Математические основы алгоритмов

А. С. Охотин

Определение 1. *Машина с произвольным доступом в память (RAM).* У нас есть память в виде ячеек на $\mathbb{Z}$, где хранятся целые числа. Будем называть

- *константой* (и писать "n") всякую целочисленную константу,
- *прямой адресацией* (и писать "x_n") получение значения по адресу, заданным константой n,
- *косвенной адресацией* (и писать "x_{x_n}") получение значения по адресу, заданным значением по адресу, заданным значением константой n.

Программы — конечные последовательности, состоящие из команд (строчек команд) следующего типа:

- присваивание: `A = B`, где `B` может быть константой или адресацией, а A может быть только адресацией;
- арифметические операции: `A = B + C`, `A = B - C`, `A = B × C`, `A = B / C`, `A = B ÷ C`, где `B` и `C` — константа или адресация, а `A` — адресация;
- перевод чтения программы на строку с номером n: `GOTO n`;
- перевод чтения программы на строку с номером x_n (значения ячейки по адресу n): `GOTO x_n`;
- `IF A == B THEN GOTO n` (вместо `A == B` может быть `A >= B`; вместо `n` может быть `x_n`), где `A` и `B` — константы или адресации;
- команда остановки: `HALT`.

Замечание. Вся суть вопросов заключена в том, чтобы вычислить какую-то функцию. В таком случае задачу можно воспринимать так, что в ячейке 0 записано количество входных значений n, а в ячейках от 1 до n записаны значения входных параметров, а требуется вычислить функцию от этих входящих значений и оставить их в таком же виде.

Также иногда можно писать менее низкоуровневые команды, если понятна их низкоуровневая реализация. Например, "провести ребро из v в u", "просумировать n конкретных значений", а не две как это реализуется командой `A = B + C` и т.д.

Пример 1. Функция вычисления факвториала n выглядит следующим образом.

```
функция f(n):
    если n = 0
        ответ 1
    иначе
        ответ n * f(n - 1)
```

что низкоуровнево может быть реализуемо как

```
1. A = n
2. RES = 1
3. IF A == 0 GOTO 7
4. RES = RES * A
5. A = A - 1
6. GOTO 3
7. HALT
```

Также нерекурсивно можно реализовать так.

```
функция f(n):
    пусть x = 1
    для i = 1, ..., n
        x = x * i
    ответ x
```

что низкоуровнево может быть реализуемо как

```
1. RES = 1
2. ITER = 1
3. IF ITER > n GOTO 7
4. RES = RES * ITER
5. ITER = ITER + 1
6. GOTO 3
7. HALT
```

Определение 2. *Сложность работы программы* — это функция $t(n)$ равная максимуму затрачиваемого времени по всем входным данным длины n. При этом время вычисляется как сумма стоимостей всех выполненных операций.

В обычной модели стоимость равна 1 для каждой операции. Но бывает проблемма, что, например, если хочется найти $a+b$ и $c+d$, то можно записать a и c в одну ячейку (например, как $a+10^k*c$), а b и d в другую и сложить их за одну операцию вместо двух. Да и вообще бывает проблема, что складывание двух больших чисел и двух маленьких в реальности являются задачами разной сложности. Поэтому есть также модель "log-cost", где каждая операция стоит логарифм от входящих в неё значений.

Определение 3. *Сложность памяти программы* — это такая функция $s(n)$ равная максимуму затрачиваемого места по всем входным данным длины n. В качестве затрачиваемого места подразумевается количество изменённых (хоть раз) ячеек.

Определение 4. Для всяких функций $f(n)$ и $g(n)$ скажем, что

- $g = o(f)$, если $\lim_{n\to\infty} \frac{|g(n)|}{|f(n)|} = 0$,
- $g = O(f)$, если $|g| \leqslant C|f|$ для некоторой константы $C > 0$ с некоторого момента,
- $g = \omega(f)$, если $\lim_{n\to\infty} \frac{|g(n)|}{|f(n)|} = +\infty$,
- $g = \Omega(f)$, если $|g| \geqslant C|f|$ для некоторой константы $C > 0$ с некоторого момента,

- $g = \Theta(f)$, если $c|f| \leqslant |g| \geqslant C|f|$ для некоторых констант $c, C > 0$ с некоторого момента.

Теорема 1. *Умножение двух чисел длины не более n можно посчитать за время $O(n^2)$.*

Доказательство. Действительно, можно посчитать произведение в столбик. В таком случае будет произведено n^2 умножений и $\approx n^2$ сложений. В таком случае произведение можно посчитать за $O(n^2)$ шагов. При этом памяти можно использовать не более $2n$ при том же времени работы, если сразу прибавлять полученные произведения к нулю. $\square$

Теорема 2 (Карацуба)**.** *Умножение двух чисел длины не более n можно посчитать за время $O(n^{\log_2(3)})$ ($\log_2(3) \approx 1.58$).*

Доказательство. Пусть даны два числа $a = \overline{A_1A_2}$ и $b = \overline{B_1B_2}$, где A_1, A_2, B_1, B_2 — последовательности цифр длины k. Тогда $c = ab$ имеет вид $\overline{C_1C_2C_3}$, где

$$C_1 = A_1B_1, \qquad C_2 = A_1B_2 + A_2B_1 = (A_1 + A_2)(B_1 + B_2) - A_1B_1 - A_2B_2, \qquad C_3 = A_2B_2.$$

Следовательно посчитать произведение можно с помощью всего трёх произведений (и операции переноса через разряды занимает линейное время от длины): A_1B_1, A_2B_2 и $(A_1 + A_2)(B_1 + B_2)$.

Давайте разобьём наши числа $\overline{a_{n-1} \dots a_0}$ и $\overline{b_{n-1} \dots b_0}$ на две приемерно равные половины: $\overline{a_{n-1} \dots a_0} = \overline{A_1A_2}$, $\overline{b_{n-1} \dots b_0} = \overline{B_1B_2}$. (Важно чтобы длины последовательностей были одинаковой длины, но можно у A_1 и B_1 добавить фиктивные нули в начале.) Тогда асимптотика перемножения для длины n будет описываться формулой

$$T(n) = 3T(n/2) + O(n).$$

Следовательно несложно понять по индукции, что $T(n) = O(n^{\log_2(3)})$. $\square$

Теорема 3.

1. *Сортировка пузырьком работает за $O(n^2)$.*

2. *Сортировка вставками (добавлением элемента) работает за $\approx n^2/2$.*

Теорема 4. *Сортировка слиянием (merge sort) работает за $O(n \log(n))$.*

Доказательство. Заметим, что слияние двух отсортированных массивов длины не более n можно реализовать за $2n$ операций. Следовательно асимптотика сортировки для длины n будет описываться формулой

$$T(n) = 2T(n/2) + O(n),$$

откуда $T(n) = O(n \log(n))$. $\square$

Теорема 5 ("быстрая сортировка", Хоар (Hoar), 1961)**.** *Быстрая сортировка работает за $O(n \log(n))$.*

Доказательство. Рассмотрим следующий алгоритм.

1. Выберем случайный элемент y.

2. Разделим весь массив без y на две части: элементы $\leqslant y$ и элементы $> y$, и поставим их в порядке "элементы $\leqslant y$, y, элементы $> y$".

3. Применим быструю сортировку к полученным частям.

Понятно, что после сортировки каждой из оставшихся частей, массив станет останется отсортированным. Давайте более конкретно опишем алгоритм:

```
function quicksort(l, m)
    if m - l >= 2 then
        i = partition(l, m)
        quicksort(l, i)
        quicksort(i+1, m)

function partition(l, m)
    choose random p among l, ..., m-1
    y = x_p
    x_p <-> x_{m-1}
    i = l
    j = m - 1
    while i < j do
        if x_i < y then
            i = i + 1
        else if x_j >= y then
            j = j - 1
        else
            x_i <-> x_j
    x_{i-1} <-> x_{m-1}
    return i-1
```

Пусть $C(n)$ — количество сравнений во время работы алгоритма. Далее несложно убедиться по индукции, что

- $C(n) = \Omega(n \log(n))$,
- минимальное количество сравнений на одном и том же массиве будет достигаться только при выборе медиан,
- $C(n) = O(n^2)$,
- максимальное количество сравнений на одном и том же массиве будет достигаться только при выборе крайних элементов.

Также заметим, что если выбор случайного элемента y имеет равновероятное распределение, то вероятность того, что y_i и y_j в отсортированном массиве будут сравнены, равна $2/(j-i+1)$. Действительно, в самом начале обрабатывается массив содержащий все элементы y_i, y_{i+1}, $\dots$, y_j, и пока никакой из этих элементов не будет выбран как опорный, то все они будут находится в одном обрабатываемом массиве и y_i и y_j не будут сравнены. Если же будет выбран элемент y_k для $i < k < j$, то y_i и y_j будут сравнены с y_k, попадут в разные обрабатываемые массивы и больше никогда не будут сравнены. Если же будет выбран один из y_i и y_j, то они будут сравнены единожды. Значит вероятность того, что y_i и y_j будут сравнены равна $2/(j-i+1)$. Значит мат. ожидание количества сравнений равно

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \approx \sum_{i=1}^{n-1} 2\ln(n-i) \approx 2n\ln(n).$$

□