Краткий экскурс по основным алгоритмам и структурам данных спортивного программирования

9 февраля 2025 г.

Содержание

1 Введение

1.1 Кратко о спортивном программировании

2 Асимптотика

2.1 Сравнение различных программ и O большое

Ключевой характеристикой программы является время её работы. Оно, по сути, прямо пропорционально количеству операций, которые необходимо произвести компьютеру перед тем, как программа прекратит свою работу. Чаще всего это количество некоторым образом зависит от количества данных, подаваемых на вход программе. Для последующих рассуждений обозначим эту зависимость как $f(n)$, где n — количество входных данных.

Итак, для оценки времени работы программы нужно знать *точное* количество операций, которые она затребует после запуска. В особо острых ситуациях, когда очень важно быстродействие, так и есть, но, бывает, можно обойтись и следующей более грубой оценкой: будем считать, что программа имеет линейное время работы, если при достаточно больших значениях n значение $f(n)$ не превосходит значение другой функции $g(n) = C \cdot n$, где C — это ненулевая конечная константа. Или, более строго, для некоторых ненулевых конечных констант N и C, для любого n, большего N, выполняется $f(n) \leq C \cdot n$. То есть:

$$\frac{f(n)}{n} \leq C.$$

Если выполняется это условие, то считается, что время работы алгоритма — $O(n)$.

Помимо линейной функции можно использовать и любые другие. Например, если

$$\frac{f(n)}{n^3} \leq C,$$

при тех же ограничениях на n, C, и N, что и до этого, то время работы алгоритма — $O(n^3)$.

Таким же образом можно оценить и количество памяти, задействованной программой.

Замечание 1

Часто встречаются алгоритмы, время работы которых — $O(\log_2 n)$. В таких случаях основание логарифма обычно не пишут, так как в этой нотации оно может без потери корректности оценки быть заменено другим числом, не равным единице, (обозначим это число как a). В самом деле, по свойству логарифма имеем:

$$\frac{\log_2 n}{\log_2 a} = \log_a n$$

$$\frac{f(n)}{\log_2 n} \leq C \Rightarrow \frac{f(n)}{\log_2 n} = \frac{f(n)}{\log_a n \cdot \log_2 a} \leq C \Rightarrow \frac{f(n)}{\log_a n} \leq C \cdot \log_2 a.$$

Так как $C \cdot \log_2 a$ — тоже ненулевая конечная константа, получаем, что время работы того же самого алгоритма — не только $O(\log_2 n)$, но ещё и $O(\log_a n)$, а потому константа в основании логарифма не существенна и обычно отбрасывается.

Замечание 2

Если значение $f(n)$ не превосходит некоторую ненулевую конечную константу C, то считается, что время работы алгоритма — $O(1)$, так как

$$\frac{f(n)}{1} \leq C.$$

Замечание 3

Если известны ограничения на n (например, что n — целое число, которое обязательно находится в некотором конечном промежутке $[a; b]$) и алгоритм всегда завершает свою работу за конечное число шагов (то есть $\forall n \in [a; b] \quad f(n) < \infty$), то, имея определённую смелость, можно заявить, что время работы программы — $O(1)$, так как

$$\frac{f(n)}{1} \leq C = \max_{n \in [a;b]} f(n).$$

Таким образом, любой конечный алгоритм работает за константное время, если взять достаточно большую константу. По понятным причинам, такая оценка применяется нечасто.

2.2 Задача о поиске в отсортированном массиве

Пусть дан упорядоченный набор чисел $a_1, a_2, \ldots, a_n$ такой, что $a_1 \leq a_2 \leq \ldots \leq a_n$. Нужно проверить наличие элемента, равного некоторому целевому значению t.

Наивное решение

Будем проверять все элементы по очереди. Если значение какого-то из них равно t, то, очевидно, искомый элемент имеется. В противном случае — не имеется.

Код решения использует контейнер `vector`, который будет рассмотрен позже. Здесь нас интересует только сложность алгоритма.

```
bool lookFor(vector<int> &a, int t)
{
    int n = a.size();
    for (int i = 0; i < n; i++)
    {
        if (a[i] == t)
        {
            return true;
        }
    }
    return false;
}
```

Количество операций в каждой итерации цикла не зависит от n, а, следовательно, эти итерации имеют константную сложность. Так как цикл произведёт не более n итераций (он остановится раньше, если найдёт искомый элемент), знаем, что сложность цикла — линейная. Вычисление размера массива и возвращение ответа через *return* выполняются за константное время. В итоге, всё решение имеет сложность $O(n)$.

Оптимальное решение

Решение можно ускорить, если воспользоваться фактом отсортированности массива. Из него следует, что если некоторый элемент a_i массива *меньше* t, то все элементы *слева* от a_i так же обязаны быть меньше t и, следовательно, рассмотрению не подлежат. Аналогично, все элементы справа от элемента, превосходящего t, не интересуют нас.

Возьмём элемент a_i в середине массива. Получим три случая:

1. a_i равен t. Искомый элемент найден, задача решена.
2. a_i меньше t. Все элементы слева исключаем из рассмотрения.
3. a_i больше t. Исключаем все элементы справа.

После такого, если только алгоритм не завершит работу, зона поиска сократится приблизительно вдвое. Теперь проведём те же рассуждения с элементом a_j, находящимся посередине новой зоны поиска.

Видно, что зона поиска не может сократиться вдвое более $\lceil \log_2 n \rceil$ раз, где угловые скобки означают округление вверх. Таким образом, алгоритм совершит не более $\lceil \log_2 n \rceil$ итераций, каждая из которых выполняется за константое время. Заключаем, что общая сложность — $O(\log n)$ (округление вверх убрано, так как оно в худшем случае добавляет одну итерацию и позволима оценка $\lceil \log_2 n \rceil \leq 2 \cdot \log_2 n$).

Такой процесс называется *двоичным поиском* и будет рассмотрен в более общем случае позже.

2.3 Амортизированная сложность

3 Базовая матчасть

3.1 Быстрое возведение в степень

Текущая задача ставится очень просто: возвести некоторое число a в степень b. Ограничимся тем, что a и b — неотрицательные целые числа. На этом примере будет явно видна суть использования эффективных алгоритмов.

Итак, возведение в степень можно по определению расписать в виде кратного умножения.

$$a^b = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ раз}}.$$

Главная проблема этой формулы заключается в том, что в ней гораздо больше операций умножения (а именно $b-1$, что является $O(b)$), чем необходимо. Формально говоря, ничего плохого в этом нет, однако, если попробовать воспользоваться этим разложением для подсчёта большой степени на компьютере, то окажется, что при больших значениях b программа начинает работать очень долго.

При помощи алгоритма двоичного возведения в степень можно ускорить решение. Рассмотрим случай с чётным ненулевым b ($b = 2k, k \in \mathbb{N}$).

$$a^b = a^{2k} = (a^2)^k = \underbrace{a^2 \cdot a^2 \cdot \ldots \cdot a^2}_{k \text{ раз}}.$$

Для вычисления a^2 мы используем 1 операцию, а для возведения его в степень k ещё $k-1$ штуку. Таким образом, необходимое количество операций сократилось почти в два раза, однако есть возможность повторно ускорить подсчёт этим же рассуждением, если k тоже окажется чётным.

Возникает вполне закономерный вопрос: что делать с нечётным b. Нечётное b можно разложить как $2k+1, k \in \mathbb{N}$. Тогда формула обретёт вид:

$$a^b = a^{2k+1} = (a^2)^k \cdot a = \underbrace{a^2 \cdot a^2 \cdot \ldots \cdot a^2}_{k \text{ раз}} \cdot a.$$

Всего $k+1$ операция, что тоже чуть больше половины от исходных $b = 2k+1$ штук.

Так как мы разобрались, как можно разложить как чётную степень, так и нечётную, можем продолжить раскладывать её (теперь k) дальше. Такой способ позволяет возводить в степень значительно быстрее обычного. Этот алгоритм называется *алгоритмом двоичного возведения в степень* и работает он за $O(\log_2 b)$ операций.

Пример реализации:

```
long long bin_pow(long long a, long long b)
{
    if (b == 0) return 1;
    if (b % 2 == 0) return bin_pow(a * a, b / 2);
    return bin_pow(a * a, b / 2) * b;
}
```

3.2 Вычисления по модулю

Мы научились быстро возводить число даже в большую степень, но не сделали важное уточнение. Результат этого возведения может не уместиться в основные встроенные числовые типы многих языков программирования. Поэтому при решении задач, требующих оперирования многозначными

числами, часто ответ принимается *по модулю* некоторого числа MOD, которое обычно считается равным какому-то простому числу порядка 10^9 (мы возьмём значение $MOD = 10^9 + 7$). Это значит, что вместо ответа, который оказался равным MOD или больше, нужно выводить только остаток от деления ответа на MOD. Может показаться, что это больше усложняет задачу, но по сравнению с альтернативой (длинной арифметикой) это значимое упрощение. Хранение чисел по модулю несколько меняет то, как над ними должны происходить арифметические действия.

В *C++* числовой тип *long long* может хранить числа порядка $9 \cdot 10^{18}$, что при модуле $MOD = 10^9 + 7$ позволяет проводить операции сложения и умножения непосредственным образом.

```
ll a; // 0 <= a < MOD
ll b; // 0 <= b < MOD
ll sum = (a + b) % MOD;   // OK
ll product = (a * b) % MOD;   // OK
```

Встроенный оператор $\%$ при попытке деления отрицательного числа на положительное выдаёт отрицательный остаток от деления, поэтому во время вычитания по модулю могут получаться разные значения в зависимости от знака разности чисел.

```
cout << 10 % 7;   // 3
cout << (-10) % 7;   // -3
```

Чтобы поддерживать все используемые числа в промежутке $[0, MOD)$ можно к результату вычитания прибавить MOD.

```
ll a; // 0 <= a < MOD
ll b; // 0 <= b < MOD
ll diff = (a + MOD - b) % MOD;   // OK
```

Для возведения в целую неотрицательную степень будем использовать модифицированный алгоритм быстрого возведения в степень. Единственное его отличие от обычного заключается в том, что все операции внутри реализации считаются по модулю (кроме деления).

```
ll bin_pow(ll a, ll b)
{
    if (b == 0) return 1;
    if (b % 2 == 0) return bin_pow((a * a) % MOD, b / 2);
    return (bin_pow((a * a) % MOD, b / 2) * b) % MOD;
}
```

Здесь не было необходимости вычислять деление по модулю, так как b ни в какой момент времени не достигло MOD. Это означает, что оно никаким образом не отличалось от обычного числа, хранящегося не по модулю. Хотя в общем случае деление двух чисел по модулю по-обычному сделать не получится.

```
ll a = 3;
ll b = 4;
ll MOD = 7;
ll product = (a * b) % MOD;   // 12 % 7 = 5
cout << (product / b) % MOD;   // (5 / 4) % 7 = 1 != a
```

Для решения этой проблемы воспользуемся тем фактом, что MOD — простое число. По малой теореме Ферма имеем, что

$$\frac{a}{b} = a \cdot b^{-1} = a \cdot b^{-2} \cdot b \equiv a \cdot b^{-2} \cdot b^{MOD} \equiv a \cdot b^{MOD-2} \pmod{MOD}.$$

Чтобы возвести число b в степень $MOD - 2$, которая обычно оказывается большим числом, следует применить алгоритм быстрого возведения в степень по модулю.

```
ll inverse(ll x)
{
    return bin_pow(x, MOD - 2);
}
```

Тогда

```
ll a;  // 0 <= a < MOD
ll b;  // 0 < b < MOD
ll div = (a * inverse(b)) % MOD;  // OK
```

3.3 XOR

XOR, или операция *исключающего или*, — математическая операция, принимающая два операнда (обозначается $a \oplus b$) и имеющая несколько полезных свойств:

- Перемена мест слагаемых не меняет ответ.
- $x \oplus x = 0$ для любого x.
- $x \oplus 0 = x$ для любого x.

Из этих свойств следует, что если некоторое слагаемое встречается в XOR-сумме чётное количество раз, то все его повторения можно исключить, а если нечётное — то исключить все, кроме одного.

В этом параграфе разберём следующую задачу, удобно решающуюся с помощью исключающего или. Исходная задача здесь. Даны три строки, в каждой из которых три символа (“A”, “B”, “C”). Как в судоку, в каждой строке и в каждом столбце символы не повторяются. Но вместо одной из букв стоит знак вопроса. Задача заключается в определении того, какой символ заменён знаком вопроса.

Решать задачу можно уймой разных способов. Однако очень к месту приходится операция XOR. Рассмотрим XOR-сумму всех девяти данных символов, а так же трёх символов “A”, трёх символов “B”, трёх символов “C” и одного вопросительного знака. В этой сумме каждому знаку, кроме искомого, найдётся парный элемент, вместе с которым он сократится. После всех сокращений и останется только один искомый символ.

Например, пусть вопросительным знаком заменена “A”. Это не умаляет общности примера, потому что при другом недостающем символе все буквы можно переобозначить так, чтобы ответом на задачу оказалась “A”. Тогда в исходных данных будут две буквы “A”, три буквы “B”, три буквы “C” и вопросительный знак. В силу того, что в XOR-сумме можно менять местами слагаемые и порядок операций, имеем:

$$(A \oplus A \oplus B \oplus B \oplus B \oplus C \oplus C \oplus C \oplus ?) \oplus (A \oplus A \oplus A \oplus B \oplus B \oplus B \oplus C \oplus C \oplus C \oplus ?) =$$

$$= (A \oplus A \oplus A \oplus A \oplus A) \oplus (B \oplus B \oplus B \oplus B \oplus B \oplus B) \oplus (C \oplus C \oplus C \oplus C \oplus C \oplus C) \oplus (? \oplus ?) =$$

$$= A \oplus 0 \oplus 0 \oplus 0 = A.$$

Один из вариантов записать это решение:

```
#include <iostream>
using namespace std;

void solve()
{
    string s1, s2, s3;
    cin >> s1 >> s2 >> s3;

    char result = 0;
    for (char c : s1) result = result ^ c;
    for (char c : s2) result = result ^ c;
    for (char c : s3) result = result ^ c;
    for (char c : "AAABBBCCC?") result = result ^ c;
    cout << result << '\n';
```

```
}

int main()
{
    int t; cin >> t;
    while(t--) solve();
}
```

Есть много других задач, решаемых с помощью XOR. Нередко их объединяют какие-либо рассуждения о чётности.

3.4 Алгоритм Евклида

Задача о поиске *наибольшего общего делителя (НОД, или GCD)* и *наименьшего общего кратного (НОК, или LCM)* нескольких чисел нередко возникает при упрощении математических моделей предметов с циклическими свойствами.

Для доказательства работоспособности достаточно эффективного вычисления НОД двух чисел приведём следующие рассуждения. Пусть есть целые неотрицательные числа x и y. Не умаляя общности, обозначим их так, что $x \geq y$. Если число d — их некоторый общий делитель, то он также будет делителем $(x - y)$. Также и наоборот, если d — общий делитель $(x - y)$ и y, то он также и общий делитель x и y. Получается, что при замене x на $(x - y)$ общие делители не пропадают и не появляются. В частности

$$\gcd(x, y) = gcd(x - y, y).$$

Это рекуррентное выражение можно преобразовать, если расписать x как $q \cdot y + r$, где q и r — целая часть и остаток от деления x на y. Кратно применив вышеописанную формулу, получаем:

$$\gcd(x, y) = \gcd(q \cdot y + r, y) = \gcd(q \cdot y + r - y, y) = \gcd((q - 1) \cdot y + r, y) =$$
$$= \gcd((q - 1) \cdot y + r - y, y) = \gcd((q - 2) \cdot y + r, y) = \cdots =$$
$$= \gcd((q - q) \cdot y + r, y) = \gcd(r, y).$$

Выходит, что в случае двух аргументов больший можно заменить остатком от деления на меньший. После нескольких таких замен неизбежно придём к тривиальному случаю $\gcd(0, w) = w$. Запишем этот алгоритм, называемый *алгоритмом Евклида*.

```
ll gcd(ll x, ll y)
{
    if (x < y) return gcd(y, x);

    if (y == 0) return x;

    return gcd(y, x % y);
}
```

Если числа x и y в своих разложениях имеют общий делитель d, то он обязательно присутствует в разложении $\gcd(x, y)$. Также, если хотя бы в одном разложении x или y имеется делитель d, то он есть и в разложении $\mathrm{lcm}(x, y)$. Нетрудно заметить, что в произведении $x \cdot y$ общие делители будут дублироваться, а уникальные появятся единожды. Из этого следует равенство:

$$x \cdot y = \gcd(x, y) \cdot \mathrm{lcm}(x, y).$$

Тогда формула для НОК принимает вид $\mathrm{lcm}(x, y) = \frac{x \cdot y}{\gcd(x, y)}$.

3.5 Решето Эратосфена

4 Стандартные контейнеры

4.1 vector

`vector` представляет собой упорядоченный список элементов одинакового типа. Создавать его можно разными способами:

```cpp
#include <vector>
using namespace std;

int main()
{
    vector<int> a;
    vector<pair<int, int>> b(10);
    vector<char> c(15, 'a');
    vector<int> d = {1, 2, 3};
}
```

Сверху создались три `vector`. Первый — пустой вектор, способный содержать в себе целые числа. Второй — вектор из десяти пар целых чисел (все числа изначально — нули). Третий — вектор, содержащий пятнадцать букв `'a'`. Четвёртый — состоит из целых чисел `1, 2, 3`.

Определим вектор и число:

```cpp
vector<int> numbers = {1, 2, 3};
int number = 4;
```

Векторы поддерживает следующие основные операции (больше можно найти на cppreference):

Операция	Сложность	Пример
Добавление элемента в конец списка	$O(1)$	`numbers.push_back(number)`
Удаление последнего элемента	$O(1)$	`numbers.pop_back()`
Просмотр произвольного элемента	$O(1)$	`numbers[i]`