Competitive Programming Algorithms and Topics

Ubiratan Neto

1. Data Structures

1.1. Segment Tree using Pointers

```cpp
/* Segment Tree implementation using pointers */
/* Can be adapted to Persistent Segment Tree */

struct node {
  node *left, *right;
  //attributes of node
  node() {
    //initialize attributes
    left = NULL;
    right = NULL;
  }
};
void combine(node *ans, node *left, node *right){
  //combine operation
}
void build(node *root, int l, int r){
  if(l == r){
    root->sum = v[l];
    return;
  }
  int m = (l+r) >> 1;
  if(!root->left) root->left = new node();
  if(!root->right) root->right = new node();
  build(root->left, l, m);
  build(root->right, m+1, r);
  combine(root, root->left, root->right);
}

void update(node *root, int l, int r, int idx, int val){
  if(l == r && l == idx){
    //do leaf operation
    return;
  }
  int m = (l+r) >> 1;
  if(idx <= m){
    if(!root->left) root->left = new node();
    update(root->left, l, m, idx, val);
  }
  else {
    if(!root->right) root->right = new node();
    update(root->right, m+1, r, idx, val);
  }
  combine(root, root->left, root->right);
}

node* query(node *root, int l, int r, int a, int b){
  if(l == a && r == b){
    return root;
  }
  int m = (l+r) >> 1;
  if(b <= m){
    if(!root->left) root->left = new node();
    return query(root->left, l, m, a, b);
  }
  else if(m < a){
    if(!root->right) root->right = new node();
    return query(root->right, m+1, r, a, b);
  }
  if(!root->left) root->left = new node();
  if(!root->right) root->right = new node();
  node *left = query(root->left, l,m,a,m);
```

```
  node *right = query(root->right, m+1, r, m+1, b);
  node *ans = new node();
  combine(ans, left, right);
  return ans;
}
```

1.2. Range Update Segment Tree

```
/* Range update Segment Tree Implementation */
/* The first node (ROOT) is defined to 1 (1 - index impl) */
/* N is the maximum number of elements given by the statement */
/* Lazy can be inside node structure instead of being another structure */

#define ROOT 1
#define N MAX_INPUT

struct node{
  //attributes of node
};

node tree[4*N];
node lazy[4*N];

node combine(node a, node b){
  node res;
  //combine operations
  return res;
}

void propagate(int root, int l , int r){
  //return if there is no update
  //update tree using lazy node
  if(l != r){
    //propagate for left and right child
  }
  //reset lazy node
}

void range_update(int root, int l, int r, int a, int b, long long val){
  if(l == a && r == b){
    //lazy operation using val
    return;
  }

  int m = (l+r)/2;

  if(b <= m) range_update(2*root, l, m, a, b, val);
  else if(m < a) range_update(2*root+1, m+1, r, a, b, val);
  else {
    range_update(2*root, l, m, a, m, val);
    range_update(2*root+1, m+1, r, m+1, b, val);
  }

  propagate(root, l , r);
  propagate(2*root, l, m);
  propagate(2*root+1, m+1, r);
  tree[root] = combine(tree[2*root], tree[2*root+1]);
}

node query(int root, int l, int r, int a, int b){
  propagate(root, l, r);
  if(l == a && r == b) return tree[root];

  int m = (l+r)/2;
  if(b <= m) return query(2*root, l, m, a, b);
  else if(m < a) return query(2*root+1, m+1, r, a, b);
  else {
    node left = query(2*root, l, m, a, m);
    node right = query(2*root+1, m+1, r, m+1, b);
    node ans = combine(left, right);
    return ans;
  }
}
```

1.3. Range Update Binary Indexed Tree

```
/* Range Update Binary Indexed Tree Implementation */
/* Tree is 1 - index */
/* Point Update Binary Indexed Tree operations are used as auxiliar */
/* N is defined as the maximum number of elements (given by the statement) */

#define N MAX_INPUT

int bit[2][N+1];

void init(int n){
  for(int i=1; i<=n; i++){
    bit[0][i] = 0;
    bit[1][i] = 0;
  }
}

//auxiliar functions

void update(int *bit, int idx, int val, int n){
  for(int i = idx; i <= n; i += i&-i){
    bit[i]+=val;
  }
}

int query(int *bit, int idx){
  int ans = 0;
  for(int i=idx; i>0; i -= i&-i){
    ans += bit[i];
  }
  return ans;
}

//end of auxiliar functions

void range_update(int l, int r, int val, int n){
  update(bit[0], l, val, n);
  update(bit[0], r+1, -val, n);
  update(bit[1], l, val*(l-1), n);
  update(bit[1], r+1, -val*r, n);
}

int prefix_query(int idx){
  return query(bit[0],idx)*idx - query(bit[1], idx);
}

int range_query(int l, int r){
  return prefix_query(r) - prefix_query(l-1);
}
```

2. Uncategorized

2.1. Longest Increasing Subsequence

```cpp
/* Longest Increasing Subsequence Implementation */
/* N is defined as the maximum array size given by the statement */

#define N MAX_N

int v[N];
int lis[N+1];

void calculate_lis(int n){
  for(int i=1; i<=n; i++) lis[i] = INT_MAX;
  lis[0] = INT_MIN;
  for(int i=0; i<n; i++){
    int index = lower_bound(lis, lis+n+1, v[i]) - lis;
    index--;
    lis[index+1] = min(lis[index+1], v[i]);
  }
}
```

3. Geometry

3.1. Closest Pair of Points

```cpp
/* Closest Pair of Points Problem Implementation */
/* Divide and Conquer Approach */
/* Using the observation of only checking points inside min_dist x min_dist from mid */
/* Binary search boosts search for the right border start point */

struct vec2 {
  long long x, y;
};

bool cmp(vec2 a, vec2 b) {
  return a.x < b.x || (a.x == b.x && a.y < b.y);
}

pair<vec2, vec2> ans;

long long solve(vector<vec2> &a) {

  long long mid = a[a.size()/2].x;
  int n = a.size();

  vector<vec2> l;
  vector<vec2> r;
  int i = 0;
  for(; i < a.size()/2; i++) l.push_back(a[i]);
  for(; i < a.size(); i++) r.push_back(a[i]);

  long long d = LLONG_MAX;

  if(l.size() > 1) {
    d = min(d, solve(l));
  } if(r.size() > 1) {
    d = min(d, solve(r));
  }

  a.clear();

  vector<vec2> ll;
  vector<vec2> rr;


  int j = 0;
  i = 0;
  for(int k=0; k<n; k++){
    if(i < l.size() && j < r.size()){
      if(r[j].y <= l[i].y){
        if((r[j].x - mid)*(r[j].x - mid) < d) {
          rr.push_back(r[j]);
        }
        a.push_back(r[j++]);
      }
      else {
        if((l[i].x - mid)*(l[i].x - mid) < d) {
          ll.push_back(l[i]);
        }
        a.push_back(l[i++]);
      }
    }
    else if(i < l.size()){
      if((l[i].x - mid)*(l[i].x - mid) < d) {
        ll.push_back(l[i]);
      }
      a.push_back(l[i++]);
    }
    else {
      if((r[j].x - mid)*(r[j].x - mid) < d) {
          rr.push_back(r[j]);
      }
        a.push_back(r[j++]);
    }
  }

  for(int i = 0; i < ll.size(); i++) {

    int ini = 0, end = rr.size()-1;
    int j;
    while(ini < end) {
      j = (ini + end) / 2;
      if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d && rr[j].y < ll[i].y)
        ini = j+1;
      else end = j;
    }
    j = ini;

    for(; j < rr.size(); j++) {
      if((rr[j].y - ll[i].y)*(rr[j].y - ll[i].y) > d) break;
      long long cur =  (ll[i].x - rr[j].x)*(ll[i].x - rr[j].x)
            +(ll[i].y - rr[j].y)*(ll[i].y - rr[j].y);
      if(cur < d) {
        d = cur;
        long long cur2 =  (ans.first.x - ans.second.x)*(ans.first.x - ans.second.x)
            +(ans.first.y - ans.second.y)*(ans.first.y - ans.second.y);
        if(cur < cur2)
          ans = { ll[i], rr[j] };
      }
    }
  }
  return d;
}
```

3.2. Convex Hull - Monotone Chain Algorithm

```
/* Convex Hull - Monotone Chain */
/* Generates Upper and Lower Hull */
/* It is needed to give array of points ordered by <x,y> */

vector < pair<int, int> > upper, lower;
vector< pair<int, int> > hull;


int cross(pair<int, int> & a, pair<int, int> & b, pair<int, int> & c){
    pair<int, int> vec1(b.ff - a.ff, b.ss - a.ss);
    pair<int, int> vec2(c.ff - b.ff, c.ss - b.ss);
    return vec1.ff*vec2.ss - vec1.ss*vec2.ff;
}

void calculate_upper(vector< pair<int, int> > & p){
    for(int i=0; i<p.size(); i++){
          while(upper.size() >= 2 && cross(upper[upper.size()-2],
    upper.back(), p[i]) >= 0){
               upper.pop_back();
          }
          upper.push_back(p[i]);
      }
}

void calculate_lower(vector< pair<int, int> > & p){
  for(int i=0; i<p.size(); i++){
          while(lower.size() >= 2 && cross(lower[lower.size()-2],
    lower.back(), p[i]) <= 0){
               lower.pop_back();
          }
          lower.push_back(p[i]);
      }
}

void merge_hull(){
  for(int i=0; i<upper.size(); i++) hull.push_back(upper[i]);
  for(int i=lower.size()-2; i>0; i--) hull.push_back(lower[i]);
}
```

3.3. Shoelace Formula for Polygon Area

```
/* Shoelace formula */
/* Calculate area of convex polygon */
/* Points given in clockwise/counterclockwise order */

int cross(pair<int, int> & a, pair<int, int> & b){
  return a.ff*b.ss - a.ss*b.ff;
}

int shoelace(vector< pair<int, int> > & p){
  int area = 0;
  for(int i=0; i<hull.size(); i++){
    area += cross(hull[i], hull[(i+1)%hull.size()]);
  }
  return abs(area/2);
}
```

4. Graphs

4.1. Dynammic Connectivity - connected(u,v) query

```
/* Dynammic Connectivity Implementation */
/* Uses Divide and Conquer Offline approach */
/* Able to answer if two vertex <u,v> are connected */
/* No multi-edges allowed */
/* DSU + Rollback is used to backtrack merges */
/* N is defined as the maximum graph size given by input */

#define N MAX_INPUT

int uf[N];
int sz[N];

struct event{
  int op, u, v, l, r;
  event() {}
  event(int o, int a, int b, int x, int y) : op(o), u(a), v(b), l(x), r(y) {}
};

map< pair<int, int>, int > edge_to_l;
stack< pair<int*,int> > hist;
vector<event> events;

int init(int n){
  for(int i=0; i<=n; i++){
    uf[i] = i;
    sz[i] = 1;
  }
}

int find(int u){
  if(uf[u] == u) return u;
  else return find(uf[u]);
}

void merge(int u, int v){
  int a = find(u);
  int b = find(v);
  if(a == b) return;
  if(sz[a] < sz[b]){
    hist.push(make_pair(&uf[a], uf[a]));
    uf[a] = b;
    hist.push(make_pair(&sz[b], sz[b]));
    sz[b]+= sz[a];
  }
  else {
    hist.push(make_pair(&uf[b], uf[b]));
    hist.push(make_pair(&sz[a], sz[a]));
    uf[b] = a;
    sz[a]+=sz[b];
  }
}

int snap(){
  return hist.size();
}

void rollback(int t){
  while(hist.size() > t){
    pair<int*, int> aux = hist.top();
    hist.pop();
    *aux.first = aux.second;
  }
}

void solve(int l, int r){
  if(l == r){
```

```cpp
    if(events[l].op == 2){
      if(find(events[l].u) == find(events[l].v)) cout << "YES" << endl;
      else cout << "NO" << endl;
    }
    return;
  }

  int m = (l+r)/2;
  //doing for [L,m]
  int t = snap();
  for(int i=l; i<=r; i++){
    if(events[i].op == 0 || events[i].op == 1){
      if(events[i].l <= l && m <= events[i].r) merge(events[i].u,
    events[i].v);
    }
  }
  solve(l, m);
  rollback(t);

  //doing for [m+1, R]
  t = snap();
  for(int i=l; i<=r; i++){
    if(events[i].op == 0 || events[i].op == 1){
      if(events[i].l <= m+1 && r <= events[i].r) merge(events[i].u,
    events[i].v);
    }
  }
  solve(m+1, r);
  rollback(t);
}

void offline_process(){
  int n, q;
  cin >> n >> q; //number of vertex and queries
  init(n);
  for(int i=0; i<q; i++){
    string op;
    int u,v;
    cin >> op >> u >> v; //add, remove or query for u,v
    if(u > v) swap(u,v);
    if(op == "add"){
      events.push_back(event(0, u, v, i, -1));
      edge_to_l[make_pair(u,v)] = i;
    }
    else if(op == "rem"){
      int l = edge_to_l[make_pair(u,v)];
      events.push_back(event(1, u, v, l, i));
      events[l].r = i;
    }
    else if(op == "conn"){
      events.push_back(event(2, u, v, -1, -1));
    }
  }
  for(int i=0; i<q; i++){
    if(events[i].op == 0){
      if(events[i].r == -1){
        events[i].r = events.size();
        events.push_back(event(1, events[i].u, events[i].v, events[i].l,
    events[i].r));
      }
    }
  }
}
```

5. Math and Number Theory

5.1. Binomial Coefficient DP

```cpp
/* Dynammic Programming for Binomial Coefficient Calculation */
/* Using Stiefel Rule C(n, k) = C(n-1, k) + C(n-1, k-1) */

int binomial(int n ,int k){
  int c[n+10][k + 10];
  memset(c, 0 , sizeof c);
  c[0][0] = 1;
  for(int i = 1;i<=n;i++){
    for(int j = min(i, k);j>0;j--){
      c[i][j] = c[i-1][j] + c[i-1][j-1];
    }
  }
  return c[n][k];
}
```

5.2. Erathostenes Sieve + Logn Prime Factorization

```cpp
/* Erasthostenes Sieve Implementation */
/* Calculate primes from 2 to N */
/* lf[i] stores the lowest prime factor of i(logn factorization) */

bitset<N> prime;
int lf[N];

void run_sieve(int n){
  for(int i=0; i<=n; i++) lf[i] = i;
  prime.set();
  prime[0] = false;
  prime[1] = false;
  for(int p = 2; p*p <= n; p++){
    if(prime[p]){
      for(int i=p*p; i<=n; i+=p){
        prime[i] = false;
        lf[i] = min(lf[i], p);
      }
    }
  }
}
```

5.3. Matrix Exponentiation

```cpp
/* Matrix Exponentiation Implementation */

typedef vector< vector<int> > Matrix;

Matrix operator *(const Matrix & a, const Matrix & b){
  Matrix c(a.size(), vector<int>(b[0].size()));
  for(int i = 0; i<a.size(); i++){
    for(int j = 0; j<b[0].size(); j++){
      for(int k = 0; k<b.size(); k++){
        c[i][j] += (a[i][k]*b[k][j]);
      }
    }
  }
  return c;
}

Matrix exp(Matrix & a, int k){
  if(k == 1) return a;
```

```
  Matrix c = exp(a, k/2);
  c = c*c;
  if(k%2) c = c*a;
  return c;
}
```

5.4. Fast Fourier Transform - Recursive and Iterative

```
/* Fast Fourier Transform Implementation */
/* Complex numbers implemented by hand */
/* Poly needs to have degree of next power of 2 (result poly has size
     next_pot2(2*n) */
/* Uses Roots of Unity Idea (Z^n = 1, divide and conquer strategy)
/* Inverse FFT only changes to the conjugate of Primitive Root of Unity */
/* Remember to use round to get integer value of Coefficients of Poly C */
/* Iterative FFT is way faster (bit reversal idea + straightforward conquer
     for each block of each size) */
/* std::complex doubles the execution time */

struct Complex{
  double a, b;

  Complex(double a, double b) : a(a), b(b) {}

  Complex() : a(0), b(0) {}

  Complex conjugate() const {
    return Complex(a, -b);
  }

  double size2() const {
    return a*a + b*b;
  }

  Complex operator+(const Complex & y) const {
    return Complex(a + y.a, b + y.b);
  }

  Complex operator-(const Complex & y) const {
    return Complex(a - y.a, b - y.b);
  }

  Complex operator*(const Complex & y) const {
    return Complex(a*y.a - b*y.b, a*y.b + b*y.a);
  }

  Complex operator/(const double & x) const {
    return Complex(a/x, b/x);
  }

  Complex operator/(const Complex & y) const {
    return (*this)*(y.conjugate()/y.size2());
  }

};

struct Poly{
  vector<Complex> c;
  Poly() {}

  Poly(int sz){
    c.resize(sz);
  }

  int size() const{
    return (int)c.size();
  }
};

inline Complex PrimitiveRootOfUnity(int n){
  const double PI = acos(-1);
  return Complex(cos(2*PI/(double)n), sin(2*PI/(double)n));
}

inline Complex InversePrimitiveRootOfUnity(int n){
  const double PI = acos(-1);
  return Complex(cos(-2*PI/(double)n), sin(-2*PI/(double)n));
}

void FFT(Poly & A, bool inverse){
  int n = A.size();
  int lg = 0;
  while(n > 0) lg++, n>>=1;
  n = A.size();
  lg-=2;

  for(int i=0; i<n; i++){
    int j = 0;
    for(int b=0; b <= lg; b++){
      if(i & (1 << b)) j |= (1 << (lg - b));
    }
    if(i < j) swap(A.c[i], A.c[j]);
  }

  for(int len=2; len <= n; len <<= 1){
    Complex w;
    if(inverse) w = InversePrimitiveRootOfUnity(len);
    else w = PrimitiveRootOfUnity(len);

    for(int i=0; i<n; i+=len){
      Complex x(1,0);
      for(int j=0; j<len/2; j++){
        Complex u = A.c[i+j], v = x*A.c[i+j+len/2];
        A.c[i+j] = u + v;
        A.c[i+j+len/2] = u - v;
        x = x*w;
      }
    }
  }

  if(inverse) for(int i=0; i<n; i++) A.c[i] = A.c[i]/n;
}

/* Skipable */
Poly RecursiveFFT(Poly A, int n, Complex w){
  if(n == 1) return A;

  Poly A_even(n/2), A_odd(n/2);

  for(int i=0; i<n; i+=2){
    A_even.c[i/2] = A.c[i];
    A_odd.c[i/2] = A.c[i+1];
  }

  Poly F_even = FFT(A_even, n/2, w*w);
  Poly F_odd = FFT(A_odd, n/2, w*w);
  Poly F(n);
  Complex x(1, 0);

```

```
  for(int i=0; i<n/2; i++){
    F.c[i] = F_even.c[i] + x*F_odd.c[i];
    F.c[i + n/2] = F_even.c[i] -  x*F_odd.c[i];
    x = x*w;
  }

  return F;
}
/* Skipable */

Poly Convolution(Poly & F_A, Poly & F_B){
  Poly F_C(F_A.size());
  for(int i=0; i<F_A.size(); i++) F_C.c[i] = F_A.c[i]*F_B.c[i];
  return F_C;
}

Poly operator*(Poly & A, Poly & B){
  FFT(A, false);

  FFT(B, false);

  Poly C = Convolution(A, B);

  FFT(C, true);

  return C;
}
```

6. String Algorithms

6.1. KMP Failure Function + String Matching

```
/* Knuth - Morris - Pratt Algorithm */
/* Failure Function for String Matching */

int pi[N];
string s, t;

void prefix(int n) {
  pi[0] = 0;
  for(int i = 1; i < n; i++) {
    pi[i] = pi[i-1];
    while(pi[i] > 0 && t[i] != t[pi[i]]) pi[i] = pi[pi[i]-1];
    if(t[i] == t[pi[i]]) pi[i]++;
  }
}

void matching(int n){
  int j = 0;
  for(int i=0; i<n; i++){
    while(j > 0 && s[i] != t[j]) j = pi[j-1];
    if(s[i] == t[j]) j++;
    if(j == t.size()){
      cout << "match in " << j-t.size()+1 << endl;
      j = pi[j-1];
    }
  }
}
```

6.2. Z-Function

```
/* Z-function */
/* Calculate the size K of the largest substring which is a prefix */

vector<int> z;

void make(string s){
  int n = s.size();
  z.resize(n);
  z[0] = 0;
  int l = 0, r = 0;
  for(int i=1; i<n; i++){
    if(i > r){
      l = i;
      r = i;
    }
    z[i] = min(z[i-l], r-i+1);
    while(i + z[i] < n && s[i + z[i]] == s[z[i]]) z[i]++;
    if(i + z[i] > r){
      l = i;
      r = i + z[i]-1;
    }
  }
}
```

6.3. Rolling Hash

```
/* Rolling Hash Implementation */
/* Uses 1-indexed string */

long long BASE = 137
long long PRIME = (int)1e9+7;

long long hash[N+1];
long long base[N+1];
long long invBase[N+1];

long long expo(long long a, long long k){
  if(k == 0) return 1LL;
  else if(k == 1) return a;
  long long aux = expo(a, k/2);
  aux %= PRIME;
  aux *= aux;
  aux %= PRIME;
  if(k%2) aux *= a;
  aux %= PRIME;
  return aux;
}

void calculate(string a){
  base[0] = 1;
  invBase[0] = 1;
  hash[0] = 0;
  for(int i=1; i<=a.size(); i++){
    hash[i]+= BASE*hash[i-1] + a[i-1];
    hash[i] % = PRIME;
    base[i] = base[i-1]*BASE;
    base[i] %= PRIME;
    invBase[i] = expo(base[i], PRIME-2);
  }
}

long long range_hash(int i, int j){
  return ((h[j] - h[i-1])*invBase[j-i+1])%PRIME;
}
```

6.4. Suffix Array + Linear Sort

```
/* Suffix Array using Counting Sort Implementation */
/* rnk is inverse of sa array */
/* aux arrays are needed for sorting step */
/* inverse sorting (using rotating arrays and blocks of power of 2) */
/* rmq data structure needed for calculating lcp of two non adjacent
    suffixes sorted */

int rnk[N],tmp[N], sa[N], sa_aux[N], lcp[N];
int block=0, n;
string s;

bool suffixcmp(int i, int j){
  if(rnk[i] != rnk[j]) return rnk[i] < rnk[j];
  i+=block, j+=block;
  i%=n;
  j%=n;
  return rnk[i] < rnk[j];
}

void suffixSort(int MAX_VAL){
  for(int i=0; i<=MAX_VAL; i++) tmp[i] = 0;
  for(int i=0; i<n; i++) tmp[rnk[i]]++;
  for(int i=1; i<=MAX_VAL; i++) tmp[i] += tmp[i-1];
  for(int i = n-1; i>=0; i--){
      int aux = sa[i]-block;
      aux%=n;
      if(aux < 0) aux+=n;
      sa_aux[--tmp[rnk[aux]]] = aux;
  }
  for(int i=0; i<n; i++) sa[i] = sa_aux[i];
  tmp[0] = 0;
  for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
  for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
}

void build_sa(){
  s+='\0';
  n++;
  for(int i=0; i<n; i++){
    sa[i] = i;
    rnk[i] = s[i];
    tmp[i] = 0;
  }
  suffixSort(256);
  block = 1;
  while(tmp[n-1] != n-1){
    suffixSort(tmp[n-1]);
    block*=2;
  }
  for(int i=0; i<n-1; i++) sa[i] = sa[i+1];
  n--;
  tmp[0] = 0;
  for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + suffixcmp(sa[i-1], sa[i]);
  for(int i=0; i<n; i++) rnk[sa[i]] = tmp[i];
}

void calculate_lcp(){
  int last = 0;
  for(int i=0; i<n; i++){
    if(rnk[i] == n-1) continue;
    int x = rnk[i];
    lcp[x] = max(0,last-1);
    while(sa[x] + lcp[x] < n && sa[x+1] + lcp[x] < n && s[sa[x]+lcp[x]] ==
     s[sa[x+1]+lcp[x]]){
      lcp[x]++;
    }
    last = lcp[x];
  }
}

int lcp(int x, int y){
  if(x == y) return n - x;
  if(rnk[x] > rnk[y]) swap(x,y);
  return rmq(rnk[x], rnk[y]-1);
}
```