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. А. Мариничев
Преподаватель: Н. С. Капралов
Группа: М8О-208Б

Дата: 23.04.21
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №7. Игра с числом

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:

Имеется натуральное число **n**. За один ход с ним можно произвести следующие действия: вычесть единицу, разделить на два, разделить на три. При этом стоимость каждой операции — текущее значение **n**. Стоимость преобразования — суммарная стоимость всех операций в преобразовании. Вам необходимо с помощью последовательностей указанных операций преобразовать число **n** в единицу таким образом, чтобы стоимость преобразования была наименьшей. Делить можно только нацело.

Формат входных данных

В первой строке строке задано $2 \leq n \leq 10^7$.

Формат результата

Выведите на первой строке искомую наименьшую стоимость. Во второй строке должна содержаться последовательность операций. Если было произведено деление на 2 или на 3, выведите /2 (или /3). Если же было вычитание, выведите -1. Все операции выводите разделяя пробелом.

1 Описание

Согласно [1], динамическое программирование позволяет решать задачи, комбинируя решения вспомагательных подзадач (термин "программирование"в данном контексте означает табличный метод, а не составление копьютерного кода). Динамическое програмирование находит применение тогда, когда подзадачи перекрываются, т.е. когда разные подзадачи используют решения одних и тех же подзадач. В алгоритме динамического программирования каждая подзадача решается только один раз, после чего ответ сохраняется в таблице. Это позволяет избежать одних и тех же вычислений каждый раз, когда встречается данная, уже решенная ранее, подзадача.
Динамическое программирование, как правило, применяется к задачам оптимизации. Такая задача может иметь много возможных решений. С каждым вариантом решения можно сопотавить какое-то значение, нам нужно найти среди них решение с оптимальным (минимальным или максимальным) значением.
Процесс разработки алгоритмов динамического программирования можно разделить на четыре перечисленных ниже этапа.

1. Описание структуры оптимального решения.
2. Определение значения, соответствующего оптимальному решению, с использованием рекурсии.
3. Вычисление значения, соответствующего оптимальному решению, обычно с помощью метода восходящего анализа.
4. Составление оптмального решения на основе информации, полученной на предыдущих этапах.

Перейдем непосредственно к нашей задаче. Будем решать ее методом восходящего анализа. Будем идти от 0 до нашего числа **n** и записывать в таблицу для каждого числа минимальный путь до этого числа, выбирая из всех возможных предыдущих чисел с посчитанным минимальным путём. Так мы значительно сократим количество вычислений в сравнении с наивным алгоритмом, т.к. будем проходить только по минимальным путям, при этом используя уже посчитанные данные. В итоге мы получим в последней ячейке нашей таблицы искомую наименьшую стоимость. После этого восстановим наш оптимальный путь из **n** в 1. Будем выбирать для текущего числа то число, которое можно получить вычитанием единицы, делением нацело на два или три, у которого соответствующее значение в таблице минимльно. Так будем двигаться, параллельно выводя выбранные операции (/2, /3, -1), пока не дойдем до единицы.

2 Исходный код

Теперь поговорим о реализации данного метода на языке C++. В качестве таблицы будем использовать динамический массив **memento** размером $n + 1$. Программа состоит из двух циклов. В цикле **for** заполняется массив минимальных путей. В цикле **while** происходит реализация решения при помощи полученных данных и вывод ответа в требуемом формате.

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

using namespace std;

int Min(int a, int b, int c) {
    return (min(a, b) < min(b, c)) ? min(a, b) : min(b,c);
}

int main() {
    int n, tmp;

    cin >> n;

    int * memento = new int [n + 1];

    // go from 0 to n and find shortest path to every number
    memento[0] = 0;
    memento[1] = 0;
    for (int i = 2; i <= n; i++) {
        // if current number can be divided by 2 and 3
        if (i % 2 == 0 && i % 3 == 0) {
            // then we choose best option by value from 3 previous points
            memento[i] = i + Min(memento[i / 2], memento[i / 3], memento[i - 1]);
            continue;
        }
        // if current number can be divided only by 3
        if (i % 3 == 0) {
            // then choose best option from 2 previous points
            if (memento[i / 3] < memento[i - 1]) {
                memento[i] = i + memento[i / 3];
            } else {
                memento[i] = i + memento[i - 1];
            }
            continue;
        }
        // if current number can be divided only by 2
        if (i % 2 == 0) {
            // then choose best option from 2 previous points
            if (memento[i / 2] < memento[i - 1]) {
```

```
42                  memento[i] = i + memento[i / 2];
43              } else {
44                  memento[i] = i + memento[i - 1];
45              }
46              continue;
47          }
48          // we get here if current number can't be divided by 2 or 3, so there is only
                one previous point to choose
49          memento[i] = i + memento[i - 1];
50      }
51
52      // displays value of the shortest path from 1 to n
53      cout << memento[n] << endl;
54
55      // now we go through memento from n to 1 finding lowest values and displaying
            corresponding operation
56      tmp = n;
57      while (tmp > 1) {
58          // if current number can be divided by 2 and 3
59          if ((tmp % 3 == 0) && (tmp % 2 == 0)) {
60              // then we choose best option by value from 3 previous points
61              if(memento[tmp / 3] <= memento[tmp / 2] && memento[tmp / 3] <= memento[tmp
                    - 1]) {
62                  cout << "/3" << ' ';
63                  tmp /= 3;
64              }
65              else if(memento[tmp / 2] <= memento[tmp / 3] && memento[tmp / 2] <= memento
                    [tmp - 1]) {
66                  cout << "/2" << ' ';
67                  tmp /= 2;
68              }
69              else {
70                  cout << "-1";
71                  tmp--;
72              }
73              continue;
74          }
75          // if current number can be divided only by 3
76          if (tmp % 3 == 0) {
77              // then choose best option from 2 previous points
78              if (memento[tmp / 3] <= memento[tmp - 1]) {
79                  cout << "/3" << ' ';
80                  tmp /= 3;
81              } else {
82                  cout << "-1" << ' ';
83                  tmp--;
84              }
85              continue;
86          }
```

```
87          // if current number can be divided only by 2
88          if (tmp % 2 == 0) {
89              // then choose best option from 2 previous points
90              if (memento[tmp / 2] <= memento[tmp - 1]) {
91                  cout << "/2" << ' ';
92                  tmp /= 2;
93              } else {
94                  cout << "-1" << ' ';
95                  tmp--;
96              }
97              continue;
98          }
99          // we get here if current number can't be divided by 2 or 3, so there is only
                one previous point to choose
100         cout << "-1" << ' ';
101         tmp--;
102     }
103     cout << endl;
104
105     delete[] memento;
106
107     return 0;
108 }
```

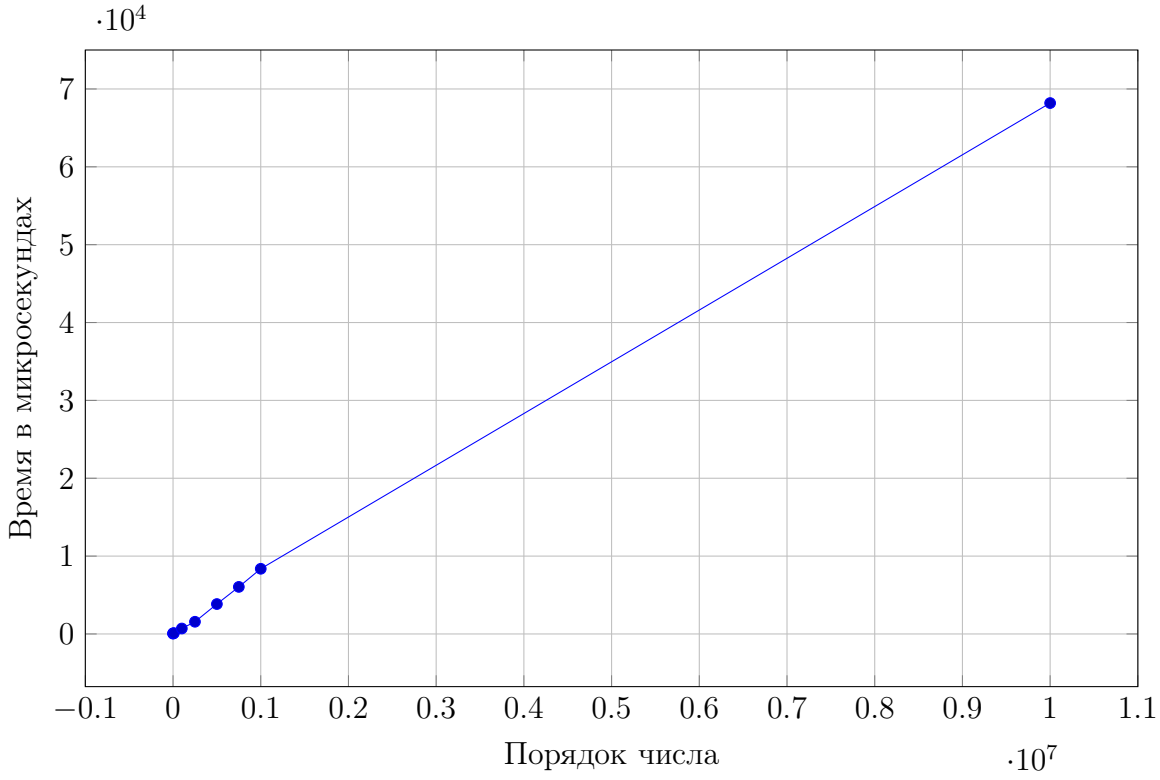
3 Тест производительности

Перейдем к оценке времени выполнения алгоритма и объема затрачиваемой оперативной памяти. Сложность алгоритма линейная. Объём дополнительной затраченной памяти O(n).
Теперь посмотрим на то, как же ведет себя программа на практке. Проверим время работы программы на тестах с числами разного порядка: 100, 1000, 10000, 100000, 250000, 500000, 750000, 1000000, 10000000. Время выводится микросекундах. Для измерения времени использовалась бибилиотека **chrono**.

```
// Тест 1. Проверка зависимости времени решения задачи методом динамического
программирования от размера входных данных.
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ make run
./solution <../tests/test_100.t
221
Time of Dynamic Programming method: 50 ms
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ make run
./solution <../tests/test_1000.t
1748
Time of Dynamic Programming method: 73 ms
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ make run
./solution <../tests/test_10000.t
25693
Time of Dynamic Programming method: 106 ms
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ make run
./solution <../tests/test_100000.t
252298
Time of Dynamic Programming method: 699 ms
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ make run
./solution <../tests/test_250000.t
948359
Time of Dynamic Programming method: 1567 ms
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ make run
./solution <../tests/test_500000.t
1022773
Time of Dynamic Programming method: 3842 ms
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ make run
./solution <../tests/test_750000.t
1758791
Time of Dynamic Programming method: 6045 ms
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ make run
```

```
./solution <../tests/test_1000000.t
2035891
Time of Dynamic Programming method: 8372 ms
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ make run
./solution <../tests/test_10000000.t
38038883
Time of Dynamic Programming method: 68191 ms
```

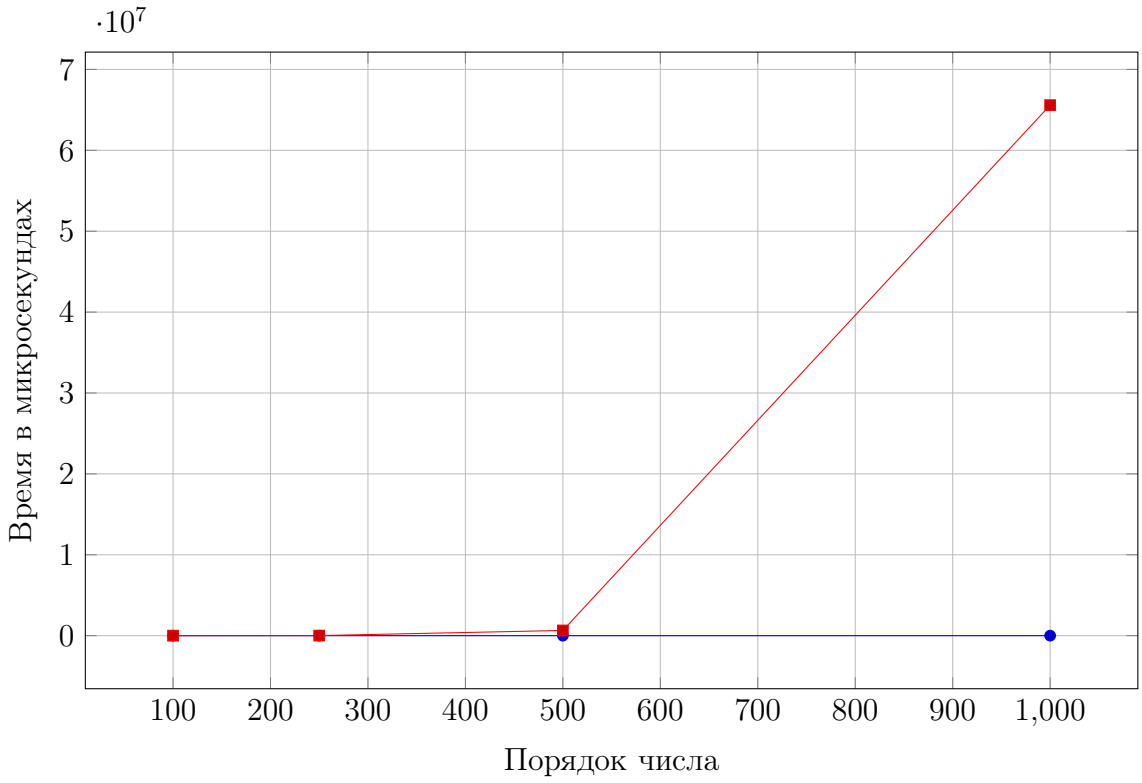
Представим полученные результаты в более наглядной форме, а именно в виде графика.



Из графика видно, что время работы программы возрастает прямо пропорционально объему входных данных, следовательно сложность программы действительно равна O(n).

Теперь проведем второй тест. Сравним время получения решения данной задачи методом динамического программирования и наивный алгоритм. В наивном алгоритме мы будем рекурсивно проходиться по всем путям от **n** до 1 и выберем из них минимальный.

```
// Тест 2. Сравнение решения задачи методом динамического программирования
и наивным алгоритмом.
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ make run
./solution <../tests/test_100.t
221
Time of Dynamic Programming method: 81 ms
221
Time of Naive Algorithm: 283 ms
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ ./solution
245
773
Time of Dynamic Programming method: 95 ms
773
Time of Naive Algorithm: 10737 ms
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ ./solution
535
1522
Time of Dynamic Programming method: 113 ms
1522
Time of Naive Algorithm: 644826 ms
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab7/benchmark$ make run
./solution <../tests/test_1000.t
1748
Time of Dynamic Programming method: 209 ms
1748
Time of Naive Algorithm: 65581859 ms
```



Разбреремся в результатах. Можно заметить, что на небольших размерах входных данных метод динамического программирования сохраняет почти неизменное время выполнения. При этом разница с наивным алгоритмом незначительна. Но при росте данных время выполнения задачи методом ДП продолжает расти линейно, в то время как скорость работы наивного алгоритма значительно снижается, и при порядке числа **n** превышающем 10^4 время ожидания превысило 5 минут.

Тесты создавались с помощью программы на языке Python:

```
1  from random import choice
2
3  NUMBER = [102, 1000, 10000, 100000, 250000, 500000, 750000, 1000000, 9999900]
4
5  length_of_list = len(NUMBER)
6
7  def write_n():
8      file.write(str(n))
9
10 for i in range(length_of_list):
11     n = NUMBER[i] + choice(range(-100, 100))
12     if (NUMBER[i] == 102):
13         with open('test_100.t', 'w') as file:
14             write_n()
```

```
    elif (NUMBER[i] == 9999900):
        with open(f'test_10000000.t', 'w') as file:
            write_n()
    else:
        with open(f'test_{NUMBER[i]}.t', 'w') as file:
            write_n()
```

4 Выводы

Выполнив седьмую лабораторную работу по курсу «Дискретный анализ», я изучил метод динамического программирования. Обычно имеется два эквивалентных способа реализации подхода динамического программирования.
Первый подход - **нисходящий с запоминанием** (memoization). При таком подходе мы пишем процедуру рекурсивно, как обычно, но модифицируя ее таким образом, чтобы она запоминала решение каждой подзадачи.
Второй подход - **восходящий**. Обычно он зависит от некоторого естественного понятия "размера"подзадачи, такого, что решение любой конкретной подзадачи зависит только от решения "меньших"подзадач. Каждую подзадачу мы решаем только один раз, и к моменту, когда мы впервые с ней сталкиваемся, все необходимые для ее решения подзадачи уже решены.

С помощью ДП решается большинство задач оптимизации, например, оптимальное хранение, оптимальное производство, оптимальный порядок и др. Однако в реальности задачи могут быть настолько большими, что время, затраченное на их решение, может слишком большим. Поэтому стоит помнить и о других алгоритмах решения задач, например о жадных алгоритмах.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))

[2] *Dynamic Programming*
URL: `https://www.geeksforgeeks.org/dynamic-programming/` (дата обращения: 21.04.2021).