Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №7 по курсу «Дискретный анализ»

Студент: И. П. Моисеенков
Преподаватель: Н. С. Капралов
Группа: М8О-208Б-19

Дата: 18.04.2021
Оценка: хорошо
Подпись:

Москва, 2021

Лабораторная работа №7

Задача: При помощи метода динамического программирования разработать алгоритм решения задачи, определяемой своим вариантом; оценить время выполнения алгоритма и объем затрачиваемой оперативной памяти. Перед выполнением задания необходимо обосновать применимость метода динамического программирования.

Разработать программу на языке C или C++, реализующую построенный алгоритм. Формат входных и выходных данных описан в варианте задания.

Вариант 3. 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 так же делятся на m без остатка.

Формат входных данных В первой строке задано $1 \leq n \leq 10^{18}$ и $1 \leq m \leq 10^5$.

Формат результата Необходимо вывести количество искомых чисел.

1 Описание

Согласно [1], динамическое программирование — это метод решения задачи путём её разбиения на несколько одинаковых подзадач, рекуррентно связанных между собой.

Наиболее быстрый и оптимальный способ решения предложенной задачи как раз базируется на методе динамического программирования. Продемонстрирую это сначала на примере. Пусть нам необходимо найти количество натуральных чисел, делящихся на какое-то число m, которые меньше лексикографически и меньше по значению числа, скажем, 567. Для нахождения ответа мы можем найти все такие числа, лежащие на интервале $(0; 5)$, на интервале $(10; 56)$ и на интервале $(100; 567)$ и сложить все полученные значения. При рассмотрении каждого из таких интервалов мы гарантируем, что числа, лежащие в нем, меньше нашего исходного числа и по значению, и лексикографически. Найти количество чисел, кратных m, на каждом интервале - тривиальная задача.

Введем следующее обозначение: $ans(0, x)$ - количество чисел на промежутке от 0 до x, удовлетворяющих условию задачи. Будем рассматривать заданное число n в виде $\overline{n_0 n_1 n_2 \dots n_k}$,Тогда решение задачи можно формализовать следующей формулой:

$$ans(0, \overline{n_0 n_1 \dots n_k}) = \sum_{i=0}^{k} ans(10^i, \overline{n_0 \dots n_i})$$

Сложность такого алгоритма по времени и по памяти будет составлять $O(k)$, где $k = |n|$ - количество разрядов в заданном числе.

Если мы бы решали эту задачу наивным переборным алгоритмом, то получили бы временную сложность $O(n)$, что гораздо хуже сложности алгоритма динамического программирования.

2 Исходный код

Реализация динамического алгоритма чрезмерно проста. Мне потребовалось написать всего лишь один цикл и функцию $Count$, которая непосредственно вычисляет количество чисел, делящихся на m на интервале $(leftBound; rightBound)$. Функция вычисляет значения первого числа $firstNum$ и последнего числа $lastNum$ из промежутка, которые делятся на m без остатка и считает количество кратных чисел между ними по формуле $(lastNum - firstNum)/m + 1$.

Учитываем, что в задаче ищутся все числа, которые строго меньше заданного. Поэтому необходимо убедиться, что заданное число не учитывается в ответе.

```
#include <iostream>

long long Count(long long leftBound, long long rightBound, int m) {
    long long firstNum, lastNum; // first and last suitable numbers
    if (leftBound % m != 0) {
        firstNum = leftBound + m - leftBound % m;
    }
    else {
        firstNum = leftBound;
    }
    lastNum = rightBound - rightBound % m;

    if (firstNum <= lastNum) {
        return (lastNum - firstNum) / m + 1;
    }
    return 0;
}

int main() {
    std::string n;
    int m;
    std::cin >> n >> m;
    long long answer = 0;
    long long leftBound = 1, rightBound = 0;
    for (int i = 0; i < n.length(); ++i) {
        if (i != 0) {
            leftBound *= 10;
            rightBound *= 10;
        }
        rightBound += n[i] - '0';
        answer += Count(leftBound, rightBound, m);
    }
    if (rightBound % m == 0) {
        --answer;
    }
    std::cout << answer << "\n";
}
```

3 Консоль

```
mosik@LAPTOP-69S778GL:~/da_lab7$ g++ main.cpp -o dynamic
mosik@LAPTOP-69S778GL:~/da_lab7$ ./dynamic
42 3
11
mosik@LAPTOP-69S778GL:~/da_lab7$ ./dynamic
125376616786 18
1566457825
mosik@LAPTOP-69S778GL:~/da_lab7$ ./dynamic
20 238497
0
```

4 Тест производительности

Будем сравнивать реализованный динамический алгоритм с наивным перебором. Для сравнения я подготовил три теста. Первый тест - число порядка 10^6, второй - порядка 10^9, третий - 10^{18}. В качестве m во всех тестах используется 3.

```
mosik@LAPTOP-69S778GL:~/da_lab7$ g++ main.cpp -o dynamic
mosik@LAPTOP-69S778GL:~/da_lab7$ g++ naive.cpp -o naive
mosik@LAPTOP-69S778GL:~/da_lab7$ ./dynamic <test1e6
33537
Time: 0.0004017 s
mosik@LAPTOP-69S778GL:~/da_lab7$ ./naive <test1e6
33537
Time: 0.0186652 s
mosik@LAPTOP-69S778GL:~/da_lab7$ ./dynamic <test1e9
274855323
Time: 0.0004111 s
mosik@LAPTOP-69S778GL:~/da_lab7$ ./naive <test1e9
274855323
Time: 84.8031 s
mosik@LAPTOP-69S778GL:~/da_lab7$ ./dynamic <test1e18
199745960722083468
Time: 0.0004081 s
mosik@LAPTOP-69S778GL:~/da_lab7$ ./naive <test1e18
```

Наивный алгоритм решал задачу с числом порядка 10^{18} больше 5 минут. Окончательного результата я дожидаться не стал.

Как видно, наивный алгоритм во много раз проиграл динамическому программированию. Это не удивительно, ведь само число растет гораздо быстрее, чем количество его разрядов.

5 Выводы

Выполнив седьмую лабораторную работу по курсу «Дискретный анализ», я познакомился с методом динамического программирования. Динамическое программирование позволяет разбить задачу, которую непонятно как решать, на несколько задач поменьше, которые тоже непонятно как решать. Но если мы будем знать, как решать самую простую задачу из этого разбиения и сможем придумать способ возврата от мелких задач к большой, то мы с легкостью сможем решить исходную задачу.

С помощью динамического программирования решается большинство задач оптимизации: оптимальное хранение, оптимальное производство, оптимальный порядок действий и многие другие. Однако в реальности задачи могут быть настолько большими, что компьютер будет искать их решение очень долго (например, несколько лет). Поэтому для особо больших задач предпочитают отходить от динамического программирования и использовать жадные алгоритмы.

Список литературы

[1] *Динамическое программирование*
URL: `https://tproger.ru/articles/dynprog-starters/` (дата обращения: 18.04.2021).