分割統治法

問題を小さく分けてまとめよう

tsutaj (@_TTJR_)

Hokkaido University M1

February 7, 2019

分割統治法とは？

- そのままでは計算に時間を要する問題を適切に分割し、小さい問題にして解いた後にそれらをまとめることで、効率よく計算する方法のこと
- 様々なバリエーション
  - 数列に対する分割統治
  - 木に対する分割統治
  - 平面に対する分割統治
- 今回はそれぞれについて簡単に説明
  - 蟻本の内容とほぼ同じなので、すでに読んでる方はすみません

転倒数 (バブルソートの交換回数)

転倒数 (バブルソートの交換回数)

長さ N の数列 A が与えられる。A の i 番目の要素を A_i と表記する。以下の条件を満たす組 (i, j) の個数を求めよ。

- $1 \leq i < j \leq N$
- $A_i > A_j$

入力制約

- $1 \leq N \leq 10^5$
- $1 \leq A_i \leq 10^9$

3	1	4	1	5	9	2	6

転倒数: i < j で、i 番目の数よりも j 番目の数が小さいような (i, j) の個数
この例では (1, 2), (1, 4), (1, 7), (3, 4), (3, 7), (5, 7), (6, 7), (6, 8) の 8 個

転倒数 (バブルソートの交換回数)

- Binary Indexed Tree, Segment Tree などの「区間和」を高速に求められるデータ構造を用いると $O(N \log N)$
  - Binary Indexed Tree を使った解法は別スライドを参照 Link
  - 実はこれらのデータ構造を使わなくても解ける！！

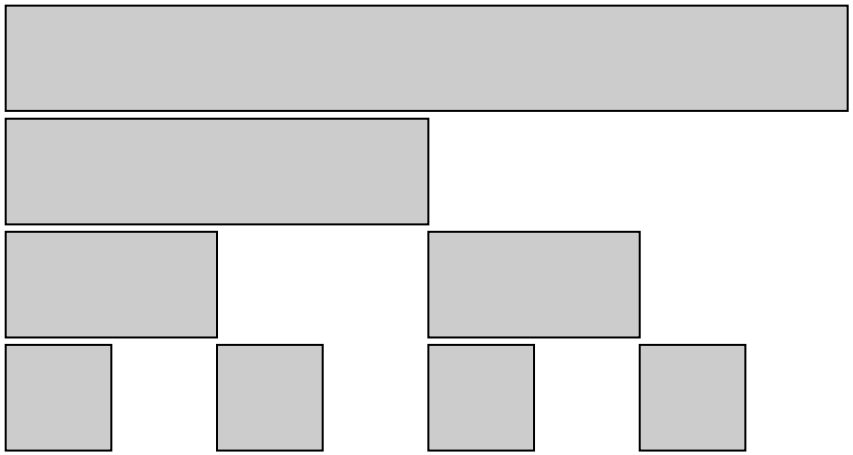
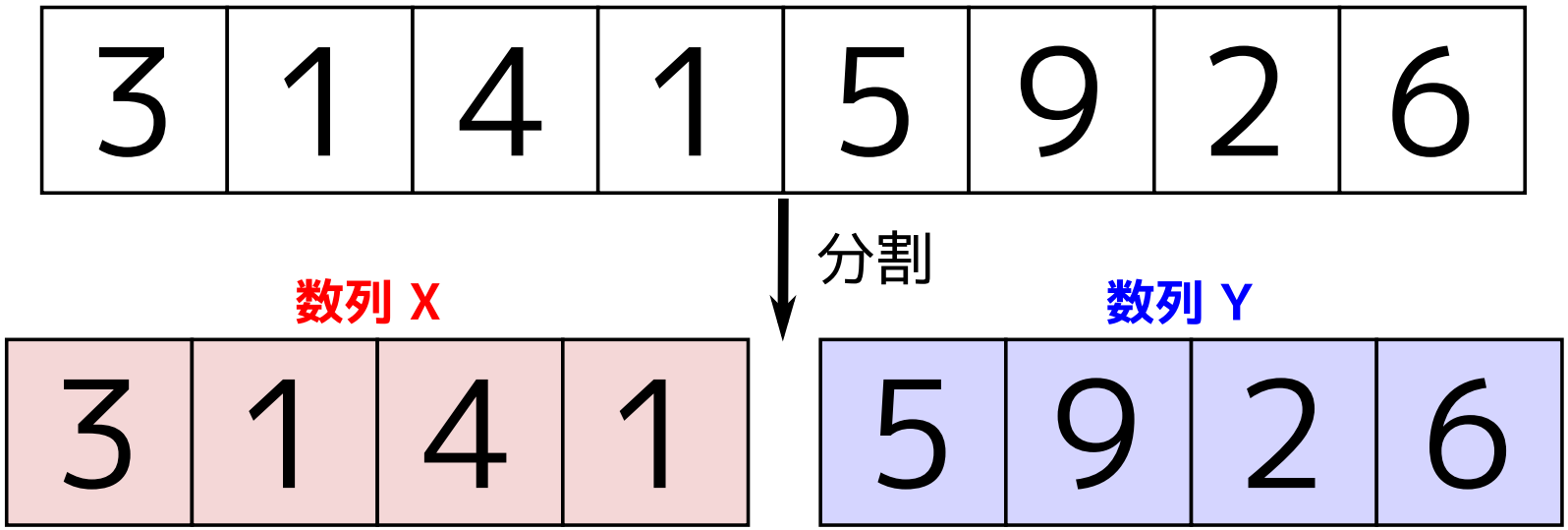


Figure: (参考) Binary Indexed Tree の外観

- 分割統治法で解くことを考える
  - どのように問題を分けて、まとめればよい・・・？
  - 数列を適切に分割して問題を小さくし、それらをまとめよう

転倒数 (バブルソートの交換回数)

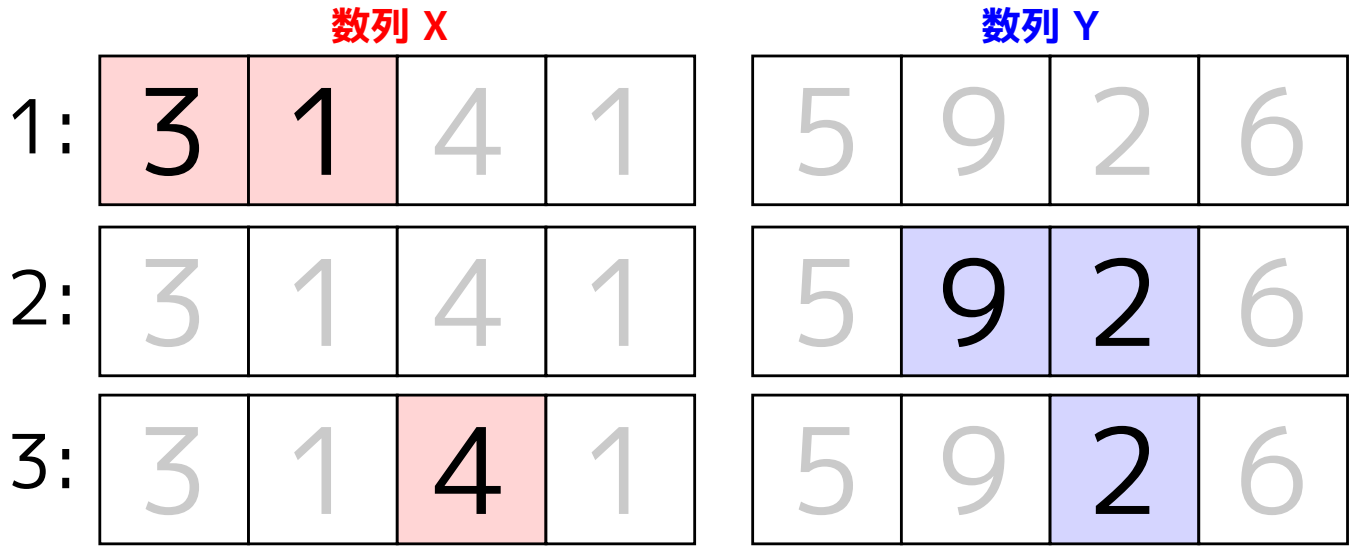
- 元の数列を、サイズができるだけ等しくなるよう 2 つに分割！！
- 分割後の数列をそれぞれ X, Y とおくと、転倒が発生するパターンはどう書けるか？



転倒数 (バブルソートの交換回数)

転倒には、次の 3 パターンがある

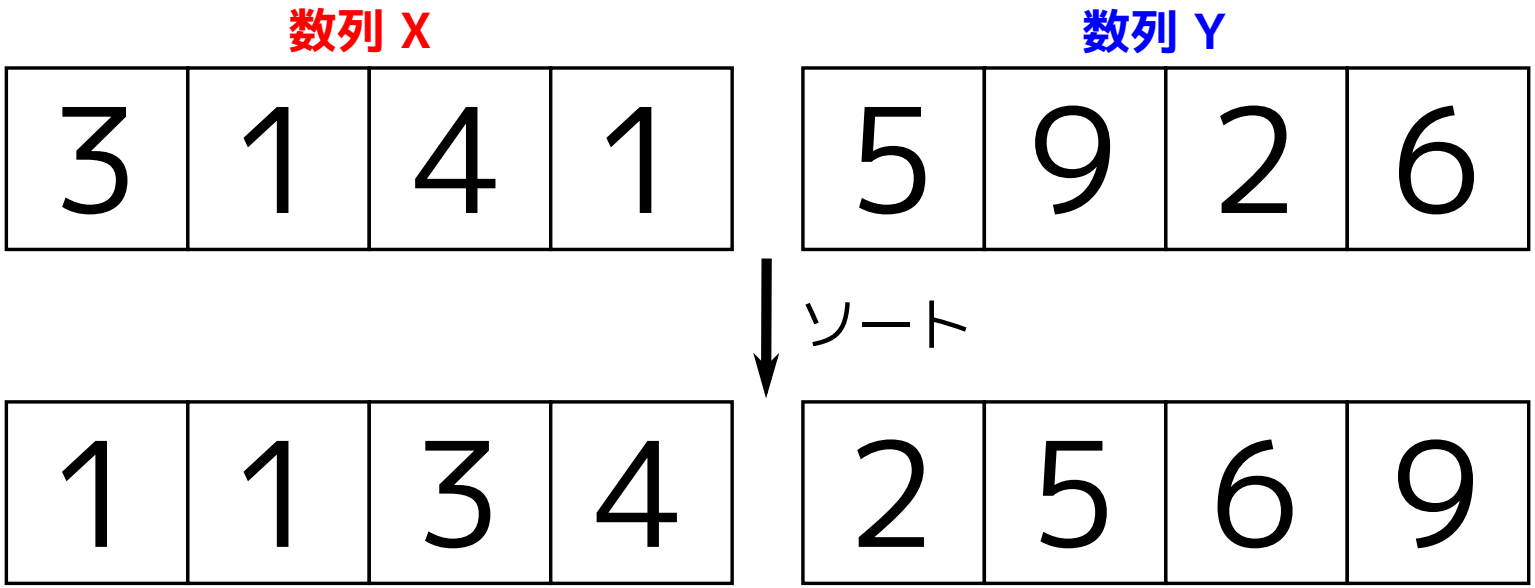
1. X 内で転倒が起こる
   - これは X をもう一度分割し、再帰的に処理すると求められる
2. Y 内で転倒が起こる
   - 上と同様に再帰的に求められる
3. X と Y の間で転倒が起こる
   - これは真面目に求める
   - X の各要素 e に対して、「Y 内にある、 e より小さい要素はいくつあるか」を求める



転倒数 (バブルソートの交換回数)

X と Y の間の転倒を求める方法

- 再帰的に処理する際に、同時に数列をソートするようにする
  - マージソート Link と同じことをしている
- ソート済みであれば、転倒しているものの個数は尺取り法のように求められる

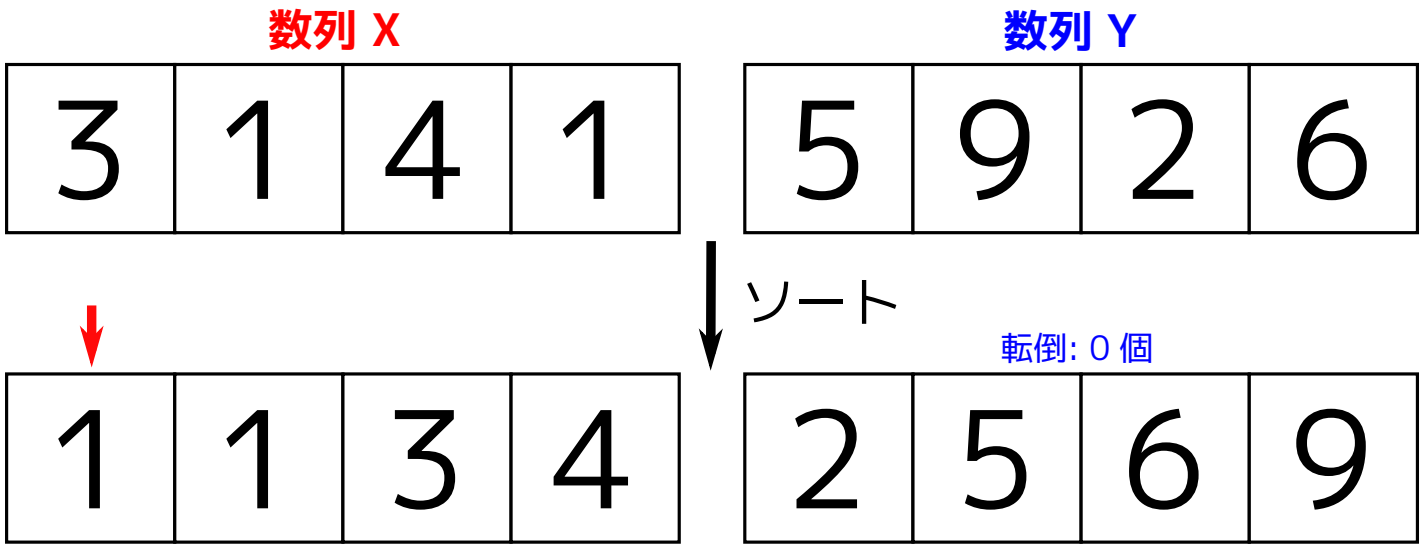


※再帰的に処理すると、このようなソート済みの列が作れる
cf. マージソート

転倒数 (バブルソートの交換回数)

X と Y の間の転倒を求める方法

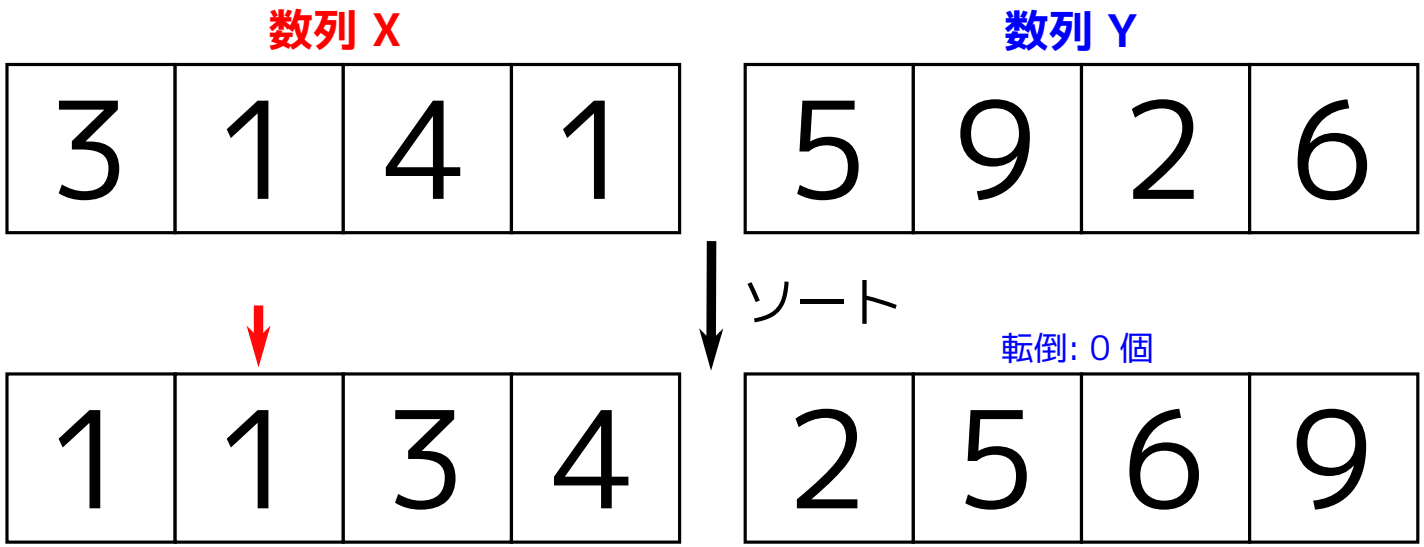
- 再帰的に処理する際に、同時に数列をソートするようにする
  - マージソート Link と同じことをしている
- ソート済みであれば、転倒しているものの個数は尺取り法のように求められる



転倒数 (バブルソートの交換回数)

X と Y の間の転倒を求める方法

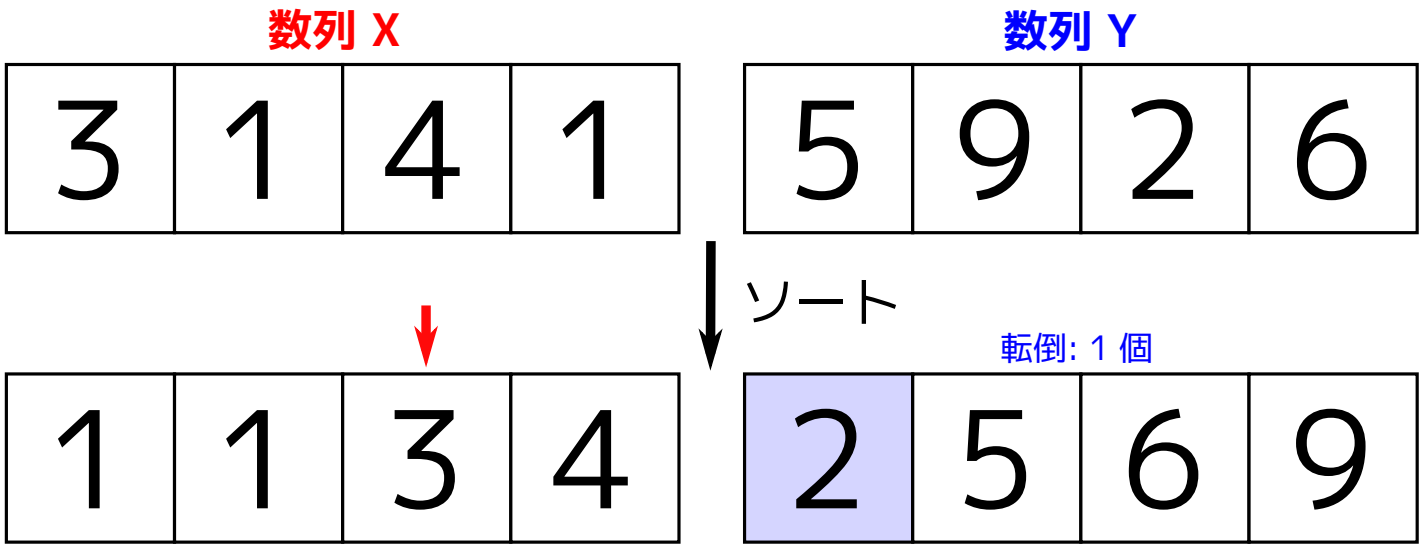
- 再帰的に処理する際に、同時に数列をソートするようにする
  - マージソート Link と同じことをしている
- ソート済みであれば、転倒しているものの個数は尺取り法のように求められる



転倒数 (バブルソートの交換回数)

X と Y の間の転倒を求める方法

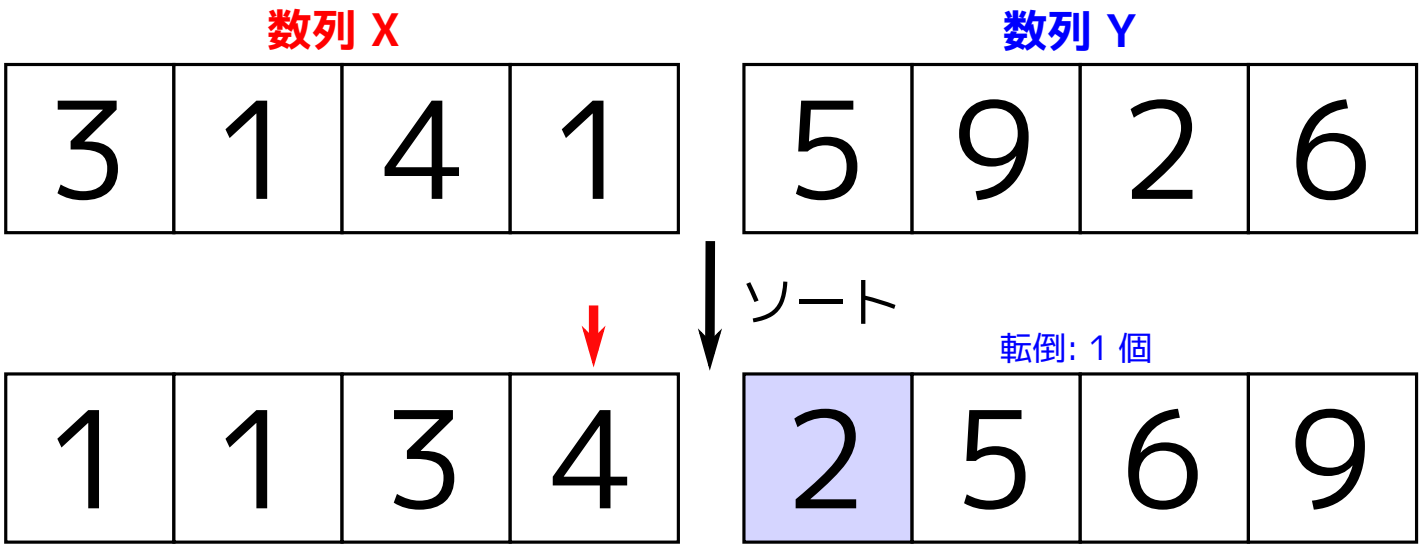
- 再帰的に処理する際に、同時に数列をソートするようにする
  - マージソート Link と同じことをしている
- ソート済みであれば、転倒しているものの個数は尺取り法のように求められる



転倒数 (バブルソートの交換回数)

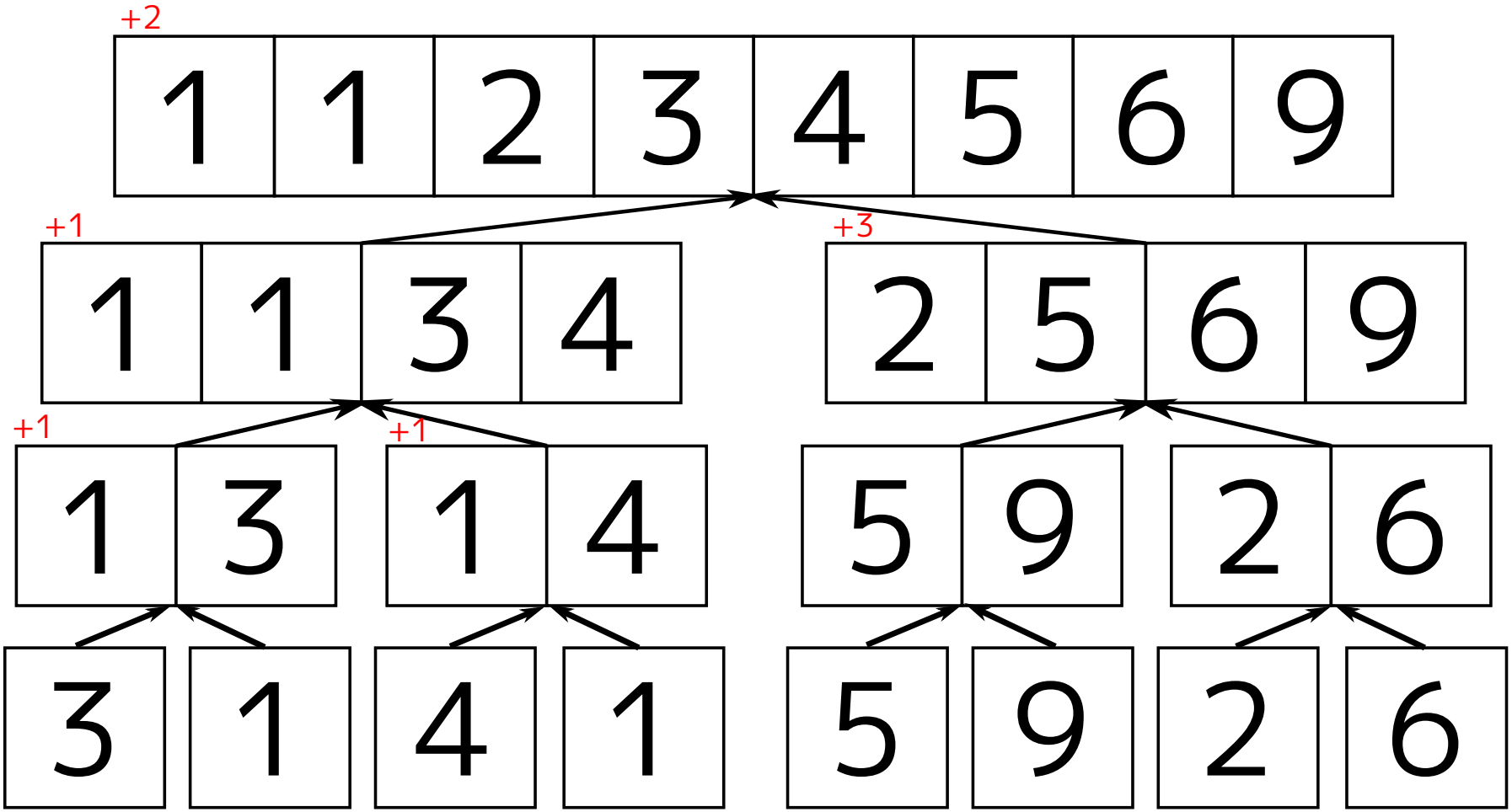
X と Y の間の転倒を求める方法

- 再帰的に処理する際に、同時に数列をソートするようにする
  - マージソート Link と同じことをしている
- ソート済みであれば、転倒しているものの個数は尺取り法のように求められる



転倒数 (バブルソートの交換回数)

やっていること全体を見ると、マージソートをしながら転倒数を数えていることに他ならない



転倒数 (バブルソートの交換回数)

実装例

```cpp
#include <cstdio>
#include <vector>
using namespace std;

long long int getInvNum(vector<int> &A, int l, int r) {
    if(r - l == 1) return 0;

    // X の内部、Y の内部で転倒しているものの合計を得る
    // この時点で左半分・右半分はソート済み
    int mid = (l + r) / 2;
    long long int res = getInvNum(A, l, mid) + getInvNum(A, mid, r);

    // 数列を 2 つに分割
    vector<int> L, R;
    for(int i=l; i<r; i++) (i < mid ? L : R).push_back(A[i]);
    int N = L.size(), M = R.size();

    // 尺取り
    int cur = l, j = 0;
    for(int i=0; i<N; i++) {
        while(j < M and L[i] > R[j]) A[cur++] = R[j++];
        res += j; A[cur++] = L[i];
    }
    for(; j<M; j++) A[cur++] = R[j];
    return res;
}

int main() {
    int N; scanf("%d", &N);
    vector<int> A(N);
    for(int i=0; i<N; i++) scanf("%d", &A[i]);
    printf("%lld\n", getInvNum(A, 0, N));
    return 0;
}
```

ふたつの和

ふたつの和

長さ N の数列 A がある。以下の条件を満たす組 i, j $(i < j)$ をそれぞれ数えよ。

1. $A_i + A_j \leq K$ を満たすもの
2. $A_i + A_j = K$ を満たすもの

入力制約

- $1 \leq N \leq 10^5$
- $1 \leq A_i \leq 10^9$

1 つめのほうが難しそうに見えますが、実は 1 つめの解法を活かして 2 つめを解く、という流れになります

ふたつの和

- $A_i + A_j \leq K$ となる (i, j) を数えることを考えよう
- 先程の「転倒数」でやっていた尺取りパートを、「大小比較」から「和の比較」に変えるだけ！
  - ソート済み数列 X, Y に対して尺取りをするならば、X を降順に見て、Y に対するカーソルを昇順に動かせば、和が K 以下になる個数を尺取りで求められる
- よって、先ほどと同じ方針で解ける

ふたつの和

- $A_i + A_j = K$ となる (i, j) を数えることを考えよう
- 重要な言い換え: (和がちょうど K の個数) $\Longleftrightarrow$ (和が K 以下の個数) $-$ (和が $K-1$ 以下の個数)
- 先程の問題を K の場合と $K-1$ の場合について解き、引くことで求められる！
- 次のページで実装を示します

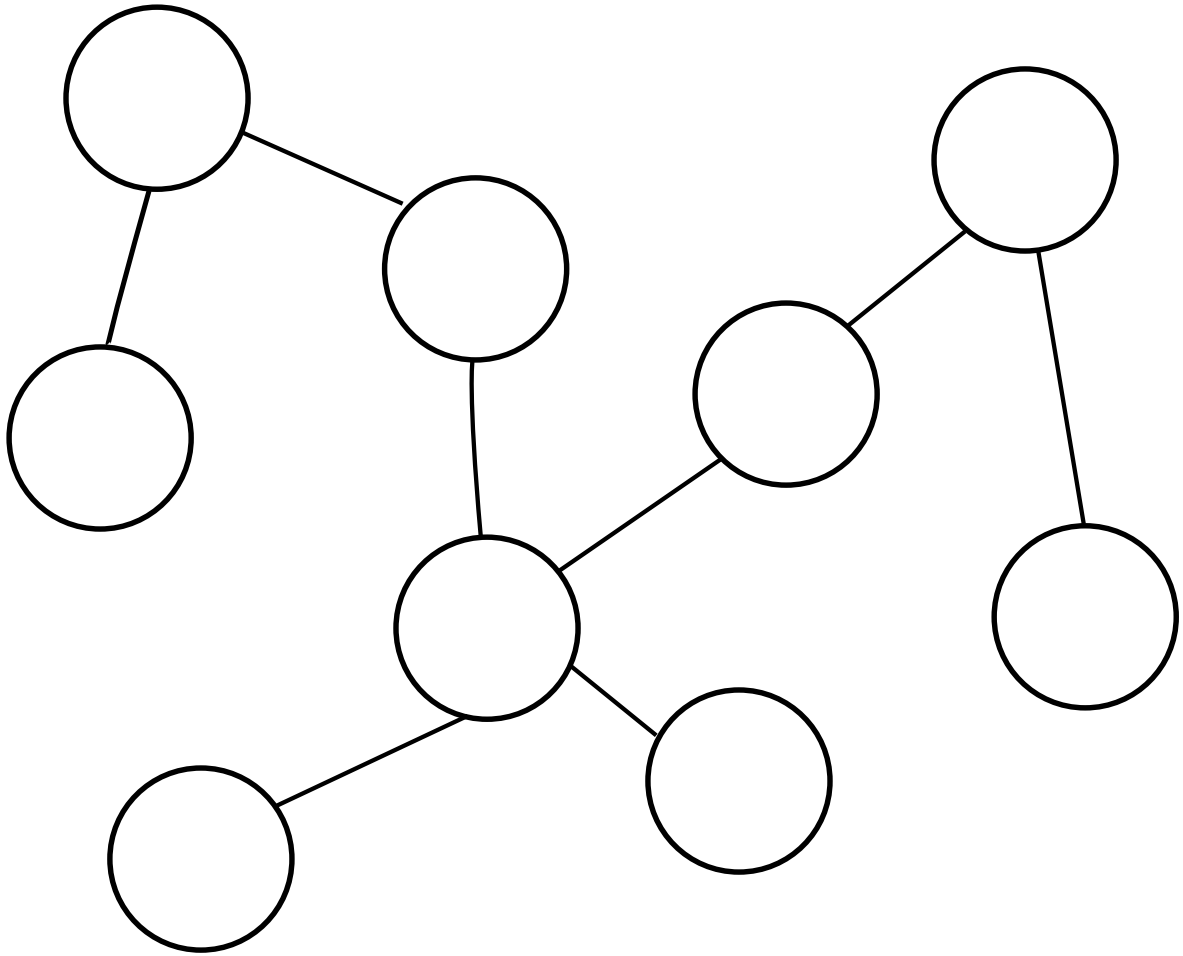
ふたつの和

ちょうど K であるものを求める実装例

```cpp
#include <cstdio>
#include <vector>
using namespace std;

// 数列中の 2 要素を持ってきて、
// その和が K 以下になるような組がいくつあるか数える
// Verify できる問題が見つからなかったのでしていません・・・（参考程度に見て）
long long int cntTwoSum(vector<int> &A, int l, int r, int K) {
    if(r - l == 1) return 0;
    int mid = (l + r) / 2;
    long long int res = cntTwoSum(A, l, mid, K) + cntTwoSum(A, mid, r, K);
    vector<int> L, R;
    for(int i=l; i<r; i++) (i < mid ? L : R).push_back(A[i]);

    // 尺取り
    int N = L.size(), M = R.size(), idx1 = 0;
    for(int i=N-1; i>=0; i--) {
        while(idx1 < M and L[i] + R[idx1] <= K) idx1++;
        res += idx1;
    }

    // マージ操作
    int cur = l, idx2 = 0;
    for(int i=0; i<N; i++) {
        while(idx2 < M and R[idx2] < L[i]) A[cur++] = R[idx2++];
        A[cur++] = L[i];
    }
    for(; idx2<M; idx2++) A[cur++] = R[idx2];
    return res;
}

int main() {
    int N, K; scanf("%d%d", &N, &K);
    vector<int> A(N);
    for(int i=0; i<N; i++) {
        scanf("%d", &A[i]);
    }
    // 1 回目の cntTwoSum の終了時に A がソート済みになるが、
    // 今回の問題では順番関係ないのでそのまま使う
    printf("%lld\n", cntTwoSum(A, 0, N, K) - cntTwoSum(A, 0, N, K-1));
    return 0;
}
```

木に対する分割統治

- 先ほどまで「列に対する分割統治」を考えていた
- 実は、木に対しても同じようなことができる！



木の重心

木の重心

- N 頂点の木 T について考える
- T 内のある頂点 v について、v を取り除くことでできる部分木のサイズが全て $\frac{N}{2}$ 以下になるとき、v を「木の重心」と呼ぶ

木の重心に関する重要な事実

- 任意の木に対して、重心となる頂点が必ず存在
- 任意の木に対して、重心となる頂点は高々 2 個

参考になるページ

- ツリーの重心分解の図解 (drken さん) Link
- 木の重心列挙アルゴリズム (こうき さん) Link

木の重心

木の重心

- N 頂点の木 T について考える
- T 内のある頂点 v について、v を取り除くことでできる部分木のサイズが全て $\frac{N}{2}$ 以下になるとき、v を「木の重心」と呼ぶ

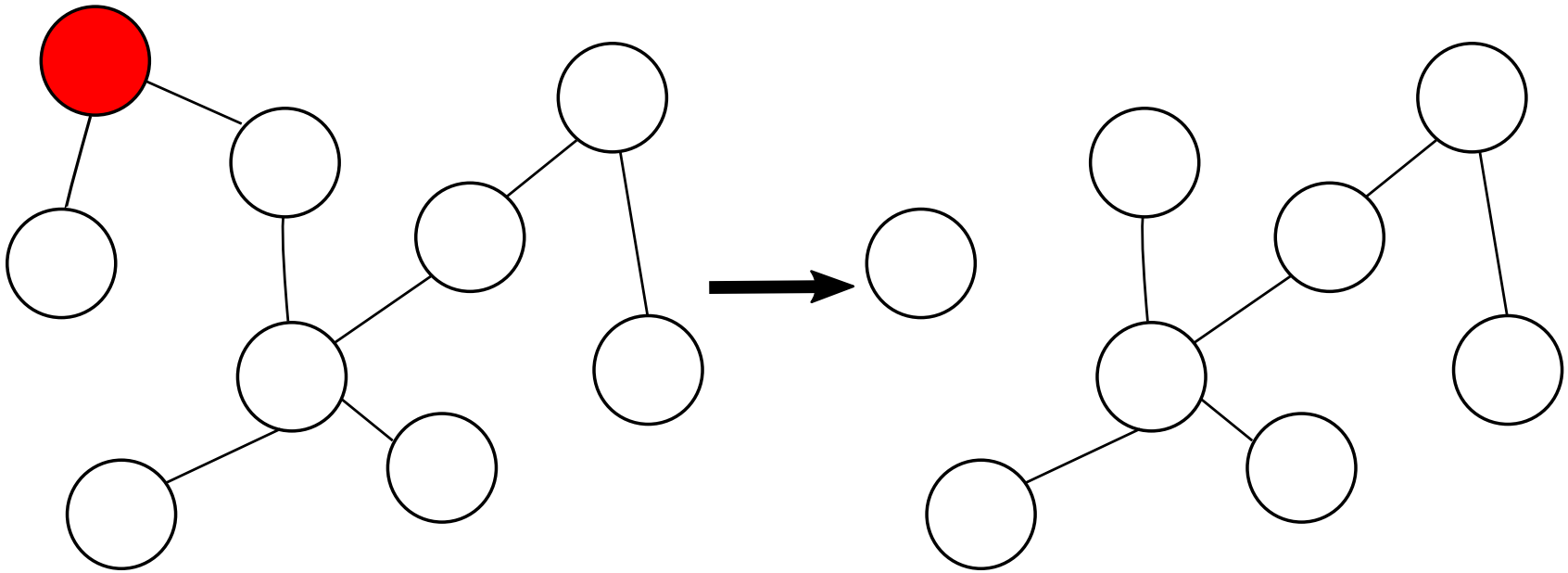


Figure: 木の重心でない例

木の重心

木の重心

- N 頂点の木 T について考える
- T 内のある頂点 v について、v を取り除くことでできる部分木のサイズが全て $\frac{N}{2}$ 以下になるとき、v を「木の重心」と呼ぶ

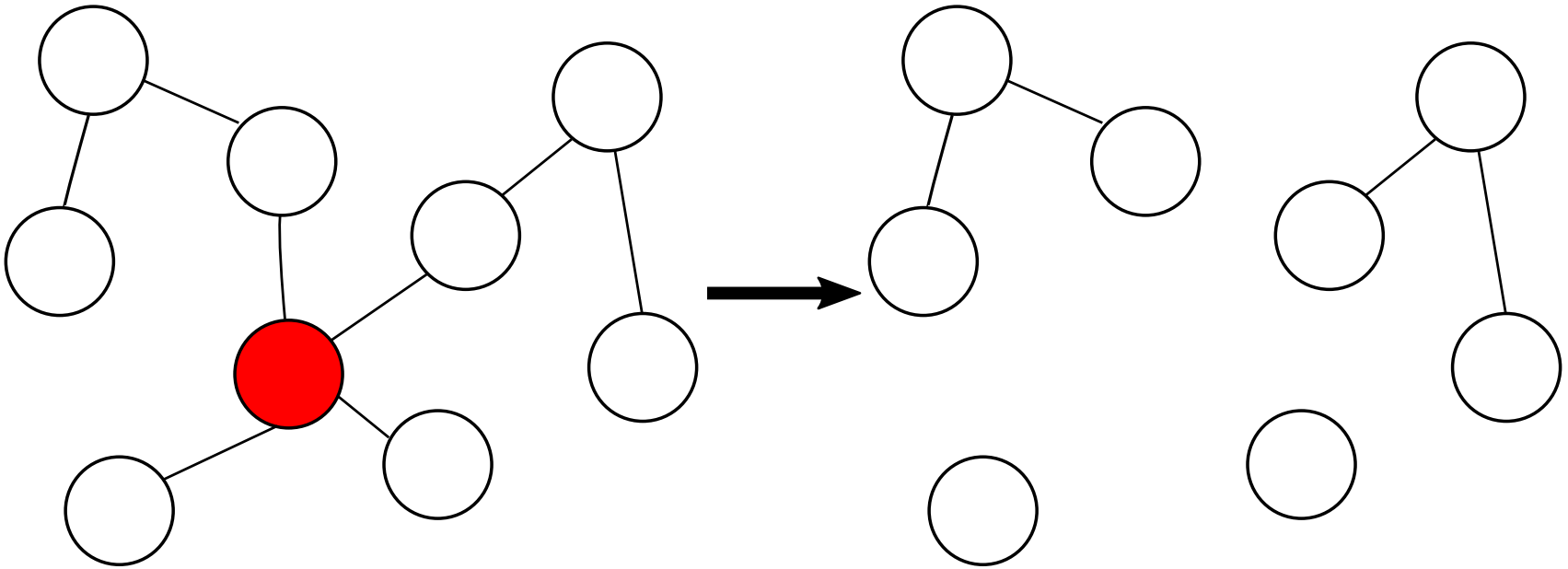
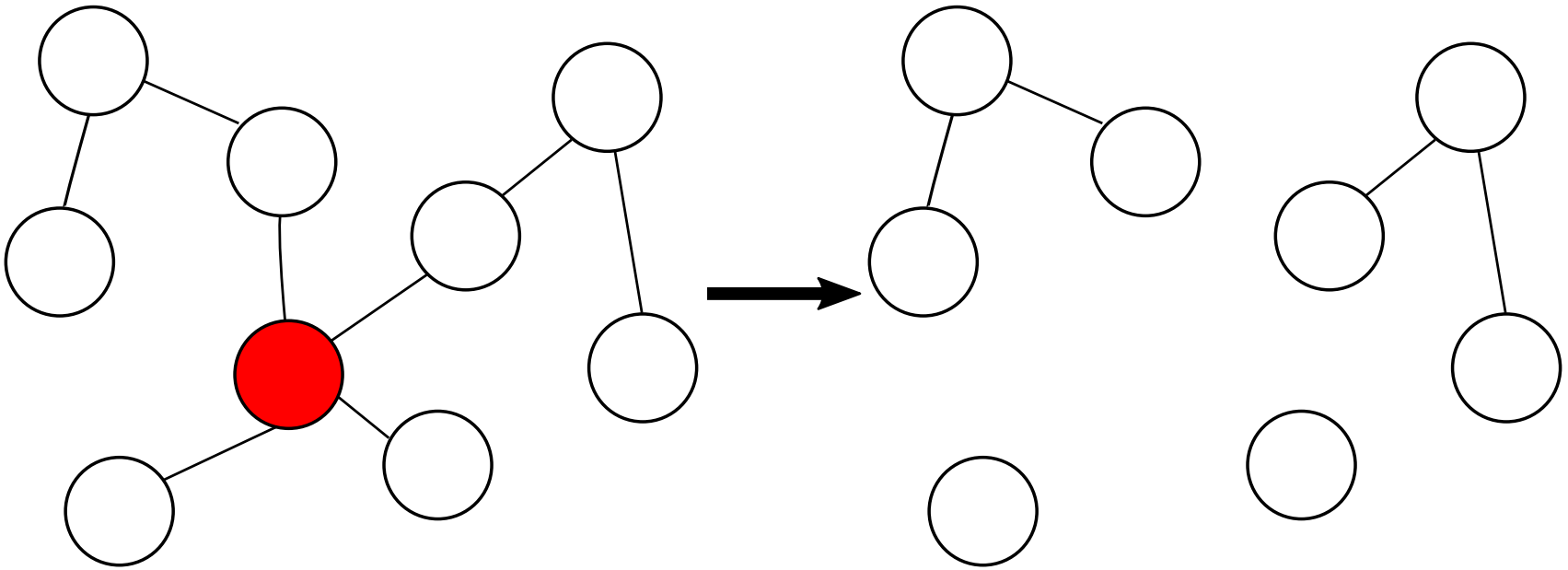


Figure: 木の重心である例

木の重心

- 木を重心で分割し、それによってできた部分木をその重心で分割し・・・という再帰的な処理を考える
  - 重心で分割してできる部分木は、元の木のサイズの半分以下！
  - SegmentTree と同じように考えると、$O(\log N)$ の深さまで分割すれば全てバラバラに
- 先程の「列に対する分割統治」と同様に分割統治を考えることができる！



重心の求め方

- 重心を使って木を分解すれば分割統治ができそうであることはわかった
- では、重心はどのように求めればよいか？

重心を求めるアルゴリズム (1)

1. 木の頂点 r を適当に選び、r を根とした根つき木にする
2. 各頂点 v に対して、v を根とする部分木のサイズ sub_v を求める
3. 各頂点 v に対して、v のすべての子 c について $\mathrm{sub}_c \leq \frac{N}{2}$ であり、かつ $N - \mathrm{sub}_v \leq \frac{N}{2}$ である場合、v は木の重心である
   - $N - \mathrm{sub}_v$ とは、v の親方向にある部分木を指している

重心の求め方

実装例

```cpp
// 木の重心を列挙する
// 重心 v → v を根とした時に、 v の子を根とする部分木のサイズが全て N/2 以下である
// DFS をすることで容易に判定可能 O(N)
vector<int> treeCentroid(Graph &G) {
    int N = G.size();
    vector<int> centroid, val(N);
    function<void (int, int)> dfs = [&](int cur, int par) {
        bool is_centroid = true;
        val[cur] = 1;
        for(auto to : G[cur]) {
            if(to == par) continue;
            dfs(to, cur);
            val[cur] += val[to];
            if(val[to] > N / 2) is_centroid = false;
        }
        if(N - val[cur] > N / 2) is_centroid = false;
        if(is_centroid) centroid.push_back(cur);
    };

    dfs(0, -1);
    return centroid;
}
```

重心の求め方

- 重心を求めるアルゴリズムはもうひとつある！
- 重心分解する際は、重心は 1 つ求まれば十分なのでこちらのアルゴリズムを使うことが多そう

重心を求めるアルゴリズム (2)

- 木の頂点 r を適当に選び、r を根とした根付き木にする
- 各頂点 v に対して、v を根とする部分木のサイズ sub_v を求める
  - 重心分解する際は、すでに切った頂点は考慮しないことに注意！
- $v \leftarrow r$ とする
- while true:
  - v のすべての子 c に対してその部分木のサイズ sub_c を見て、最もサイズが大きい頂点 $c_{\max}$ について $sub_{c_{\max}} > \frac{N}{2}$ ならば、$v \leftarrow c_{\max}$ とする
  - そうでなければ、v が木の重心であるので v を返す

重心の求め方

実装例

```cpp
// 部分木の重心をひとつ求める (通行できない頂点は dead で管理)
// 子を根としたときの部分木のサイズが N/2 を超えていれば、その子を新たに根として探索
int oneCentroid(Graph &G, int root, vector<bool> &dead) {
    // 毎回確保すると遅いので必ず static にしよう
    static vector<int> val((int)G.size());
    function<void (int, int)> get_size = [&](int cur, int par) {
        val[cur] = 1;
        for(auto to : G[cur]) {
            if(to == par || dead[to]) continue;
            get_size(to, cur);
            val[cur] += val[to];
        }
    };

    get_size(root, -1);
    int N = val[root];
    function<int (int, int)> dfs = [&](int cur, int par) {
        for(auto to : G[cur]) {
            if(to == par || dead[to]) continue;
            if(val[to] > N / 2) return dfs(to, cur);
        }
        return cur;
    };
    return dfs(root, -1);
}
```

Tree

重心分解の練習問題を考えてみよう！

Tree

N 頂点からなる木が与えられ、各辺には重み l_i がついている。この木に含まれる長さ K 以下のパスを数え上げよ。

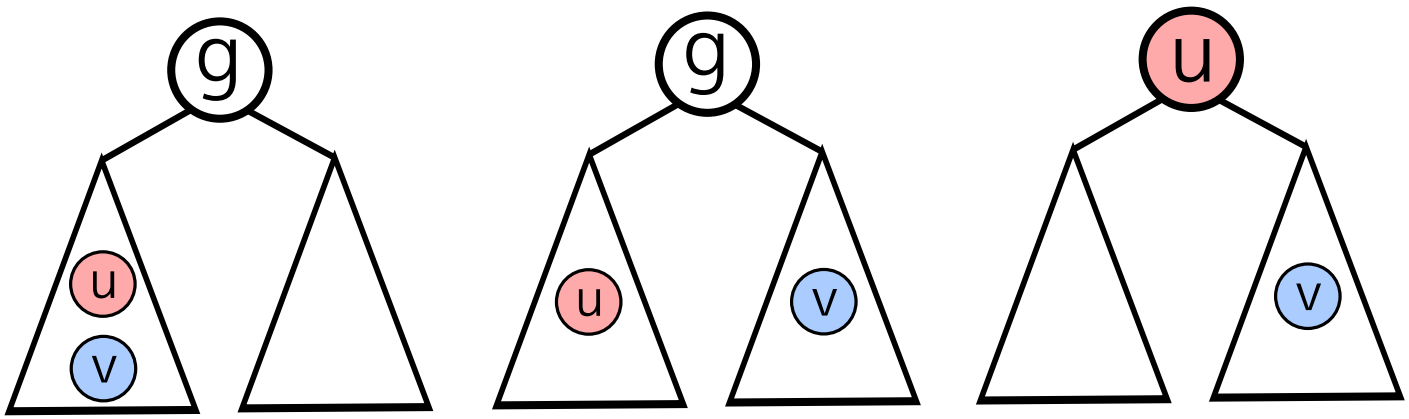
- $1 \leq N \leq 10,000$
- $1 \leq l_i \leq 1,000$

出典: POJ 1741 Link

- コード量が多すぎてスライドに貼れないので、実装例は省きます
- これとやることがほぼ同じ問題 (みんなのプロコン 2018 決勝 C) のソースコードのリンクを代わりに貼るので、そちらも参考にしてください
  - 問題 Link
  - ソースコード Link

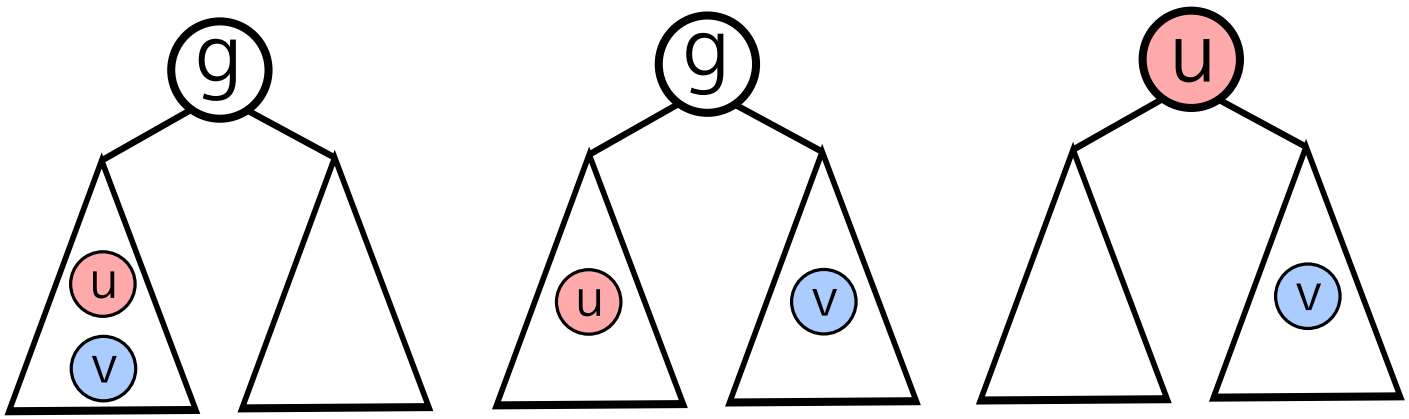
Tree

- 木を重心 g で分割し、いくつかの部分木に分けることを考える
- パス (u, v) について、考慮すべき場合は以下の 3 通り
  1. u も v も同じ部分木に属する
  2. u と v が異なる部分木に属する
  3. u, v のいずれかが g と一致する
- 「列に対する分割統治」でやったことを思い出すと・・・？



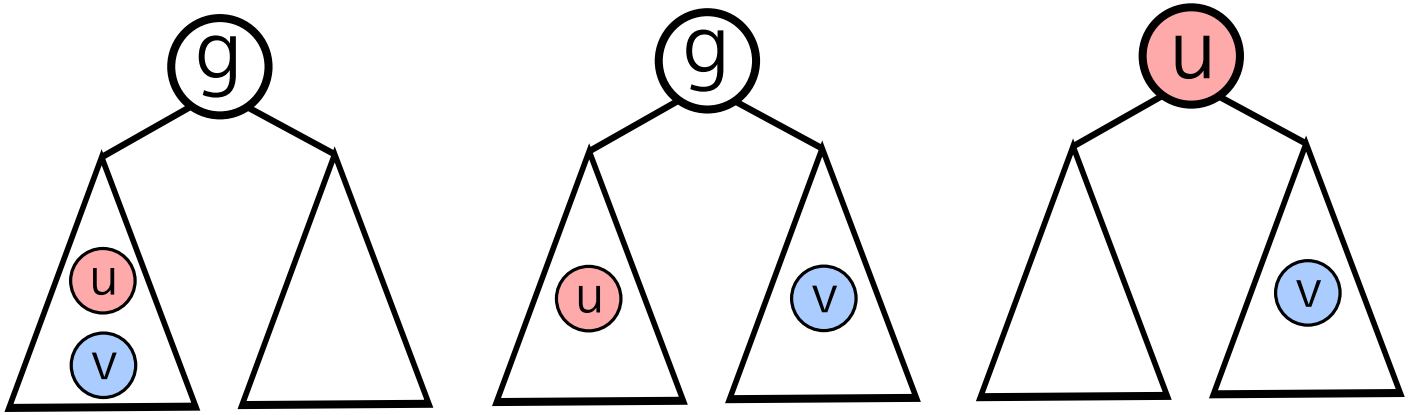
Tree

- 木を重心 g で分割し、いくつかの部分木に分けることを考える
- パス (u, v) について、考慮すべき場合は以下の 3 通り
  1. u も v も同じ部分木に属する
  2. u と v が異なる部分木に属する
  3. u, v のいずれかが g と一致する
- 「列に対する分割統治」でやったことを思い出すと・・・？
  - パターン 1 は再帰的に処理可能
  - パターン 2 は、g から各頂点までパスの長さを全て記録した配列をソートし、尺取りすることで可能
  - パターン 3 は、パターン 2 に対する処理で g を「距離 0 の頂点」とみなせば可能



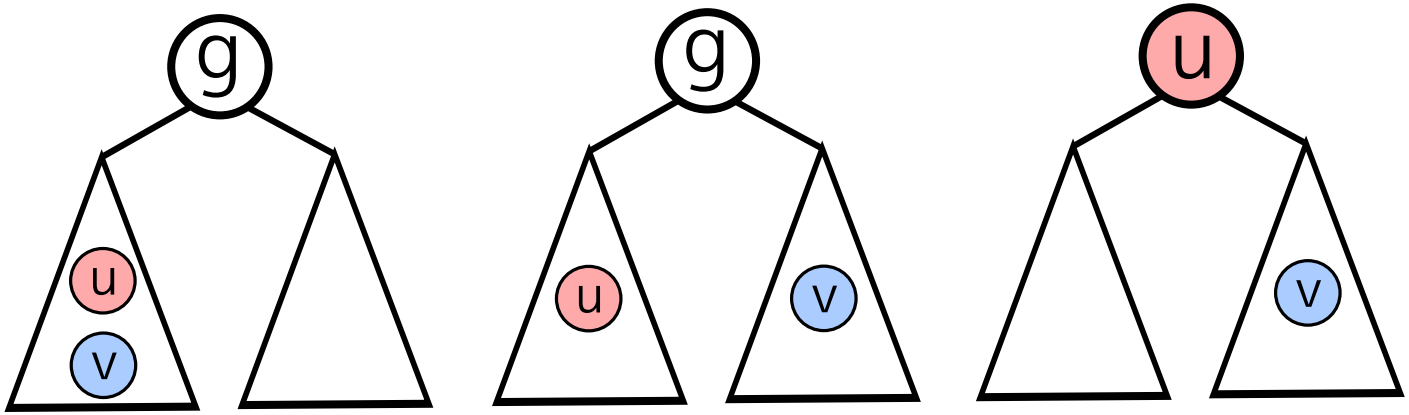
Tree

- 木を重心 g で分割し、いくつかの部分木に分けることを考える
- パス (u, v) について、考慮すべき場合は以下の 3 通り
  1. u も v も同じ部分木に属する
  2. u と v が異なる部分木に属する
  3. u, v のいずれかが g と一致する
- 「列に対する分割統治」でやったことを思い出すと・・・？
  - パターン 1 は再帰的に処理可能
  - パターン 2 は、g から各頂点までパスの長さを全て記録した配列をソートし、尺取りすることで可能
  - パターン 3 は、パターン 2 に対する処理で g を「距離 0 の頂点」とみなせば可能
- パターン 1 を重複して数えないように注意！！



Tree

- 木を重心 g で分割し、いくつかの部分木に分けることを考える
- パス (u, v) について、考慮すべき場合は以下の 3 通り
  1. u も v も同じ部分木に属する
  2. u と v が異なる部分木に属する
  3. u, v のいずれかが g と一致する
- 「列に対する分割統治」でやったことを思い出すと・・・？
  - パターン 1 は再帰的に処理可能
  - パターン 2 は、g から各頂点までパスの長さを全て記録した配列をソートし、尺取りすることで可能
  - パターン 3 は、パターン 2 に対する処理で g を「距離 0 の頂点」とみなせば可能
- 同様の方針で「長さがちょうど K」のパスも数えられます



最近点対

最近点対

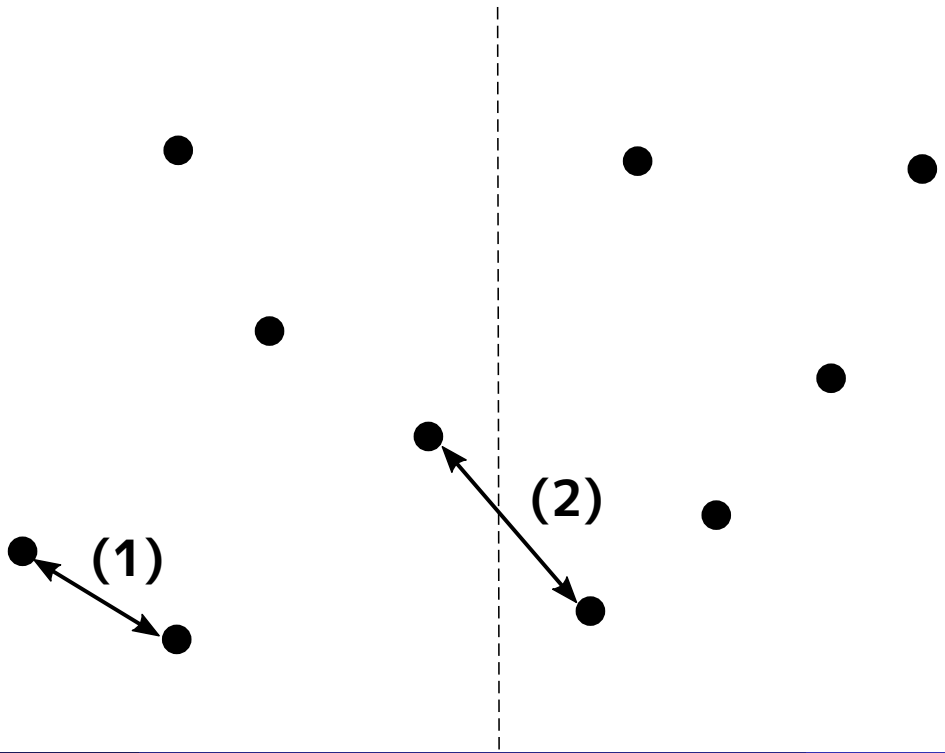
平面上に N 個の点 (x_i, y_i) がある。最も近い 2 点の距離を求めよ。

- $2 \leq N \leq 10^5$
- $-100 \leq x_i, y_i \leq 100$

どのような分割を考えて、まとめればよいか・・・？

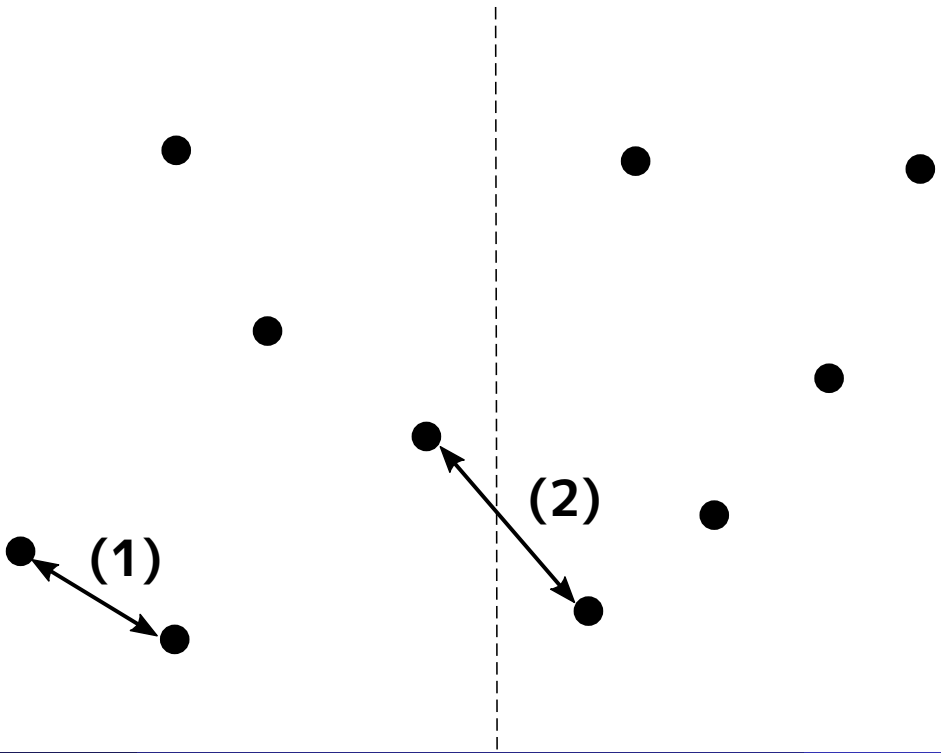
最近点対

- 点の集合を、x 座標を基準に半分に分けてみよう
- 点対 (a, b) について、考慮すべき場合は以下の 2 通り
  1. a, b が同じ領域内に属する
  2. a, b が異なる領域内に属する



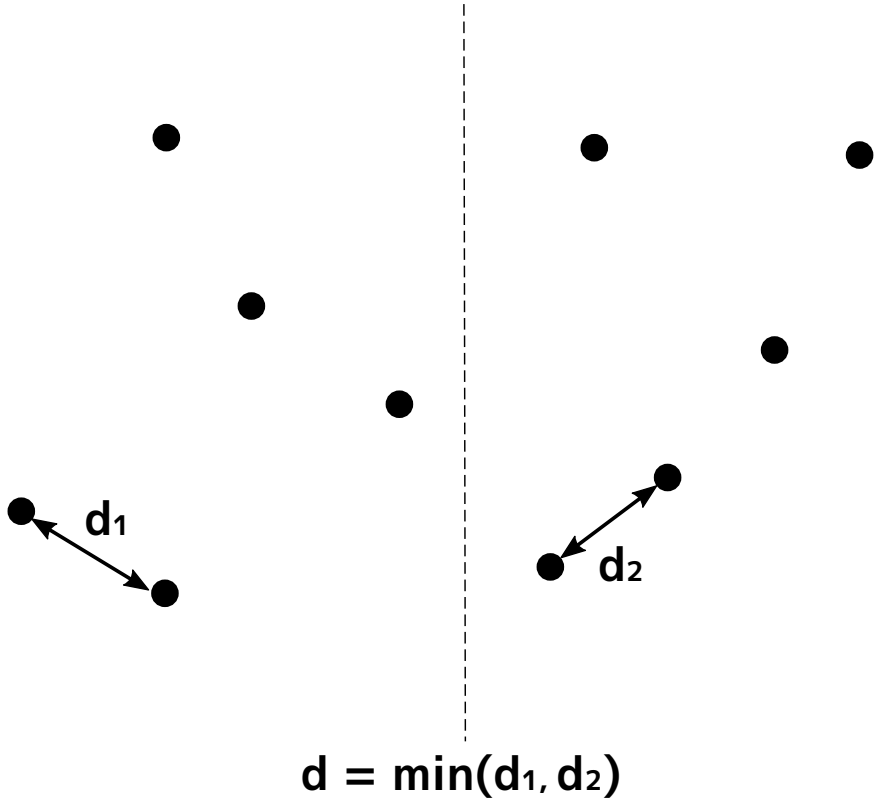
最近点対

- 点の集合を、x 座標を基準に半分に分けてみよう
- 点対 (a, b) について、考慮すべき場合は以下の 2 通り
  1. a, b が同じ領域内に属する
  2. a, b が異なる領域内に属する
- パターン 1 は再帰的に処理すれば実現できそう
- パターン 2 はどうする？



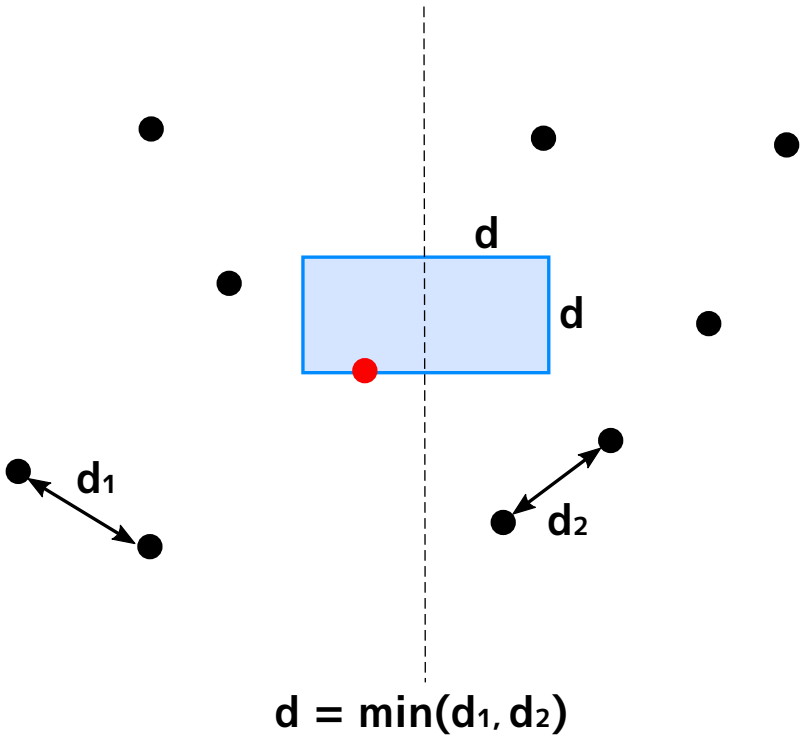
最近点対

- ほしい情報は「最も近い 2 点の距離」なので、それになりえないものの情報はいらない
- パターン 1 を計算して、最近点対の距離が少なくとも d 以下であることが判明したとする



最近点対

- パターン 2 の計算時には、以下のように幅 d のスリット内にある点だけを考えて計算すれば良い！
  - このスリット内に無い点は、そもそも距離が d より大きいため考慮する必要がない
  - スリット内にある点の数は非常に少ないことが証明可能 (詳しくは蟻本 P.325 を参照)



実装例

分割は x 座標について行い、その後 y 座標でソートし直して実装しています

```cpp
using namespace std;
using Point = pair<double, double>;
bool compare_y(Point a, Point b) { return a.second < b.second; }

double closest_pair(vector<Point> &points, int l, int r) {
    if(r - l <= 1) return 1e100;
    int mid = (l + r) / 2;
    double x = points[mid].first;
    double d = min(closest_pair(points, l, mid), closest_pair(points, mid, r));
    auto iti = points.begin(), itl = iti + l, itm = iti + mid, itr = iti + r;
    inplace_merge(itl, itm, itr, compare_y);

    vector<Point> near_line;
    for(int i=l; i<r; i++) {
        if(abs(points[i].first - x) >= d) continue;

        int sz = near_line.size();
        for(int j=sz-1; j>=0; j--) {
            double dx = points[i].first  - near_line[j].first;
            double dy = points[i].second - near_line[j].second;
            if(dy >= d) break;
            d = min(d, sqrt(dx * dx + dy * dy));
        }
        near_line.push_back(points[i]);
    }
    return d;
}
```

練習問題

分割統治に関する練習問題

- POJ 1854: Evil Straw Warts Live ▸ Link
- GCJ 2009 World Finals B: Min Perimeter ▸ Link
- Yandex. Algorithm 2011 Finals B: Superset ▸ Link
- POJ 2114: Boatherds ▸ Link
- UVa 12161: Ironman Race in Treeland ▸ Link
- SPOJ QTREE5: Query on a tree V ▸ Link
- hamayan さんのブログにも関連問題が豊富に載っています！ ▸ Link