Лабораторная работа № 5 по курсу дискретного анализа: Динамическое программирование

Выполнил студент группы М80-308Б-22 МАИ *Кочкожаров Иван.*

Условие

Краткое описание задачи:

1. При помощи метода динамического программирования разработать алгоритм решения задачи, определяемой своим вариантом
2. *Вариант:* Количество чисел
3. *Задача:* Задано целое число n. Необходимо найти количество натуральных (без нуля) чисел, которые меньше n по значению **и** меньше n лексикографически (если сравнивать два числа как строки), а также делятся на m без остатка.

Метод решения

Будем на каждой итерации цикла применять функцию CountMultiplesInRange(long long l, long long r, int m) для подсчета чисел, делящихся на m без остатка, лежащих в диапазоне l и r . Динамическое программирование основано на переходе из состояния $dp[n][m]$, в состояние $dp[n/10][m]$ и так далее, пока n не станет меньше 10. Это решение можно представить рекурсивной функцией: dp[n][m] = CountMultiplesInRange(pow(10, len(n)-1), n, m) + dp[n/10][m]. Таким образом мы подсчитываем все CountMultiplesInRange(pow(10, len(n)-1), n, m) для всех префиксов числа n, очевидно что эти префиксы меньше и лексикографически меньше n, и суммируем все эти результаты. Это и будет ответом. Сложность алгоритма - O(len(n))

Описание программы

Разделение по файлам, описание основных типов данных и функций.

- Реализация алгоритма

```
#include <iostream>
#include <cmath>

int LengthOfNum(long long x){
    int len=1;
    while(x>9){ len++; x/=10; }
    return len;
}
```

```cpp
long long CountMultiplesInRange(long long l, long long r, int m) {
    long long firstMultiple = (l % m == 0) ? l : l + (m - l % m);
    long long lastMultiple = (r % m == 0) ? r : r - (r % m);
    if (firstMultiple > lastMultiple) {
        return 0;
    }
    return (lastMultiple - firstMultiple) / m + 1;
}

long long Solve(long long n, int m) {
    int length = LengthOfNum(n);
    long long result = 0;
    long long a = pow(10, length-1);
    long long b = n;
    for (int i = 0; i < length; ++i) {
        result += CountMultiplesInRange(a, b, m);
        a /= 10;
        b /= 10;
    }
    if (n % m == 0) {
        return result-1;
    }
    return result;
}

int main() {
    long long n;
    int m;
    std::cin >> n >> m;
    std::cout << Solve(n, m) << std::endl;
    return 0;
}
```

Дневник отладки

Во время реализации я столкнулся с небольшими проблемами:

1. Проблема с алгоритмом. Изначально для подзадач я брал следующий срез числа: $n[len(n) - k; len(n)]$ или же просто брал правые k чисел. Очевидно, что это неправильно

2. Проблема со временем. Также перейдя к нисходящей концепции я решил задачу

рекурсивно. Хотя задача и успешно была проверена на чекере, рекурсия тратит значительно больше времени, нежели итерация.

Тест производительности

Как видно из результатов теста, алогитм с использованием ДП гораздо быстрее, засчет асимптотики O(len(n)) вместо наивного лексикографического сравнения всех чисел за O(n*len(n))

```
ivan@asus-vivobook ~/c/d/b/lab7 (master)> ./lab7_benchmark < test1 \
&& ./lab7_naive_benchmark < test1 && cat test1
Bench
Time: 0.000095 sec
Naive Bench
Time: 2.219022 sec
1248129480 342
1248129480 345
1248129480 341
```

Ниже приведена программы, использовавштеся для засечения времени работы функций:

```
#include <iostream>
#include <string>


int main() {
    long long n;
    int m;
    double start_time =  clock();
    while (std::cin >> n >> m){
        long long count = 0;
        for (long long multiple = m; multiple < n; multiple += m) {
            if (std::to_string(multiple) < std::to_string(n)) {
                count++;
            }
        }
    }
    double end_time = clock();
    double search_time = end_time - start_time;
    printf("Naive Bench\nTime: %f sec\n", (double)search_time/CLOCKS_PER_SE
    return 0;
}
```

```
#include <iostream>
```

```
#include <string>
#include <math.h>
#include <time.h>


int LengthOfNum(long long x){
    int len=1;
    while(x>9){ len++; x/=10; }
    return len;
}

long long CountMultiplesInRange(long long l, long long r, int m) {
    long long firstMultiple = (l % m == 0) ? l : l + (m - l % m);
    long long lastMultiple = (r % m == 0) ? r : r - (r % m);
    if (firstMultiple > lastMultiple) {
        return 0;
    }
    return (lastMultiple - firstMultiple) / m + 1;
}

long long Solve(long long n, int m) {
    int length = LengthOfNum(n);
    long long result = 0;
    long long a = pow(10, length-1);
    long long b = n;
    for (int i = 0; i < length; ++i) {
        result += CountMultiplesInRange(a, b, m);
        a /= 10;
        b /= 10;
    }
    if (n % m == 0) {
        return result-1;
    }
    return result;
}


int main() {
    long long n;
    int m;
    double start_time =  clock();
    while (std::cin >> n >> m){
```

```
        long long x = Solve(n, m);
    }
    double end_time = clock();
    double search_time = end_time - start_time;
    printf("Bench\nTime: %f sec\n", (double)search_time/CLOCKS_PER_SEC);
    return 0;
}
```

Выводы

В ходе выполнения лабораторной работы я изучил классические задачи динамического программирования и их методы решения, реализовал алгоритм для своего варианта задания.

Также в очередной раз убедился в том, что рекурсия - хорошее средство для ленивого программиста, но занимает достаточно много времени. Поэтому если есть возможность пользоваться итерацией, не следует ей пренебрегать.

Динамическое программирование позволяет разработать точные и относительно быстрые алгоритмы для решения сложных задач, в то время, как решение перебором слишком медленное, а жадный алгоритм не всегда даёт правильный результат.