Unibo Team

Reference Document

06/05/2018

Contents

1. Data Structures

1.1. **Union-Find.** An implementation of the Union-Find disjoint sets data structure.

```cpp
struct union_find {
  vector<int> parent, rank;
  union_find(int N) : parent(N, 0), rank(N, 0) {
    for(int i=0; i<N; ++i)
      parent[i] = i;
  }
  int find_set(int i) {
    return parent[i] == i ? i : (parent[i] = find(parent[i]));
  }
  bool is_same_set(int i, int j) {
    return find_set(i) == find_set(j);
  }
  bool unite(int i, int j) {
    if(!is_same_set(i, j)) {
      int x = find_set(i), y = find_set(j);
      if(rank[x] > rank[y]) p[y] = x;
      else p[x] = y;
      if(rank[x] == rank[y]) ++rank[y];
    }
  }
};
```

1.2. **Sparse Segment Tree.** An implementation of a Segment Tree.

```cpp
//remember to update merge, nullvalue
struct segment{
  const int nullValue = 0;  // EDIT HERE
  int merge(int a, int b){ return a+b; }
  int l, r, val;
  segment *left, *right;
  segment(int _l, int _r){
    l = _l, r = _r;
    left = right = 0;
    val = nullValue;
  }
  int size(){ return r-l+1; }
  int query(int a, int b){
    if(l >= a && r <= b) return val;
    if(r < a || l > b) return nullValue;
    return merge(left ? left->query(a, b) : nullValue,
          right ? right->query(a, b) : nullValue);
  }
  void update(int pos, int k){
    if(l > pos || r < pos) return;
    if(size() == 1){
      val += k;
      return;
    }
    int m = (l+r)/2;
    if(left == 0)
      left = new segment(l, m);
    if(right == 0)
      right = new segment(m+1, r);
    left->update(pos, k);
    right->update(pos, k);
    sync();
  }
  void sync(){ val = merge(right->val, left->val); }
};

//use this in main
struct sparsetree{
  segment *root;
  sparsetree(int range){
    root = new segment(0, range);
  }
  int rsq(int l, int r){
    return root->query(l, r);
  }
  void update(int i, int k){
    root->update(i, k);
  }
};
```

1.3. **Segment Tree AND Segment Tree 2D.** An implementation of a Segment Tree.

```
#define LEFT(i) ((i)*2+1)
#define RIGHT(i) ((i)*2+2)
#define ROOT(i) (((i)-1)/2)
constexpr int closest_pow(int n){ return 1<<(int)ceil(log2(n)); }
/* versione per RMQ (maximum)
  0 based - per modificarlo per le somme, fare attenzione a tutte le funzioni
  UTILIZZO 1D -> [0 based RMQ]
    NB : point update e range add ATTUALMENTE NON si devono mischiare,
      se si decide di utilizzare la lazy propagation usiamo dei range
      add per simulare le point update O mettiamo mano al codice
    rangemax(l, r) = massimo nel range l, r
    update(i, k) = l'indice i diventa k
    range_add(l, r, k) = tutti i valori tra l e r inclusi aumentano di k */
struct rmqtree{
  vector<int> v,u;
  int off, a, b, nullvalue = INT_MIN;
  void update_node(int i){
    v[i] = max(v[LEFT(i)], v[RIGHT(i)]);
  }
  void lazyupdate(int i){
      v[i] += u[i];
      if(i<v.size()/2){
        u[RIGHT(i)] += u[i];
        u[LEFT(i)] += u[i];
      }
      u[i] = 0;
  }
  void build(vi &source){
    v.resize(closest_pow(source.size())*2 - 1, nullvalue);
    u.resize(v.size(),0);
    for(int i=0;i< source.size();i++)
      v[i+v.size()/2]= source[i];
    for(int i=v.size()/2 - 1;i>=0;i--)
      update_node(i);
  }
  rmqtree(vi &source){ build(source); }
  rmqtree(){}
  int rmq(int i,int l ,int r){
    lazyupdate(i);
    if(l>=a && r<=b) return v[i];
    if(r<a || l>b) return nullvalue;
    int m = (l+r)/2;
    return max(rmq(LEFT(i),l,m), rmq(RIGHT(i),m+1,r));
  }
  int rangemax(int qa,int qb){
    a = qa, b = qb;
    return rmq(0,0,v.size()/2);
  }
  void update_up(int i){
    update_node(i);
    if(i!=0) update_up(ROOT(i));
  }
  void update(int i, int k){
    int x = v.size()/2 + i;
    v[x] = k;
    update_up(ROOT(x));
  }
  void range_add(int i,int l,int r){
    lazyupdate(i);
    if(l>=a && r<=b){
      int x=off;
      v[i]+=x;
      if(l!=r)
        u[LEFT(i)] += x/2, u[RIGHT(i)]+= x/2;
      return;
    }
    if(r<a || l>b) return;
    int m = (l+r)/2;
    range_add(LEFT(i),l,m);
    range_add(RIGHT(i),m+1,r);
    update_node(i);
  }
  void range_up(int qa,int qb,int k){
    a=qa,b=qb, off=k;
    range_add(0, 0, v.size()/2);
  }
};

rmqtree merge(rmqtree &t1, rmqtree &t2){
  vi ans(t1.v.size()/2 + 1);
  for(int i = 0; i < ans.size(); i++)
    ans[i] = max(t1.v[i+t1.v.size()/2], t2.v[i+t2.v.size()/2]);
  return rmqtree(ans);
}
/* UTILIZZO 2D -> [0 based RMQ]
    la lazy propagation non c'è quindi neanche range update.
    volendo, si possono fare range update su una singola riga
    ma serve modificare un minimo il codice
    point_update(r, c, k) = setta a Q il valore [r][c]
    rangemax(r1, c1, r2, c2) = il max nel quadrato [r1, c1], [r2, c2] */
struct rmq2d{
  vector<rmqtree> v;
  int null_value = INT_MIN;
  void build(vvi &source){
    int n1 = closest_pow(source.size())*2 - 1;
    int n2 = closest_pow(source.front().size())*2 - 1;
    v.resize(n1);
    for(int i = n1/2; i < v.size(); i++)
      v[i].build(source[i - n1/2]);
    for(int i = n1/2 - 1; i >= 0; i--)
      v[i] = merge(v[LEFT(i)], v[RIGHT(i)]);
  }
  rmq2d(vvi &source){
    build(source);
  }
  rmq2d(){}
  int qr1, qr2, qc1, qc2, qk;
  int rmq(int i, int l, int r){
    if(l >= qr1 && r <= qr2)
      return v[i].rangemax(qc1, qc2);
    if(r < qr1 || l > qr2)
      return null_value;
    int m = (l+r)/2;
    return max(rmq(LEFT(i), l, m), rmq(RIGHT(i), m + 1, r));
  }
  int rangemax(int r1, int c1, int r2, int c2){
    qr1 = r1, qc1 = c1;
    qr2 = r2, qc2 = c2;
    if(qr1 > qr2)
      swap(qr1, qr2);
    if(qc1 > qc2)
      swap(qc1, qc2);
    return rmq(0, 0, v.size()/2);
  }
  void update_node(int i, int j){
    if(i < v.size())
      v[i].v[j] = max(v[LEFT(i)].v[j], v[RIGHT(i)].v[j]);
  }
  void update_up(int r, int c){
    v[r].update(qc1, qk);
    if(r) update_up(ROOT(r), c);
  }
  void update(int i, int l, int r){
    if(qr1 > r || qr1 < l) return;
    if(l == r){
      v[i].update(qc1, qk);
      update_up(ROOT(i), qc1+v[i].v.size()/2);
    } else update(LEFT(i), l, (l+r)/2), update(RIGHT(i), (l+r)/2 + 1, r);
  }
  void point_update(int r, int c, int k){
    qr1 = r, qc1 = c, qk = k;
    update(0, 0, v.size()/2);
  }
};
```

1.4. **Fenwick Tree.** A Fenwick Tree is a data structure that represents an array of n numbers. It supports adjusting the i-th element in $O(\log n)$ time, and computing the sum of numbers in the range $i..j$ in $O(\log n)$ time. It only needs $O(n)$ space.

```
struct fenwick_tree {
  int LSB(int x) { return x & (-x); }
  vector<int> ft;
  fenwick_tree(int N) : ft(N+1, 0) {}
  fenwick_tree(vector<int> &s) : ft(s.size()+1, 0) {
    for(int i=1; i<ft.size(); ++i) {
      ft[i] += s[i-1];
      if(i + LSB(i) < ft.size())
        ft[i + LSB(i)] += ft[i];
    }
  }
  int range_sum(int i) {
    int sum = 0;
    for(; i; i-=LSB(i))
      sum += ft[i];
    return sum;
  }
  int range_sum(int l, int r) {   // fenwick 1 to N, range sum [l, r]
    return range_sum(r) - (l == 1 ? 0 : range_sum(l - 1));
  }
  void add(int i, int k) {
    for(; i<ft.size(), i+=LSB(i))
      ft[i] += k;
  }
};
```

1.5. **AVL Tree.** A fast, easily augmentable, balanced binary search tree.

```
typedef unsigned char uchar;
typedef unsigned int uint;
struct node{
  int key;
  uchar height;
  int size;
  node *left, *right;
  node(int k){
    size = 1;
    height = 1;
    key = k;
    left = right = 0;
  }
};
uchar height(node *n){ return n ? n->height : 0; }
int size(node *n){ return n ? n->size : 0; }
//right - left
int bfactor(node *n){ return height(n->right) - height(n->left); }
//supposes that children are already fixed
void fix_height(node *n){
  uchar l = height(n->left);
  uchar r = height(n->right);
  n->height = max(l, r) + 1;
  n->size = size(n->left) + size(n->right) + 1;
}
//it essentially means that we are taking node n and moving it to\
 his right sub-tree so we need to take the guy on his left and make\
  him become the new root this also means that the new root would become\
  losing his old right child. that right child is so given to his \
  "now-right-child-b4-father" as left child. node bfactor return bfactor(n) + 1
node* rotate_right(node *n){
  node *left = n->left; // copy of the new father so we can lose pointers
  n->left = left->right; //n steals his new-father right-child
  left->right = n; //we make left father of his old father
  // fix those nodes height, start from the bottom
  fix_height(n);
  fix_height(left);
  //we may now return the father in charge of this AVL-freakin' tree
  return left;
}
//same as above, but symmetrical\
  bfactor of node return is bfactor(n) - 1
node* rotate_left(node *n){
  node *right = n->right;
  n->right = right->left;
  right->left = n;
  fix_height(n);
  fix_height(right);
  return right;
}
node* balance(node *n){
  fix_height(n);
  int b = bfactor(n);
  if(b == 2){
//now we need to check if n's right child is left balanced:\
  in that case, a left rotation wouldn't balance the tree because\
  we'll give too much weight to the left child.\
  in this case, we need to perform a right rotation on that child\
  so that we spread it's weight unbalance on his children
    if(bfactor(n->right) < 0)
      n->right = rotate_right(n->right);
    return rotate_left(n);
  }
  else if(b == -2){
    //as above, we rotate to the right but before check for child balance:\
      we don't like it as +1
    if(bfactor(n->left) > 0)
      n->left = rotate_left(n->left);
    return rotate_right(n);
  }
  return n;
}
node *insert(node *n, int k){
  //if we reach the end of the tree
  if(!n) return new node(k);
  //insert left
  if(k <= n->key) n->left = insert(n->left, k);
  else n->right = insert(n->right, k);
  //as we inserted a node, we may need to rebalance
  return balance(n);
}
node *get_min(node *n){ return n->left ? get_min(n->left) : n; }
node *remove_min(node *n){
  if(n->left){
    n->left = remove_min(n->left);
    return balance(n);
  }
  return n->right;
}
node *find(node *n, int k){
  if(!n) return 0;
  if(n->key < k) return find(n->left, k);
  if(n->key > k) return find(n->right, k);
  return n;
}
//0 based
node *get_kth(node *n, int k){
  if(!n) return 0;
  if(size(n->left) < k)
    return get_kth(n->right, k-size(n->left)-1);
  if(size(n->left) > k)
    return get_kth(n->left, k);
  if(size(n->left) == k)
    return n;
}
//removes given key from tree
node *remove_key(node *n, int k){
  if(!n) return 0;
  if(k < n->key)
    n->left = remove_key(n->left, k);
  else if(k > n->key)
```

```
    n->right = remove_key(n->right, k);
  else{
  //if found, we have to replace it with the next element\
    which is min of right subtree
    node *l = n->left, *r = n->right;
    //destroys the struct :(
    delete n;
    //if we have no right-child, just return left one
    if(!r) return l;
    node *min = get_min(r);
    min->right = remove_min(r);
    min->left = l;
    return balance(min);
  }
  return balance(n);
}
struct avl{
  node* _root = 0;
  int _size = 0;
  int size(){ return _size; }
  void insert(int k){
    _root = ::insert(_root, k);
    _size++;
  }
  void remove(int k){ _root = remove_key(_root, k); }
  bool contains(int k){ return find(_root, k); }
  int get_nth(int n){ return ::get_kth(_root, n)->key; }
};
```

1.6. **AVL Tree K-*th*.** A fast, easily augmenta ble, balanced binary search tree.

```
typedef unsigned char uchar;
typedef unsigned int uint;
struct node{
  int key;
  uchar height;
  int size;
  node *left, *right;
  node(int k){
    size = 1;
    height = 1;
    key = k;
    left = right = 0;
  }
};
uchar height(node *n){
  return n ? n->height : 0;
}
int size(node *n){
  return n ? n->size : 0;
}
int bfactor(node *n){
  return height(n->right) - height(n->left);
}
void fix_height(node *n){
  uchar l = height(n->left);
  uchar r = height(n->right);
  n->height = max(l, r) + 1;
  n->size = size(n->left) + size(n->right) + 1;
}
node *rotate_right(node *n){
  node *left = n->left;
  n->left = left->right;
  left->right = n;
  fix_height(n);
  fix_height(left);
  return left;
}
node *rotate_left(node *n){
  node *right = n->right;
  n->right = right->left;
  right->left = n;
  fix_height(n);
  fix_height(right);
  return right;
}
node *balance(node *n){
  fix_height(n);
  int b = bfactor(n);
  if(b == 2){
    if(bfactor(n->right) < 0)
      n->right = rotate_right(n->right);
    return rotate_left(n);
  }
  else if(b == -2){
    if(bfactor(n->left) > 0)
      n->left = rotate_left(n->left);
    return rotate_right(n);
  }
  return n;
}
node *insert_kth(node *n, int k, int value){
  if(!n) return new node(value);
  if(k <= size(n->left))
    n->left = insert_kth(n->left, k, value);
  else if(k > size(n->left))
    n->right = insert_kth(n->right, k - size(n->left) - 1, value);
  return balance(n);
}
node *get_min(node *n){
  return n->left ? get_min(n->left) : n;
}
node *remove_min(node *n){
  if(n->left){
    n->left = remove_min(n->left);
    return balance(n);
  }
  return n->right;
}
node *get_kth(node *n, int k){
  if(!n)
    return 0;
  if(size(n->left) < k)
    return get_kth(n->right, k-size(n->left)-1);
  if(size(n->left) > k)
    return get_kth(n->left, k);
  if(size(n->left) == k)
    return n;
}
static int ans;
node *remove_kth(node *n, int k){
  if(k < size(n->left))
    n->left = remove_kth(n->left, k);
  else if(k > size(n->left))
    n->right = remove_kth(n->right, k - size(n->left) - 1);
  else{
    node *l = n->left, *r = n->right;
    ans = n->key;
    delete n;
    if(!r) return l;
    node *min = get_min(r);
    min->right = remove_min(r);
    min->left = l;
    return balance(min);
  }
  return balance(n);
}
node *remove_key(node *n, int k){
  if(!n) return 0;
  if(k < n->key)
    n->left = remove_key(n->left, k);
  else if(k > n->key)
    n->right = remove_key(n->right, k);
  else{
    node *l = n->left, *r = n->right;
    delete n;
    if(!r) return l;
    node *min = get_min(r);
    min->right = remove_min(r);
    min->left = l;
    return balance(min);
  }
  return balance(n);
}
struct avl{
  node* _root = 0;
  int _size = 0;
  int size(){
    return _size;
  }
  void insert(int k, int value){
    _root = ::insert_kth(_root, k, value);
    _size++;
  }
  int remove(int k){
    _root = remove_kth(_root, k);
    return ans;
  }
  int get_nth(int n){
    return ::get_kth(_root, n)->key;
  }
};
```

1.7. **Wavelettree.** Count values less (or equal) elements, retrieve k-th minimal element

```
// 1-indexed!
struct wavelet {
    int lo, hi;
    vector<int> b;
    wavelet *l = 0, *r = 0;
    wavelet(vector<int>::iterator pl, vector<int>::iterator pr, int x, int y) {
        lo = x, hi = y;
        if (lo == hi || pl >= pr) return;
        int mid = lo + (hi - lo) / 2;
        auto f = [mid](int x) {
            return x <= mid;
        };
        b.reserve(pr - pl + 1);
        b.push_back(0);
        for (auto it = pl; it != pr; it++)
            b.push_back(b.back() + f(*it));
        auto it = stable_partition(pl, pr, f);
        l = new wavelet(pl, it, lo, mid), r = new wavelet(it, pr, mid + 1, hi);
    }
    // retrieve the k_th minimal value in range [l, r]
    int kth(int l, int r, int k) {
        if (l > r) return 0;
        if (lo == hi) return lo;
        int le = b[l - 1], ri = b[r];
        if (ri - le >= k) return this->l->kth(le + 1, ri, k);
        return this->r->kth(l - le, r - ri, k - (ri - le));
    }
    // count values less than k in range [l, r]
    int clt(int l, int r, int k) {
        if (l > r || k <= lo) return 0;
        if (k > hi) return r - l + 1;
        int le = b[l - 1], ri = b[r];
        return this->l->clt(le + 1, ri, k) + this->r->clt(l - le, r - ri, k);
    }
    // count equal to k in range [l, r]
    int ce(int l, int r, int k) {
        if (k < lo || k > hi || l > r) return 0;
        if (lo == hi) return r - l + 1;
        int le = b[l - 1], ri = b[r];
        return this->l->ce(le + 1, ri, k) + this->r->ce(l - le, r - ri, k);
    }
    ~wavelet() { delete l, delete r; }
};
```

1.8. **Treap.** Implementation of implicit treap

```
int get_random(){ return rand() << 15 | rand(); }
struct node{
  int priority = get_random();
  int size = 1;
  int value;
  node *l = 0, *r = 0;
  node(int v) : value(v){}
  ~node(){ delete l, r; }
  node* update(){
    size = 1 + (l ? l->size : 0) + (r ? r->size : 0);
    return this;
  }
  int kth(int k){
    int ls = l ? l->size : 0;
    int rs = r ? r->size : 0;
    if(k < ls) return l->kth(k);
    if(k == ls) return value;
    if(k > ls) return r->kth(k - ls - 1);
  }
};
struct treap{
  node *root = 0;
  ~treap(){ delete root; }
  node *merge(node *l, node *r){
    if(!l) return r;
    if(!r) return l;
    if(l->priority < r->priority){
      l->r = merge(l->r, r);
      return l->update();
    } else {
      r->l = merge(l, r->l);
      return r->update();
    }
  }
  pair<node*, node*> split(node *n, int k){
    if(!n) return {0, 0};
    int lf = n->l ? n->l->size : 0;
    if(k < lf){
      auto f = split(n->l, k);
      n->l = f.second;
      f.second = n->update();
      return f;
    } else {
      auto f = split(n->r, k - lf - 1);
      n->r = f.first;
      f.first = n->update();
      return f;
    }
  }
  //k = position
  void insert(int k, int val){
    auto t = split(root, k-1);
    root = merge(t.first, merge(new node(val), t.second));
  }
  void erase(int k){
    auto t1 = split(root, k - 1);
    auto t2 = split(t1.second, 0);
    if(t2.first) delete(t2.first);
    t1.second = t2.second;
    root = merge(t1.first, t1.second);
  }
  int size(){ return root->size; }
  int operator[](int k){ return root->kth(k); }
};
```

1.9. Mergesort tree array.

```
typedef vector<int> vi;
vi merge(vi &a, vi &b){
  vi v;
  int l = 0, r = 0;
  v.reserve(a.size() + b.size());
  while(l + r < a.size() + b.size()){
    if(l == a.size())
      v.push_back(b[r++]);
    else if(r == b.size() || a[l] < b[r])
      v.push_back(a[l++]);
    else
      v.push_back(b[r++]);
  }
  return move(v);
}
int leftMost(vi &v, int x, int a = -1, int b = -1){
  if(a == -1) a = 0, b = v.size();
  if(a+1 == b) return a;
  int m = (a + b)/2;
  if(v[m] < x) return leftMost(v, x, m, b);
  return leftMost(v, x, a, m);
}
int rightMost(vi &v, int x, int a = -1, int b = -1){
  if(a == -1) a = 0, b = v.size();
  if(a == b) return a;
  int m = (a + b)/2;
  if(v[m] > x) return rightMost(v, x, a, m);
  return rightMost(v, x, m+1, b);
}
constexpr int right(int i){ return i*2 + 2; }
constexpr int left(int i){ return i*2 + 1; }
constexpr int dad(int i){ return (i-1)/2; }
struct mt_node{
  vi v;
  mt_node(){}
  void build(int i, vi &source, vector<mt_node> &tree){
    int pos = i - source.size() + 1;
    if(pos >= 0)
      v.push_back(source[pos]);
    else
      v = merge(tree[left(i)].v, tree[right(i)].v);
  }
  int count(int k){
    int l = leftMost(v, k);
    int r = rightMost(v, k);
    return max(0 , r-l-1);
  }
};
struct mergetree{
  vector<mt_node> v;
  mergetree(vi &source){
    v.resize((1<<(int)(ceil(log2(source.size())+1)))-1);
    for(int i = source.size()*2 - 2; i >= 0; i--)
      v[i].build(i, source, v);
  }
  const int nullQuery = 0;
  int qa, qb, qx;
  int count(int i, int l, int r){
    if(l >= qa && r <= qb) return v[i].count(qx);
    if(l > qb || r < qa) return nullQuery;
    int m = (l+r)/2;
    return count(left(i), l, m) + count(right(i), m+1, r);
  }
  int query(int a, int b, int x){
    qa = a, qb = b, qx = x;
    int l = 0, r = v.size()/2;
    return count(0, l, r);
  }
};
```

1.10. ***k*-d Tree.** A *k*-dimensional tree supporting fast construction, adding points, and nearest neighbor queries.

```
#define INC(c) ((c) == K - 1 ? 0 : (c) + 1)
template <int K> struct kd_tree {
  struct pt {
    double coord[K];
    pt() {}
    pt(double c[K]) { rep(i,0,K) coord[i] = c[i]; }
    double dist(const pt &other) const {
      double sum = 0.0;
      rep(i,0,K) sum += pow(coord[i] - other.coord[i], 2.0);
      return sqrt(sum); } };
  struct cmp {
    int c;
    cmp(int _c) : c(_c) {}
    bool operator ()(const pt &a, const pt &b) {
      for (int i = 0, cc; i <= K; i++) {
        cc = i == 0 ? c : i - 1;
        if (abs(a.coord[cc] - b.coord[cc]) > EPS)
          return a.coord[cc] < b.coord[cc];
      }
      return false; } };
  struct bb {
    pt from, to;
    bb(pt _from, pt _to) : from(_from), to(_to) {}
    double dist(const pt &p) {
      double sum = 0.0;
      rep(i,0,K) {
        if (p.coord[i] < from.coord[i])
          sum += pow(from.coord[i] - p.coord[i], 2.0);
        else if (p.coord[i] > to.coord[i])
          sum += pow(p.coord[i] - to.coord[i], 2.0);
      }
      return sqrt(sum); }
    bb bound(double l, int c, bool left) {
      pt nf(from.coord), nt(to.coord);
      if (left) nt.coord[c] = min(nt.coord[c], l);
      else nf.coord[c] = max(nf.coord[c], l);
      return bb(nf, nt); } };
  struct node {
    pt p; node *l, *r;
    node(pt _p, node *_l, node *_r)
      : p(_p), l(_l), r(_r) {  } };
  node *root;
  // kd_tree() : root(NULL) { }
  kd_tree(vector<pt> pts) {
    root = construct(pts, 0, size(pts) - 1, 0); }
  node* construct(vector<pt> &pts, int from, int to, int c) {
    if (from > to) return NULL;
    int mid = from + (to - from) / 2;
    nth_element(pts.begin() + from, pts.begin() + mid,
          pts.begin() + to + 1, cmp(c));
    return new node(pts[mid],
            construct(pts, from, mid - 1, INC(c)),
            construct(pts, mid + 1, to, INC(c))); }
  bool contains(const pt &p) { return _con(p, root, 0); }
  bool _con(const pt &p, node *n, int c) {
    if (!n) return false;
    if (cmp(c)(p, n->p)) return _con(p, n->l, INC(c));
    if (cmp(c)(n->p, p)) return _con(p, n->r, INC(c));
    return true; }
  void insert(const pt &p) { _ins(p, root, 0); }
  void _ins(const pt &p, node* &n, int c) {
    if (!n) n = new node(p, NULL, NULL);
    else if (cmp(c)(p, n->p)) _ins(p, n->l, INC(c));
    else if (cmp(c)(n->p, p)) _ins(p, n->r, INC(c)); }
  void clear() { _clr(root); root = NULL; }
  void _clr(node *n) {
    if (n) _clr(n->l), _clr(n->r), delete n; }
  pair<pt, bool> nearest_neighbour(const pt &p,
      bool allow_same=true) {
    double mn = INFINITY, cs[K];
    rep(i,0,K) cs[i] = -INFINITY;
    pt from(cs);
    rep(i,0,K) cs[i] = INFINITY;
    pt to(cs), resp;
    _nn(p, root, bb(from, to), mn, resp, 0, allow_same);
    return make_pair(resp, !std::isinf(mn)); }
  void _nn(const pt &p, node *n, bb b,
      double &mn, pt &resp, int c, bool same) {
    if (!n || b.dist(p) > mn) return;
    bool l1 = true, l2 = false;
    if ((same || p.dist(n->p) > EPS) && p.dist(n->p) < mn)
      mn = p.dist(resp = n->p);
    node *n1 = n->l, *n2 = n->r;
    rep(i,0,2) {
      if (i == 1 || cmp(c)(n->p, p)) swap(n1,n2),swap(l1,l2);
      _nn(p, n1, b.bound(n->p.coord[c], c, l1), mn,
          resp, INC(c), same); } } };
```

1.11. Sparse Table.

```
struct sparse_table { vvi m;
  sparse_table(vi arr) {
    m.push_back(arr);
    for (int k = 0; (1<<(++k)) <= size(arr); ) {
      m.push_back(vi(size(arr)-(1<<k)+1));
      rep(i,0,size(arr)-(1<<k)+1)
        m[k][i] = min(m[k-1][i], m[k-1][i+(1<<(k-1))]); } }
  int query(int l, int r) {
    int k = 0; while (1<<(k+1) <= r-l+1) k++;
    return min(m[k][l], m[k][r-(1<<k)+1]); } };
```

1.12. Minque.

```
class minqueue {
  deque<pair<int,int>> q;
  int l = 0, r = 0;
  void push(int val){
    while(q.size() && q.back().first >= val)
      q.pop_back();
    q.push_back({val, r++});
  }
  void pop(){
    if(q.front().second == l)
      q.pop_front();
    l++;
  }
  int get_min(){ return q.front().first; }
};
```

2. Graphs

2.1. Single-Source Shortest Paths.

2.1.1. *Dijkstra's algorithm.* It runs in $\Theta(|E| \log |V|)$ time.

```
struct link {
  int node;
  long long weight;
  bool operator()(link &a, link &b) { return a.weight > b.weight; }
};
void dijkstra(int s, vector<vector<link>> &G) {
  vector<int> dist(G.size(), INF);    // local/global
  priority_queue<link, vector<link>, link> PQ;
  PQ.push({s, 0});
  dist[s] = 0;
  while(!PQ.empty()){
    link u = PQ.top(); PQ.pop();
    if(u.weight > dist[u.node]) continue;
    for(link &v : G[u.node]) {
      if(dist[v.node] > dist[u.node] + v.weight){
        dist[v.node] = dist[u.node] + v.weight;
        PQ.push({v.node, dist[v.node]});
      }
    }
  }
}
```

2.1.2. *SPFA.*

```
vi spfa(graph &g, int s){
  // check if node already in queue
  queue<int> q;
  q.push(s);
  vi dist(g.size(), inf);
  dist[s] = 0;
  while(!q.empty()){
    int current = q.front();
    q.pop();
    for(edge &e : g[curr])
      if(dist[e.to] > dist[curr] + e.weight){
        dist[e.to] = dist[curr] + e.weight;
        q.push(e.to);
      }
  } return dist;
}
```

2.2. All-Pairs Shortest Paths.

2.2.1. *Floyd-Warshall algorithm.* The Floyd-Warshall algorithm solves the all-pairs shortest paths problem in $O(|V|^3)$ time.

```
void floyd_warshall(vvi &adjMatrix, vvi &dist){
  for(int i = 0; i < adjMatrix.size(); i++) {
    dist[i][i] = 0;
    for(int j = 0; j < adjMatrix.size(); j++)
      if (adjMatrix[i][j] != -1)
        dist[i][j] = adjMatrix[i][j];
      else if(i != j)
        dist[i][j] = INF;
  }
  for (int i = 0; i < adjMatrix.size(); i++){
    for (int j = 0; j < adjMatrix.size(); j++) if (j != i){
      for (int k = 0; k < adjMatrix.size(); k++) {
        dist[j][k] = min(dist[j][k], dist[j][i] + dist[i][k]);
      }
    }
  }
}
```

2.3. Strongly Connected Components.

2.3.1. *Kosaraju's algorithm.* Kosarajus's algorithm finds strongly connected components of a directed graph in $O(|V| + |E|)$ time. Returns a Union-Find of the SCCs, as well as a topological ordering of the SCCs. Note that the ordering specifies a random element from each SCC, not the UF parents!

```
void dfs_order(vvi &g, int i, vb &vis, vi &dfso){
  vis[i] = 1;
  for(int &p : g[i])
    if(!vis[p])
      dfs_order(g, p, vis, dfso);
  dfso.push_back(i);
}
void l_dfs(vvi &g, int i, vb &vis, vi &labels, int &l){
  if(!vis[i]){
    vis[i] = 1;
    ++l;
  }
  labels[i] = l;
  for(int &x : g[i])
    if(!vis[x]){
      vis[x] = 1;
    l_dfs(g, x, vis, labels, l);
  }
}q
vvi kosaraju(vvi &g, vi &labels){
  //we make a stack based on dfs order (in a way that no node is\
    before someone reachable by him - except for SCC case!
  vi dfso; dfso.reserve(g.size());
  vb vis(g.size());
  for(int i = 0; i < g.size(); i++)
    if(!vis[i])
      dfs_order(g, i, vis, dfso);
  //we need the reverse graph
  vvi rg(g.size());
  for(int i = 0; i < g.size(); i++)
    for(int &p : g[i])
      rg[p].push_back(i);
  //now, we just go backwards from last note visited and build the scc
  fill(vis.begin(), vis.end(), 0);
  int l = -1;
  for(int i = g.size() - 1; i >= 0; i--)
    if(!vis[dfso[i]])
      l_dfs(rg, dfso[i], vis, labels, l);
  //labels are set, we construct the graph
  vvi scc(l+1);//l started as -1, so we add 1
  set<pii> edges;//to avoid multiple edges, consider dsu if TLE
  for(int i = 0; i < g.size(); i++){
    for(int &p : g[i])
      if(labels[i] != labels[p]){
        if(edges.count({labels[i], labels[p]}) == 0){
          scc[labels[i]].push_back(labels[p]);
          edges.insert({labels[i], labels[p]});
        }
      }
  }
  return scc;
}
```

2.4. **Cut Points and Bridges.**

```
#define MAXN 5000
int low[MAXN], num[MAXN], curnum;
void dfs(const vvi &adj, vi &cp, vii &bri, int u, int p) {
  low[u] = num[u] = curnum++;
  int cnt = 0; bool found = false;
  rep(i,0,size(adj[u])) {
    int v = adj[u][i];
    if (num[v] == -1) {
      dfs(adj, cp, bri, v, u);
      low[u] = min(low[u], low[v]);
      cnt++;
      found = found || low[v] >= num[u];
      if (low[v] > num[u]) bri.push_back(ii(u, v));
    } else if (p != v) low[u] = min(low[u], num[v]); }
  if (found && (p != -1 || cnt > 1)) cp.push_back(u); }
pair<vi,vii> cut_points_and_bridges(const vvi &adj) {
  int n = size(adj);
  vi cp; vii bri;
  memset(num, -1, n << 2);
  curnum = 0;
  rep(i,0,n) if (num[i] == -1) dfs(adj, cp, bri, i, -1);
  return make_pair(cp, bri); }
```

2.5. **Minimum Spanning Tree.**

2.5.1. *Kruskal's algorithm.*

```
#include "../data-structures/union_find.cpp"
vector<pair<int, ii> > mst(int n,
    vector<pair<int, ii> > edges) {
  union_find uf(n);
  sort(edges.begin(), edges.end());
  vector<pair<int, ii> > res;
  rep(i,0,size(edges))
    if (uf.find(edges[i].second.first) !=
        uf.find(edges[i].second.second)) {
      res.push_back(edges[i]);
      uf.unite(edges[i].second.first,
               edges[i].second.second); }
  return res; }
```

2.6. **Topological Sort.**

2.6.1. *Modified Depth-First Search.*

```
void tsort_dfs(int cur, char* color, const vvi& adj,
    stack<int>& res, bool& cyc) {
  color[cur] = 1;
  rep(i,0,size(adj[cur])) {
    int nxt = adj[cur][i];
    if (color[nxt] == 0)
      tsort_dfs(nxt, color, adj, res, cyc);
    else if (color[nxt] == 1)
      cyc = true;
    if (cyc) return; }
  color[cur] = 2;
  res.push(cur); }
vi tsort(int n, vvi adj, bool& cyc) {
  cyc = false;
  stack<int> S;
  vi res;
  char* color = new char[n];
  memset(color, 0, n);
  rep(i,0,n) {
    if (!color[i]) {
      tsort_dfs(i, color, adj, S, cyc);
      if (cyc) return res; } }
  while (!S.empty()) res.push_back(S.top()), S.pop();
  return res; }
```

2.7. **Euler Path.** Finds an euler path (or circuit) in a directed graph, or reports that none exist.

```
#define MAXV 1000
#define MAXE 5000
vi adj[MAXV];
int n, m, indeg[MAXV], outdeg[MAXV], res[MAXE + 1];
ii start_end() {
  int start = -1, end = -1, any = 0, c = 0;
  rep(i,0,n) {
    if (outdeg[i] > 0) any = i;
    if (indeg[i] + 1 == outdeg[i]) start = i, c++;
    else if (indeg[i] == outdeg[i] + 1) end = i, c++;
    else if (indeg[i] != outdeg[i]) return ii(-1,-1); }
  if ((start == -1) != (end == -1) || (c != 2 && c != 0))
    return ii(-1,-1);
  if (start == -1) start = end = any;
  return ii(start, end); }
bool euler_path() {
  ii se = start_end();
  int cur = se.first, at = m + 1;
  if (cur == -1) return false;
  stack<int> s;
  while (true) {
    if (outdeg[cur] == 0) {
      res[--at] = cur;
      if (s.empty()) break;
      cur = s.top(); s.pop();
    } else s.push(cur), cur = adj[cur][--outdeg[cur]]; }
  return at == 0; }
```

And an undirected version, which finds a cycle.

```
multiset<int> adj[1010];
list<int> L;
list<int>::iterator euler(int at, int to,
    list<int>::iterator it) {
  if (at == to) return it;
  L.insert(it, at), --it;
  while (!adj[at].empty()) {
    int nxt = *adj[at].begin();
    adj[at].erase(adj[at].find(nxt));
    adj[nxt].erase(adj[nxt].find(at));
    if (to == -1) {
      it = euler(nxt, at, it);
      L.insert(it, at);
      --it;
    } else {
      it = euler(nxt, to, it);
      to = -1; } }
  return it; }
// euler(0,-1,L.begin())
```

2.8. Bipartite Matching.

2.8.1. *Alternating Paths algorithm.* The alternating paths algorithm solves bipartite matching in $O(mn^2)$ time, where m, n are the number of vertices on the left and right side of the bipartite graph, respectively.

```
vi* adj;
bool* done;
int* owner;
int alternating_path(int left) {
  if (done[left]) return 0;
  done[left] = true;
  rep(i,0,size(adj[left])) {
    int right = adj[left][i];
    if (owner[right] == -1 ||
        alternating_path(owner[right])) {
      owner[right] = left; return 1; } }
  return 0; }
```

2.8.2. *Hopcroft-Karp algorithm.* An implementation of Hopcroft-Karp algorithm for bipartite matching. Running time is $O(|E|\sqrt{|V|})$.

```
#define MAXN 5000
int dist[MAXN+1], q[MAXN+1];
#define dist(v) dist[v == -1 ? MAXN : v]
struct bipartite_graph {
  int N, M, *L, *R; vi *adj;
  bipartite_graph(int _N, int _M) : N(_N), M(_M),
    L(new int[N]), R(new int[M]), adj(new vi[N]) {}
  ~bipartite_graph() { delete[] adj; delete[] L; delete[] R; }
  bool bfs() {
    int l = 0, r = 0;
    rep(v,0,N) if(L[v] == -1) dist(v) = 0, q[r++] = v;
      else dist(v) = INF;
    dist(-1) = INF;
    while(l < r) {
      int v = q[l++];
      if(dist(v) < dist(-1)) {
        iter(u, adj[v]) if(dist(R[*u]) == INF)
          dist(R[*u]) = dist(v) + 1, q[r++] = R[*u]; } }
    return dist(-1) != INF; }
  bool dfs(int v) {
    if(v != -1) {
      iter(u, adj[v])
        if(dist(R[*u]) == dist(v) + 1)
          if(dfs(R[*u])) {
            R[*u] = v, L[v] = *u;
            return true; }
      dist(v) = INF;
      return false; }
    return true; }
  void add_edge(int i, int j) { adj[i].push_back(j); }
  int maximum_matching() {
    int matching = 0;
    memset(L, -1, sizeof(int) * N);
    memset(R, -1, sizeof(int) * M);
    while(bfs()) rep(i,0,N)
      matching += L[i] == -1 && dfs(i);
    return matching; } };
```

2.8.3. *Minimum Vertex Cover in Bipartite Graphs.*

```cpp
#include "hopcroft_karp.cpp"
vector<bool> alt;
void dfs(bipartite_graph &g, int at) {
  alt[at] = true;
  iter(it,g.adj[at]) {
    alt[*it + g.N] = true;
    if (g.R[*it] != -1 && !alt[g.R[*it]]) dfs(g, g.R[*it]); } }
vi mvc_bipartite(bipartite_graph &g) {
  vi res; g.maximum_matching();
  alt.assign(g.N + g.M,false);
  rep(i,0,g.N) if (g.L[i] == -1) dfs(g, i);
  rep(i,0,g.N) if (!alt[i]) res.push_back(i);
  rep(i,0,g.M) if (alt[g.N + i]) res.push_back(g.N + i);
  return res; }
```

2.9. **Maximum Flow.**

2.9.1. ***Dinic's algorithm.*** An implementation of Dinic's algorithm that runs in $O(|V|^2|E|)$. It computes the maximum sflow of a flow network.

```cpp
#define MAXV 2000
int q[MAXV], d[MAXV];
struct flow_network {
  struct edge { int v, nxt, cap;
    edge(int _v, int _cap, int _nxt)
      : v(_v), nxt(_nxt), cap(_cap) { } };
  int n, *head, *curh; vector<edge> e, e_store;
  flow_network(int _n) : n(_n) {
    curh = new int[n];
    memset(head = new int[n], -1, n*sizeof(int)); }
  void reset() { e = e_store; }
  void add_edge(int u, int v, int uv, int vu=0) {
    e.push_back(edge(v, uv, head[u])); head[u] = size(e)-1;
    e.push_back(edge(u, vu, head[v])); head[v] = size(e)-1; }
  int augment(int v, int t, int f) {
    if (v == t) return f;
    for (int &i = curh[v], ret; i != -1; i = e[i].nxt)
      if (e[i].cap > 0 && d[e[i].v] + 1 == d[v])
        if ((ret = augment(e[i].v, t, min(f, e[i].cap))) > 0)
          return (e[i].cap -= ret, e[i^1].cap += ret, ret);
    return 0; }
  int max_flow(int s, int t, bool res=true) {
    e_store = e;
    int l, r, f = 0, x;
    while (true) {
      memset(d, -1, n*sizeof(int));
      l = r = 0, d[q[r++] = t] = 0;
      while (l < r)
        for (int v = q[l++], i = head[v]; i != -1; i=e[i].nxt)
          if (e[i^1].cap > 0 && d[e[i].v] == -1)
            d[q[r++] = e[i].v] = d[v]+1;
      if (d[s] == -1) break;
      memcpy(curh, head, n * sizeof(int));
      while ((x = augment(s, t, INF)) != 0) f += x; }
    if (res) reset();
    return f; } };
```

2.10. **Minimum Cost Maximum Flow.** An implementation of **Edmonds Karp's algorithm**, modified to find shortest path to augment each time (instead of just any path). It computes the maximum flow of a flow network, and when there are multiple maximum flows, finds the maximum flow with minimum cost. Running time is $O(|V|^2|E|\log|V|)$.

```cpp
#define MAXV 2000
int d[MAXV], p[MAXV], pot[MAXV];
struct cmp { bool operator ()(int i, int j) {
    return d[i] == d[j] ? i < j : d[i] < d[j]; } };
struct flow_network {
  struct edge { int v, nxt, cap, cost;
    edge(int _v, int _cap, int _cost, int _nxt)
      : v(_v), nxt(_nxt), cap(_cap), cost(_cost) { } };
  int n; vi head; vector<edge> e, e_store;
  flow_network(int _n) : n(_n), head(n,-1) { }
  void reset() { e = e_store; }
  void add_edge(int u, int v, int cost, int uv, int vu=0) {
    e.push_back(edge(v, uv, cost, head[u]));
    head[u] = size(e)-1;
    e.push_back(edge(u, vu, -cost, head[v]));
    head[v] = size(e)-1; }
  ii min_cost_max_flow(int s, int t, bool res=true) {
    e_store = e;
    memset(pot, 0, n*sizeof(int));
    rep(it,0,n-1) rep(i,0,size(e)) if (e[i].cap > 0)
      pot[e[i].v] =
        min(pot[e[i].v], pot[e[i^1].v] + e[i].cost);
    int v, f = 0, c = 0;
    while (true) {
      memset(d, -1, n*sizeof(int));
      memset(p, -1, n*sizeof(int));
      set<int, cmp> q;
      d[s] = 0; q.insert(s);
      while (!q.empty()) {
        int u = *q.begin();
        q.erase(q.begin());
        for (int i = head[u]; i != -1; i = e[i].nxt) {
          if (e[i].cap == 0) continue;
          int cd = d[u] + e[i].cost + pot[u] - pot[v = e[i].v];
          if (d[v] == -1 || cd < d[v]) {
            q.erase(v);
            d[v] = cd; p[v] = i;
            q.insert(v); } } }
      if (p[t] == -1) break;
      int at = p[t], x = INF;
      while (at != -1)
        x = min(x, e[at].cap), at = p[e[at^1].v];
      at = p[t], f += x;
      while (at != -1)
        e[at].cap -= x, e[at^1].cap += x, at = p[e[at^1].v];
      c += x * (d[t] + pot[t] - pot[s]);
      rep(i,0,n) if (p[i] != -1) pot[i] += d[i]; }
    if (res) reset();
    return ii(f, c); } };
```

2.11. All Pairs Maximum Flow.

2.11.1. *Gomory-Hu Tree.* An implementation of the Gomory-Hu Tree. The spanning tree is constructed using Gusfield's algorithm in $O(|V|^2)$ plus $|V|-1$ times the time it takes to calculate the maximum flow. If Dinic's algorithm is used to calculate the max flow, the running time is $O(|V|^3|E|)$. NOTE: Not sure if it works correctly with disconnected graphs.

```
#include "dinic.cpp"
bool same[MAXV];
pair<vii, vvi> construct_gh_tree(flow_network &g) {
  int n = g.n, v;
  vii par(n, ii(0, 0)); vvi cap(n, vi(n, -1));
  rep(s,1,n) {
    int l = 0, r = 0;
    par[s].second = g.max_flow(s, par[s].first, false);
    memset(d, 0, n * sizeof(int));
    memset(same, 0, n * sizeof(bool));
    d[q[r++] = s] = 1;
    while (l < r) {
      same[v = q[l++]] = true;
      for (int i = g.head[v]; i != -1; i = g.e[i].nxt)
        if (g.e[i].cap > 0 && d[g.e[i].v] == 0)
          d[q[r++] = g.e[i].v] = 1; }
    rep(i,s+1,n)
      if (par[i].first == par[s].first && same[i])
        par[i].first = s;
    g.reset(); }
  rep(i,0,n) {
    int mn = INF, cur = i;
    while (true) {
      cap[cur][i] = mn;
      if (cur == 0) break;
      mn = min(mn, par[cur].second), cur = par[cur].first; } }
  return make_pair(par, cap); }
int compute_max_flow(int s, int t, const pair<vii, vvi> &gh) {
  int cur = INF, at = s;
  while (gh.second[at][t] == -1)
    cur = min(cur, gh.first[at].second),
    at = gh.first[at].first;
  return min(cur, gh.second[at][t]); }
```

2.12. Heavy-Light Decomposition.

```
#include "../data-structures/segment_tree.cpp"
const int ID = 0;
int f(int a, int b) { return a + b; }
struct HLD {
  int n, curhead, curloc;
  vi sz, head, parent, loc;
  vvi adj; segment_tree values;
  HLD(int _n) : n(_n), sz(n, 1), head(n),
                parent(n, -1), loc(n), adj(n) {
    vector<ll> tmp(n, ID); values = segment_tree(tmp); }
  void add_edge(int u, int v) {
    adj[u].push_back(v); adj[v].push_back(u); }
  void update_cost(int u, int v, int c) {
    if (parent[v] == u) swap(u, v); assert(parent[u] == v);
    values.update(loc[u], c); }
  int csz(int u) {
    rep(i,0,size(adj[u])) if (adj[u][i] != parent[u])
      sz[u] += csz(adj[parent[adj[u][i]] = u][i]);
    return sz[u]; }
  void part(int u) {
    head[u] = curhead; loc[u] = curloc++;
    int best = -1;
    rep(i,0,size(adj[u]))
      if (adj[u][i] != parent[u] &&
          (best == -1 || sz[adj[u][i]] > sz[best]))
        best = adj[u][i];
    if (best != -1) part(best);
    rep(i,0,size(adj[u]))
      if (adj[u][i] != parent[u] && adj[u][i] != best)
        part(curhead = adj[u][i]); }
  void build(int r = 0) {
    curloc = 0, csz(curhead = r), part(r); }
  int lca(int u, int v) {
    vi uat, vat; int res = -1;
    while (u != -1) uat.push_back(u), u = parent[head[u]];
    while (v != -1) vat.push_back(v), v = parent[head[v]];
    u = size(uat) - 1, v = size(vat) - 1;
    while (u >= 0 && v >= 0 && head[uat[u]] == head[vat[v]])
      res = (loc[uat[u]] < loc[vat[v]] ? uat[u] : vat[v]),
      u--, v--;
    return res; }
  int query_upto(int u, int v) { int res = ID;
    while (head[u] != head[v])
      res = f(res, values.query(loc[head[u]], loc[u]).x),
      u = parent[head[u]];
    return f(res, values.query(loc[v] + 1, loc[u]).x); }
  int query(int u, int v) { int l = lca(u, v);
    return f(query_upto(u, l), query_upto(v, l)); } };
```

2.13. Least Common Ancestors, Binary Jumping.

```
struct node {
  node *p, *jmp[20];
  int depth;
  node(node *_p = NULL) : p(_p) {
    depth = p ? 1 + p->depth : 0;
    memset(jmp, 0, sizeof(jmp));
    jmp[0] = p;
    for (int i = 1; (1<<i) <= depth; i++)
      jmp[i] = jmp[i-1]->jmp[i-1]; } };
node* st[100000];
node* lca(node *a, node *b) {
  if (!a || !b) return NULL;
  if (a->depth < b->depth) swap(a,b);
  for (int j = 19; j >= 0; j--)
    while (a->depth - (1<<j) >= b->depth) a = a->jmp[j];
  if (a == b) return a;
  for (int j = 19; j >= 0; j--)
    while (a->depth >= (1<<j) && a->jmp[j] != b->jmp[j])
```

```
      a = a->jmp[j], b = b->jmp[j];
  return a->p; }
```

2.14. **Maximum Density Subgraph.** Given (weighted) undirected graph G. Binary search density. If g is current density, construct flow network: (S, u, m), $(u, T, m + 2g - d_u)$, $(u, v, 1)$, where m is a large constant (larger than sum of edge weights). Run floating-point max-flow. If minimum cut has empty S-component, then maximum density is smaller than g, otherwise it's larger. Distance between valid densities is at least $1/(n(n-1))$. Edge case when density is 0. This also works for weighted graphs by replacing d_u by the weighted degree, and doing more iterations (if weights are not integers).

2.15. **Maximum-Weight Closure.** Given a vertex-weighted directed graph G. Turn the graph into a flow network, adding weight ∞ to each edge. Add vertices S, T. For each vertex v of weight w, add edge (S, v, w) if $w \geq 0$, or edge $(v, T, -w)$ if $w < 0$. Sum of positive weights minus minimum $S - T$ cut is the answer. Vertices reachable from S are in the closure. The maximum-weight closure is the same as the complement of the minimum-weight closure on the graph with edges reversed.

2.16. **Maximum Weighted Independent Set in a Bipartite Graph.** This is the same as the minimum weighted vertex cover. Solve this by constructing a flow network with edges $(S, u, w(u))$ for $u \in L$, $(v, T, w(v))$ for $v \in R$ and (u, v, ∞) for $(u, v) \in E$. The minimum S, T-cut is the answer. Vertices adjacent to a cut edge are in the vertex cover.

2.17. **Max flow with lower bounds on edges.** Change edge $(u, v, l \leq f \leq c)$ to $(u, v, f \leq c - l)$. Add edge (t, s, ∞). Create super-nodes S, T. Let $M(u) = \sum_v l(v, u) - \sum_v l(u, v)$. If $M(u) < 0$, add edge $(u, T, -M(u))$, else add edge $(S, u, M(u))$. Max flow from S to T. If all edges from S are saturated, then we have a feasible flow. Continue running max flow from s to t in original graph.

3. Strings

3.1. **The Z algorithm.** Given a string S, $Z_i(S)$ is the longest substring of S starting at i that is also a prefix of S. The Z algorithm computes these Z values in $O(n)$ time, where $n = |S|$. Z values can, for example, be used to find all occurrences of a pattern P in a string T in linear time. This is accomplished by computing Z values of $S = PT$, and looking for all i such that $Z_i \geq |P|$.

```
int* z_values(const string &s) {
  int n = size(s);
  int* z = new int[n];
  int l = 0, r = 0;
  z[0] = n;
  rep(i,1,n) {
    z[i] = 0;
    if (i > r) {
      l = r = i;
      while (r < n && s[r - l] == s[r]) r++;
      z[i] = r - l; r--;
    } else if (z[i - l] < r - i + 1) z[i] = z[i - l];
    else {
      l = i;
      while (r < n && s[r - l] == s[r]) r++;
      z[i] = r - l; r--; } }
  return z; }
```

3.2. **Trie.** A Trie class.

```
template <class T>
struct trie {
  struct node {
    map<T, node*> children;
    int prefixes, words;
    node() { prefixes = words = 0; } };
  node* root;
  trie() : root(new node()) {  }
  template <class I>
  void insert(I begin, I end) {
    node* cur = root;
    while (true) {
      cur->prefixes++;
      if (begin == end) { cur->words++; break; }
      else {
        T head = *begin;
        typename map<T, node*>::const_iterator it;
        it = cur->children.find(head);
        if (it == cur->children.end()) {
          pair<T, node*> nw(head, new node());
          it = cur->children.insert(nw).first;
        } begin++, cur = it->second; } } }
  template<class I>
  int countMatches(I begin, I end) {
    node* cur = root;
    while (true) {
      if (begin == end) return cur->words;
      else {
        T head = *begin;
        typename map<T, node*>::const_iterator it;
        it = cur->children.find(head);
        if (it == cur->children.end()) return 0;
        begin++, cur = it->second; } } }
  template<class I>
  int countPrefixes(I begin, I end) {
    node* cur = root;
    while (true) {
      if (begin == end) return cur->prefixes;
      else {
        T head = *begin;
        typename map<T, node*>::const_iterator it;
        it = cur->children.find(head);
        if (it == cur->children.end()) return 0;
        begin++, cur = it->second; } } } };

//########################################################################
struct t_node {  //trie with O(NlogA) operations
  char c;
  map<char, t_node*> v;
  t_node (char c = 0) : c(c){}
};
//add with $ if perfect matching
struct trie{
  t_node root;
  void insert(string &s){
    t_node *current = &root;
    for(char c : s){
      if(current->v.count(c) == 0)
        current->v[c] = new t_node(c);
      current = current->v[c];
    }
  }
  bool find(string &s){
    t_node *current = &root;
    for(char c : s){
      if(current->v.count(c) == 0) return 0;
      current = current->v[c];
    }
    return 1;
  }
};
```

3.3. **Suffix Array.** An $O(n \log n)$ construction of a Suffix Tree.

```
//O(NlogN) suffixarray - add a char at the end
struct suffarray{
  vi sa;
  string &s;
  void radix_sort(vi &rank, int k){
    vi count(max(256, (int)sa.size()));
    for(int i : sa) count[((i + k) < rank.size() ? rank[i+k]+1 : 1)]++;
    int last = 0;
    for(int &i : count) i = last += i;
    vi temp(sa.size());
    for(int i : sa) temp[count[(i + k < rank.size() ? rank[i + k] : 0)]++] = i;
    fill(count.begin(), count.end(), 0);
    for(int &i : rank) count[i+1]++;
    last = 0;
    for(int &i : count) i = last += i;
    for(int i : temp) sa[count[rank[i]]++] = i;
  }
  void update_ranks(vi &rank, int k){
    vi old(rank);
    int r = rank[sa[0]] = 0;
    for(int i = 1; i < rank.size(); i++)
      rank[sa[i]] = (old[sa[i]] == old[sa[i-1]] && (sa[i] + k < old.size() ?
      old[sa[i]+k] : 0) <= (sa[i-1]+k < old.size() ? old[sa[i-1]+k] : 0)) ? r : ++r;
  }
  suffarray(string &s) : sa(s.size()), s(s){
    int n = s.size();
    for(int i = 0; i < n; i++)
      sa[i] = i;
    vi rank(s.size());
    for(int i = 0; i < s.size(); i++)
      rank[i] = s[i];
    for(int k = 1; k <= n; k<<=1){
      radix_sort(rank, k);
      update_ranks(rank, k);
      if(rank[sa.back()] == s.size()-1) break;
    }
  }
};
```

3.4. **LCP Kasai.** An $O(n)$ construction of a LCP array from suffix array.

```
//ignore the $ sign?
vi lcp_kasai(suffarray &x){
  vi &sa = x.sa;
  string &s = x.s;
  int n = sa.size();
  vi rev(n), lcp(n);
  for(int i = 1; i < n; i++)
    rev[sa[i]] = i;
  for(int i = 1, k = 0; i < n; i++){
    if(rev[i] == n-1) continue;
    int next = sa[rev[i]+1];
    while(i+k < n && next+k < n && s[i+k] == s[next+k])
      k++;
    lcp[rev[i]] = k;
    if(k) k--;
  }
  return lcp;
}
```

3.5. **Aho-Corasick Algorithm.** An implementation of the Aho-Corasick algorithm. Constructs a state machine from a set of keywords which can be used to search a string for any of the keywords.

```
//multiple pattern matching, care for "sonda"
struct node{
  node *parent, *fall, *output;
  char c = '$';
  bool ending = 0; //make this int for speed
  unordered_map<char, node*> f;
  node(node *p, char c) : parent(p), c(c){}
  node(){}
};
struct aho_trie{
  node *root = new node();
  void insert(string s){
    node *curr = root;
    for(char c : s)
      curr = curr->f.count(c) ? curr->f[c] : (curr->f[c] = new node(curr, c));
  }
  void set_falls(){
    queue<node*> q;
    q.push(root);
    while(!q.empty()){
      node *curr = q.front(); q.pop();
      for(auto &x : curr->f)
        q.push(x.second);
      if(curr == root) continue;
      curr->fall = curr->parent->fall;
      while(curr->fall->f.count(curr->c) == 0 && curr->fall != root)
        curr->fall = curr->fall->fall;
      if(curr->fall != curr->parent && curr->fall->f.count(curr->c))
        curr->fall = curr->fall->f[curr->c];
      curr->output = curr->fall->ending ? curr->fall : curr->fall->output;
    }
  }
  int query(string s){
    node *stato = root;
    int ans = 0;
    for(char c : s){
      while(stato != root && stato->f.count(c) == 0)
```

```
        stato = stato->fall;
      if(stato->f.count(c))
        stato = stato->f[c];
      node *sonda = stato->output;
      while(sonda != root)
        ans++, sonda = sonda->output;
      ans += stato->ending;
    }
    return ans;
  }
  aho_trie(){ root->parent = root->fall = root->output = root; }
};
```

3.6. eerTree.

eerTree. Constructs an eerTree in $O(n)$, one character at a time.

```
#define MAXN 100100
#define SIGMA 26
#define BASE 'a'
char *s = new char[MAXN];
struct state {
  int len, link, to[SIGMA];
} *st = new state[MAXN+2];
struct eertree {
  int last, sz, n;
  eertree() : last(1), sz(2), n(0) {
    st[0].len = st[0].link = -1;
    st[1].len = st[1].link = 0; }
  int extend() {
    char c = s[n++]; int p = last;
    while (n - st[p].len - 2 < 0 || c != s[n - st[p].len - 2])
      p = st[p].link;
    if (!st[p].to[c-BASE]) {
      int q = last = sz++;
      st[p].to[c-BASE] = q;
      st[q].len = st[p].len + 2;
      do { p = st[p].link;
      } while (p != -1 && (n < st[p].len + 2 ||
               c != s[n - st[p].len - 2]));
      if (p == -1) st[q].link = 1;
      else st[q].link = st[p].to[c-BASE];
      return 1; }
    last = st[p].to[c-BASE];
    return 0; } };
```

3.7. **Hashing.** Modulus should be a large prime. Can also use multiple instances with different moduli to minimize chance of collision.

```
struct hasher { int b = 311, m; vi h, p;
  hasher(string s, int _m)
    : m(_m), h(size(s)+1), p(size(s)+1) {
    p[0] = 1; h[0] = 0;
    rep(i,0,size(s)) p[i+1] = (ll)p[i] * b % m;
    rep(i,0,size(s)) h[i+1] = ((ll)h[i] * b + s[i]) % m; }
  int hash(int l, int r) {
    return (h[r+1] + m - (ll)h[l] * p[r-l+1] % m) % m; } };
```

4. Mathematics

4.1. **Binomial Coefficients.** The binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is the number of ways to choose k items out of a total of n items. Also contains an implementation of Lucas' theorem for computing the answer modulo a prime p. Use modular multiplicative inverse if needed, and be very careful of overflows.

```
int nck(int n, int k) {
  if (n < k) return 0;
  k = min(k, n - k);
  int res = 1;
  rep(i,1,k+1) res = res * (n - (k - i)) / i;
  return res; }
int nck(int n, int k, int p) {
  int res = 1;
  while (n || k) {
    res = nck(n % p, k % p) % p * res % p;
    n /= p, k /= p; }
  return res; }
```

4.2. **Euclidean algorithm.** The Euclidean algorithm computes the greatest common divisor of two integers a, b.

```
ll gcd(ll a, ll b) { return b == 0 ? a : gcd(b, a % b); }
```

The extended Euclidean algorithm computes the greatest common divisor d of two integers a, b and also finds two integers x, y such that $a \times x + b \times y = d$.

```
ll egcd(ll a, ll b, ll& x, ll& y) {
  if (b == 0) { x = 1; y = 0; return a; }
  ll d = egcd(b, a % b, x, y);
  x -= a / b * y; swap(x, y); return d; }
```

4.3. **Sieve of Eratosthenes.** An optimized implementation of Eratosthenes' Sieve.

```
vi prime_sieve(int n) {
  int mx = (n - 3) >> 1, sq, v, i = -1;
  vi primes;
  bool* prime = new bool[mx + 1];
  memset(prime, 1, mx + 1);
  if (n >= 2) primes.push_back(2);
  while (++i <= mx) if (prime[i]) {
    primes.push_back(v = (i << 1) + 3);
    if ((sq = i * ((i << 1) + 6) + 3) > mx) break;
    for (int j = sq; j <= mx; j += v) prime[j] = false; }
  while (++i <= mx)
    if (prime[i]) primes.push_back((i << 1) + 3);
  delete[] prime; // can be used for O(1) lookup
  return primes; }
```

4.4. **Divisor Sieve.** A O(n) prime sieve. Computes the smallest divisor of any number up to n.

```
vi divisor_sieve(int n) {
  vi mnd(n+1, 2), ps;
  if (n >= 2) ps.push_back(2);
  mnd[0] = 0;
  for (int k = 1; k <= n; k += 2) mnd[k] = k;
  for (int k = 3; k <= n; k += 2) {
    if (mnd[k] == k) ps.push_back(k);
    rep(i,1,size(ps))
```

```
      if (ps[i] > mnd[k] || ps[i]*k > n) break;
      else mnd[ps[i]*k] = ps[i]; }
  return ps; }
```

4.5. Modular Exponentiation.

A function to perform fast modular exponentiation.

```
template <class T>
T mod_pow(T b, T e, T m) {
  T res = T(1);
  while (e) {
    if (e & T(1)) res = smod(res * b, m);
    b = smod(b * b, m), e >>= T(1); }
  return res; }
```

4.6. Modular Multiplicative Inverse.

A function to find a modular multiplicative inverse. Alternatively use mod_pow(a,m-2,m) when m is prime.

```
#include "egcd.cpp"
ll mod_inv(ll a, ll m) {
  ll x, y, d = egcd(a, m, x, y);
  return d == 1 ? smod(x,m) : -1; }
```

A sieve version:

```
vi inv_sieve(int n, int p) {
  vi inv(n,1);
  rep(i,2,n) inv[i] = (p - (ll)(p/i) * inv[p%i] % p) % p;
  return inv; }
```

4.7. Primitive Root.

```
#include "mod_pow.cpp"
ll primitive_root(ll m) {
  vector<ll> div;
  for (ll i = 1; i*i <= m-1; i++) {
    if ((m-1) % i == 0) {
      if (i < m) div.push_back(i);
      if (m/i < m) div.push_back(m/i); } }
  rep(x,2,m) {
    bool ok = true;
    iter(it,div) if (mod_pow<ll>(x, *it, m) == 1) {
      ok = false; break; }
    if (ok) return x; }
  return -1; }
```

4.8. Summatory Phi.

The summatory phi function $\Phi(n) = \sum_{i=1}^{n} \phi(i)$. Let $L \approx (n \log \log n)^{2/3}$ and the algorithm runs in $O(n^{2/3})$.

```
#define N 10000000
ll sp[N];
unordered_map<ll,ll> mem;
ll sumphi(ll n) {
  if (n < N) return sp[n];
  if (mem.find(n) != mem.end()) return mem[n];
  ll ans = 0, done = 1;
  for (ll i = 2; i*i <= n; i++) ans += sumphi(n/i), done = i;
  for (ll i = 1; i*i <= n; i++)
    ans += sp[i] * (n/i - max(done, n/(i+1)));
  return mem[n] = n*(n+1)/2 - ans; }
void sieve() {
  for (int i = 1; i < N; i++) sp[i] = i;
  for (int i = 2; i < N; i++) {
    if (sp[i] == i) {
      sp[i] = i-1;
      for (int j = i+i; j < N; j += i) sp[j] -= sp[j] / i; }
    sp[i] += sp[i-1]; } }
```

4.9. Number of Integer Points under Line.

Count the number of integer solutions to $Ax+By \leq C$, $0 \leq x \leq n$, $0 \leq y$. In other words, evaluate the sum $\sum_{x=0}^{n} \left\lfloor \frac{C-Ax}{B} + 1 \right\rfloor$. To count all solutions, let $n = \left\lfloor \frac{c}{a} \right\rfloor$. In any case, it must hold that $C - nA \geq 0$. Be very careful about overflows.

```
ll floor_sum(ll n, ll a, ll b, ll c) {
  if (c == 0) return 1;
  if (c < 0) return 0;
  if (a % b == 0) return (n+1)*(c/b+1)-n*(n+1)/2*a/b;
  if (a >= b) return floor_sum(n,a%b,b,c)-a/b*n*(n+1)/2;
  ll t = (c-a*n+b)/b;
  return floor_sum((c-b*t)/b,b,a,c-b*t)+t*(n+1); }
```

4.10. Numbers and Sequences.

Some random prime numbers: 1031, 32771, 1048583, 33554467, 1073741827, 34359738421, 1099511627791, 35184372088891, 1125899906842679, 36028797018963971.

More random prime numbers: $10^3 + \{-9, -3, 9, 13\}$, $10^6 + \{-17, 3, 33\}$, $10^9 + \{7, 9, 21, 33, 87\}$.

Some maximal divisor counts:

840	32
720 720	240
735 134 400	1 344
963 761 198 400	6 720
866 421 317 361 600	26 880
897 612 484 786 617 600	103 680

4.11. Game Theory.

Useful identity:

$$\oplus_{x=0}^{a-1} x = [0, a-1, 1, a][a\%4]$$

5. Geometry

5.1. Primitives.

Geometry primitives.

```
#define P(p) const point &p
#define L(p0, p1) P(p0), P(p1)
#define C(p0, r) P(p0), double r
#define PP(pp) pair<point,point> &pp
typedef complex<double> point;
double dot(P(a), P(b)) { return real(conj(a) * b); }
double cross(P(a), P(b)) { return imag(conj(a) * b); }
point rotate(P(p), double radians = pi / 2,
             P(about) = point(0,0)) {
  return (p - about) * exp(point(0, radians)) + about; }
point reflect(P(p), L(about1, about2)) {
  point z = p - about1, w = about2 - about1;
  return conj(z / w) * w + about1; }
point proj(P(u), P(v)) { return dot(u, v) / dot(u, u) * u; }
point normalize(P(p), double k = 1.0) {
  return abs(p) == 0 ? point(0,0) : p / abs(p) * k; }
double ccw(P(a), P(b), P(c)) { return cross(b - a, c - b); }
bool collinear(P(a), P(b), P(c)) {
  return abs(ccw(a, b, c)) < EPS; }
double angle(P(a), P(b), P(c)) {
  return acos(dot(b - a, c - b) / abs(b - a) / abs(c - b)); }
double signed_angle(P(a), P(b), P(c)) {
  return asin(cross(b - a, c - b) / abs(b - a) / abs(c - b)); }
double angle(P(p)) { return atan2(imag(p), real(p)); }
point perp(P(p)) { return point(-imag(p), real(p)); }
double progress(P(p), L(a, b)) {
  if (abs(real(a) - real(b)) < EPS)
    return (imag(p) - imag(a)) / (imag(b) - imag(a));
  else return (real(p) - real(a)) / (real(b) - real(a)); }
```

5.2. **Lines.** Line related functions.

```
#include "primitives.cpp"
bool collinear(L(a, b), L(p, q)) {
  return abs(ccw(a, b, p)) < EPS && abs(ccw(a, b, q)) < EPS; }
bool parallel(L(a, b), L(p, q)) {
  return abs(cross(b - a, q - p)) < EPS; }
point closest_point(L(a, b), P(c), bool segment = false) {
  if (segment) {
    if (dot(b - a, c - b) > 0) return b;
    if (dot(a - b, c - a) > 0) return a;
  }
  double t = dot(c - a, b - a) / norm(b - a);
  return a + t * (b - a); }
double line_segment_distance(L(a,b), L(c,d)) {
  double x = INFINITY;
  if (abs(a - b) < EPS && abs(c - d) < EPS) x = abs(a - c);
  else if (abs(a - b) < EPS)
    x = abs(a - closest_point(c, d, a, true));
  else if (abs(c - d) < EPS)
    x = abs(c - closest_point(a, b, c, true));
  else if ((ccw(a, b, c) < 0) != (ccw(a, b, d) < 0) &&
       (ccw(c, d, a) < 0) != (ccw(c, d, b) < 0)) x = 0;
  else {
    x = min(x, abs(a - closest_point(c,d, a, true)));
    x = min(x, abs(b - closest_point(c,d, b, true)));
    x = min(x, abs(c - closest_point(a,b, c, true)));
    x = min(x, abs(d - closest_point(a,b, d, true)));
  }
  return x; }
bool intersect(L(a,b), L(p,q), point &res, bool seg=false) {
  // NOTE: check parallel/collinear before
  point r = b - a, s = q - p;
  double c = cross(r, s),
         t = cross(p - a, s) / c, u = cross(p - a, r) / c;
  if (seg &&
      (t < 0-EPS || t > 1+EPS || u < 0-EPS || u > 1+EPS))
    return false;
  res = a + t * r;
  return true; }
```

5.3. **Circles.** Circle related functions.

```
#include "lines.cpp"
int intersect(C(A, rA), C(B, rB), point &r1, point &r2) {
  double d = abs(B - A);
  if ((rA + rB) <  (d - EPS) || d < abs(rA - rB) - EPS)
    return 0;
  double a = (rA*rA - rB*rB + d*d) / 2 / d,
         h = sqrt(rA*rA - a*a);
  point v = normalize(B - A, a),
        u = normalize(rotate(B-A), h);
  r1 = A + v + u, r2 = A + v - u;
  return 1 + (abs(u) >= EPS); }
int intersect(L(A, B), C(O, r), point &r1, point &r2) {
  point H = proj(B-A, O-A) + A; double h = abs(H-O);
  if (r < h - EPS) return 0;
  point v = normalize(B-A, sqrt(r*r - h*h));
  r1 = H + v, r2 = H - v;
  return 1 + (abs(v) > EPS); }
int tangent(P(A), C(O, r), point &r1, point &r2) {
  point v = O - A; double d = abs(v);
  if (d < r - EPS) return 0;
  double alpha = asin(r / d), L = sqrt(d*d - r*r);
  v = normalize(v, L);
  r1 = A + rotate(v, alpha), r2 = A + rotate(v, -alpha);
  return 1 + (abs(v) > EPS); }
void tangent_outer(point A, double rA,
                   point B, double rB, PP(P), PP(Q)) {
  if (rA - rB > EPS) { swap(rA, rB); swap(A, B); }
  double theta = asin((rB - rA)/abs(A - B));
  point v = rotate(B - A, theta + pi/2),
        u = rotate(B - A, -(theta + pi/2));
  u = normalize(u, rA);
  P.first = A + normalize(v, rA);
  P.second = B + normalize(v, rB);
  Q.first = A + normalize(u, rA);
  Q.second = B + normalize(u, rB); }
```

5.4. **Polygon.** Polygon primitives.

```
#include "primitives.cpp"
typedef vector<point> polygon;
double polygon_area_signed(polygon p) {
  double area = 0; int cnt = size(p);
  rep(i,1,cnt-1) area += cross(p[i] - p[0], p[i + 1] - p[0]);
  return area / 2; }
double polygon_area(polygon p) {
  return abs(polygon_area_signed(p)); }
#define CHK(f,a,b,c) \
    (f(a) < f(b) && f(b) <= f(c) && ccw(a,c,b) < 0)
int point_in_polygon(polygon p, point q) {
  int n = size(p); bool in = false; double d;
  for (int i = 0, j = n - 1; i < n; j = i++)
    if (collinear(p[i], q, p[j]) &&
      0 <= (d = progress(q, p[i], p[j])) && d <= 1)
      return 0;
  for (int i = 0, j = n - 1; i < n; j = i++)
    if (CHK(real, p[i], q, p[j]) || CHK(real, p[j], q, p[i]))
      in = !in;
  return in ? -1 : 1; }
// pair<polygon, polygon> cut_polygon(const polygon &poly,
//                                    point a, point b) {
```

```
//     polygon left, right;
//     point it(-100, -100);
//     for (int i = 0, cnt = poly.size(); i < cnt; i++) {
//         int j = i == cnt-1 ? 0 : i + 1;
//         point p = poly[i], q = poly[j];
//         if (ccw(a, b, p) <= 0) left.push_back(p);
//         if (ccw(a, b, p) >= 0) right.push_back(p);
//         // myintersect = intersect where
//         // (a,b) is a line, (p,q) is a line segment
//         if (myintersect(a, b, p, q, it))
//             left.push_back(it), right.push_back(it); }
//     return pair<polygon, polygon>(left, right); }
```

5.5. **Convex Hull.** An algorithm that finds the Convex Hull of a set of points. NOTE: Doesn't work on some weird edge cases. (A small case that included three collinear lines would return the same point on both the upper and lower hull.)

```
#include "polygon.cpp"
#define MAXN 1000
point hull[MAXN];
bool cmp(const point &a, const point &b) {
  return abs(real(a) - real(b)) > EPS ?
    real(a) < real(b) : imag(a) < imag(b); }
int convex_hull(polygon p) {
  int n = size(p), l = 0;
  sort(p.begin(), p.end(), cmp);
  rep(i,0,n) {
    if (i > 0 && p[i] == p[i - 1]) continue;
    while (l >= 2 &&
        ccw(hull[l - 2], hull[l - 1], p[i]) >= 0) l--;
    hull[l++] = p[i]; }
  int r = l;
  for (int i = n - 2; i >= 0; i--) {
    if (p[i] == p[i + 1]) continue;
    while (r - l >= 1 &&
        ccw(hull[r - 2], hull[r - 1], p[i]) >= 0) r--;
    hull[r++] = p[i]; }
  return l == 1 ? 1 : r - 1; }
```

Another Convex Hull implementation

```
//Andrew monotonechain + some geometry things
struct point {
  int x, y;
  bool operator ==(point &other){ return x == other.x && y == other.y;}
};
/* < 0 -> counterclockwise, = 0 -> collinear, > 0 -> clockwise */
ll orientation(point &a, point &b, point &c){
  return (b.y - a.y) * (ll)(c.x - b.x) - (b.x - a.x) * (ll)(c.y - b.y);
}
struct segment {
  point a, b;
  bool contains(point &p){
    if(orientation(a, b, p) != 0)
      return 0;
    bool x = p.x >= a.x && p.x <= b.x;
    bool y = p.y >= min(a.y, b.y) && p.y <= max(a.y, b.y);
    // bool x = unsigned(p.x - a.x) <= b.x - a.x;
    // bool y = unsigned(p.y - a.y) <= b.y - a.y;
    return x && y;
  }
  bool operator& (segment &s){
    return intersect(s);
  }
  bool intersect(segment &s){
    ll o1 = orientation(a, b, s.a);
    ll o2 = orientation(a, b, s.b);
    ll o3 = orientation(s.a, s.b, a);
    ll o4 = orientation(s.a, s.b, b);
    o1 = o1 > 0 ? 1 : (o1 < 0 ? -1 : 0);
    o2 = o2 > 0 ? 1 : (o2 < 0 ? -1 : 0);
    o3 = o3 > 0 ? 1 : (o3 < 0 ? -1 : 0);
    o4 = o4 > 0 ? 1 : (o4 < 0 ? -1 : 0);
    //if they obv intersect
    if(o1 != o2 && o3 != o4) return 1;
    return contains(s.a) || contains(s.b) || s.contains(a) || s.contains(b);
  }

  segment(point p1, point p2) : a(p1), b(p2){
    if(a.x > b.x)
      swap(a, b);
  }
};
//make a vector of point and voila
struct convex_hull : vector<point>{
  vector<point> &hull = *this;
  vector<segment> segs;
  convex_hull (vector<point> &v){
    point lowest = *min_element(v.begin(), v.end(),
      [](point &a, point &b){return tie(a.y, a.x) < tie(b.y, b.x);});
    sort(v.begin(), v.end(), [&lowest](point &a, point &b){
      ll o = orientation(lowest, a, b);
      if(o == 0) return a.x < b.x;
      return o < 0;
    });

    hull.push_back(v[0]), hull.push_back(v[1]);
    for(int i = 2; i < v.size(); i++){
      //we try to insert v[i]
      while(hull.size() >= 2){
        point &a = hull[hull.size() - 2], &b = hull.back();
        if(orientation(a, b, v[i]) >= 0)
          hull.pop_back();
        else
          break;
      }
      hull.push_back(v[i]);
    }
    segs.reserve(hull.size());
    segs.push_back(segment(hull.front(), hull.back()));
    for(int i = 0; i + 1 < hull.size(); i++)
      segs.push_back(segment(hull[i], hull[i+1]));
    sort(segs.begin(), segs.end(), [](segment &a, segment &b){
      return tie(a.a.x, a.b.x) < tie(b.a.x, b.b.x);
    });
  }
  //we also count point on the borders
  bool contains(point &p){
    int l = - 1, r = segs.size()-1, m;
    while(l <= r){
      segment &s = segs[m = (l + r)/2 + 1];
      if(p.x < s.a.x) l = m;
      if(p.x > s.b.x) r = m - 1;
    }
    segment line(p, {p.x, INF});
    vector<segments> xx;
    for(int i = lb + 1; i < segs.size() && segs[i].b.x <= p[x]; i++)
      if(segs[i].intersect(line)) xx.push_back(segs[i]);
    return 0;
  }
};
```

5.6. **Line Segment Intersection.** Computes the intersection between two line segments.

```
#include "lines.cpp"
bool line_segment_intersect(L(a, b), L(c, d), point &A,
                                               point &B) {
  if (abs(a - b) < EPS && abs(c - d) < EPS) {
    A = B = a; return abs(a - d) < EPS; }
  else if (abs(a - b) < EPS) {
    A = B = a; double p = progress(a, c,d);
    return 0.0 <= p && p <= 1.0
      && (abs(a - c) + abs(d - a) - abs(d - c)) < EPS; }
  else if (abs(c - d) < EPS) {
    A = B = c; double p = progress(c, a,b);
    return 0.0 <= p && p <= 1.0
      && (abs(c - a) + abs(b - c) - abs(b - a)) < EPS; }
  else if (collinear(a,b, c,d)) {
    double ap = progress(a, c,d), bp = progress(b, c,d);
    if (ap > bp) swap(ap, bp);
    if (bp < 0.0 || ap > 1.0) return false;
    A = c + max(ap, 0.0) * (d - c);
    B = c + min(bp, 1.0) * (d - c);
    return true; }
  else if (parallel(a,b, c,d)) return false;
  else if (intersect(a,b, c,d, A, true)) {
    B = A; return true; }
  return false; }
```

5.7. **Great-Circle Distance.** Computes the distance between two points (given as latitude/longitude coordinates) on a sphere of radius r.

```
double gc_distance(double pLat, double pLong,
         double qLat, double qLong, double r) {
  pLat *= pi / 180; pLong *= pi / 180;
  qLat *= pi / 180; qLong *= pi / 180;
  return r * acos(cos(pLat) * cos(qLat) * cos(pLong - qLong) +
         sin(pLat) * sin(qLat)); }
```

5.8. **Triangle Circumcenter.** Returns the unique point that is the same distance from all three points. It is also the center of the unique circle that goes through all three points.

```
#include "primitives.cpp"
point circumcenter(point a, point b, point c) {
  b -= a, c -= a;
  return a +
    perp(b * norm(c) - c * norm(b)) / 2.0 / cross(b, c); }
```

5.9. **Closest Pair of Points.** A sweep line algorithm for computing the distance between the closest pair of points.

```
#include "primitives.cpp"

struct cmpx { bool operator ()(const point &a,
                               const point &b) {
    return abs(real(a) - real(b)) > EPS ?
      real(a) < real(b) : imag(a) < imag(b); } };
struct cmpy { bool operator ()(const point &a,
                               const point &b) {
  return abs(imag(a) - imag(b)) > EPS ?
      imag(a) < imag(b) : real(a) < real(b); } };
double closest_pair(vector<point> pts) {
  sort(pts.begin(), pts.end(), cmpx());
  set<point, cmpy> cur;
  set<point, cmpy>::const_iterator it, jt;
  double mn = INFINITY;
  for (int i = 0, l = 0; i < size(pts); i++) {
    while (real(pts[i]) - real(pts[l]) > mn)
      cur.erase(pts[l++]);
    it = cur.lower_bound(point(-INFINITY, imag(pts[i]) - mn));
    jt = cur.upper_bound(point(INFINITY, imag(pts[i]) + mn));
    while (it != jt) mn = min(mn, abs(*it - pts[i])), it++;
    cur.insert(pts[i]); }
  return mn; }
```

5.10. **Line upper/l ower envelope.** To find the upper/lower envelope of a collection of lines $a_i + b_i x$, plot the points (b_i, a_i), add the point $(0, \pm\infty)$ (depending on if upper/lower envelope is desired), and then find the convex hull.

5.11. **Formulas.** Let $a = (a_x, a_y)$ and $b = (b_x, b_y)$ be two-dimensional vectors.

- $a \cdot b = |a||b| \cos\theta$, where θ is the angle between a and b.
- $a \times b = |a||b| \sin\theta$, where θ is the signed angle between a and b.
- $a \times b$ is equal to the area of the parallelogram with two of its sides formed by a and b. Half of that is the area of the triangle formed by a and b.
- **Euler's formula:** $V - E + F = 2$
- Side lengths a, b, c can form a triangle iff. $a + b > c$, $b + c > a$ and $a + c > b$.
- Sum of internal angles of a regular convex n-gon is $(n - 2)\pi$.
- **Law of sines:** $\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$
- **Law of cosines:** $b^2 = a^2 + c^2 - 2ac\cos B$
- Internal tangents of circles $(c_1, r_1), (c_2, r_2)$ intersect at $(c_1 r_2 + c_2 r_1)/(r_1 + r_2)$, external intersect at $(c_1 r_2 - c_2 r_1)/(r_1 + r_2)$.

6. Other Algorithms

6.1. **2SAT.** A fast 2SAT solver.

```
struct { vi adj; int val, num, lo; bool done; } V[2*1000+100];
struct TwoSat {
  int n, at = 0; vi S;
  TwoSat(int _n) : n(_n) {
    rep(i,0,2*n+1)
      V[i].adj.clear(),
      V[i].val = V[i].num = -1, V[i].done = false; }
  bool put(int x, int v) {
    return (V[n+x].val &= v) != (V[n-x].val &= 1-v); }
  void add_or(int x, int y) {
    V[n-x].adj.push_back(n+y), V[n-y].adj.push_back(n+x); }
  int dfs(int u) {
    int br = 2, res;
    S.push_back(u), V[u].num = V[u].lo = at++;
    iter(v,V[u].adj) {
      if (V[*v].num == -1) {
        if (!(res = dfs(*v))) return 0;
        br |= res, V[u].lo = min(V[u].lo, V[*v].lo);
      } else if (!V[*v].done)
        V[u].lo = min(V[u].lo, V[*v].num);
      br |= !V[*v].val; }
    res = br - 3;
    if (V[u].num == V[u].lo) rep(i,res+1,2) {
      for (int j = size(S)-1; ; j--) {
        int v = S[j];
        if (i) {
          if (!put(v-n, res)) return 0;
          V[v].done = true, S.pop_back();
        } else res &= V[v].val;
        if (v == u) break; }
      res &= 1; }
    return br | !res; }
  bool sat() {
    rep(i,0,2*n+1)
      if (i != n && V[i].num == -1 && !dfs(i)) return false;
    return true; } };
```

6.2. **Simplex.**

```
typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
const DOUBLE EPS = 1e-9;
struct LPSolver {
 int m, n;
 VI B, N;
 VVD D;
 LPSolver(const VVD &A, const VD &b, const VD &c) :
  m(b.size()), n(c.size()),
  N(n + 1), B(m), D(m + 2, VD(n + 2)) {
  for (int i = 0; i < m; i++) for (int j = 0; j < n; j++)
    D[i][j] = A[i][j];
  for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1;
    D[i][n + 1] = b[i]; }
  for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
  N[n] = -1; D[m + 1][n] = 1; }
 void Pivot(int r, int s) {
  double inv = 1.0 / D[r][s];
  for (int i = 0; i < m + 2; i++) if (i != r)
   for (int j = 0; j < n + 2; j++) if (j != s)
    D[i][j] -= D[r][j] * D[i][s] * inv;
  for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] *= inv;
  for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] *= -inv;
  D[r][s] = inv;
  swap(B[r], N[s]); }
 bool Simplex(int phase) {
  int x = phase == 1 ? m + 1 : m;
  while (true) {
   int s = -1;
   for (int j = 0; j <= n; j++) {
    if (phase == 2 && N[j] == -1) continue;
    if (s == -1 || D[x][j] < D[x][s] ||
        D[x][j] == D[x][s] && N[j] < N[s]) s = j; }
   if (D[x][s] > -EPS) return true;
   int r = -1;
   for (int i = 0; i < m; i++) {
    if (D[i][s] < EPS) continue;
    if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] /
        D[r][s] || (D[i][n + 1] / D[i][s]) == (D[r][n + 1] /
        D[r][s]) && B[i] < B[r]) r = i; }
   if (r == -1) return false;
   Pivot(r, s); } }
 DOUBLE Solve(VD &x) {
  int r = 0;
  for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1])
    r = i;
  if (D[r][n + 1] < -EPS) {
   Pivot(r, n);
   if (!Simplex(1) || D[m + 1][n + 1] < -EPS)
     return -numeric_limits<DOUBLE>::infinity();
   for (int i = 0; i < m; i++) if (B[i] == -1) {
    int s = -1;
    for (int j = 0; j <= n; j++)
     if (s == -1 || D[i][j] < D[i][s] ||
         D[i][j] == D[i][s] && N[j] < N[s])
       s = j;
    Pivot(i, s); } }
  if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
  x = VD(n);
  for (int i = 0; i < m; i++) if (B[i] < n)
    x[B[i]] = D[i][n + 1];
  return D[m][n + 1]; } };
// Two-phase simplex algorithm for solving linear programs
// of the form
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be
//              stored
// OUTPUT: value of the optimal solution (infinity if
//                    unbounded above, nan if infeasible)
// To use this code, create an LPSolver object with A, b,
// and c as arguments.  Then, call Solve(x).
// #include <iostream>
// #include <iomanip>
// #include <vector>
// #include <cmath>
// #include <limits>
// using namespace std;
// int main() {
//   const int m = 4;
//   const int n = 3;
//   DOUBLE _A[m][n] = {
//     { 6, -1, 0 },
//     { -1, -5, 0 },
//     { 1, 5, 1 },
//     { -1, -5, -1 }
//   };
//   DOUBLE _b[m] = { 10, -4, 5, -5 };
//   DOUBLE _c[n] = { 1, -1, 0 };
//   VVD A(m);
//   VD b(_b, _b + m);
//   VD c(_c, _c + n);
//   for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);
//   LPSolver solver(A, b, c);
//   VD x;
//   DOUBLE value = solver.Solve(x);
//   cerr << "VALUE: " << value << endl; // VALUE: 1.29032
//   cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
//   for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
//   cerr << endl;
//   return 0;
// }
```

6.3. **Fast Square Testing.** An optimized test for square integers.

```
long long M;
void init_is_square() {
  rep(i,0,64) M |= 1ULL << (63-(i*i)%64); }
inline bool is_square(ll x) {
  if ((M << x) >= 0) return false;
  int c = __builtin_ctz(x);
  if (c & 1) return false;
  x >>= c;
  if ((x&7) - 1) return false;
  ll r = sqrt(x);
  return r*r == x; }
```

6.4. **Fast Input Reading.** If input or output is huge, sometimes it is beneficial to optimize the input reading/output writing. This can be achieved by reading all input in at once (using fread), and then parsing it manually. Output can also be stored in an output buffer and then dumped once in the end (using fwrite). A simpler, but still effective, way to achieve speed is to use the following input reading method.

```
void readn(register int *n) {
  int sign = 1;
  register char c;
  *n = 0;
  while((c = getc_unlocked(stdin)) != '\n') {
    switch(c) {
      case '-': sign = -1; break;
      case ' ': goto hell;
      case '\n': goto hell;
      default: *n *= 10; *n += c - '0'; break; } }
hell:
  *n *= sign; }
```

6.5. **128-bit Integer.** GCC has a 128-bit integer data type named `__int128`. Useful if doing multiplication of 64-bit integers, or something needing a little more than 64-bits to represent. There's also `__float128`.

7. Useful Information

8. Misc

8.1. Debugging Tips.

- Stack overflow? Recursive DFS on tree that is actually a long path?
- Floating-point numbers
  - Getting `NaN`? Make sure `acos` etc. are not getting values out of their range (perhaps `1+eps`).
  - Rounding negative numbers?
  - Outputting in scientific notation?
- Wrong Answer?
  - Read the problem statement again!
  - Are multiple test cases being handled correctly? Try repeating the same test case many times.
  - Integer overflow?
  - Think very carefully about boundaries of all input parameters
  - Try out possible edge cases:
    - $n = 0, n = -1, n = 1, n = 2^{31} - 1$ or $n = -2^{31}$
    - List is empty, or contains a single element
    - n is even, n is odd
    - Graph is empty, or contains a single vertex
    - Graph is a multigraph (loops or multiple edges)
    - Polygon is concave or non-simple
  - Is initial condition wrong for small cases?
  - Are you sure the algorithm is correct?
  - Explain your solution to someone.
  - Are you using any functions that you don't completely understand? Maybe STL functions?
  - Maybe you (or someone else) should rewrite the solution?
  - Can the input line be empty?
- Run-Time Error?
  - Is it actually Memory Limit Exceeded?

8.2. Solution Ideas.

- Dynamic Programming
  - Parsing CFGs: CYK Algorithm
  - Drop a parameter, recover from others
  - Swap answer and a parameter
  - When grouping: try splitting in two
  - 2^k trick
  - When optimizing
    - Convex hull optimization
      - $\text{dp}[i] = \min_{j<i}\{\text{dp}[j] + b[j] \times a[i]\}$
      - $b[j] \geq b[j+1]$
      - optionally $a[i] \leq a[i+1]$
      - $O(n^2)$ to $O(n)$
    - Divide and conquer optimization
      - $\text{dp}[i][j] = \min_{k<j}\{\text{dp}[i-1][k] + C[k][j]\}$
      - $A[i][j] \leq A[i][j+1]$
      - $O(kn^2)$ to $O(kn \log n)$
      - sufficient: $C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$, $a \leq b \leq c \leq d$ (QI)
    - Knuth optimization
      - $\text{dp}[i][j] = \min_{i<k<j}\{\text{dp}[i][k] + \text{dp}[k][j] + C[i][j]\}$
      - $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$
      - $O(n^3)$ to $O(n^2)$
      - sufficient: QI and $C[b][c] \leq C[a][d]$, $a \leq b \leq c \leq d$
- Greedy
- Randomized
- Optimizations
  - Use bitset (/64)
  - Switch order of loops (cache locality)
- Process queries offline
  - Mo's algorithm
- Square-root decomposition
- Precomputation
- Efficient simulation
  - Mo's algorithm
  - Sqrt decomposition
  - Store 2^k jump pointers
- Data structure techniques
  - Sqrt buckets
  - Store 2^k jump pointers
  - 2^k merging trick
- Counting
  - Inclusion-exclusion principle
  - Generating functions
- Graphs
  - Can we model the problem as a graph?
  - Can we use any properties of the graph?
  - Strongly connected components
  - Cycles (or odd cycles)
  - Bipartite (no odd cycles)
    - Bipartite matching
    - Hall's marriage theorem
    - Stable Marriage
  - Cut vertex/bridge
  - Biconnected components
  - Degrees of vertices (odd/even)
  - Trees
    - Heavy-light decomposition
    - Centroid decomposition
    - Least common ancestor
    - Centers of the tree
  - Eulerian path/circuit
  - Chinese postman problem
  - Topological sort
  - (Min-Cost) Max Flow
  - Min Cut
    - Maximum Density Subgraph
  - Huffman Coding
  - Min-Cost Arborescence
  - Steiner Tree
  - Kirchoff's matrix tree theorem
  - Prüfer sequences
  - Lovász Toggle
  - Look at the DFS tree (which has no cross-edges)
  - Is the graph a DFA or NFA?
    - Is it the Synchronizing word problem?
- Mathematics
  - Is the function multiplicative?
  - Look for a pattern
  - Permutations
    - Consider the cycles of the permutation
  - Functions
    - Sum of piecewise-linear functions is a piecewise-linear function
    - Sum of convex (concave) functions is convex (concave)
  - Modular arithmetic
    - Chinese Remainder Theorem
    - Linear Congruence
  - Sieve
  - System of linear equations
  - Values too big to represent?
    - Compute using the logarithm
    - Divide everything by some large value
  - Linear programming
    - Is the dual problem easier to solve?
  - Can the problem be modeled as a different combinatorial problem? Does that simplify calculations?
- Logic
  - 2-SAT
  - XOR-SAT (Gauss elimination or Bipartite matching)
- Meet in the middle
- Only work with the smaller half ($\log(n)$)
- Strings
  - Trie (maybe over something weird, like bits)
  - Suffix array
  - Suffix automaton (+DP?)
  - Aho-Corasick
  - eerTree
  - Work with $S + S$
- Hashing
- Euler tour, tree to array
- Segment trees
  - Lazy propagation
  - Persistent
  - Implicit
  - Segment tree of X
- Geometry
  - Minkowski sum (of convex sets)
  - Rotating calipers
  - Sweep line (horizontally or vertically?)
  - Sweep angle
  - Convex hull
- Fix a parameter (possibly the answer).
- Are there few distinct values?
- Binary search
- Sliding Window (+ Monotonic Queue)
- Computing a Convolution? Fast Fourier Transform
- Computing a 2D Convolution? FFT on each row, and then on each column

- Exact Cover (+ Algorithm X)
- Cycle-Finding
- What is the smallest set of values that identify the solution? The cycle structure of the permutation? The powers of primes in the factorization?
- Look at the complement problem
  - Minimize something instead of maximizing
- Immediately enforce necessary conditions. (All values greater than 0? Initialize them all to 1)
- Add large constant to negative numbers to make them positive
- Counting/Bucket sort

Practice Contest Checklist

- How many operations per second ($10^{7/8}$)? Compare to local machine.
- What is the stack size?
- How to use printf/scanf with long long/long double?
- Are `__int128` and `__float128` available?
- Does MLE give RTE or MLE as a verdict? What about stack overflow?
- What is `RAND_MAX`?
- How does the judge handle extra spaces (or missing newlines) in the output?
- Look at documentation for programming languages.
- Try different programming languages: C++, Java and Python.
- Try the submit script.
- Try local programs: i?python[23], factor.
- Try submitting with `assert(false)` and `assert(true)`.
- Return-value from `main`.
- Look for directory with sample test cases.
- Make sure printing works.
- Remove this page from the notebook.