

Содержание

1	Strategy.txt	1
2	algo/flows/globalcut.cpp	2
3	algo/flows/hungary.cpp	2
4	algo/math/fft_recursive.cpp	3
5	algo/strings/automaton.cpp	4
6	algo/strings/suffix_array.cpp	4
7	algo/strings/ukkonen.cpp	5
8	algo/structures/ordered_set.cpp	6
9	algo/structures/splay.cpp	6
10	algo/structures/treap.cpp	7

1 Strategy.txt

- Проверить руками сэмплы
- Подумать как дебагать после написания
- Выписать сложные формулы и все +-1
- Проверить имена файлов
- Прогнать сэмплы
- Переполнения int, переполнения long long
- Выход за границу массива: _GLIBCXX_DEBUG
- Переполнения по модулю: в
 - ↪ псевдо-онлайн-генераторе, в функциях-обертках
- Проверить мультитест на разных тестах
- Прогнать минимальный по каждому параметру тест
- Прогнать псевдо-максимальный тест(немного чисел,
 - ↪ но очень большие или очень маленькие)
- Представить что не зайдет и заранее написать
 - ↪ assert'ы, прогнать слегка модифицированные тесты
- cout.precision: в том числе в интерактивных
 - ↪ задачах
- Удалить debug-output, отсечения для тестов,
 - ↪ вернуть оригинальный main, удалить
 - ↪ _GLIBCXX_DEBUG
- Вердикт может врать
- Если много тестов(>3), дописать в конец каждого
 - ↪ теста ответ, чтобы не забыть
- (WA) Потестить не только ответ, но и содержимое
 - ↪ значимых массивов, переменных
- (WA) Изменить тест так, чтобы ответ не менялся:
 - ↪ поменять координаты местами, сжать/растянуть
 - ↪ координаты, поменять ROOT дерева
- (WA) Подвигать размер блока в корневой или
 - ↪ битсете
- (WA) Поставить assert'ы, возможно написать чекер
 - ↪ с assert'ом
- (WA) Проверить, что программа не печатает
 - ↪ что-либо неожиданное, что должно попадать под
 - ↪ PE: inf - 2, не лекс. мин. решение, одинаковые
 - ↪ числа вместо разных, неправильное количество
 - ↪ чисел, пустой ответ, перечитать output format
- (TL) cin -> scanf -> getchar
- (TL) Упихать в кэш большие массивы, поменять
 - ↪ местами for'ы или измерения массива
- (RE) Проверить формулы на деление на 0, выход за
 - ↪ область определения(sqrt(-eps), acos(1 + eps))

2 algo/flows/globalcut.cpp

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  #define forn(i,n) for (int i = 0; i < int(n); ++i)
4  const int inf = 1e9 + 1e5;
5
6  const int maxn = 505;
7  namespace StoerWagner {
8      int g[maxn][maxn];
9      int dist[maxn];
10     bool used[maxn];
11     int n;
12
13     void addEdge(int u, int v, int c) {
14         g[u][v] += c;
15         g[v][u] += c;
16     }
17
18     int run() {
19         vector<int> vertices;
20         forn (i, n)
21             vertices.push_back(i);
22         int mincut = inf;
23         while (vertices.size() > 1) {
24             int u = vertices[0];
25             for (auto v: vertices) {
26                 used[v] = false;
27                 dist[v] = g[u][v];
28             }
29             used[u] = true;
30             forn (ii, vertices.size() - 2) {
31                 for (auto v: vertices)
32                     if (!used[v])
33                         if (used[u] || dist[v] > dist[u])
34                             u = v;
35             }
36             used[u] = true;
37             for (auto v: vertices)
38                 if (!used[v])
39                     dist[v] += g[u][v];
40         }
41         int t = -1;
42         for (auto v: vertices)
43             if (!used[v])
44                 t = v;
45         assert(t != -1);
46         mincut = min(mincut, dist[t]);
47         vertices.erase(find(vertices.begin(), vertices.end(), t));
48         for (auto v: vertices)
49             addEdge(u, v, g[v][t]);
50     }
51     return mincut;
52 };
53
54 int main() {
55     StoerWagner::n = 4;
56     StoerWagner::addEdge(0, 1, 5);
57     StoerWagner::addEdge(2, 3, 5);
58     StoerWagner::addEdge(1, 2, 4);
59     cerr << StoerWagner::run() << '\n';
60 }

```

3 algo/flows/hungary.cpp

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  #define forn(i,n) for (int i = 0; i < int(n); ++i)
4  const int inf = 1e9 + 1e5;
5
6  // left half is the smaller one
7  namespace Hungary {
8      const int maxn = 505;
9      int a[maxn][maxn];
10     int p[2][maxn];
11     int match[maxn];
12     bool used[maxn];
13     int from[maxn];
14     int mind[maxn];
15     int n, m;
16
17     int hungary(int v) {
18         used[v] = true;
19         int u = match[v];
20         int best = -1;
21         forn (i, m + 1) {
22             if (used[i])
23                 continue;
24             int nw = a[u][i] - p[0][u] - p[1][i];
25             if (nw <= mind[i]) {
26                 mind[i] = nw;
27                 from[i] = v;
28             }
29             if (best == -1 || mind[best] > mind[i])
30                 best = i;
31         }
32         v = best;
33         int delta = mind[best];
34         forn (i, m + 1) {
35             if (used[i]) {
36                 p[1][i] -= delta;
37                 p[0][match[i]] += delta;
38             } else
39                 mind[i] -= delta;
40         }
41         if (match[v] == -1)
42             return v;
43         return hungary(v);
44     }
45
46     void check() {
47         int edges = 0, res = 0;
48         forn (i, m)
49             if (match[i] != -1) {
50                 ++edges;
51                 assert(p[0][match[i]] + p[1][i] == a[match[i]][i]);
52                 res += a[match[i]][i];
53             } else
54                 assert(p[1][i] == 0);
55         assert(res == -p[1][m]);
56         forn (i, n) forn (j, m)
57             assert(p[0][i] + p[1][j] <= a[i][j]);
58     }
59
60     int run() {
61         forn (i, n)
62             p[0][i] = 0;
63         forn (i, m + 1) {
64             p[1][i] = 0;
65             match[i] = -1;
66         }
67         forn (i, n) {
68             match[m] = i;
69             fill(used, used + m + 1, false);
70             fill(mind, mind + m + 1, inf);
71             fill(from, from + m + 1, -1);
72             int v = hungary(m);
73             while (v != m) {
74                 int w = from[v];
75                 match[v] = match[w];
76                 v = w;
77             }
78         }
79         check();
80         return -p[1][m];
81     }
82 };
83
84 int main() {
85     int n = 300, m = 500;
86     Hungary::n = n, Hungary::m = m;
87     forn (i, n) forn (j, m) Hungary::a[i][j] = rand() % 200001 - 100000;
88     cerr << Hungary::run() << "\n";
89 }

```

4 algo/math/fft_recursive.cpp

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  #define forn(i, n) for (int i = 0; i < (int)(n); ++i)
4  typedef long long i64;
5
6  typedef double ld;
7
8  struct base {
9      ld re, im;
10     base(){}
11     base(ld re) : re(re), im(0) {}
12     base(ld re, ld im) : re(re), im(im) {}
13
14     base operator+(const base& o) const { return {re+o.re, im+o.im}; }
15     base operator-(const base& o) const { return {re-o.re, im-o.im}; }
16     base operator*(const base& o) const {
17         return {
18             re*o.re - im*o.im,
19             re*o.im + im*o.re
20         };
21     }
22 };
23
24 const int sz = 1<<20;
25
26 int revb[sz];
27 vector<base> ang[21];
28
29 void init(int n) {
30     int lg = 0;
31     while ((1<<lg) != n) {
32         ++lg;
33     }
34     forn(i, n) {
35         revb[i] = (revb[i>>1]>>1)^((i&1)<<(lg-1));
36     }
37
38     ld e = M_PI * 2 / n;
39     ang[lg].resize(n);
40     forn(i, n) {
41         ang[lg][i] = { cos(e * i), sin(e * i) };
42     }
43
44     for (int k = lg - 1; k >= 0; --k) {
45         ang[k].resize(1 << k);
46         forn(i, 1<<k) {
47             ang[k][i] = ang[k+1][i*2];
48         }
49     }
50 }
51
52 void fft_rec(base *a, int lg, bool rev) {
53     if (lg == 0) {
54         return;
55     }
56     int len = 1 << (lg - 1);
57     fft_rec(a, lg-1, rev);
58     fft_rec(a+len, lg-1, rev);
59
60     forn(i, len) {
61         base w = ang[lg][i];
62         if (rev) w.im *= -1;
63         base u = a[i];
64         base v = a[i+len] * w;
65         a[i] = u + v;
66         a[i+len] = u - v;
67     }
68 }
69
70 void fft(base *a, int n, bool rev) {
71     forn(i, n) {
72         int j = revb[i];
73         if (i < j) swap(a[i], a[j]);
74     }
75     int lg = 0;
76     while ((1<<lg) != n) {
77         ++lg;
78     }
79     fft_rec(a, lg, rev);
80     if (rev) forn(i, n) {
81         a[i] = a[i] * (1.0 / n);
82     }
83 }
84
85 const int maxn = 1050000;
86
87 int n;
88 base a[maxn];
89 base b[maxn];

```

```

90
91 void test() {
92     int n = 1<<19;
93     mt19937 rr(55);
94     forn(i, n) a[i] = rr() % 10000;
95     forn(j, n) b[j] = rr() % 10000;
96
97     int N = 1;
98     while (N < 2*n) N *= 2;
99
100    clock_t start = clock();
101    init(N);
102    cerr << "init time: " << (clock()-start) / 1000 << " ms" << endl;
103    fft(a, N, 0);
104    forn(i, N) a[i] = a[i] * b[i];
105    fft(a, N, 1);
106    clock_t end = clock();
107
108    ld err = 0;
109    forn(i, N) {
110        err = max(err, (ld)fabs1(a[i].im));
111        err = max(err, (ld)fabs1(a[i].re - (i64(a[i].re + 0.5))));
112    }
113
114    cerr << "Time: " << (end - start) / 1000 << " ms, err = " << err <<
115    endl;
116 }
117
118 int main() {
119     test();
120 }

```

5 algo/strings/automaton.cpp

```

1 //real 4m27.689s
2 #include <bits/stdc++.h>
3 using namespace std;
4 #define forn(i, n) for (int i = 0; i < (int)(n); ++i)
5
6 const int maxn = 100500;
7
8 int t[maxn][26], lnk[maxn], len[maxn];
9 int sz;
10 int last;
11
12 void init() {
13     sz = 3;
14     last = 1;
15     forn(i, 26) t[2][i] = 1;
16     len[2] = -1;
17     lnk[1] = 2;
18 }
19
20 void addchar(int c) {
21     int nlast = sz++;
22     len[nlast] = len[last] + 1;
23     int p = last;
24     for (; !t[p][c]; p = lnk[p]) {
25         t[p][c] = nlast;
26     }
27     int q = t[p][c];
28     if (len[p] + 1 == len[q]) {
29         lnk[nlast] = q;
30     } else {
31         int clone = sz++;
32         len[clone] = len[p] + 1;
33         lnk[clone] = lnk[q];
34         lnk[q] = lnk[nlast] = clone;
35         forn(i, 26) t[clone][i] = t[q][i];
36         for (; t[p][c] == q; p = lnk[p]) {
37             t[p][c] = clone;
38         }
39     }
40     last = nlast;
41 }
42
43 bool check(const string& s) {
44     int v = 1;
45     for (int c: s) {
46         c -= 'a';
47         if (!t[v][c]) return false;
48         v = t[v][c];
49     }
50     return true;
51 }
52
53 int main() {
54     string s;
55     cin >> s;
56     init();
57     for (int i: s) {
58         addchar(i - 'a');
59     }
60     forn(i, s.length()) {
61         assert(check(s.substr(i)));
62     }
63     cout << sz << endl;
64     return 0;
65 }

```

6 algo/strings/suffix_array.cpp

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 #define forn(i, n) for (int i = 0; i < (int)(n); ++i)
4
5 const int maxn = 100500;
6
7 string s;
8 int n;
9 int sa[maxn], new_sa[maxn], cls[maxn], new_cls[maxn], cnt[maxn],
10 ↪ lcp[maxn];
11 int n_cls;
12
13 void build() {
14     n_cls = 256;
15     forn(i, n) {
16         sa[i] = i;
17         cls[i] = s[i];
18     }
19     for (int d = 0; d < n; d = d ? d*2 : 1) {
20         forn(i, n) new_sa[i] = (sa[i] - d + n) % n;
21         forn(i, n_cls) cnt[i] = 0;
22         forn(i, n) ++cnt[cls[i]];
23         forn(i, n_cls) cnt[i+1] += cnt[i];
24         for (int i = n-1; i >= 0; --i) sa[--cnt[cls[new_sa[i]]]] =
25 ↪ new_sa[i];
26
27         n_cls = 0;
28         forn(i, n) {
29             if (i && (cls[sa[i]] != cls[sa[i-1]] ||
30 ↪ cls[(sa[i] + d) % n] != cls[(sa[i-1] + d) % n]))
31                 ++n_cls;
32             new_cls[sa[i]] = n_cls;
33         }
34         ++n_cls;
35         forn(i, n) cls[i] = new_cls[i];
36     }
37
38     // cls is also a reverse permutation of sa if a string is not cyclic
39     // (i.e. a position of i-th lexicographical suffix)
40     int val = 0;
41     forn(i, n) {
42         if (val) --val;
43         if (cls[i] == n-1) continue;
44         int j = sa[cls[i] + 1];
45         while (i + val != n && j + val != n && s[i+val] == s[j+val])
46 ↪ ++val;
47         lcp[cls[i]] = val;
48     }
49 }
50
51 int main() {
52     cin >> s;
53     s += '$';
54     n = s.length();
55     build();
56     forn(i, n) {
57         cout << s.substr(sa[i]) << endl;
58         cout << lcp[i] << endl;
59     }
60 }

```

7 algo/strings/ukkonen.cpp

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  #define sz(x) ((int) (x).size())
4  #define forn(i,n) for (int i = 0; i < int(n); ++i)
5  const int inf = int(1e9) + int(1e5);
6
7  string s;
8  const int alpha = 26;
9
10 namespace SuffixTree {
11     struct Node {
12         Node *to[alpha];
13         Node *lnk, *par;
14         int l, r;
15
16         Node(int l, int r): l(l), r(r) {
17             memset(to, 0, sizeof(to));
18             lnk = par = 0;
19         }
20     };
21
22     Node *root, *blank, *cur;
23     int pos;
24
25     void init() {
26         root = new Node(0, 0);
27         blank = new Node(0, 0);
28         forn (i, alpha)
29             blank->to[i] = root;
30         root->lnk = root->par = blank->lnk = blank->par = blank;
31         cur = root;
32         pos = 0;
33     }
34
35     int at(int id) {
36         return s[id];
37     }
38
39     void goDown(int l, int r) {
40         if (l >= r)
41             return;
42         if (pos == cur->r) {
43             int c = at(l);
44             assert(cur->to[c]);
45             cur = cur->to[c];
46             pos = min(cur->r, cur->l + 1);
47             ++l;
48         } else {
49             int delta = min(r - l, cur->r - pos);
50             l += delta;
51             pos += delta;
52         }
53         goDown(l, r);
54     }
55
56     void goUp() {
57         if (pos == cur->r && cur->lnk) {
58             cur = cur->lnk;
59             pos = cur->r;
60             return;
61         }
62         int l = cur->l, r = pos;
63         cur = cur->par->lnk;
64         pos = cur->r;
65         goDown(l, r);
66     }
67
68     void setParent(Node *a, Node *b) {
69         assert(a);
70         a->par = b;
71         if (b)
72             b->to[at(a->l)] = a;
73     }
74
75     void addLeaf(int id) {
76         Node *x = new Node(id, inf);
77         setParent(x, cur);
78     }
79
80     void splitNode() {
81         assert(pos != cur->r);
82         Node *mid = new Node(cur->l, pos);
83         setParent(mid, cur->par);
84         cur->l = pos;
85         setParent(cur, mid);
86         cur = mid;
87     }
88
89     bool canGo(int c) {
90         if (pos == cur->r)
91             return cur->to[c];
92         return at(pos) == c;
93     }
94
95     void fixLink(Node *&bad, Node *newBad) {
96         if (bad)
97             bad->lnk = cur;
98             bad = newBad;
99     }
100
101     void addCharOnPos(int id) {
102         Node *bad = 0;
103         while (!canGo(at(id))) {
104             if (cur->r != pos) {
105                 splitNode();
106                 fixLink(bad, cur);
107                 bad = cur;
108             } else {
109                 fixLink(bad, 0);
110             }
111             addLeaf(id);
112             goUp();
113         }
114         fixLink(bad, 0);
115         goDown(id, id + 1);
116     }
117
118     int cnt(Node *u, int ml) {
119         if (!u)
120             return 0;
121         int res = min(ml, u->r) - u->l;
122         forn (i, alpha)
123             res += cnt(u->to[i], ml);
124         return res;
125     }
126
127     void build(int l) {
128         init();
129         forn (i, l)
130             addCharOnPos(i);
131     }
132 };
133
134 int main() {
135     cin >> s;
136     SuffixTree::build(s.size());
137 }

```

8 algo/structures/ordered_set.cpp

```

1  #include <ext/pb_ds/assoc_container.hpp>
2  #include <ext/pb_ds/tree_policy.hpp>
3
4  typedef __gnu_pbds::tree<int, __gnu_pbds::null_type, std::less<int>,
5      __gnu_pbds::rb_tree_tag,
6      __gnu_pbds::tree_order_statistics_node_update> oset;
7
8  #include <iostream>
9
10 int main() {
11     oset X;
12     X.insert(1);
13     X.insert(2);
14     X.insert(4);
15     X.insert(8);
16     X.insert(16);
17
18     std::cout << *X.find_by_order(1) << std::endl; // 2
19     std::cout << *X.find_by_order(2) << std::endl; // 4
20     std::cout << *X.find_by_order(4) << std::endl; // 16
21     std::cout << *X.find_by_order(6) << std::endl; // 16
22     std::cout << *X.find_by_order(8) << std::endl; // 32
23     std::cout << *X.find_by_order(10) << std::endl; // 64
24     std::cout << *X.find_by_order(12) << std::endl; // 128
25     std::cout << *X.find_by_order(14) << std::endl; // 256
26     std::cout << *X.find_by_order(16) << std::endl; // 512
27 }

```

9 algo/structures/splay.cpp

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  #define forn(i, n) for (int i = 0; i < (int)(n); ++i)
4
5  const int maxn = 100500;
6
7  struct node;
8  void updson(node* p, node* v, node* was);
9
10 struct node {
11     int val;
12     node *l, *r, *p;
13     node() {}
14     node(int val) : val(val), l(r=p=NULL) {}
15
16     bool isRoot() const { return !p; }
17     bool isRight() const { return p && p->r == this; }
18     bool isLeft() const { return p && p->l == this; }
19     void setLeft(node* t) {
20         if (t) t->p = this;
21         l = t;
22     }
23     void setRight(node* t) {
24         if (t) t->p = this;
25         r = t;
26     }
27 };
28
29 void updson(node *p, node *v, node *was) {
30     if (p) {
31         if (p->l == was) p->l = v;
32         else p->r = v;
33     }
34     if (v) v->p = p;
35 }
36
37 void rightRotate(node *v) {
38     assert(v && v->l);
39     node *u = v->l;
40     node *p = v->p;
41     v->setLeft(u->r);
42     u->setRight(v);
43     updson(p, u, v);
44 }
45
46 void leftRotate(node *v) {
47     assert(v && v->r);
48     node *u = v->r;
49     node *p = v->p;
50     v->setRight(u->l);
51     u->setLeft(v);
52     updson(p, u, v);
53 }
54
55 void splay(node *v) {
56     while (v->p) {
57         if (!v->p->p) {
58             if (v->isLeft()) rightRotate(v->p);
59             else leftRotate(v->p);
60         } else if (v->isLeft() && v->p->isLeft()) {
61             rightRotate(v->p->p);
62             rightRotate(v->p);
63         } else if (v->isRight() && v->p->isRight()) {
64             leftRotate(v->p->p);
65             leftRotate(v->p);
66         } else if (v->isLeft() && v->p->isRight()) {
67             rightRotate(v->p);
68             leftRotate(v->p);
69         } else {
70             leftRotate(v->p);
71             rightRotate(v->p);
72         }
73     }
74     v->p = NULL;
75 }
76
77 node *insert(node *t, node *n) {
78     if (!t) return n;
79     int x = n->val;
80     while (true) {
81         if (x < t->val) {
82             if (t->l) {
83                 t = t->l;
84             } else {
85                 t->setLeft(n);
86                 t = t->l;
87                 break;
88             }
89         } else {
90             if (t->r) {
91                 t = t->r;
92             } else {
93                 t->setRight(n);
94                 t = t->r;
95                 break;
96             }
97         }
98     }
99 }

```

10 algo/structures/treap.cpp

```

90         if (t->r) {
91             t = t->r;
92         } else {
93             t->setRight(n);
94             t = t->r;
95             break;
96         }
97     }
98 }
99 splay(t);
100 return t;
101 }
102
103 node *insert(node *t, int x) {
104     return insert(t, new node(x));
105 }
106
107 int main() {
108     node *t = NULL;
109     forn(i, 1000000) {
110         int x = rand();
111         t = insert(t, x);
112     }
113     return 0;
114 }

```

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  #define forn(i, n) for (int i = 0; i < (int)(n); ++i)
4  const int maxn = 100500;
5
6  struct node {
7      int x, y;
8      node *l, *r;
9      node(int x) : x(x), y(rand()), l(r=NULL) {}
10 };
11
12 void split(node *t, node *&l, node *&r, int x) {
13     if (!t) return (void)(l=r=NULL);
14     if (x <= t->x) {
15         split(t->l, l, t->l, x), r = t;
16     } else {
17         split(t->r, t->r, r, x), l = t;
18     }
19 }
20
21 node *merge(node *l, node *r) {
22     if (!l) return r;
23     if (!r) return l;
24     if (l->y > r->y) {
25         l->r = merge(l->r, r);
26         return l;
27     } else {
28         r->l = merge(l, r->l);
29         return r;
30     }
31 }
32
33 node *insert(node *t, node *n) {
34     node *l, *r;
35     split(t, l, r, n->x);
36     return merge(l, merge(n, r));
37 }
38
39 node *insert(node *t, int x) {
40     return insert(t, new node(x));
41 }
42
43 node *fast_insert(node *t, node *n) {
44     if (!t) return n;
45     node *root = t;
46     while (true) {
47         if (n->x < t->x) {
48             if (!t->l || t->l->y < n->y) {
49                 split(t->l, n->l, n->r, n->x), t->l = n;
50                 break;
51             } else {
52                 t = t->l;
53             }
54         } else {
55             if (!t->r || t->r->y < n->y) {
56                 split(t->r, n->l, n->r, n->x), t->r = n;
57                 break;
58             } else {
59                 t = t->r;
60             }
61         }
62     }
63     return root;
64 }
65
66 node *fast_insert(node *t, int x) {
67     return fast_insert(t, new node(x));
68 }
69
70 int main() {
71     node *t = NULL;
72     forn(i, 1000000) {
73         int x = rand();
74         t = fast_insert(t, x);
75     }
76 }

```