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: А. О. Дубинин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2019

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 6 (Палиндромы): Задана строка S состоящая из n прописных букв латинского алфавита. Вычеркиванием из этой строки некоторых символов можно получить другую строку, которая будет являться палиндромом. Требуется найти количество способов вычеркивания из данного слова некоторого (возможно, пустого) набора таких символов, что полученная в результате строка будет являться палиндромом. Способы, отличающиееся только порядком вычеркивания символов, считаются одинаковыми.

Входные данные: Задана одна строка S ($|S| \leq 100$).

Выходные данные: Необходимо вывести одно число – ответ на задачу. Гарантируется, что он $\leq 2^{63} - 1$.

1 Описание

Согласно[1] , динамическое программирование позволяет решать сложные задачи путём разбиения их на более простые подзадачи. Он применим к задачам с оптимальной подструктурой, то есть для набор перекрывающихся подзадач, сложность которых меньше исходной. В этом случае время вычислений, по сравнению с рекурсивными и очевидными методами, можно значительно сократить.

Ключевая идея в динамическом программировании: чтобы решить поставленную задачу, требуется решить отдельные части задачи (подзадачи), из результата этих подзадач сделать выбор, после чего объединить решения подзадач в одно общее решение. Часто многие из этих подзадач одинаковы. Подход динамического программирования состоит в том, чтобы решить каждую подзадачу только один раз, сократив тем самым количество вычислений.

Для моей задачи для начала определим оптимальную подструктуру, пусть массив d[i][j] = количеству способов вычеркивание символов для получения палиндрома в S[i:j]. Определим базу рекурсии d[i][i] = 1, так как любая строка состоящая из одного символа является палиндромом. Если S[i] = S[j] в строке, то решение будет состоять из суммы:

- 1(решение состоящие из строки S[i]+S[j])
- d[i + 1][j - 1](тут мы не удаляем ни символ S[i], ни символ S[j], считаем внутренее решение + наши символы)
- d[i][j - 1](смотрим решение без правого символа)
- d[i + 1][j](смотрим решение без левого символа)
- - d[i + 1][j - 1](удаляем второе посчитаное решение, которое лишним было посчитано на предыдущем шаге)

Упростив формулу получим: d[i][j] = d[i][j - 1] + d[i + 1][j] + 1. Если S[i] $\neq$ S[j], то формула принимает вид суммы:

- d[i][j - 1](смотрим решение без правого символа)
- d[i + 1][j](смотрим решение без левого символа)
- - d[i + 1][j - 1](удаляем второе посчитаное решение, которое лишним было посчитано на предыдущем шаге)

Определим рекурентную формулу для решения задачи:

$$d[i][j] = \begin{cases} 0 & i > j \\ 1 & i = j \\ d[i][j-1] + d[i+1][j] + 1 & S[i] = S[j] \\ d[i][j-1] + d[i+1][j] - d[i+1][j-1] & S[i] \neq S[j] \end{cases}$$

2 Исходный код

Нисходящая реализация алгоритма.

main.cpp

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

int64_t matrix[101][101];

int64_t getCount(std::string &str, int i, int j) {

    if (i == j)
        return 1;

    if (i > j)
        return 0;

    if(matrix[i][j] == 0) {
        if (str[i] == str[j]) {
            matrix[i][j] = getCount(str, i + 1, j) + getCount(str, i, j - 1) + 1;
        } else {
            matrix[i][j] = getCount(str, i + 1, j) + getCount(str, i, j - 1) - getCount
                (str, i + 1, j - 1);
        }
        return matrix[i][j];
    }

    return matrix[i][j];

}


int main() {

    std::string str;
    std::cin >> str;

    std::cout << getCount(str, 0, str.size() - 1) << std::endl;

    return 0;
}
```

Восходящая реализация алгоритма. **main.cpp**

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

int64_t matrix[101][101];

```

```
6  int64_t getCountUp(std::string &str) {
7
8      for (int i = 0; i < str.size(); ++i) {
9          matrix[i][i] = 1;
10     }
11
12     for (int i = 0, j = 1; j < str.size(); ++i, ++j) {
13         if (str[i] == str[j])
14             matrix[i][j] = 3;
15         else
16             matrix[i][j] = 2;
17     }
18
19     for (int l = 2; l < str.size(); ++l) {
20         for (int i = 0, j = l; j < str.size(); ++i, ++j) {
21
22             if (str[i] == str[j]) {
23                 matrix[i][j] = matrix[i + 1][j] + matrix[i][j - 1] + 1;
24             } else {
25                 matrix[i][j] = matrix[i + 1][j] + matrix[i][j - 1] - matrix[i + 1][j -
                       1];
26             }
27
28         }
29     }
30
31    return matrix[0][str.size()-1];
32
33 }
34
35 int main() {
36
37     std::string str;
38     std::cin >> str;
39
40     std::cout << getCountUp(str) << std::endl;
41
42     return 0;
43 }
```

3 Производительность

Сравним две реализации алгоритма, восходящую и нисходящую.
Тест на 1000 операций:

```
<From up to bottom:
<Time of working 14 ms.
---
>From bottom to up:
>Time of working 7 ms.
```

Тест на 5000 операций:

```
<From up to bottom:
<Time of working 294 ms.
---
>From bottom to up:
>Time of working 247 ms.
```

Тест на 10000 операций:

```
<From up to bottom:
<Time of working 1160 ms.
---
>From bottom to up:
>Time of working 1344 ms.
```

Из тестов видно, что на маленьких тестах нисходящий алгоритм работает быстрее, ибо считает не всю таблицу, т.е. не все подзадачи. Но на большом тесте тенденция меняется в пользу иттеративного алгоритма, так как затраты на рекурсивные вызовы функций слишком затратны и проигрывают простому for циклу.

4 Выводы

Динамическое программирование применяется при наличии характерных признаков:

- оптимальность для подзадач
- наличие в задаче перекрывающихся подзадач

Действительно в данной задаче для решения сложной задачи требуется идти от более простой. Эта задача была похожа на классическую задачу о наибольшей общей подпоследовательности, именно она помогла найти способ решение к этой задаче.

Динамическое программирование показывает возможность создания очень быстрых алгоритмов для задач, в которых нужно оптимизировать какую-либо величину или посчитать количество вариантов чего-либо. В случае наивного подхода к решению подобной задачи, решение для большого теста может быть и не найдено в адекватное время, поскольку перебор всех вариантов очень и очень затратная операция.

Список литературы

[1] Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн *Алгоритмы: построение и анализ, 3-е изд.* — Издательский дом « Вильямс», 2013. Перевод с английского: ООО «И. Д. Вильямс». — 1328 с. (ISBN 978-5-8459-1794-2 (рус.))