Оглавление

Глава 1

Теория чисел

1.1 Наибольший общий делитель

Алгоритм Евклида вычисляет НОД двух чисел, используя свойство: НОД(a, b) = НОД$(b, a \bmod b)$. На каждом шаге заменяем пару (a, b) на $(b, a \bmod b)$, пока b не станет равным 0. Тогда НОД равен a.

```
int gcd(int a, int b) {
    while (b != 0) {
        a %= b;
        swap(a, b);
    }
    return a;
}
```

1.2 Наименьшее общее кратное

Вычисление НОК основано на теореме: НОД$(a, b) \cdot$ НОК$(a, b) = a \cdot b$. Сначала вычисляем НОД с помощью алгоритма Евклида, затем находим НОК по формуле НОК$(a, b) = \frac{a \cdot b}{\text{НОД}(a,b)}$.

```
int lcm(int a, int b) {
    return (1LL * a * b) / gcd(a, b);
}
```

1.3 Проверка числа на простоту

Для проверки простоты n достаточно проверить его делимость на все целые числа от 2 до $\lfloor\sqrt{n}\rfloor$. Пусть n — составное число. Очевидно, оно имеет простой делитель $p \leq \sqrt{n}$. Алгоритм проверяет делимость на все числа до $\sqrt{n}$, поэтому обязательно обнаружит этот делитель. Если n простое, то у него нет делителей кроме 1 и самого себя, поэтому алгоритм вернёт истину.

```
bool check_for_prime(int n) {
    if (n == 0 || n == 1) {
        return false;
    }
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

1.4 Поиск всех делителей числа

Алгоритм находит все делители числа n. Если d делит n, то n/d также делит n. Поэтому достаточно перебрать все i от 1 до $\lfloor\sqrt{n}\rfloor$ и для каждого делителя i добавить i и n/i в список.

```
vector<int> find_divisors(int n) {
    vector<int> divisors;
    for (int i = 1; i * i <= n; ++i) {
        if (n % i == 0) {
            divisors.push_back(i);
            if (i != n / i) {
                divisors.push_back(n / i);
            }
        }
    }
    sort(divisors.begin(), divisors.end());
    return divisors;
}
```

1.5 Факторизация чисел

Алгоритм находит разложение числа n на простые множители. Последовательно проверяем делимость n на числа от 2 до $\lfloor\sqrt{n}\rfloor$. Если n делится на i, то i добавляется в разложение, и процесс продолжается для n/i. По теореме 2, если n составное, то его наименьший простой делитель не превосходит $\sqrt{n}$.

```
vector<int> factorize(int n) {
    vector<int> factors;
    for (int i = 2; i * i <= n; i++) {
        while (n % i == 0) {
            n /= i;
            factors.push_back(i);
        }
    }
    if (n > 1) { factors.push_back(n); }
    return factors;
}
```

1.6 Решето Эратосфена

Алгоритм находит все простые числа до n, последовательно вычёркивая составные числа. Для каждого числа i от 2 до $\lfloor\sqrt{n}\rfloor$, если i простое, то вычёркиваются все числа, кратные i, начиная с i^2. Любое составное число $m \leq n$ имеет наименьший простой делитель $p \leq \sqrt{m}$, который будет найден и вычеркнет m как кратное.

```
vector<bool> sieve(int n) {
    vector<bool> a(n+1, true);
    a[0] = false; a[1] = false;
    for (int i = 2; i * i <= n; ++i) {
        if (a[i] == true) {
            for (int j = i * i; j <= n; j += i) {
                a[j] = false;
            }
        }
    }
    return a;
}
```

Глава 2

Модульная арифметика

2.1 Базовые операторы

В задачах по комбинаторике часто приходится работать с очень большими числами, особенно при вычислении факториалов, сочетаний и перестановок. Прямые вычисления таких величин приводят к переполнению даже 64-битных целых типов. Решением этой проблемы является **модульная арифметика** — математический аппарат, работающий с остатками от деления на фиксированное число (модуль).

Листинг 2.1: Реализация оператора +

```
int Sum(int a, int b) {
    int s = a + b;
    if (s > mod) {
        s -= mod;
    } return s;
}
```

Первый оператор который мы реализуем - сумма. Сумма двух чисел, очевидно, может превысить mod (но будет находиться в промежутке $[0; 2 * mod - 2]$ за счет того, что каждое из слагаемых строго меньше модуля) - если это произошло, вычтем mod и получим корректный неотрицательный результат, взятый по модулю (теперь он будет лежать в $[0; mod - 2]$).

Листинг 2.2: Реализация оператора -

```
int Sub(int a, int b) {
    int s = a - b;
    if (s < 0) {
        s += mod;
    } return s;
}
```

Следующий оператор - разность. Разность двух чисел может оказаться отрицательной (но будет находиться в промежутке $[-(mod - 1); mod - 1]$, так как каждое из чисел строго меньше модуля). Если это произошло, прибавим mod и получим корректный неотрицательный результат, взятый по модулю (теперь он будет лежать в $[1; mod - 1]$).

Листинг 2.3: Реализация оператора *

```
int Mul(int a, int b) {
    return (1LL * a * b) % mod;
}
```

Третий базовый оператор — умножение. При перемножении двух чисел результат может достигать $(mod - 1)^2$, что требует использования 64-битного типа для промежуточного хранения. Умножение выполняется с приведением к long long (1LL), после чего результат берется по модулю mod (Максимальное значение произведения: $(mod - 1) \times (mod - 1) = mod^2 - 2mod + 1$. Для $mod \approx 10^9$ это около 10^{18}, что требует 64-битного типа. Операция % mod гарантирует результат в $[0; mod - 1]$).

2.2 Деление чисел в модульной арифметике через малую теорему Ферма

Четвертый базовый оператор — деление. В модульной арифметике деление заменяется умножением на обратный элемент. Для простого модуля mod обратный элемент существует для любого $x \neq 0$ и вычисляется с помощью малой теоремы Ферма как $x^{mod-2} \pmod{mod}$. Действительно, если $a^{p-1} \equiv 1 \pmod{p}$, то $a^{p-2} \equiv a^{-1} \pmod{p}$. Но тут возникает проблема: как быстро возвести число по модулю? Можно было бы использовать предпосчет, однако mod может быть слишком большим. На самом деле существует алгоритм быстрого возведения в степень:

Быстрое возведение в степень

Для эффективного вычисления $a^b \mod m$ используется алгоритм **быстрого возведения в степень** (также известный как **бинарное возведение в степень**). Этот алгоритм позволяет выполнить возведение в степень за $O(\log b)$ операций умножения по модулю, что особенно важно при работе с большими числами. Основная идея заключается в разложении степени b в двоичное представление и использовании свойств степеней:

$$a^b = a^{b_0 \cdot 2^0 + b_1 \cdot 2^1 + \cdots + b_k \cdot 2^k} = a^{b_0 \cdot 2^0} \cdot a^{b_1 \cdot 2^1} \cdot \ldots \cdot a^{b_k \cdot 2^k},$$

где b_i — биты числа b.

Таким образом, можно вычислить $a^b \mod m$, последовательно возводя a в степени 2^i и перемножая только те множители, для которых $b_i = 1$.

Листинг 2.4: binpow

```cpp
long long binpow(long long a, long long b, long long m) {
    if (b == 0) return 1;

    vector<long long> bits;
    long long x = b;
    while (x != 0) {
        bits.push_back(x % 2);
        x /= 2;
    }

    long long k = bits.size();
    vector<long long> dp(k);

    dp[0] = a % m;
    for (long long i = 1; i < k; i++) {
        dp[i] = (dp[i - 1] * dp[i - 1]) % m;
    }

    long long result = 1;
    for (long long i = 0; i < k; i++) {
        if (bits[i] == 1) {
            result = (result * dp[i]) % m;
        }
    }

    return result;
}
```

Используя быстрое возведение в степень, реализуем операцию деления:

Листинг 2.5: Реализация оператора /

```cpp
int Div(int a, int b) {
    int inv_b = binpow(b, mod - 2, mod);
    return (1LL * a * inv_b) % mod;
}
```

Глава 3

Хеширование строк

3.1 Постановка задачи

Дана строка s длины n и m запросов вида "равны ли подстроки $S[l_1..r_1]$ и $S[l_2..r_2]$?". Требуется ответить на все запросы эффективно.

3.2 Наивное решение

Простейшее решени — сравнение подстрок символов за линейное время. Алгоритм достаточно прост в реализации, но неэффективен для множества запросов на достаточно больших строках.

Листинг 3.1: Линейная проверка для каждого запроса

```
string get(string& s, int l, int r) {
    string ans = "";
    for (int i = l; i <= r; i++) {
        ans += s[i];
    }
    return ans;
}

void Solve() {
    int n, m; cin >> n >> m;
    string s; cin >> s;
    for (int i = 0; i < m; i++) {
        int l1, r1; cin >> l1 >> r1;
        int l2, r2; cin >> l2 >> r2;
        if (r1 - l1 + 1 != r2 - l2 + 1) {
            cout << "no\n";
        }
        else {
            if (get(s, l1, r1) == get(s, l2, r2)) {
                cout << "yes\n";
            }
            else {
                cout << "no\n";
            }
        }
    }
}
```

- **Временная сложность**: $O(n)$ на запрос
- **Пространственная сложность**: $O(n)$

3.3 Оптимальное решение: хеширование строк

Сравнение строк работает очень долго и хотелось бы вместо линейной проверки на равество сопоставить каждой подстроке какое-то целое число, которое называется хешом. Тогда для ответа на запрос можно просто посмотреть совпадают ли значения соответствующих хешей (int-значения сравниваются за $O(1)$).

Определим хеш сроки $S = s_0 s_1 \dots s_{n-1}$, как полином:

$$\text{hash}(S) = s_0 \cdot p^0 + s_1 \cdot p^1 + \dots + s_{n-2} \cdot p^{n-2} + s_{n-1} \cdot p^{n-1}$$

где p - произвольное число, большее размера алфавита. Теперь нужно быстро получать хеш любой подстроки. Воспользуемся уже известным нам методом - префиксные суммы: prefhash[i] = hash(S[0 . . . i]).

Стоит отметить, что значения хешей будут получаться очень большими, поэтому хранить их стоит по какому-то простому модулю, например, $1e9 + 7$.

Листинг 3.2: Precalc prefhash

```
int mod = 1e9 + 7;
int p = 31;

vector<int> P;
vector<int> prefhash;

void Precalc(string& s, int n) {
    P.resize(n + 1);
    P[0] = 1;
    for (int i = 1; i <= n; i++) {
        P[i] = (P[i - 1] * p) % mod;
    }

    prefhash.resize(n);
    prefhash[0] = (s[0] - 'a' + 1) % mod;
    for (int i = 1; i < (int)s.size(); i++) {
        prefhash[i] = (prefhash[i - 1] + ((s[i] - 'a' + 1) * P[i]) % mod) % mod;
    }
}
```

Полиномиальный хеш для подстроки $S[l...r]$ определим как:

- если $l = 0$: polihash(S[l..r]) $= prefhash[r] * P[n]$, взятое по модулю mod.
- если $l \neq 0$: polihash(S[l..r]) $= (prefhash[r] - prefhash[l-1] + mod) * P[n-l]$, взятое по модулю mod.

Листинг 3.3: Take hash

```
int Take_hash(int l, int r, int n) {
    if (l == 0) {
        return prefhash[r] * P[n] % mod;
    }
    else {
        return (prefhash[r] - prefhash[l - 1] + mod) * P[n - l] % mod;
    }
}
```

Утверждается, что с большой вероятностью при совпадении хешей равны сравниваемые строки. И в обратную сторону: если хеши отличаются, то соответствующие строки скорее всего различны. Чтобы в этом убедиться рассмотрим: polihash($S[l..r]$) = (prefhash[r] − prefhash[$l-1$]) $\cdot P^{n-l}$, где:

- prefhash[r] $= s_0 \cdot p^0 + s_1 \cdot p^1 + \dots + s_r \cdot p^r$
- prefhash[$l-1$] $= s_0 \cdot p^0 + s_1 \cdot p^1 + \dots + s_{l-1} \cdot p^{l-1}$

Разность prefhash$[r]$ − prefhash$[l-1]$ даёт:

$$s_l \cdot p^l + s_{l+1} \cdot p^{l+1} + \cdots + s_r \cdot p^r$$

Умножение на P^{n-l} (где $P[i] = p^i$) приводит хеш к виду:

$$s_l \cdot p^n + s_{l+1} \cdot p^{n+1} + \cdots + s_r \cdot p^{n+r-l}$$

Это эквивалентно полиномиальному хешу подстроки $S[l..r]$, домноженному на p^n, что не влияет на сравнение, так как p и mod взаимно просты.

Вероятность коллизии

Для различных строк s_1, s_2 длины n вероятность коллизии:

$$P(polihash(s_1) = polihash(s_2)) \leq \frac{n}{m}$$

Рассмотрим разность хешей как многочлен:

$$D(p) = polihash(s_1) - polihash(s_2) \equiv 0 \mod m$$

Многочлен степени $n-1$ может иметь не более $n-1$ корней. При случайном выборе простого mod и p, большего размера алфавита:

$$P(D(p) \equiv 0) \leq \frac{n-1}{m} \approx \frac{n}{m}$$

Полиномиальное хеширование было впервые предложено в 1987 году. С тех пор метод широко используется в спортивном программировании благодаря своей простоте и эффективности. Двойное хеширование стало популярным его надежностью против коллизий. Предлагается хешировать строку S дважды (по разным модулям и основаниям), после чего сравнивать оба хеша поочередно.

3.4 Задачи

3.4.1 Задача 1: Быстрое сравнение строк

Условие: Даны n строк одинаковой длины. Требуется обработать q запросов на сравнение двух строк по их номерам.

Решение: Вычисляем полиномиальный хеш для каждой строки один раз при вводе. При запросе сравниваем только хеши, что выполняется за константное время.

3.4.2 Задача 2: Поиск всех вхождений

Условие: Даны строка T и строка P. Найти все позиции вхождения P в T.

Решение: Предподсчитываем хеши всех префиксов T. Вычисляем хеш строки P. Затем последовательно вычисляем хеши всех подстрок T длины $|P|$ и сравниваем с хешом P.

3.4.3 Задача 3: Строчки

Условие: Даны две строки s и t одинаковой длины. Проверить, можно ли получить t из s циклическим сдвигом.

Решение: Строим удвоенную строку $s + s$. Ищем вхождение строки t как подстроки в этой удвоенной строке с помощью хеширования. Если такое вхождение найдено на позиции i, то минимальный сдвиг равен $(n - i) \mod n$.

3.4.4 Задача 4: Циклическая строка

Условие: Дана строка, полученная из бесконечного повторения некоторой строки S. Найти минимально возможную длину S.

Решение: Ищем минимальный период строки. Для проверки, является ли длина len периодом, используем хеширование: сравниваем хеш префикса длины $n - len$ с хешом суффикса, начиная с позиции len.

3.4.5 Задача 5: Максимальный подпалиндром

Условие: Дана строка s. Найти длину её наибольшей подстроки-палиндрома.

Решение: Для каждой возможной центральной позиции палиндрома используем бинарный поиск по радиусу. Для проверки, является ли подстрока палиндромом, сравниваем прямой хеш подстроки с обратным хешом той же подстроки с помощью заранее вычисленных хешей прямой и обратной строк.