Московский авиационный институт
(национальный исследовательский университет)

Институт информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №7 по курсу «Дискретный анализ»

Студент: И. С. Глушатов
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б-19
Дата:
Оценка:
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:

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.

1 Описание

Динамическое программирование - это способ решения сложных задач путем разбиения на более простые.

В моем задании наивное решение предполагает рекурсивный вызов по рекуррентной формуле:

$$f(n) = \begin{cases} 0, & n = 1 \\ min\{f(n-1)+n; f(n/2)+n; f(n/3)+n\}, & n \vdots 2,\, n \vdots 3 \\ min\{f(n-1)+n; f(n/2)+n\}, & n \vdots 2 \\ min\{f(n-1)+n; f(n/3)+n\}, & n \vdots 3 \\ f(n-1)+n, & \end{cases}$$

Вычислим примерно сложность такого решения:

Для каждого n можно построить дерево вызовов. Пусть функция g(n) возвращает количество вершин этого дерева.

$g(1) = 1;$
$g(2) = g(1) + g(1) + 1 = 3;$
$g(3) = g(2) + g(1) + 1 = 5;$
$\vdots$
$g(6) = g(5) + g(3) + g(2) + 1 = 18;$
$\vdots$
$g(n) \leq g(n-1) + g(n/2) + g(n/3) + 1;$

$$g(n/2) < g(n-1)$$
$$g(n/3) + 1 < g(n-1)$$

$=> g(n) < 3g(n-1)$
$=> g(n) < 3^2 g(n-2)$
$=> g(n) < 3^3 g(n-2)$
$=> g(n) < 3^{n-1} g(1)$
$=> g(n) < 3^{n-1}$

Отсюда следует, что верхняя оценка сложности наивного алгоритма $O(3^n)$. При нижней оценке мы предполагаем, что рекурсия вызывается только для $g(n-1)$, поэтому она будет равняться $O(n)$.

Эту задачу можно решить эффективнее за $O(n)$ в любом случае. Для этого мы избавимся от одних и тех же подсчетов f(n). Введем массив длины n, где для каждого числа будем хранить минимальную стоимость. Вычислять эти значения мы будем восходящим путем лишь единожды, за счет чего будет обеспечена линейная сложность как по времени, так и по памяти. В итоге результат будет храниться в последней ячейке массива.

2 Исходный код

```
unsigned int n;
std::cin >> n;

unsigned int* table = new unsigned int[n]();

for (unsigned int ind = 1; ind < n; ind++) {
  unsigned int number = ind + 1;

  if (number % 2 == 0 and number % 3 == 0) {
    if (table[ind - 1] < table[ind / 2]) {
      if (table[ind - 1] < table[ind / 3]) {
        table[ind] = table[ind - 1] + number;
      } else {
        table[ind] = table[ind / 3] + number;
      }
    } else {
      if (table[ind / 2] < table[ind / 3]) {
        table[ind] = table[ind / 2] + number;
      } else {
        table[ind] = table[ind / 3] + number;
      }
    }
  } else if (number % 3 == 0) {
    if (table[ind - 1] < table[ind / 3]) {
      table[ind] = table[ind - 1] + number;
    } else {
      table[ind] = table[ind / 3] + number;
    }
  } else if (number % 2 == 0) {
    if (table[ind - 1] < table[ind / 2]) {
      table[ind] = table[ind - 1] + number;
    } else {
      table[ind] = table[ind / 2] + number;
    }
  } else {
    table[ind] = table[ind - 1] + number;
  }
}
```

Востановление результата происходит с конца. Мы знаем, что стоимость на последнем шаге ровно на n больше, чем на предыдущем. Поэтому из конечной стоимости вычитается n и уже новая стоимость ищется в табличке. Найдя её мы узнаем, какую операцию мы совершили и продолжаем так, пока не дойдем до n = 0.

```
std::string res = "";

unsigned int last_n = n;
unsigned int new_n = n;
long int i = n - 1;
unsigned int delta = table[i] - (i + 1);

while (i >= 0) {
  if (delta == table[i]) {
    last_n = i + 1;

    if (last_n * 3 == new_n) {
      res += "/3 ";
    } else if (last_n * 2 == new_n) {
      res += "/2 ";
```

```
    } else if (last_n + 1 == new_n) {
      res += "-1 ";
    } else {
      i--;
      continue;
    }

    new_n = last_n;
    delta -= last_n;
  }

  i--;
}

std::cout << table[n-1] << std::endl;
std::cout << res << std::endl;
```

3 Консоль

```
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/lab7$ ./main.out
576
894
/3 /3 /2 /2 /2 /2 /2 /2
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/lab7$ ./main.out
123456789
208825723
/3 /3 -1 /2 /2 -1 /3 -1 /3 /3 -1 /2 /2 -1 /3 /3 /3 /3 /2 /2 /2 -1 /3 /2 /2
/2 /2
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/lab7$
```

4 Тест производительности

Для тестов я использвал утилиту gnuplot для построения графиков зависимости времени работы программы от величины числа. Так же для сравнения использовал библиотеку chrono для замера времени.

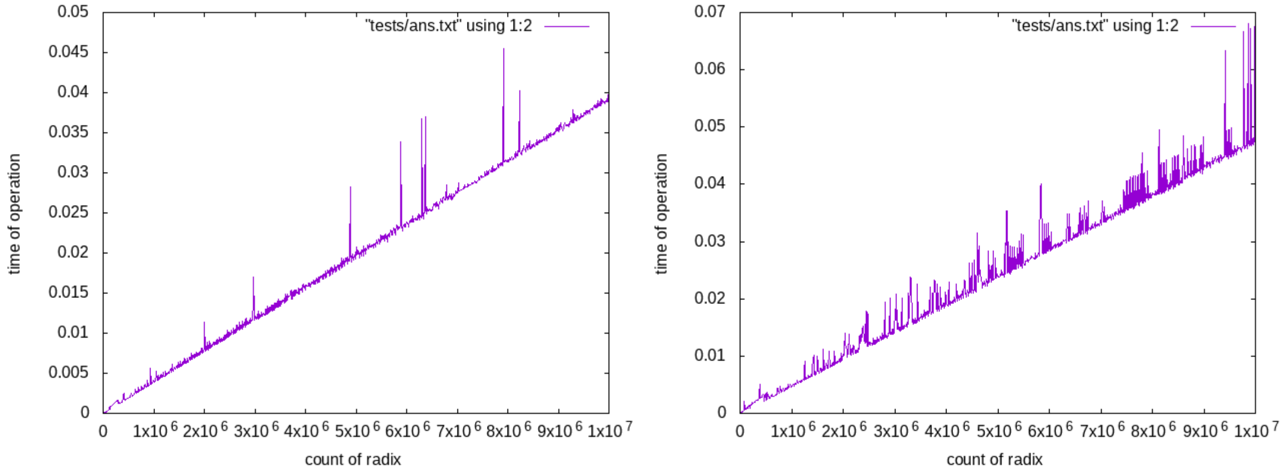

Рис. 1: Графики работы с поиском пути и без него

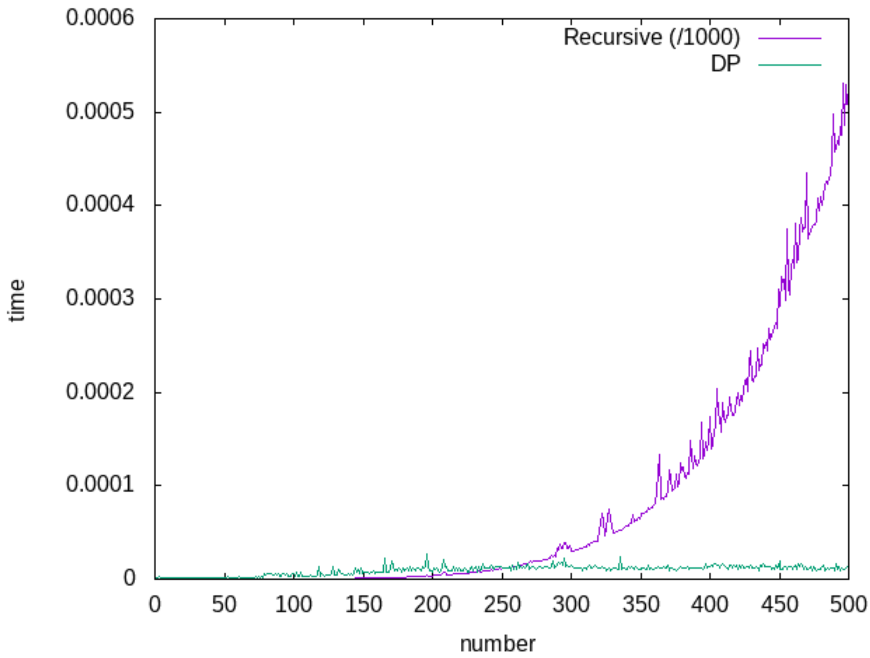

Рис. 2: Графики работы наивного алгоритма и с приминением методов динамического программирования

5 Выводы

В ходе седьмой лабораторной работы я познакомился с методами динамического программирования. Увидел, насколько рекурсивные алгоритмы могут быть плохие и понял, что при возможности их надо избегать. Особых проблем с лабораторной работой не выявилось.

Список литературы

[1] *Поисковик - Google.*
URL: `https://www.google.com/`

[2] *Сайт с подробной документацией библиотек C++*
URL: `https://en.cppreference.com/`