Алгоритмы и структуры данных. Рекурсия и сортирвки

densine

8 ноября 2025 г.

Содержание

1. Рекурсия

Суть рекурсии – вызов функцией самой себя. Ей мы можем манипулировать зная, что функции выполняются в порядке стека. Исходя из этого и пишутся рекурсии. Частый пример – факториалы или числа Фибоначчи. Этим и займемся.

1.1. Числа Фибоначчи

Суть – каждое следующее число равно сумме предыдущих, при условии $F_0 = 1$ и $F_1 = 1$.

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Но сложность такого алгоритма – $O(2^n)$. Она обуславливается тем, что на каждой итерации вызывается дважды функция: для $n-1$ и $n-2$. Решением будет запоминать значения функции для раличных n.

```
#include <map>
map<int, int> memo = {{0, 0}, {1, 1}};
...
int fib(int n) {
    if (memo.find(n) != memo.end()) return memo[n];
    return memo[n] = f(n - 1) + f(n - 2);
}
```

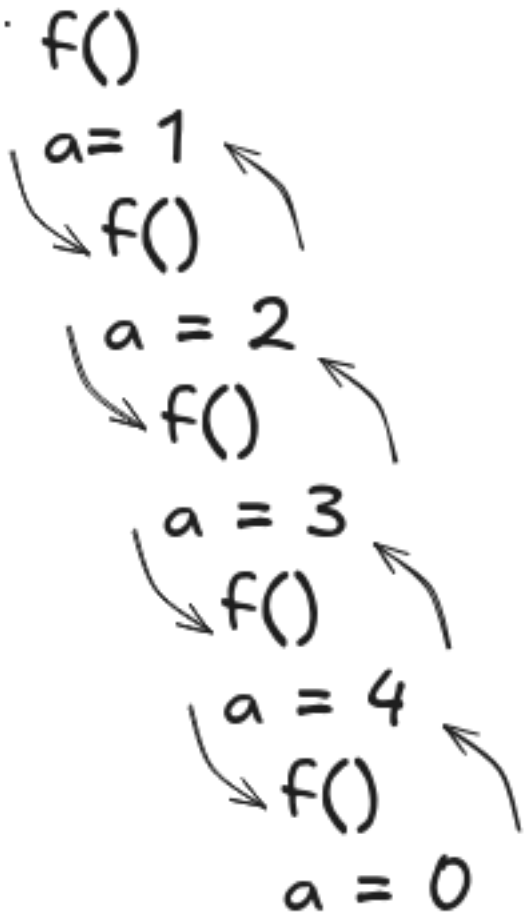
Теперь сложность алгоритма – $O(n*log(n))$. Если заменить map на vector сложность станет линейной.

1.2.

Задан массив $[a_1, a_2 \dots a_n, 0]$ (с неизвестным количеством элементов, оканчивающихся нулем). Как вывести его в обратном порядке без использования массивов? С помощью рекурсии.

```
void f() {
    int a;
    cin >> a;
    if (a == 0) return;
    f();
    cout << a << ' ';
}
```

Суть в том, что перед выводом вызывается еще $n-i$ функций, и в момент, когда $a = 0$, рекурсия останавливается, стек освождается.



1.3. Перебор двоичных строк длины N

Напишем функцию, которая перебирает все возможные суффиксы для строки длиной n.

```
void gen(string &s, int n) {
    if (s.size() == n) {
        cout << s << '\n';
        return;
    }

    s += '0';
    gen(s, n);
    s.back() = '1';
    gen(s, n);
    s.pop_back();
}
```

В общем виде эта программа перебирает все возможные суффиксы для любого (и пустого в том числе) префикса. Такой алгоритм работает за $O(2^n)$.

1.4. Перебор строк длины N с заданным k единиц

```
void gen(string &s, int n, int k, int cnt) {
```

```
    if (cnt > k || cnt + n - s.size() < k) return;

    if (s.size() == n) {
        cout << s << '\n';
        return;
    }

    s += '0';
    gen(s, n, k, cnt);
    s.back() = '1';
    gen(s, n, k, cnt + 1);
    s.pop_back();
}
```

1.5. Перестановки элементов массива

```
void gen(vector<int>& p, int n) {
    if (p.size() == n) {
        cout << p << '\n';
        return;
    }

    for (int i = 1; i <= n; i++) {
        p.push_back(i);
        if (p.find(i) == p.end()) {
            gen(p, n);
        }
        p.pop_back();
    }
}
```

Сложность такого алгоритма – $O(n! * n)$. Чтобы снизить ее до $O(n!)$, храним массив из булевых величин, где храним записи о уже использованных элементах.

1.6. Разложение на слагаемые

Дано число N. Найти все его разложения на любые слагаемые.

```
void gen(int N, vector<int>& s) {
    if (N == 0){
        // вывод s
        return;
    }

    for (int i = 1; i <= n; i++) {
        s.push_back(i);
        gen(N - i, s);
        s.pop_back();
    }
}
```

Если нам неважен порядок перебора, то можно перебирать в отсортированном порядке. Для этого достаточно вместо 1 начальным элементом перебора сделать $s.back()$.

2. Сортировки

2.1. Сортировка слиянием (Merge Sort)

Суть – в разбиении массива на две части рекурсивно, пока не останется 2 элемента. Затем просто соединяем половинки и получаем отсортированный массив.

```
vector<int> merge_sort(const vector<int> &a) {
        if (a.size() == 1) return a;
        int m = a.size() / 2;
        vector<int> l, r;
        for (ll i = 0; i < m; i++) {
                l.push_back(a[i]);
        }
        for (ll i = m; i < a.size(); i++) {
                r.push_back(a[i]);
        }
        auto sl = merge_sort(l);
        auto rl = merge_sort(r);
        return merge(ls, lr);
}

vector<int> merge(const vector<int> &a, const vector<int> &b) {
        int cur_a = 0, cur_b = 0;
        vector <int> res;
        while (cur_a < a.size() && cur_b < b.size()) {
            // чтобы была стабильная сортировка:
            // a[cur_a] <= b[cur_b]
            res.push_back(a[cur_a] < b[cur_b] ?
                      a[cur_a++] : b[cur_b++]);
        }

        while (cur_a < a.size()) res.push_back(a[cur_a++]);
        while (cur_b < b.size()) res.push_back(b[cur_b++]);

        return res;
}
```

Каждый раз происходит деление массива на, значит таких делений будет $log(n)$ штук. Затем, на каждом уровне совершается n операций, следовательно сложность такого алгоритма – $O(n * log(n))$.

2.2. Стабильная и нестабильная сортировка

- Стабильной называют сортировку, если при равенстве элементов компаратор их местами не меняет.

- Нестабильной называют сортировку, если при равенстве элементов их порядок меняется.

2.3. Сортировка подсчета (Bucket sort)

Суть этой сортировки – в создании заданного количество ячеек, каждой из которых присваивается какое-то конкретное число. Для сортировки считаем сколько всего в массиве элементов каждого типа.

```
int MAXN = 1e6 + 1;
vector<int> hist(MAXN);
for (int i : a) hist[i]++;
vector<int> res;
for (int i = 0; i < MAXN; ++i) {
        for (int j = 0; j < hist[i]; ++j) {
                res.push_back(i);
        }
}
```

Работает за $O(n)$, но есть подвох. Очевидно, что она работает только для целых чисел. При этом среди них сразу должен быть известен максимум.