МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
высшего образования
«Московский технологический университет»

МИРЭА

Институт информационных технологий (ИТ)
Кафедра инструментального и прикладного программного обеспечения (ИППО)

Отчет по практической работе № 3
по дисциплине
«Структуры и алгоритмы обработки данных»

Выполнил студент группы ИСБОп-01-14 Карих Д.С.

Принял Шмелёва Д.В.

Москва 2017

Динамическое программирование. Задача о форматировании

В ходе выполения данной практической работы было реализовано три алгоритма форматирования текста: жадный (greedy), грубой силы (brute_force) и использующий динамическое программирование (dynamic).

Функция generator создаёт случайный вектор целых чисел длиной N (количество слов), характеризующих длину каждого слова в тексте.

```
std::vector<int> generator(int N) {
    std::vector<int> result;
    REPEAT(result.push_back(rand() % 5 + 3), N);
    return result;
}
```

Все три функции расстановки переносов принимают на вход данный вектор и возвращают число типа long int. При переводе данного числа в двоичный вид, позиции единиц будут совпадать с позициями переносов.

Жадный алгоритм заполняет каждую строку до тех пор, пока в ней есть место, а затем переходит на следующую. Такой подход не всегда приводит к идеальной расстановке переносов, однако работает очень быстро. Примерная сложность алгоритма: O(N) (линейная).

```
long int greedy(std::vector<int> words) {
    long int solution = 0;
    int current_line_size = 0;

    for (unsigned int i = 0; i < words.size(); i++) {
        if (LINE_MAX - current_line_size < words[i]) {
            solution |= 1 << (i - 1);
            current_line_size = 0;
        }
        current_line_size += words[i] + 1;
    }

    return solution;
}
```

Алгоритм **полного перебора** проверяет все возможные комбинации переносов. Количество таких комбинаций равно 2^n, поэтому полный перебор на большом количестве слов может занять очень много времени. Сложность этого алгоритма составляет $O(2^n)$.

```
long int brute_force(std::vector<int> words) {
    long int best_state = 0;
    int best_quality = -1;

    for (long int state = 0;
         state < pow(2, words.size());
         state++) {

        int quality = check_state(words, state);

        if (quality == -1)
            continue;

        if (quality < best_quality
            || best_quality == -1) {
            best_state = state;
            best_quality = quality;
        }
    }

    return best_state;
}
```

Динамический алгоритм подходит к проблеме с другой стороны: сначала рассчитываются остатки для всех возможных переносов $(N^2/2$ операций$)$, затем полным перебором находит наилучший вариант $(\text{тоже } N^2/2 \text{ операций})$. Таким образом, сложность алгоритма составляет $O(N^2)$.

```
long int dynamic(std::vector<int> words) {
    long int result = 0;
    int S[words.size()][words.size()] = {};
```

```
    int R[words.size() + 1] = {};
    int B[words.size()] = {};

    for (unsigned int i = 0; i < words.size(); i++) {
        S[i][i] = LINE_MAX - words[i];
        for (unsigned int j = i+1; j < words.size(); j++)
            S[i][j] = S[i][j-1] - words[j] - 1;
    }

    for (unsigned int j = 0; j < words.size(); j++) {
        for (int i = j; i >= 0; i--) {
            // Игнорируем слишком длинные строки
            if (S[i][j] < 0) continue;

            int cost = R[i] + pow(S[i][j], 3);
            if (R[j+1] > cost || R[j+1] == 0) {
                R[j+1] = cost;
                B[j] = i;
            }
        }
    }

    int j = words.size();
    while (j > 0) {
        int i = B[j-1];
        if (result != 0) {
            result |= 1;
            result = result << (j-i);
        } else {
            result = 1;
        }
        j = i;
    }

    return result >> 1;
}
```

Для проверки и отображения результата были созданы функции check_state – рассчёт суммы кубов остатков для всех строк и print_vector – вывод результата в наглядном виде. Ниже приведён пример работы программы для N = 24.

```
greedy: 2378020 (1911) 158 µs
AAAAA AAAA AAAAA    | 4
AAAAAA AAA AAAAAA   | 3
AAA AAAAAAA AAAA    | 4
AAAAAA AAAAAA AAAAAA| 0
AAAA AAAAAA AAAAA   | 3
AAAA AAAAAA AAA AAA | 1
AAAAA AAAAAA AAAAAAA| 0
AAA AAA             | 13
--
brute_force: 1198372 (435) 29490832 µs
AAAAA AAAA AAAAA    | 4
AAAAAA AAA AAAAAA   | 3
AAA AAAAAAA AAAA    | 4
AAAAAA AAAAAA AAAAAA| 0
AAAA AAAAAA AAAAA   | 3
AAAA AAAAAA AAA     | 5
AAA AAAAA AAAAAA    | 4
AAAAAAA AAA AAA     | 5
--
dynamic: 1198372 (435) 120 µs
AAAAA AAAA AAAAA    | 4
AAAAAA AAA AAAAAA   | 3
AAA AAAAAAA AAAA    | 4
AAAAAA AAAAAA AAAAAA| 0
AAAA AAAAAA AAAAA   | 3
AAAA AAAAAA AAA     | 5
AAA AAAAA AAAAAA    | 4
AAAAAAA AAA AAA     | 5
```

В данном случае полный перебор занял 29.5 секунд. С каждым новым словом это время стабильно увеличивается в 2 раза. Также можно заметить недостаток жадного алгоритма (1911 против 435).

Заключение

Динамическое программирование значительно превосходит по производительности метод полного перебора, не уступая при этом по качеству получаемых результатов. Динамические алгоритмы гораздо сложнее в реализации, однако выигрыш в производительности делает их очень значимыми в обработке больших объёмов данных.

Список источников

1. Википедия: Жадный алгоритм // https://ru.wikipedia.org/wiki/Жадный_алгоритм (дата обращения: 21.05.2017)
2. Википедия: Полный перебор // https://ru.wikipedia.org/wiki/Полный_перебор (дата обращения: 21.05.2017)
3. Википедия: Динамическое программирование // https://ru.wikipedia.org/wiki/Динамическое_программирование (дата обращения: 21.05.2017)
4. Документация C++: std::vector // http://en.cppreference.com/w/cpp/container/vector (дата обращения: 21.05.2017)