Национальный исследовательский университет
"Московский авиационный институт"
Факультет №8 "Информационные технологии и прикладная математика"
Кафедра 806 "Вычислительная математика и программирование"

ЛАБОРАТОРНАЯ РАБОТА №7
ПО КУРСУ “ДИСКРЕТНЫЙ АНАЛИЗ”
4 СЕМЕСТР

Выполнил студент:	Поляков А.И.
Группа:	М80-208Б-19
Оценка:	
Подпись:	

Москва 2022

Лабораторная работа №7

Вариант № 4 - Игра с числом

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 $2 \leq n \leq 107$.

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.

Описание

Согласно Кормену, 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.

Рассмотрим метод решения предложенной в данной работе задачи. 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.

Исходный код

Код программы состоит из двух файлов: *lib.h* и *main.cpp*. В *main.cpp* содержится код для ввода принятых значений, вызова функций из библиотеки. Файл *lib.h* содержит три функции: *TrippleMin* для поиска минимального числа из трех, поданных на вход.

```
int TrippleMin(int a, int b, int c) {
    return (std::min(a, b) < std::min(b, c)) ? std::min(a, b) : std::min(b, c);
}
```

MinPath для рассчета наименьшего пути получения всех чисел от 0 до n. После выполнения, данная функция возвращает "цену"последнего элемента.

```
int MinPath(int* array, int n) {
    array[0] = 0;
    array[1] = 0;
    for (int i = 2; i <= n; i++) {
     if (i % 2 == 0 && i % 3 == 0) {
      array[i] = i + TrippleMin(array[i / 2], array[i / 3], array[i - 1]);
      continue;
     }
     if (i % 3 == 0) {
      if (array[i / 3] < array[i - 1]) {
       array[i] = i + array[i / 3];
      } else {
       array[i] = i + array[i - 1];
      }
      continue;
     }

     if (i % 2 == 0) {
      if (array[i / 2] < array[i - 1]) {
       array[i] = i + array[i / 2];
      } else {
       array[i] = i + array[i - 1];
      }
      continue;
     }
     array[i] = i + array[i - 1];
    }
    return array[n];
}
```

ReconstructPath - функция для вывода пути к итоговому числу.

```
void ReconstructPath(int* array, int n) {
    int temp = n;
    while (temp > 1) {
     if ((temp % 3 == 0) && (temp % 2 == 0)) {
      if (array[temp / 3] <= array[temp / 2] && array[temp / 3] <= array[temp - 1]) {
       std::cout << "/3" << ' ';
       temp /= 3;
      } else if (array[temp / 2] <= array[temp / 3] && array[temp / 2] <= array[temp - 1]) {
       std::cout << "/2" << ' ';
       temp /= 2;
      } else {
```

```
       std::cout << "-1";
       temp--;
      }
      continue;
     }
     if (temp % 3 == 0) {
      if (array[temp / 3] <= array[temp - 1]) {
       std::cout << "/3" << ' ';
       temp /= 3;
      } else {
       std::cout << "-1" << ' ';
       temp--;
      }
      continue;
     }
     if (temp % 2 == 0) {
      if (array[temp / 2] <= array[temp - 1]) {
       std::cout << "/2" << ' ';
       temp /= 2;
      } else {
       std::cout << "-1" << ' ';
       temp--;
      }
      continue;
     }
     std::cout << "-1" << ' ';
     temp--;
    }
    std::cout << std::endl;
}
```

Проведем оценку сложности алгоритма и количество дополнительной памяти. При выполнении алгоритма мы исполняем максимум два прохода по массиву длинны n, из чего следует что итоговая сложность алгоритма $O(n)$. Затраты дополнительной памяти - $O(n)$.

Тест производительности

Произведем тестирование для различного размера входных данных. Можно заметить, что из-за линейной сложности алгоритма, при увеличении числа входных данных, время работы алгоритма пропорционально увеличивается.

```
user@AN-LAP-1110:/mnt/c/Users/Andrew/Desktop/ДА 4/lab7$ ./solution < test_
221
Execution time: 50 ms
user@AN-LAP-1110:/mnt/c/Users/Andrew/Desktop/ДА 4/lab7$ ./solution < test_
1748
Execution time: 73 ms
user@AN-LAP-1110:/mnt/c/Users/Andrew/Desktop/ДА 4/lab7$ ./solution < test_
25693
Execution time: 106 ms
user@AN-LAP-1110:/mnt/c/Users/Andrew/Desktop/ДА 4/lab7$ ./solution < test_
252298
Execution time: 699 ms
user@AN-LAP-1110:/mnt/c/Users/Andrew/Desktop/ДА 4/lab7$ ./solution < test_
948359
Execution time: 1567 ms
user@AN-LAP-1110:/mnt/c/Users/Andrew/Desktop/ДА 4/lab7$ ./solution < test_
1022773
Execution time: 3842 ms
user@AN-LAP-1110:/mnt/c/Users/Andrew/Desktop/ДА 4/lab7$ ./solution < test_
1758791
```

В отличии от наивного алгоритма перебора данных, в котором мы проходим все пути от 1 до n и затем выбираем наименьший, данный алгоритм не проверяет все пути, а только наименьшие, вследствие чего время его работы значительно сокращается.

Вывод

Выполнив 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. 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.

Литература

[1] Кормен Томас Х., Лейзерсон Чарльз И. Алгоритмы. Построение и анализ. Третье издание. (2019)

[2] Дэн Гасфилд. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология. — Издательский дом «Невский Диалект», 200ц. Перевод с английского: И. В. Романовского.

[3] Dynamic Programming URL: https://www.geeksforgeeks.org/dynamic-programming/ (дата обращения: 12.11.2021).