Team Note of (교내)우승후보

overnap

Compiled on July 20, 2022

Contents

1 Data Structures For Range Query

1.1 Segment Tree w/ Lazy Propagation

Usage: `update(1, 0, n-1, l, r, v)`
Time Complexity: $\mathcal{O}(\log N)$

```
struct lazySeg {
  vector<ll> tree, lazy;
  void push(int n, int s, int e) {
        tree[n] += lazy[n] * (e - s + 1);
        if (s != e) {
            lazy[n*2] += lazy[n];
            lazy[n*2+1] += lazy[n];
        }
        lazy[n] = 0;
  }
  void update(int n, int s, int e, int l, int r, int v) {
    push(n, s, e);
    if (e < l || r < s)
      return;
    if (l <= s && e <= r) {
      lazy[n] += v;
      push(n, s, e);
    } else {
      int m = (s + e) / 2;
      update(n*2, s, m, l, r, v);
      update(n*2+1, m+1, e, l, r, v);
      tree[n] = tree[n*2] + tree[n*2+1];
    }
  }
  ll query(int n, int s, int e, int l, int r) {
    push(n, s, e);
    if (e < l || r < s)
      return 0;
    if (l <= s && e <= r)
      return tree[n];
        int m = (s + e) / 2;
        return query(n*2, s, m, l, r) + query(n*2+1, m+1, e, l, r);
  }
};
```

1.2 Sparse Table

Usage: RMQ l r: `min(lift[l][len], lift[r-(1<<len)+1][len])`
Time Complexity: $\mathcal{O}(N) - \mathcal{O}(1)$

```
int k = ceil(log2(n));
vector<vector<int>> lift(n, vector<int>(k));
for (int i=0; i<n; ++i)
    lift[i][0] = lcp[i];
for (int i=1; i<k; ++i) {
    for (int j=0; j<=n-(1<<i); ++j)
        lift[j][i] = min(lift[j][i-1], lift[j+(1<<(i-1))][i-1]);
}
vector<int> bits(n+1);
for (int i=2; i<=n; ++i) {
    bits[i] = bits[i-1];
    while (1 << bits[i] < i)
        bits[i]++;
    bits[i]--;
}
```

1.3 Merge Sort Tree

Time Complexity: $\mathcal{O}(N \log N) - \mathcal{O}(\log^2 N)$

```
struct mst {
  int n;
  vector<vector<int>> tree;
  void init(vector<int> &arr) {
    n = 1 << (int)ceil(log2(arr.size()));
    tree.resize(n*2);
    for (int i=0; i<arr.size(); ++i)
      tree[n+i].push_back(arr[i]);
    for (int i=n-1; i>0; --i) {
      tree[i].resize(tree[i*2].size() + tree[i*2+1].size());
      merge(tree[i*2].begin(), tree[i*2].end(),
        tree[i*2+1].begin(), tree[i*2+1].end(), tree[i].begin());
    }
  }
  int sum(int l, int r, int k) {
    int ret = 0;
```

```
    for (l+=n, r+=n; l<=r; l/=2, r/=2) {
      if (l%2)
        ret += upper_bound(tree[l].begin(), tree[l].end(), k) -
tree[l].begin(), l++;
      if (r%2 == 0)
        ret += upper_bound(tree[r].begin(), tree[r].end(), k) -
tree[r].begin(), r--;
    }
    return ret;
  }
};
```

1.4 Binray Search In Segment Tree

Time Complexity: $\mathcal{O}(\log N)$

```
int query(int x) {
  int acc, i;
  for (acc=0, i=1; i<n;) {
    if (acc + tree[i*2] < x) {
      acc += tree[i*2];
      i = i*2+1;
    } else
      i = i*2;
  }
  return i - n;
}
```

1.5 Persistence Segment Tree

Time Complexity: $\mathcal{O}(\log^2 N)$

```
struct pst {
  struct node {
    int count;
    array<int, 2> go;
    node() {
      count = 0;
      fill(go.begin(), go.end(), -1);
    }
  };
  vector<node> nodes;
  vector<int> roots;
  void init() {
    nodes.emplace_back();
    roots.push_back(0);
    function<void (int, int)> dfs = [&] (int x, int depth) {
      if (depth == 20)
        return;
      nodes[x].go[0] = nodes.size();
      nodes.emplace_back();
      dfs(nodes[x].go[0], depth+1);
      nodes[x].go[1] = nodes.size();
      nodes.emplace_back();
      dfs(nodes[x].go[1], depth+1);
    };
    dfs(0, 0);
  }
  void insert(int x) {
    roots.push_back(nodes.size());
    nodes.emplace_back();
    int curr = roots.back(), prev = roots[roots.size()-2];
    for (int i=0; i<20; ++i) {
      const int next = (x >> 19 - i) & 1;
      nodes[curr].count = nodes[prev].count + 1;
      nodes[curr].go[next] = nodes.size();
      nodes.emplace_back();
      nodes[curr].go[!next] = nodes[prev].go[!next];
      curr = nodes[curr].go[next];
      prev = nodes[prev].go[next];
    }
    nodes[curr].count = nodes[prev].count + 1;
  }
  void remove(int k) {
    nodes.resize(roots[roots.size()-k]);
    roots.resize(roots.size()-k);
  }
  int sum(int q, int x) {
    int curr = roots[q], ret = 0;
    for (int i=0; i<20; ++i) {
      const int next = (x >> 19 - i) & 1;
```

```
      if (next)
        ret += nodes[nodes[curr].go[!next]].count;
      curr = nodes[curr].go[next];
      if (nodes[curr].count == 0)
        break;
    }
    ret += nodes[curr].count;
    return ret;
  }
};
```

2 Graph

2.1 Min Cut Max Flow

2.2 Strongly Connected Component

Time Complexity: $\mathcal{O}(N)$

```
int idx = 0, scnt = 0;
vector<int> scc(n, -1), vis(n, -1), st;
function<int (int)> dfs = [&] (int x) {
  int ret = vis[x] = idx++;
  st.push_back(x);
  for (int next : e[x]) {
    if (vis[next] == -1)
      ret = min(ret, dfs(next));
    else if (scc[next] == -1)
      ret = min(ret, vis[next]);
  }
  if (ret == vis[x]) {
    while (!st.empty()) {
      const int t = st.back();
      st.pop_back();
      scc[t] = scnt;
      if (t == x)
        break;
    }
    scnt++;
  }
```

```
  return ret;
};
```

2.3 Biconnected Component

Time Complexity: $\mathcal{O}(N)$

```
int idx = 0;
vector<int> vis(n, -1);
vector<pii> st;
vector<vector<pii>> bcc;
vector<bool> cut(n); // articulation point
function<int (int, int)> dfs = [&] (int x, int p) {
    int ret = vis[x] = idx++;
    int child = 0;
    for (int next : e[x]) {
        if (next == p)
            continue;
        if (vis[next] < vis[x])
            st.emplace_back(x, next);
        if (vis[next] != -1)
            ret = min(ret, vis[next]);
        else {
            int res = dfs(next, x);
            ret = min(ret, res);
            child++;
            if (vis[x] <= res) {
                if (p != -1)
                    cut[x] = true;
                bcc.emplace_back();
                while (st.back() != pii{x, next}) {
                    bcc.back().push_back(st.back());
                    st.pop_back();
                }
                bcc.back().push_back(st.back());
                st.pop_back();
            } // vis[x] < res to find bridges
        }
    }
    if (p == -1 && child > 1)
        cut[x] = true;
```

```
    return ret;
};
```

2.4 Lowest Common Ancestor

Usage: Query with the sparse table
Time Complexity: $\mathcal{O}(N \log N) - \mathcal{O}(\log N)$

```
for (int i=1; i<16; ++i) {
    for (int j=0; j<n; ++j)
        par[j][i] = par[par[j][i-1]][i-1];
}
```

2.5 Heavy-Light Decomposition

Usage: Query with the ETT number and it's root node
Time Complexity: $\mathcal{O}(N) - \mathcal{O}(\log N)$

```
vector<int> par(n), ett(n), root(n), depth(n), sz(n);
function<void (int)> dfs1 = [&] (int x) {
    sz[x] = 1;
    for (int &next : e[x]) {
        if (next == par[x])
            continue;
        depth[next] = depth[x]+1;
        par[next] = x;
        dfs1(next);
        sz[x] += sz[next];
        if (e[x][0] == par[x] || sz[e[x][0]] < sz[next])
            swap(e[x][0], next);
    }
};
int idx = 1;
function<void (int)> dfs2 = [&] (int x) {
    ett[x] = idx++;
    for (int next : e[x]) {
        if (next == par[x])
            continue;
        root[next] = next == e[x][0] ? root[x] : next;
        dfs2(next);
    }
};
```

3 Geometry

3.1 Counter Clockwise

Usage: It returns $\{-1, 0, 1\}$ - the ccw of $b - a$ and $c - b$
Time Complexity: $\mathcal{O}(1)$

```
auto ccw = [] (const pii &a, const pii &b, const pii &c) {
    pii x = { b.first - a.first, b.second - a.second };
    pii y = { c.first - b.first, c.second - b.second };
    ll ret = 1LL * x.first * y.second - 1LL * x.second * y.first;
    return ret == 0 ? 0 : (ret > 0 ? 1 : -1);
};
```

3.2 Line intersection

Usage: Check the intersection of (x_1, x_2) and (y_1, y_2). It requires an additional condition when they are parallel
Time Complexity: $\mathcal{O}(1)$

```
ccw(x1, x2, y1) != ccw(x1, x1, y2) && ccw(y1, y2, x1) != ccw(y1, y2,
x2)
```

3.3 Graham Scan

Time Complexity: $\mathcal{O}(N \log N)$

```
struct point {
    int x, y, p, q;
    point() { x = y = p = q = 0; }
    bool operator < (const point& other) {
        if (1LL * other.p * q != 1LL * p * other.q)
            return 1LL * other.p * q < 1LL * p * other.q;
        else if (y != other.y)
            return y < other.y;
        else
            return x < other.x;
    }
};
swap(points[0], *min_element(points.begin(), points.end()));
for (int i=1; i<points.size(); ++i) {
    points[i].p = points[i].x - points[0].x;
```

```
    points[i].q = points[i].y - points[0].y;
}
sort(points.begin()+1, points.end());
vector<int> hull;
for (int i=0; i<points.size(); ++i) {
    while (hull.size() >= 2 && ccw(points[hull[hull.size()-2]],
points[hull.back()], points[i]) < 1)
        hull.pop_back();
    hull.push_back(i);
}
```

3.4 Monotone Chain

Usage: Get the upper and lower hull of the convex hull
Time Complexity: $\mathcal{O}(N \log N)$

```
pair<vector<pii>, vector<pii>> getConvexHull(vector<pii> pt){
    sort(pt.begin(), pt.end());
    vector<pii> uh, dh;
    int un=0, dn=0; // for easy coding
    for (auto &tmp : pt) {
        while(un >= 2 && ccw(uh[un-2], uh[un-1], tmp))
            uh.pop_back(), --un;
        uh.push_back(tmp); ++un;
    }
    reverse(pt.begin(), pt.end());
    for (auto &tmp : pt) {
        while(dn >= 2 && ccw(dh[dn-2], dh[dn-1], tmp))
            dh.pop_back(), --dn;
        dh.push_back(tmp); ++dn;
    }
    return {uh, dh};
} // ref: https://namnamseo.tistory.com
```

3.5 Rotating Calipers

Usage: Get the maximum distance of the convex hull
Time Complexity: $\mathcal{O}(N)$

```
auto ccw4 = [&] (point& a1, point& a2, point& b1, point& b2) {
    return 1LL * (a2.x - a1.x) * (b2.y - b1.y) > 1LL * (a2.y - a1.y)
* (b2.x - b1.x);
};
auto dist = [] (point& a, point& b) {
    return 1LL * (a.x - b.x) * (a.x - b.x) + 1LL * (a.y - b.y) *
(a.y - b.y);
};
ll maxi = 0;
for (int i=0, j=1; i<hull.size();) {
    maxi = max(maxi, dist(hull[i], hull[j]));
    if (j < hull.size()-1 && ccw4(hull[i], hull[i+1], hull[j],
hull[j+1]))
        j++;
    else
        i++;
}
```

4 Fast Fourier Transform

4.1 Fast Fourier Transform

Usage: FFT and multiply polynomials
Time Complexity: $\mathcal{O}(N \log N)$

```
using cd = complex<double>;
void fft(vector<cd> &f, cd w) {
  int n = f.size();
  if (n == 1)
    return;
  vector<cd> odd(n/2), even(n/2);
  for (int i=0; i<n; ++i)
    (i%2 ? odd : even)[i/2] = f[i];
  fft(odd, w*w);
  fft(even, w*w);
  cd x(1, 0);
  for (int i=0; i<n/2; ++i) {
    f[i] = even[i] + x * odd[i];
    f[i+n/2] = even[i] - x * odd[i];
    x *= w; // get through power to better accuracy
  }
}
vector<cd> mult(vector<cd> a, vector<cd> b) {
```

```
  int n;
  for (n=1; n<a.size() || n<b.size(); n*=2);
  n *= 2;
  vector<cd> ret(n);
  a.resize(n);
  b.resize(n);
  static constexpr double PI = 3.14159265358979323846;
  cd w(cos(PI*2/n), sin(PI*2/n));
  fft(a, w);
  fft(b, w);
  for (int i=0; i<n; ++i)
    ret[i] = a[i] * b[i];
  fft(ret, cd(1, 0)/w);
  for (int i=0; i<n; ++i) {
    ret[i] /= cd(n, 0);
    ret[i] = cd(round(ret[i].real()), round(ret[i].imag()));
  }
  return ret;
}
```

4.2 Number Theoretic Transform

Usage: FFT with integer - to get better accuracy
Time Complexity: $\mathcal{O}(N \log N)$

```
// w is the root of mod e.g. 3/998244353 and 5/1012924417
void ntt(vector<ll> &f, const ll w, const ll mod) {
  const int n = f.size();
  if (n == 1)
    return;
  vector<ll> odd(n/2), even(n/2);
  for (int i=0; i<n; ++i)
    (i&1 ? odd : even)[i/2] = f[i];
  ntt(odd, w*w%mod, mod);
  ntt(even, w*w%mod, mod);
  ll x = 1;
  for (int i=0; i<n/2; ++i) {
    f[i] = (even[i] + x * odd[i] % mod) % mod;
    f[i+n/2] = (even[i] - x * odd[i] % mod + mod) % mod;
    x = x*w%mod;
  }
}
```

4.3 Fast Walsh Hadamard Transform

Usage: XOR convolution
Time Complexity: $\mathcal{O}(N \log N)$

```
void fwht(vector<ll> &f) {
  const int n = f.size();
  if (n == 1)
    return;
  vector<ll> odd(n/2), even(n/2);
  for (int i=0; i<n; ++i)
    (i&1 ? odd : even)[i/2] = f[i];
  fwht(odd);
  fwht(even);
  for (int i=0; i<n/2; ++i) {
    f[i*2] = even[i] + odd[i];
    f[i*2+1] = even[i] - odd[i];
  }
}
```

5 String

5.1 Knuth-Moris-Pratt

Time Complexity: $\mathcal{O}(N)$

```
vector<int> fail(m);
for (int i=1, j=0; i<m; ++i) {
    while (j > 0 && p[i] != p[j])
        j = fail[j-1];
    if (p[i] == p[j])
        fail[i] = ++j;
}
vector<int> ans;
for (int i=0, j=0; i<n; ++i) {
    while (j > 0 && t[i] != p[j])
        j = fail[j-1];
```

```
    if (t[i] == p[j]) {
        if (j == m-1) {
            ans.push_back(i-j);
            j = fail[j];
        } else
            j++;
    }
}
```

5.2 Rabin-Karp

Usage: The Rabin fingerprint for const-length hashing

Time Complexity: $\mathcal{O}(N)$

```
ull hash, p;
vector<ull> ht;
for (int i=0; i<=l-mid; ++i) {
    if (i == 0) {
        hash = s[0];
        p = 1;
        for (int j=1; j<mid; ++j) {
            hash = hash * pi + s[j];
            p = p * pi; // pi is the prime e.g. 13
        }
    } else
        hash = (hash - p * s[i-1]) * pi + s[i+mid-1];
    ht.push_back(hash);
}
```

5.3 Manacher

5.4 Suffix Array and LCP Array

5.5 Aho-Corasick

6 Offline Query

6.1 Mo's

6.2 Parallel Binary Search

7 DP Optimization

7.1 Convex Hull Optimization w/ Stack

7.2 Convex Hull Optimization w/ Li-Chao Tree

7.3 Knuth Optimization

7.4 Slope Trick

8 Number Theory

8.1 Fermat's Little Theorem

8.2 Modular Inverse in $\mathcal{O}(N)$

8.3 Extended Euclidean

8.4 Miller-Rabin

8.5 Chinese Remainder Theorem

8.6 Pollard Rho

9 ETC

9.1 Catalan Number

9.2 Ternery Search

9.3 제출하기 전 생각해볼 것

9.4 자주 쓰이는 문제 접근법