

# University of Tartu ICPC Team Notebook (2017-2018)

October 28, 2017

## Contents

1	gcc ordered set	1
2	Triangle centers	1
3	Min cost max flow $O(\text{flow} \cdot n^2)$	2
4	Aho Corasick $O( \text{alpha}  \sum \text{len})$	3
5	Suffix automaton $O((n + q) \log( \text{alpha} ))$	4
6	Templated multi dimensional BIT $O(\log(n)^{\text{dim}})$	5

## 1 gcc ordered set

---

```
1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4 template <typename T>
5 using ordered_set = tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
6 int main(){
7     ordered_set<int> cur;
8     cur.insert(1);
9     cur.insert(3);
10    cout << cur.order_of_key(2) << endl; // the number of elements in the set less than 2
11    cout << *cur.find_by_order(0) << endl; // the 0-th smallest number in the set(0-based)
12    cout << *cur.find_by_order(1) << endl; // the 1-th smallest number in the set(0-based)
13 }
```

---

## 2 Triangle centers

---

```
1 const double min_delta = 1e-13;
2 const double coord_max = 1e6;
3 typedef complex < double > point;
4 point A, B, C; // vertices of the triangle
5 bool collinear(){
6     double min_diff = min(abs(A - B), min(abs(A - C), abs(B - C)));
7     if(min_diff < coord_max * min_delta)
8         return true;
9     point sp = (B - A) / (C - A);
10    double ang = M_PI/2 - abs(abs(arg(sp)) - M_PI/2); //positive angle with the real line
11    return ang < min_delta;
12 }
13 point circum_center(){
14     if(collinear())
15         return point(NAN, NAN);
16     //squared lengths of sides
17     double a2, b2, c2;
18     a2 = norm(B - C);
19     b2 = norm(A - C);
20     c2 = norm(A - B);
21     //barycentric coordinates of the circumcenter
22     double c_A, c_B, c_C;
23     c_A = a2 * (b2 + c2 - a2); //sin(2 * alpha) may be used as well
24     c_B = b2 * (a2 + c2 - b2);
25     c_C = c2 * (a2 + b2 - c2);
```

```

26 double sum = c_A + c_B + c_C;
27 c_A /= sum;
28 c_B /= sum;
29 c_C /= sum;
30 // cartesian coordinates of the circumcenter
31 return c_A * A + c_B * B + c_C * C;
32 }
33 point centroid(){ //center of mass
34 return (A + B + C) / 3.0;
35 }
36 point ortho_center(){ //euler line
37 point O = circum_center();
38 return O + 3.0 * (centroid() - O);
39 };
40 point nine_point_circle_center(){ //euler line
41 point O = circum_center();
42 return O + 1.5 * (centroid() - O);
43 };
44 point in_center(){
45 if(collinear())
46 return point(NAN,NAN);
47 double a, b, c; //side lengths
48 a = abs(B - C);
49 b = abs(A - C);
50 c = abs(A - B);
51 //trilinear coordinates are (1,1,1)
52 //barycentric coordinates
53 double c_A = a, c_B = b, c_C = c;
54 double sum = c_A + c_B + c_C;
55 c_A /= sum;
56 c_B /= sum;
57 c_C /= sum;
58 // cartesian coordinates of the incenter
59 return c_A * A + c_B * B + c_C * C;
60 }

```

### 3 Min cost max flow $O(\text{flow} \cdot n^2)$

```

1 const int nmax=1055;
2 const ll inf=1e14;
3 int t, n, v;
4 ll rem_flow[nmax][nmax];
5 ll cost[nmax][nmax];
6 ll min_dist[nmax];
7 int prev_node[nmax];
8 ll node_flow[nmax];
9 bool visited[nmax];
10 ll tot_cost, tot_flow;
11 void mincmxf(){
12 tot_cost=0;
13 tot_flow=0;
14 ll sink_pot=0;
15 while(true){
16 for(int i=0; i<=v; ++i){
17 min_dist[i]=inf;
18 visited[i]=false;
19 }
20 min_dist[0]=0;
21 node_flow[0]=inf;
22 int min_node;
23 while(true){
24 int min_node=v;
25 for(int i=0; i<v; ++i){
26 if((!visited[i]) && min_dist[i]<min_dist[min_node]){
27 min_node=i;
28 }
29 }
30 if(min_node==v){
31 break;
32 }
33 visited[min_node]=true;
34 for(int i=0; i<v; ++i){
35 if((!visited[i]) && min_dist[min_node]+cost[min_node][i] < min_dist[i]){

```

```

36     min_dist[i]=min_dist[min_node]+cost[min_node][i];
37     prev_node[i]=min_node;
38     node_flow[i]=min(node_flow[min_node], rem_flow[min_node][i]);
39 }
40 }
41 }
42 if(min_dist[v-1]==inf){
43     break;
44 }
45 for(int i=0; i<v; ++i){
46     for(int j=0; j<v; ++j){
47         if(cost[i][j]!=inf){
48             cost[i][j]+=min_dist[i];
49             cost[i][j]-=min_dist[j];
50         }
51     }
52 }
53 sink_pot+=min_dist[v-1];
54 tot_flow+=node_flow[v-1];
55 tot_cost+=sink_pot*node_flow[v-1];
56 int cur=v-1;
57 while(cur!=0){
58     rem_flow[prev_node[cur]][cur]-=node_flow[v-1];
59     rem_flow[cur][prev_node[cur]]+=node_flow[v-1];
60     cost[cur][prev_node[cur]]=0;
61     if(rem_flow[prev_node[cur]][cur]==0){
62         cost[prev_node[cur]][cur]=inf;
63     }
64     cur=prev_node[cur];
65 }
66 }
67 }

```

## 4 Aho Corasick $O(|\alpha| \sum \text{len})$

```

1 const int alpha_size=26;
2 struct node{
3     node *nxt[alpha_size]; //May use other structures to move in trie
4     node *suffix;
5     node(){
6         memset(nxt, 0, alpha_size*sizeof(node *));
7     }
8     int cnt=0;
9 };
10 node *aho_corasick(vector<vector<char> > &dict){
11     node *root= new node;
12     root->suffix = 0;
13     vector<pair<vector<char> *, node *> > cur_state;
14     for(vector<char> &s : dict)
15         cur_state.emplace_back(&s, root);
16     for(int i=0; !cur_state.empty(); ++i){
17         vector<pair<vector<char> *, node *> > nxt_state;
18         for(auto &cur : cur_state){
19             node *nxt=cur.second->nxt[(*(cur.first)[i]);
20             if(nxt){
21                 cur.second=nxt;
22             }else{
23                 nxt = new node;
24                 cur.second->nxt[(*(cur.first)[i]] = nxt;
25                 node *suf = cur.second->suffix;
26                 cur.second = nxt;
27                 nxt->suffix = root; //set correct suffix link
28                 while(suf){
29                     if(suf->nxt[(*(cur.first)[i]]){
30                         nxt->suffix = suf->nxt[(*(cur.first)[i]);
31                         break;
32                     }
33                     suf=suf->suffix;
34                 }
35             }
36             if(cur.first->size() > i+1)
37                 nxt_state.push_back(cur);
38         }

```

```

39     cur_state=nxt_state;
40 }
41 return root;
42 }
43 node *walk(node *cur, char c){
44     while(true){
45         if(cur->nxt[c])
46             return cur->nxt[c];
47         if(!cur->suffix){
48             return cur;
49         }
50         cur = cur->suffix;
51     }
52 }
53 void cnt_matches(node *root, vector<char> &match_in){
54     node *cur = root;
55     for(char c : match_in){
56         cur = walk(cur, c);
57         ++cur->cnt;
58     }
59 }
60 void add_cnt(node *root){
61     vector<node *> to_visit = {root};
62     for(int i=0; i<to_visit.size(); ++i){
63         node *cur = to_visit[i];
64         for(int j=0; j<alpha_size; ++j){
65             if(cur->nxt[j]){
66                 to_visit.push_back(cur->nxt[j]);
67             }
68         }
69     }
70     for(int i=to_visit.size()-1; i>0; --i){
71         to_visit[i]->suffix->cnt += to_visit[i]->cnt;
72     }
73 }

```

## 5 Suffix automaton $O((n + q) \log(|\alpha|))$

```

1 class AutoNode {
2 private:
3     map< char, AutoNode * > nxt_char; // Map is faster than hashtable and unsorted arrays
4 public:
5     int len;
6     AutoNode *suf;
7     bool has_nxt(char c) const {
8         return nxt_char.count(c);
9     }
10    AutoNode *nxt(char c) {
11        if (!has_nxt(c))
12            return NULL;
13        return nxt_char[c];
14    }
15    void set_nxt(char c, AutoNode *node) {
16        nxt_char[c] = node;
17    }
18    AutoNode *split(int new_len, char c) {
19        AutoNode *new_n = new AutoNode;
20        new_n->nxt_char = nxt_char;
21        new_n->len = new_len;
22        new_n->suf = suf;
23        suf = new_n;
24        return new_n;
25    }
26    // Extra functions for matching and counting
27    AutoNode *lower_depth(int depth) {
28        if (suf->len >= depth)
29            return suf->lower_depth(depth);
30        return this;
31    }
32    AutoNode *walk(char c, int depth, int &match_len) {
33        match_len = min(match_len, len);
34        if (has_nxt(c)) {
35            ++match_len;

```

```

36     return nxt(c)->lower_depth(depth);
37 }
38 if (suf)
39     return suf->walk(c, depth, match_len);
40 return this;
41 }
42 int paths_to_end = 0;
43 void set_as_end() {
44     paths_to_end = 1;
45     if (suf) suf->set_as_end();
46 }
47 bool vis = false;
48 void calc_paths_to_end() {
49     if (!vis) {
50         vis = true;
51         for (auto cur : nxt_char) {
52             cur.second->calc_paths_to_end();
53             paths_to_end += cur.second->paths_to_end;
54         }
55     }
56 }
57 };
58 struct SufAutomaton {
59     AutoNode *last;
60     AutoNode *root;
61     void extend(char new_c) {
62         AutoNode *new_end = new AutoNode; // The equivalence class containing the whole new string
63         new_end->len = last->len + 1;
64         AutoNode *suf_w_nxt = last; // The whole old string class
65         while (suf_w_nxt && !suf_w_nxt->has_nxt(new_c)) { // is turned into the longest suffix which
66                                                         // can be turned into a substring of old state
67                                                         // by appending new_c
68             suf_w_nxt->set_nxt(new_c, new_end);
69             suf_w_nxt = suf_w_nxt->suf;
70         }
71         if (!suf_w_nxt) { // The new character isn't part of the old string
72             new_end->suf = root;
73         } else {
74             AutoNode *max_sbstr = suf_w_nxt->nxt(new_c); // Equivalence class containing longest
75                                                         // substring which is a suffix of the new state.
76             if (suf_w_nxt->len + 1 == max_sbstr->len) { // Check whether splitting is needed
77                 new_end->suf = max_sbstr;
78             } else {
79                 AutoNode *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1, new_c);
80                 new_end->suf = eq_sbstr;
81                 // Make suffixes of suf_w_nxt point to eq_sbstr instead of max_sbstr
82                 AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
83                 while (w_edge_to_eq_sbstr != 0 && w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
84                     w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
85                     w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
86                 }
87             }
88         }
89         last = new_end;
90     }
91     SufAutomaton(string to_suffix) {
92         root = new AutoNode;
93         root->len = 0;
94         root->suf = NULL;
95         last = root;
96         for (char c : to_suffix) extend(c);
97     }
98 };

```

## 6 Templated multi dimensional BIT $O(\log(n)^{\dim})$

```

1 // Fully overloaded any dimensional BIT, use any type for coordinates, elements, return_value.
2 // Includes coordinate compression.
3 template < typename elem_t, typename coord_t, coord_t n_inf, typename ret_t >
4 class BIT {
5     vector< coord_t > positions;
6     vector< elem_t > elems;
7     bool initiated = false;

```

```

8
9 public:
10 BIT() {
11     positions.push_back(n_inf);
12 }
13 void initiate() {
14     if (initiated) {
15         for (elem_t &c_elem : elems)
16             c_elem.initiate();
17     } else {
18         initiated = true;
19         sort(positions.begin(), positions.end());
20         positions.resize(unique(positions.begin(), positions.end()) - positions.begin());
21         elems.resize(positions.size());
22     }
23 }
24 template < typename... loc_form >
25 void update(coord_t cord, loc_form... args) {
26     if (initiated) {
27         int pos = lower_bound(positions.begin(), positions.end(), cord) - positions.begin();
28         for (; pos < positions.size(); pos += pos & -pos)
29             elems[pos].update(args...);
30     } else {
31         positions.push_back(cord);
32     }
33 }
34 template < typename... loc_form >
35 ret_t query(coord_t cord, loc_form... args) { //sum in open interval (-inf, cord)
36     ret_t res = 0;
37     int pos = (lower_bound(positions.begin(), positions.end(), cord) - positions.begin())-1;
38     for (; pos > 0; pos -= pos & -pos)
39         res += elems[pos].query(args...);
40     return res;
41 }
42 };
43 template < typename internal_type >
44 struct wrapped {
45     internal_type a = 0;
46     void update(internal_type b) {
47         a += b;
48     }
49     internal_type query() {
50         return a;
51     }
52     // Should never be called, needed for compilation
53     void initiate() {
54         cerr << 'i' << endl;
55     }
56     void update() {
57         cerr << 'u' << endl;
58     }
59 };
60 int main() {
61     // return type should be same as type inside wrapped
62     BIT< BIT< wrapped< ll >, int, INT_MIN, ll >, int, INT_MIN, ll > fenwick;
63     int dim = 2;
64     vector< tuple< int, int, ll > > to_insert;
65     to_insert.emplace_back(1, 1, 1);
66     // set up all positions that are to be used for update
67     for (int i = 0; i < dim; ++i) {
68         for (auto &cur : to_insert)
69             fenwick.update(get< 0 >(cur), get< 1 >(cur)); // May include value which won't be used
70         fenwick.initiate();
71     }
72     // actual use
73     for (auto &cur : to_insert)
74         fenwick.update(get< 0 >(cur), get< 1 >(cur), get< 2 >(cur));
75     cout << fenwick.query(2, 2)<<'\n';
76 }

```