible. In the case that $x_i = 4$ a single four or eight is also a possible answer.

Then, for each multiple of x_i between 0 and 99 we can check if we can make the last two digits of a_i equal to the multiple similarly to Subtask 1. To find the answer we keep track of the longest answer we have seen so far, taking care to exclude the answer "00".

As an example, if $x_i = 25$, any number divisible by 25 must end in either "00", "25", "50", or "75". We can greedily remove digits from the back to form any such suffixes.

C++ Solution

The code for this solution can be found at the end of this document.

Subtask 4

In this subtask we are given that $x_i = 3$. To solve this subtask we can make use of the fact that a number is divisible by 3 iff the sum of its digits is divisible by 3. Let S_i be the sum of the digits in a_i. Then, if S_i is equal to 0 mod 3 then a_i is already divisible by 3. If S_i is equal to 1 mod 3 then we can remove either a single one of $1, 4, 7$ or two of $2, 5, 8$. Similarly, if S_i equals 2 mod 3 then we can remove either a single one of $2, 5, 8$ or two of $1, 4, 7$. We should also take care to prioritise removing matching digits from the end to avoid leading zeros, and we take whatever option minimises the number of digits removed.

C++ Solution

The code for this solution can be found at the end of this document.

Subtask 5

In this subtask we are given that $x_i = 9$. A number is divisible by 9 iff the sum of its digits is a multiple of 9. We can use a DP (Dynamic Programming) approach to solve this subtask. Our DP state will consist of (length of prefix, sum of digits modulo x_i) and will store the maximum possible length of a subsequence of the prefix with the corresponding sum of digits, or whether a given sum is impossible. Let v_j denote the j-th digit in a_i. Then, we can use the following DP recurrence to solve this subtask:

$$\mathrm{DP}(0,k) = \begin{cases} 1 \text{ if } k \equiv v_0 \mod x_i \\ -1 \text{ otherwise} \end{cases}$$

For $j \geq 1$:

$$\mathrm{DP}(j,k) = \max \begin{cases} \mathrm{DP}(j-1,k) \\ \mathrm{DP}\big(j-1,(k-v_j) \bmod x_i\big) + 1 \text{ if the dp state is not equal to -1} \\ 1 \text{ if } v_j = k \text{ and } v_j \neq 0 \quad \text{to handle leading zeroes} \end{cases}$$

By storing the DP state that we came from for each state, we can use backtracking to find the actual digits of the answer.

C++ Solution

The code for this solution can be found at the end of this document .

Subtask 6

In this subtask we are given that $x_i = 11$. A number is divisible by 11 iff the alternating sum of its digits is divisible by 11, e.g. 7293 is divisible by 11 since $7 - 2 + 9 - 3 = 11$. We can use a similar DP approach to the previous subtask, but we need to store a DP state of (no of digits seen, alternating sum of digits modulo x_i, length is even).

C++ Solution

The code for this solution can be found at the end of this document .

Subtask 7

In this subtask we are given that x_i is prime. Thus, excluding when $x_i = 2$ and $x_i = 5$ we can find the modular multiplicative inverse of 10 under modulo x_i which is given by 10^{x_i-2}. Then, we can use a DP state of (number of digits seen so far, sum modulo x_i) to maximise the number of included digits which form a number with the given sum. This subtask differs from the full solution in that we can simply use both a bottom up and top down approach as we can easily calculate i such that $i \times 10 + j = k$ to find the DP state we need to transition from.

C++ Solution

The code for this solution can be found at the end of this document .

Full Solution

For the full solution we can use a DP state of (no of digits seen so far, sum modulo x_i) to maximise the number of included digits. We need to use a bottom up approach to solve this DP as there can be multiple i such that $i \times 10 + j = k$ so a given sum k will have a variable number of DP state transitions.

Alternatively, it is still possible to do the DP top-down if one pre-calculates the pre-images of all numbers under multiplication by 10 modulo x_i but we will not delve into this solution.

C++ Solution

The code for this solution can be found at the end of this document .

Big O Complexity

Computer scientists like to compare programs using something called Big O notation. This works by choosing a parameter, usually one of the inputs, and seeing what happens as this parameter increases in value. For example, let's say we have a list N items long. We often call the measured parameter N. For example, a list of length N.

In contests, problems are often designed with time or memory constraints to make you think of a more efficient algorithm. You can estimate this based on the problem's constraints. It's often reasonable to assume a computer can perform around 100 million (100,000,000) operations per second. For example, if the problem specifies a time limit of 1 second and an input of N as large as 100,000, then you know that an $O(N^2)$ algorithm might be too slow for large N since $100{,}000^2 = 10{,}000{,}000{,}000$, or 10 billion operations.

Time Complexity

The time taken by a program can be estimated by the number of processor operations. For example, an addition $a + b$ or a comparison $a < b$ is one operation.

$O(1)$ time means that the number of operations a computer performs does not increase as N increases (i.e. does not depend on N). For example, say you have a program containing a list of N items and want to access the item at the i-th index. Usually, the computer will simply access the corresponding location in memory. There might be a few calculations to work out which location in memory the entry i corresponds to, but these will take the same amount of computation regardless of N. Note that time complexity does not account for constant factors. For example, if we doubled the number of calculations used to get each item in the list, the time complexity is still $O(1)$ because it is the same for all list lengths. You can't get a better algorithmic complexity than constant time.

$O(\log N)$ time suggests the program takes a constant number of extra operations every time N doubles in size. For example, finding a number in a sorted list using binary search might take 3 operations when $N = 8$, but it will only take one extra operation if we double N to 16. As far as efficiency goes, this is pretty good, since N generally has to get very, very large before a computer starts to struggle.

$O(N)$ time means you have an algorithm where the number of operations is directly proportional to N. For example, a maximum finding algorithm `max()` will need to compare against every item in a list of length N to confirm you have indeed found the maximum. Usually, if you have one loop that iterates N times your algorithm is $O(N)$.

$O(N^2)$ time means the number of operations is proportional to N^2 . For example, suppose you had an algorithm which compared every item in a list against every other item to find similar items. For a list of N items, each item has to check against the remaining $N - 1$ items. In total, $N(N - 1)$ checks are done. This expands to $N^2 - N$. For Big O, we always take the most significant term as the dominating factor, which gives $O(N^2)$. This is generally not great for large values of N, which can take a very long time to compute. As a general rule of thumb in contests, $O(N^2)$ algorithms are only useful for input sizes of $N \lesssim 10{,}000$. Usually, if you have a nested loop in your program (loop inside a loop) then your solution is $O(N^2)$ if both these loops run about N times.

Additional Example Solutions

Teacups Full Solution (Python)

```python
N = int(input())
teacups = [int(input()) for _ in range(N)]
teacups.sort()

idx = 0
shelf1 = []
shelf2 = []
while idx < N:
    if idx < N - 1 and teacups[idx] == teacups[idx + 1]:
        shelf1.append(teacups[idx])
        shelf1.append(teacups[idx])
        idx += 2
    else:
        shelf2.append(teacups[idx])
        idx += 1

print("Shelf 1:", *shelf1)
print("Shelf 2:", *shelf2)
```

Ribbon's Subtask 2 Solution (Python)

```python
N, M, K = map(int, input().split())
colours = list(map(int, input().split()))

possible_lengths = []
for start in range(N):
    for end in range(start, min(start + M, N)):
        unique_colours = set(colours[start:end+1])
        if len(unique_colours) <= K:
            possible_lengths.append(end - start + 1)
possible_lengths.sort()

print(possible_lengths[-1], possible_lengths.count(possible_lengths[-1]))
```

Ribbons Subtask 3 Solution (Python)

```python
N, M, K = map(int, input().split())
colours = list(map(int, input().split()))

possible_lengths = []
for start in range(N):
    unique_colours = set()
    for end in range(start, min(start + M, N)):
        unique_colours.add(colours[end])
        if len(unique_colours) > K:
            possible_lengths.append(end - start)
            break
    else:
        possible_lengths.append(end - start + 1)
possible_lengths.sort()

print(possible_lengths[-1], possible_lengths.count(possible_lengths[-1]))
```

Ribbons Full Solution (Python)

```python
from collections import defaultdict

N, M, K = map(int, input().split())
colours = list(map(int, input().split()))

best_length = 0
best_count = 0

colour_counts = defaultdict(int)
unique_colours = 0
lower_bound = 0
for upper_bound in range(N):
    if colour_counts[colours[upper_bound]] == 0: unique_colours += 1
    colour_counts[colours[upper_bound]] += 1

    # Adjust lower_bound until (lower_bound, upper_bound) is a valid section
    while unique_colours > K or lower_bound == upper_bound - M:
        colour_counts[colours[lower_bound]] -= 1
        if colour_counts[colours[lower_bound]] == 0: unique_colours -= 1
        lower_bound += 1

    length = upper_bound - lower_bound + 1
    if best_length < length:
        best_length = length
        best_count = 1
    elif best_length == length:
        best_count += 1

print(best_length, best_count)
```

Pablo's Homework Subtask 1 Solution (Python)

```python
N = int(input())
for _ in range(N):
    a = input()
    x = int(input())
    substrings = [""]
    for c in a:
        for s in substrings.copy():
            substrings.append(s + c)

    ans = ""
    for s in substrings:
        # Empty
        if len(s) == 0: continue
        # Leading zero
        if s[0] == '0' and len(s) > 1: continue
        # Suboptimal
        if len(s) <= len(ans): continue
        if int(s) % x == 0:
            ans = s
    if ans == "": print(-1)
    else: print(ans)
```

Pablo's Homework Subtask 2 Solution (Python)

```python
N = int(input())
for _ in range(N):
    a = input()
    x = int(input())
    last_valid = -1
    if x == 2:
        for i in range(len(a)):
            if a[i] in "02468": last_valid = i
    else:
        for i in range(len(a)):
            if a[i] in "05": last_valid = i

    if last_valid == -1: print(-1)
    else: print(a[:last_valid+1])
```

Pablo's Homework Subtask 2 Solution (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;

string makeEndWith(string& s, string end) {
    auto pos = s.find_last_of(end);
    if (pos == s.npos) return "";
    return s.substr(0, pos + 1);
}

int main() {
    int N;
    cin >> N;
    while (N--) {
        string A;
        int X;
        cin >> A >> X;

        string ans;
        if (X == 2) ans = makeEndWith(A, "02468");
        else ans = makeEndWith(A, "05");

        if (ans.empty()) cout << "-1\n";
        else cout << ans << "\n";
    }
    return 0;
}
```

Pablo's Homework Subtask 3 Solution (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;

string makeEndWith(string& s, string end) {
    auto pos = s.find_last_of(end);
    if (pos == s.npos) return "";
    return s.substr(0, pos + 1);
}

int main() {
    int N;
```

```
    cin >> N;
    while (N--) {
        string A;
        int X;
        cin >> A >> X;

        string ans;
        if (X == 2) ans = makeEndWith(A, "02468");
        else if (X == 5) ans = makeEndWith(A, "05");
        else if (X == 10) ans = makeEndWith(A, "0");
        else {
            if (X == 4) {
                auto pos = A.find_first_of("048");
                if (pos != A.npos) ans = A[pos];
            }
            else if (A.find_first_of('0') != A.npos) ans = "0";

            for (int i = 0; i < 10; ++i) {
                for (int j = 0; j < 10; ++j) {
                    if ((i * 10 + j) % X != 0) continue;

                    auto nans = makeEndWith(A, to_string(j));
                    // Answer cannot have a leading zero
                    if (nans.empty() || nans[0] == '0') continue;
                    // Handle case where i == j
                    nans.pop_back();
                    nans = makeEndWith(nans, to_string(i));
                    if (nans.empty()) continue;
                    nans.append(to_string(j));

                    if (nans.size() > ans.size()) swap(ans, nans);
                }
            }
        }

        if (ans.empty()) cout << "-1\n";
        else cout << ans << "\n";
    }
    return 0;
}
```

Pablo's Homework Subtask 4 Solution (C++)

```
#include <bits/stdc++.h>
using namespace std;

string removeChar(string a, string remove) {
    auto pos = a.find_last_of(remove);
    if (pos == a.npos) return "";
    a.erase(pos, 1);
    if (a[0] == '0') {
        pos = a.find_first_not_of('0');
        if (pos == a.npos) return "";
        a = a.substr(pos);
    }
    return a;
}
```

```cpp
int main() {
    int N;
    cin >> N;
    while (N--) {
        string A;
        int X;
        cin >> A >> X;

        string ans;
        if (A.find_first_of('0') != A.npos) ans = "0";

        int mod = 0;
        for (char c : A) mod = (mod + (c - '0')) % 3;

        if (mod == 0) ans = A;
        else if (mod == 1) {
            auto nans = removeChar(A, "147");
            if (ans.size() < nans.size()) ans = nans;
            nans = removeChar(removeChar(A, "258"), "258");
            if (ans.size() < nans.size()) ans = nans;
        }
        else {
            auto nans = removeChar(A, "258");
            if (ans.size() < nans.size()) ans = nans;
            nans = removeChar(removeChar(A, "147"), "147");
            if (ans.size() < nans.size()) ans = nans;
        }

        if (ans.empty()) cout << "-1\n";
        else cout << ans << "\n";
    }
    return 0;
}
```

Pablo's Homework Subtask 5 Solution (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> ii;

int main() {
    int N;
    cin >> N;
    while (N--) {
        string A;
        int X;
        cin >> A >> X;

        vector<vector<ii>> dp(A.size(), vector<ii>(X, {-1, -1}));
        dp[0][(A[0] - '0') % X] = {1, 0};
        for (int i = 1; i < A.size(); ++i) {
            int charVal = (A[i] - '0') % X;
            for (int j = 0; j < X; ++j) dp[i][j] = {dp[i - 1][j].first, -1};

            if(A[i] != '0') dp[i][charVal] = max(dp[i][charVal], {1, 0});
```

```
            for (int j = 0; j < X; ++j) {
                if (dp[i - 1][j].first == -1) continue;
                int pos = (j + charVal) % X;
                dp[i][pos] = max(dp[i][pos], {dp[i - 1][j].first + 1, j});
            }
        }

        if (dp.back()[0].first == -1) {
            if (A.find_first_of('0') != A.npos) cout << "0\n";
            else cout << "-1\n";
        }
        else {
            string ans;
            int pos = 0;
            for (int i = A.size() - 1; i >= 0; --i) {
                if (dp[i][pos].second >= 0) {
                    ans.push_back(A[i]);
                    pos = dp[i][pos].second;
                }
            }
            reverse(ans.begin(), ans.end());
            cout << ans << "\n";
        }
    }
    return 0;
}
```

Pablo's Homework Subtask 6 Solution (C++)

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> ii;

int main() {
    int N;
    cin >> N;
    while (N--) {
        string A;
        int X;
        cin >> A >> X;

        vector<vector<ii>> dpOdd(A.size(), vector<ii>(X, {-1, -1}));
        vector<vector<ii>> dpEven(A.size(), vector<ii>(X, {-1, -1}));
        dpOdd[0][A[0] - '0'] = {1, 0};
        for (int i = 1; i < A.size(); ++i) {
            int charValOdd = A[i] - '0';
            int charValEven = (X - (A[i] - '0')) % X;
            for (int j = 0; j < X; ++j) {
                dpOdd[i][j] = {dpOdd[i - 1][j].first, -1};
                dpEven[i][j] = {dpEven[i - 1][j].first, -1};
            }

            if(A[i] != '0') dpOdd[i][charValOdd] = max(dpOdd[i][charValOdd], {1, 0});

            for (int j = 0; j < X; ++j) {
                if (dpEven[i - 1][j].first == -1) continue;
                int pos = (j + charValOdd) % X;
```

```
                dpOdd[i][pos] = max(dpOdd[i][pos], {dpEven[i - 1][j].first + 1, j});
            }
            for (int j = 0; j < X; ++j) {
                if (dpOdd[i - 1][j].first == -1) continue;
                int pos = (j + charValEven) % X;
                dpEven[i][pos] = max(dpEven[i][pos], {dpOdd[i - 1][j].first + 1, j});
            }
        }

        bool oddSol = (dpOdd.back()[0].first != -1);
        bool evenSol = (dpEven.back()[0].first != -1);
        if (oddSol || evenSol) {
            string ans;
            bool odd = false;
            if (!evenSol || (oddSol && dpEven.back()[0].first < dpOdd.back()
[0].first)) odd = true;

            int pos = 0;
            for (int i = A.size() - 1; i >= 0; --i) {
                if (odd) {
                    if (dpOdd[i][pos].second >= 0) {
                        ans.push_back(A[i]);
                        pos = dpOdd[i][pos].second;
                        odd = false;
                    }
                }
                else {
                    if (dpEven[i][pos].second >= 0) {
                        ans.push_back(A[i]);
                        pos = dpEven[i][pos].second;
                        odd = true;
                    }
                }
            }
            reverse(ans.begin(), ans.end());
            cout << ans << '\n';
        }
        else if (A.find_first_of('0') != A.npos) cout << "0\n";
        else cout << "-1\n";
    }
    return 0;
}
```

Pablo's Homework Subtask 7 Solution (C++)

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> ii;

string makeEndWith(string& s, string end) {
    auto pos = s.find_last_of(end);
    if (pos == s.npos) return "";
    return s.substr(0, pos + 1);
}

int main() {
    int N;
```

```
    cin >> N;
    while (N--) {
        string A;
        int X;
        cin >> A >> X;

        // Subtask 2 solution
        if (X == 2 || X == 5) {
            string ans;
            if (X == 2) ans = makeEndWith(A, "02468");
            else ans = makeEndWith(A, "05");
            if (ans.empty()) cout << "-1\n";
            else cout << ans << "\n";
            continue;
        }

        // Calculate the modular inverse of 10 under mod X (only works if 10 is
coprime to X)
        int inv = 1;
        for (int i = 0; i < X - 2; ++i) inv = (inv * 10) % X;

        vector<vector<ii>> dp(A.size(), vector<ii>(X, {-1, -1}));
        dp[0][(A[0] - '0') % X] = {1, 0};
        for (int i = 1; i < A.size(); ++i) {
            int charVal = (A[i] - '0') % X;
            for (int j = 0; j < X; ++j) {
                dp[i][j] = {dp[i - 1][j].first, -1};
                // Find value such that prevPos * 10 + charVal == j under modulo X
                int prevPos = ((j - charVal + X) * inv) % X;
                if (dp[i - 1][prevPos].first != -1) {
                    dp[i][j] = max(dp[i][j], {dp[i - 1][prevPos].first + 1,
prevPos});
                }
            }
            if (A[i] != '0' && dp[i][charVal].first == -1) dp[i][charVal] = {1, 0};
        }

        if (dp.back()[0].first != -1) {
            string ans;
            int pos = 0;
            for (int i = A.size() - 1; i >= 0; --i) {
                if (dp[i][pos].second >= 0) {
                    ans.push_back(A[i]);
                    pos = dp[i][pos].second;
                }
            }
            reverse(ans.begin(), ans.end());
            cout << ans << "\n";
        }
        else if (A.find_first_of('0') != A.npos) cout << "0\n";
        else cout << "-1\n";
    }
    return 0;
}
```

Pablo's Homework Full Solution (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> ii;

int main() {
    int N;
    cin >> N;
    while (N--) {
        string A;
        int X;
        cin >> A >> X;

        vector<vector<ii>> dp(A.size(), vector<ii>(X, {-1, -1}));
        dp[0][(A[0] - '0') % X] = {1, 0};
        for (int i = 1; i < A.size(); ++i) {
            for (int j = 0; j < X; ++j) dp[i][j] = {dp[i - 1][j].first, -1};
            int charVal = (A[i] - '0') % X;
            for (int j = 0; j < X; ++j) {
                if (dp[i - 1][j].first == -1) continue;
                int nextPos = (j * 10 + charVal) % X;
                dp[i][nextPos] = max(dp[i][nextPos], {dp[i - 1][j].first + 1, j});
            }
            if (A[i] != '0' && dp[i][charVal].first == -1) dp[i][charVal] = {1, 0};
        }

        if (dp.back()[0].first != -1) {
            string ans;
            int pos = 0;
            for (int i = A.size() - 1; i >= 0; --i) {
                if (dp[i][pos].second >= 0) {
                    ans.push_back(A[i]);
                    pos = dp[i][pos].second;
                }
            }
            reverse(ans.begin(), ans.end());
            cout << ans << "\n";
        }
        else if (A.find_first_of('0') != A.npos) cout << "0\n";
        else cout << "-1\n";
    }
    return 0;
}
```