# 衝突判定法をソート・検索アルゴリズムの観点から眺める

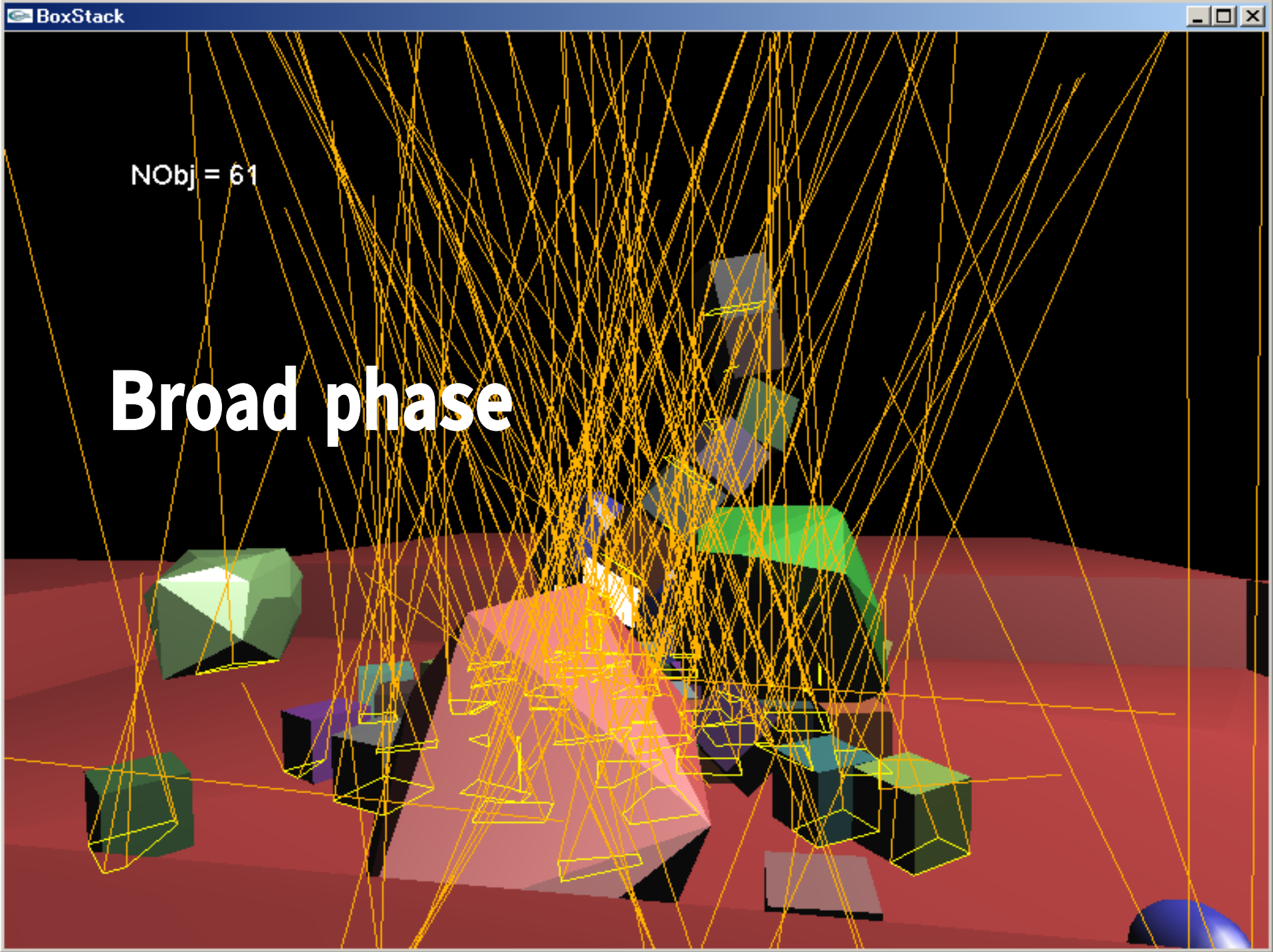
長谷川晶一, 須佐育弥, 松永昇悟
電気通信大学知能システム工学科
http://springhead.info/

# はじめに

- 衝突判定にはさまざまなアルゴリズムが必要
  - BSP, BVH, GJK, Lin-Canny, Quick hull, Andrew, …
  - 変わった衝突判定には，独自のアルゴリズムを作ることも
    - Ex: 肩の関節の可動域
    - ひとつひとつ覚えていくのは，無理＆無駄．
- 検索，ソートはアルゴリズムの基本
  - 接触判定アルゴリズムを検索･ソートの視点から俯瞰
    - 衝突判定アルゴリズムを ソート･検索 に対応付ける
    - ある衝突判定アルゴリズムが，どのくらい賢いかわかる
    - 速さの秘密はどこにあるのか

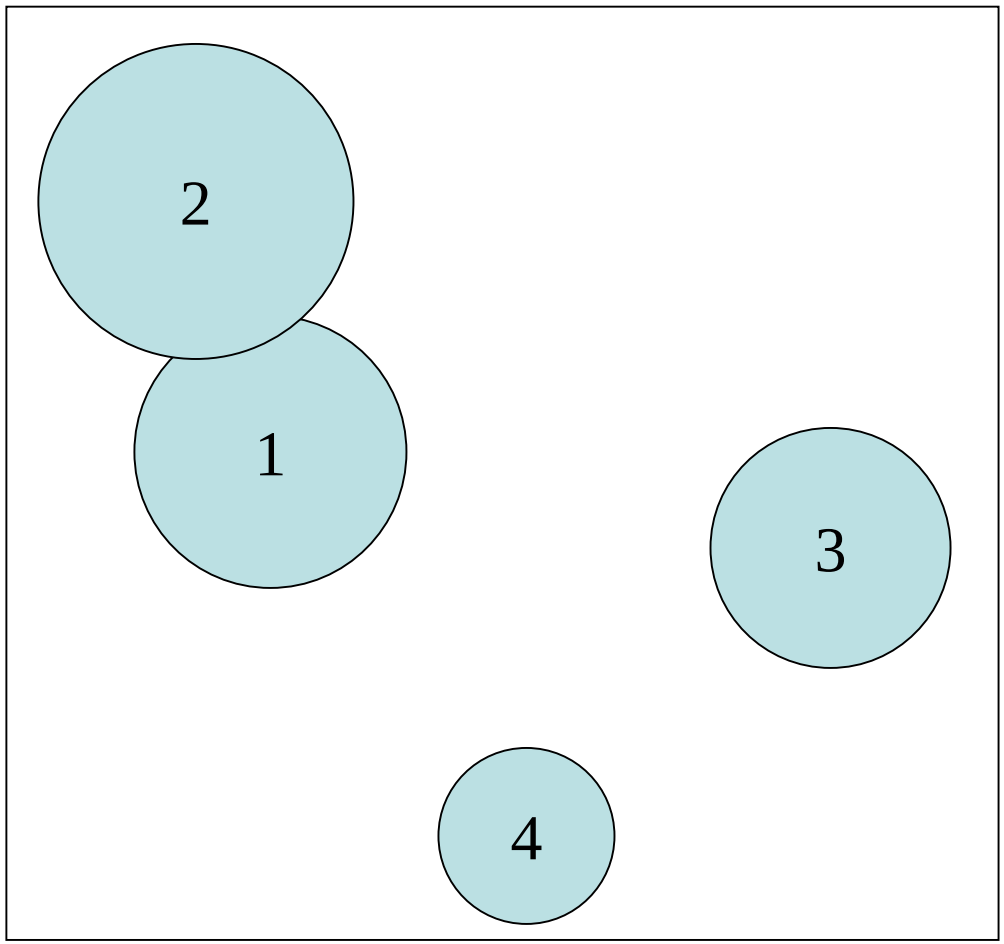
# 目次

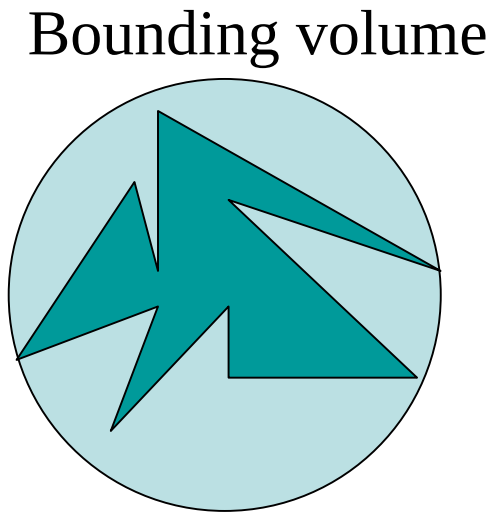
- Broad phase(大まかな枝刈り)のアルゴリズム
  - 接触判定とソート
    - 総当り と Sweep and prune
  - BSPとQuick Sort
  - バケツソート と Uniform Grid
  - BVHと2分木の検索
  - 検索とソートのまとめ
- Narrow phase (ポリゴン同士の判定など) のアルゴリズム
  - 物理シミュレーションに必要な接触情報
  - 接触点と法線
    - 最小と極小
    - GJK, Lin-Canny と 探索
  - 交差部分の切り口 (領域)
  - 交差部分の3次元形状 (体積)
    - Convex Hull
      - Andrewのアルゴリズム, Quick Hull


BoxStack
NObj = 61
Broad phase

# 接触判定とソート

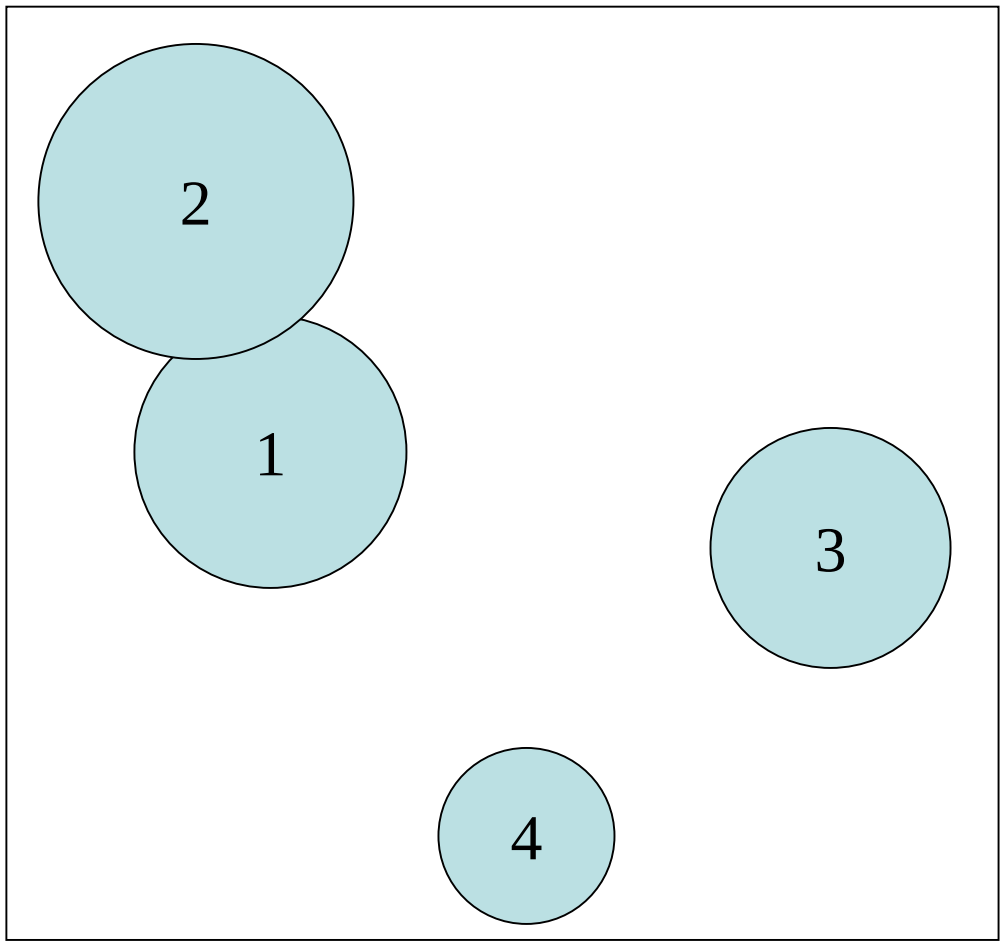
- Broad phase
  - Bounding Volume 同士の大まかな枝刈り



# 総当り法

- すべてのペアをチェック
  - n･(n-1)/2 個のペアをチェック



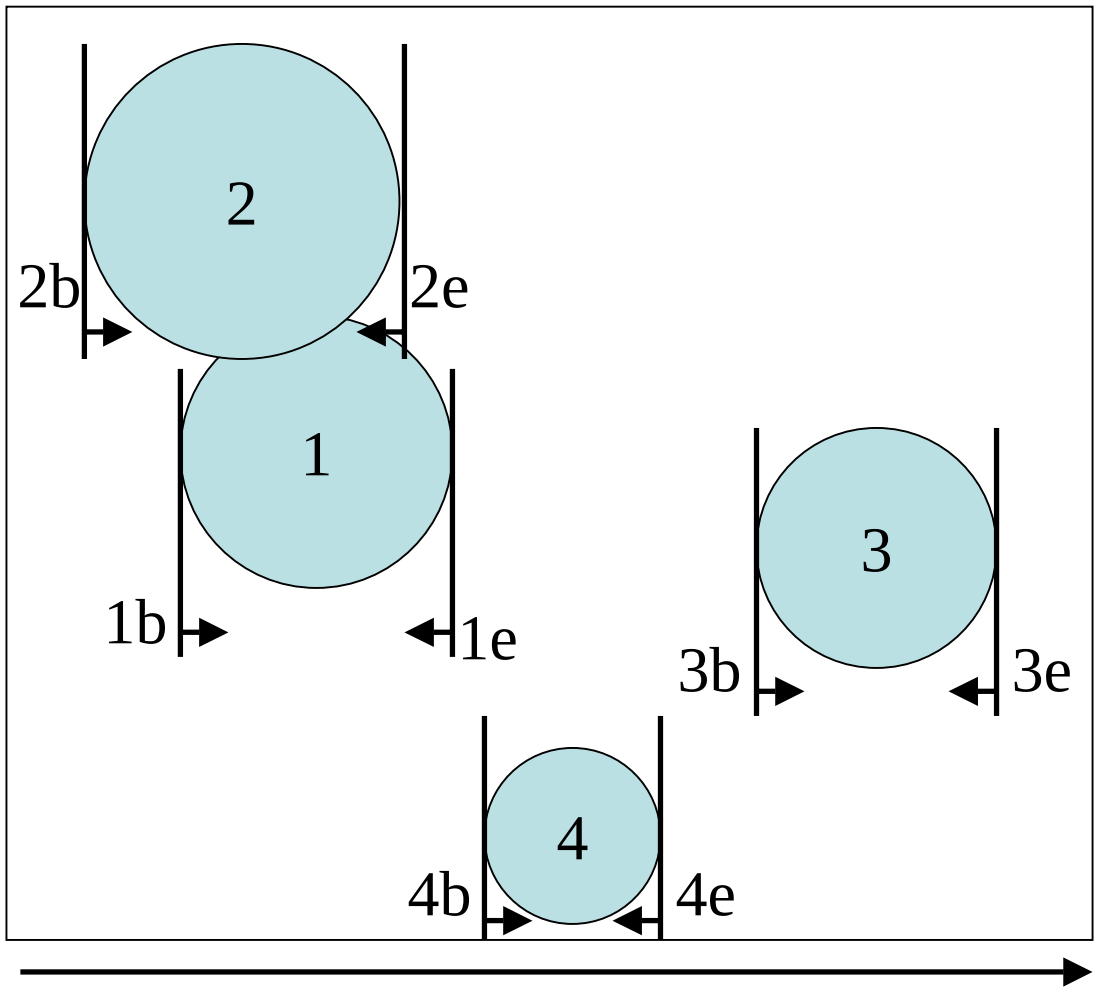

1-2
1-3
1-4
2-3
2-4
3-4

$\frac{n \cdot (n-1)}{2}$回

重なっているかチェック

# Sweep and prune

- 最初にある軸についてソート
  - 重なっている可能性のあるものだけを判定

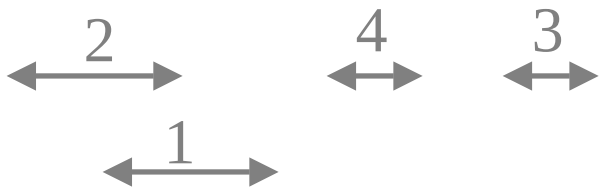



ソート

2b 1b 2e 1e 4b 4e 3b 3e

左から重なりをチェック

2b 1b 2e 1e 4b 4e 3b 3e




1-2

n回チェック

ソート+n回チェック

# ソートのアルゴリズム

- **Selection sort　(総当り)**

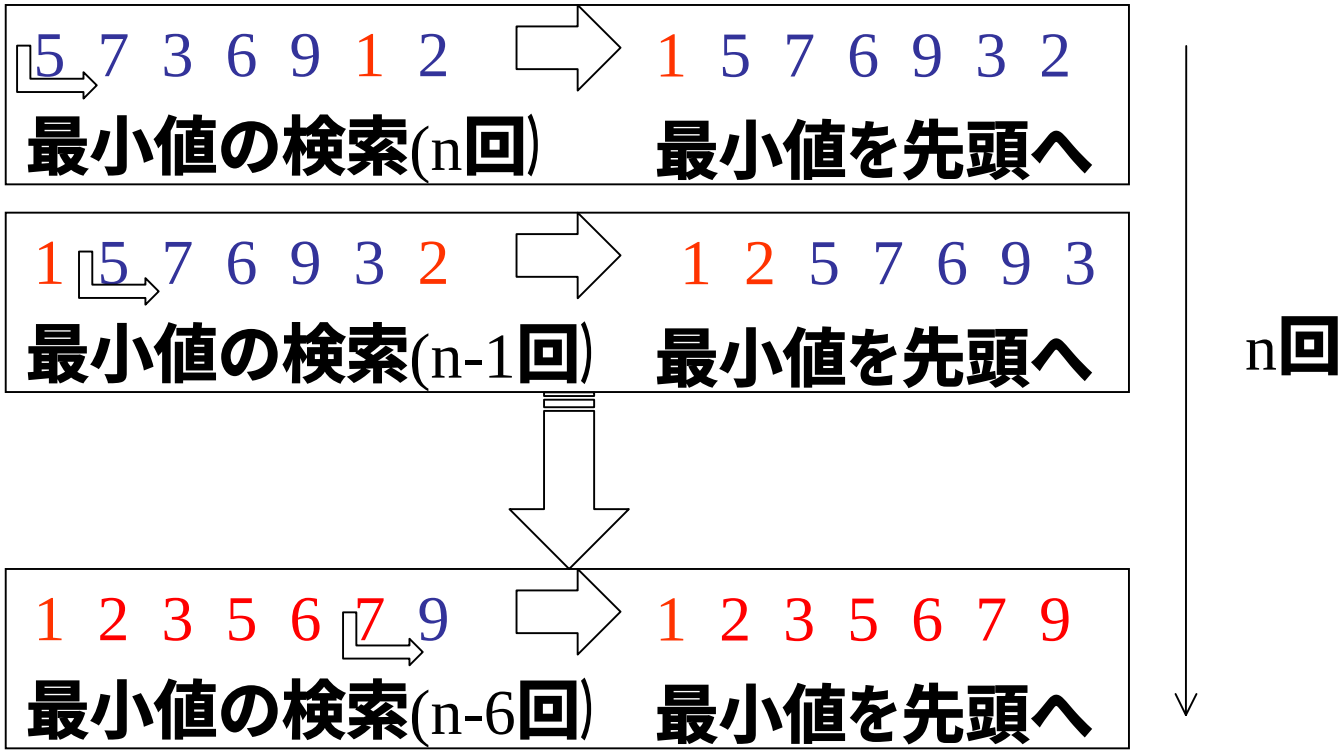
  $O(n^2)$

- **Quick sort**

  $O(n \log n)$

# Selection sort の計算量

ソート前： 5 7 3 6 9 1 2 (n=7)

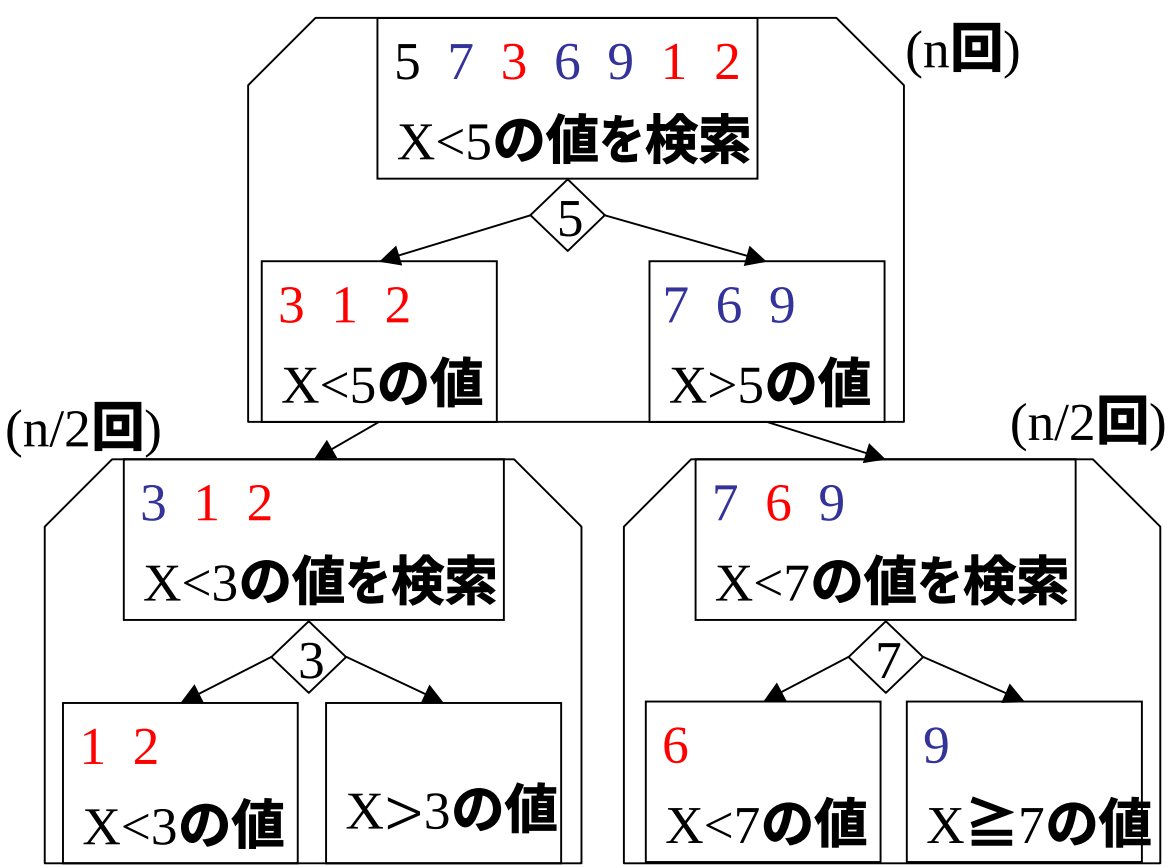



比較の回数：$\sum_{i=1}^{n} n-i = \frac{(n-1)(n-2)}{2}$

計算量のオーダー：$O(n^2)$

# Quick sort の計算量

ソート前： 5 7 3 6 9 1 2 (n=7)




$\log n$ 階層

比較の回数：$n + 2\frac{n}{2} + 2^2\frac{n}{2^2} + \ldots = \sum_{i=1}^{\log_2 n} 2^{(i-1)}\frac{n}{2^{(i-1)}} = n\log_2 n$

計算量のオーダー：$O(n\log n)$

# 計算量の考え方

- 計算量のオーダー
  - 計算量：CPUで動かすときの命令数
    - 対象データが変われば，命令数も変わる
      - アルゴリズムごとの計算速度の比較にはなりにくい
      - データによらない，と〜っても大雑把な指標が欲しい
  - n → ∞ のときの計算量を比較
    - 当然計算量は，∞　　でも　∞にも速さの違いがあります．
    - 計算量のオーダー O( f(n) )
      - アルゴリズムA がある．
      - データの数がnのときのAの計算の命令数 = a(n) とする
      - データの数が∞の時の命令数を考える

$$\lim_{n\to\infty}\frac{a(n)}{n^2}=\text{定数}\Leftrightarrow O(a(n))=O(n^2)$$

このとき，アルゴリズムAのオーダーは $O(n^2)$

# 計算量のオーダー

- アルゴリズムの速さの大雑把な比較に便利
  - 詳細な比較をしても，データやCPUによるので
    アルゴリズム自体の速さの比較にはなりにくい
  - オーダーの性質

$$O(1) = O(10) = O(\text{定数})$$

$$O(n) = O(10n) = O(n\text{の定数倍})O(n) = O(10n) = O(n\text{の定数倍})$$

$$O(n^2) = O(3n^2 + 2n) = O(n\text{次式})$$

  - 代表的な意味のあるオーダー

$O(1)$

$O(\log n), O(\log^2 n), \ldots$

$O(n)$

$O(n \log n)$

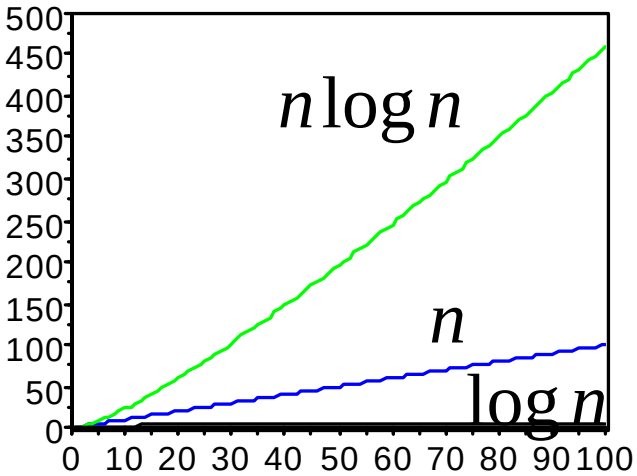
$O(n^2), O(n^3), \ldots$

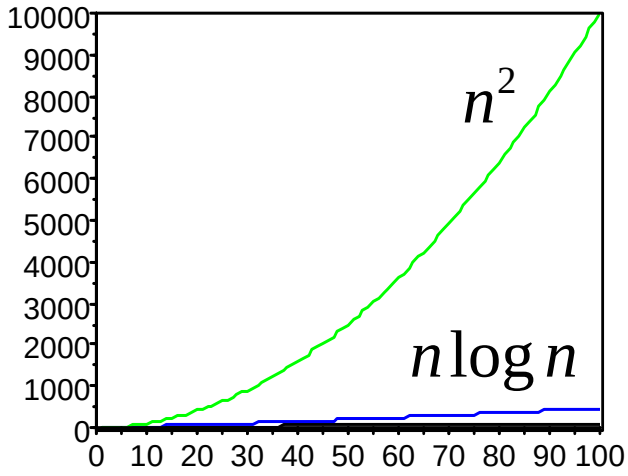
$O(2^n)$

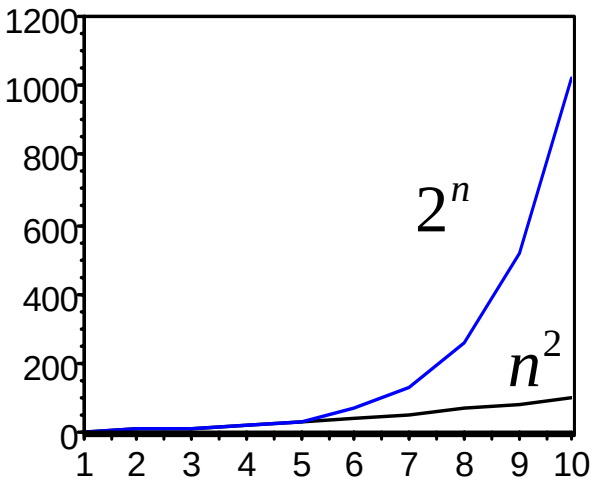
$O(n!)$

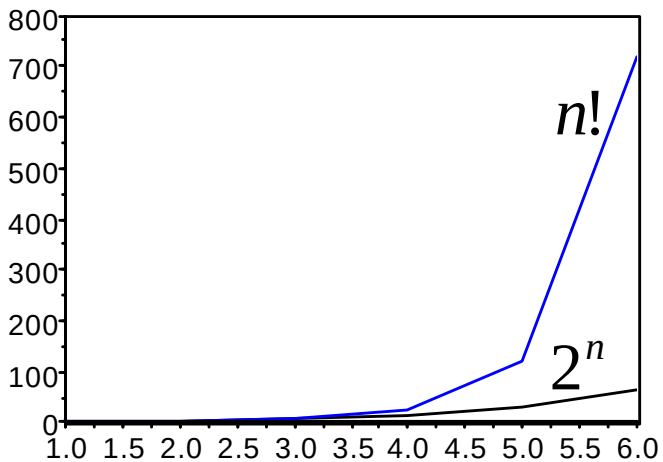
# 計算量のオーダー

- nが大きい場合はオーダーの違いが利く



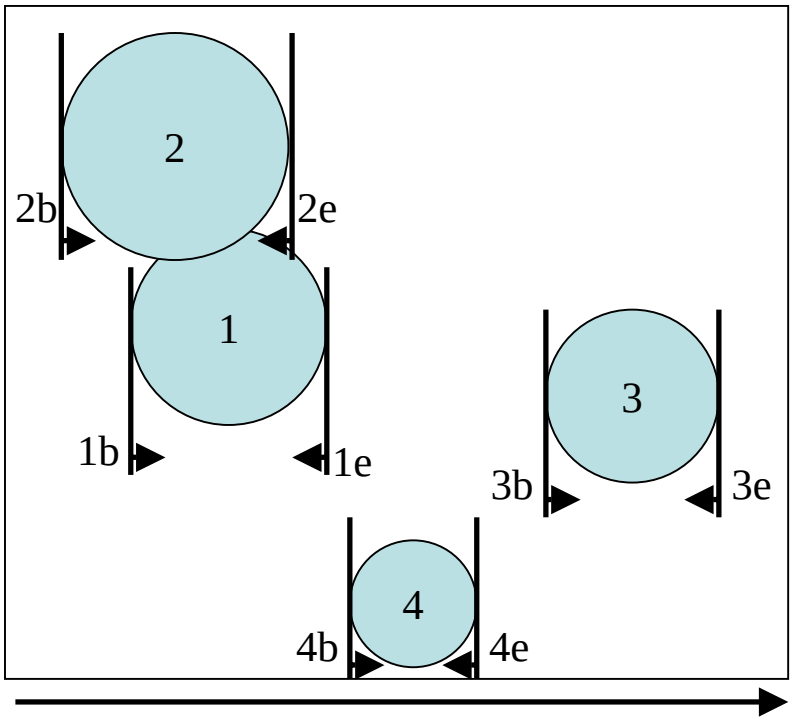



# Sweep and prune

- ソート→重なっている可能性のあるものだけを判定




ソート

2b 1b 2e 1e 4b 4e 3b 3e

左から重なりをチェック

2b 1b 2e 1e 4b 4e 3b 3e

2 4 3

1

1-2

n回チェック

ソート1個分の計算量

チェック1個分の計算量

計算量：ソート＋$n$回チェック

$$O(\text{QuickSort}+n\text{回チェック}) = O(c_1 n\log n + c_2 n)$$

$$= O(n\log n + n) = O(n\log n)$$

オーダーには影響しない

# オーダーの計算の確認

- $O(c_1 n \log n + c_2 n) = O(n \log n)$の確認

$$\lim_{n\to\infty} \frac{c_1 n \log n + c_2 n}{n \log n} = \lim_{n\to\infty} \frac{c_1 n \log n}{n \log n} + \frac{c_2 n}{n \log n}$$

$$= c_1 + \lim_{n\to\infty} \frac{c_2}{\log n} = c_1$$

- 参考：$O(c_1 n \log n + c_2 n) \neq O(n^2)$の確認

$$\lim_{n\to\infty} \frac{c_1 n \log n + c_2 n}{n^2} = \lim_{n\to\infty} \frac{c_1 n \log n}{n^2} + \frac{c_2 n}{n^2}$$

$$= \lim_{n\to\infty} \frac{c_1 \log n}{n} + \frac{c_2}{n} = 0$$

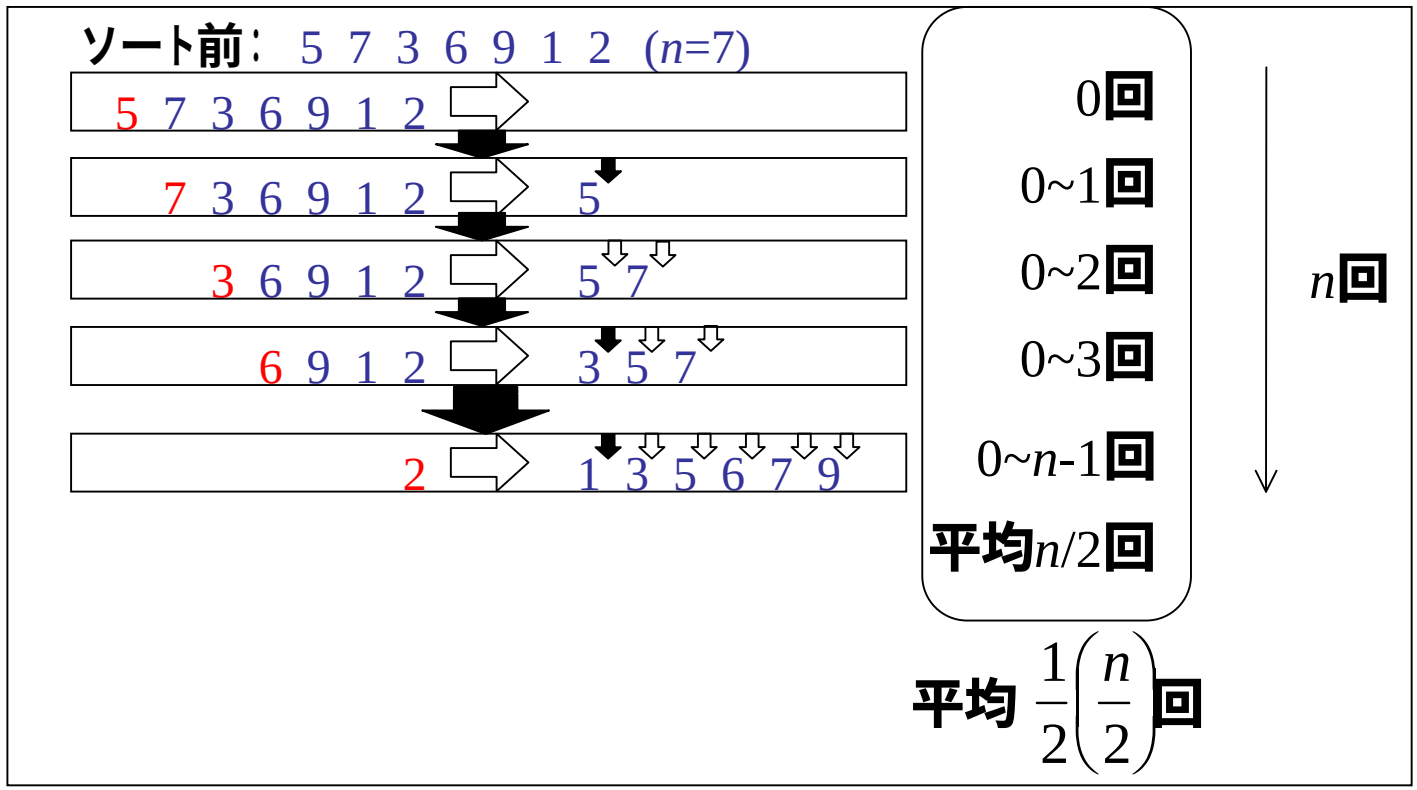
# Sweep and pruneの計算量

- 総当りの判定 → $O(n^2)$
  - Selection sort → $O(n^2)$
- Sweep and prune → $O(n \log n)$
  - Quick sort → $O(n \log n)$

- Sweep and pruneとソート
  - 物体間に ＜ 演算 を導入
  - 高速なソートのアルゴリズムを利用

# Sweep and prune に最適なソート

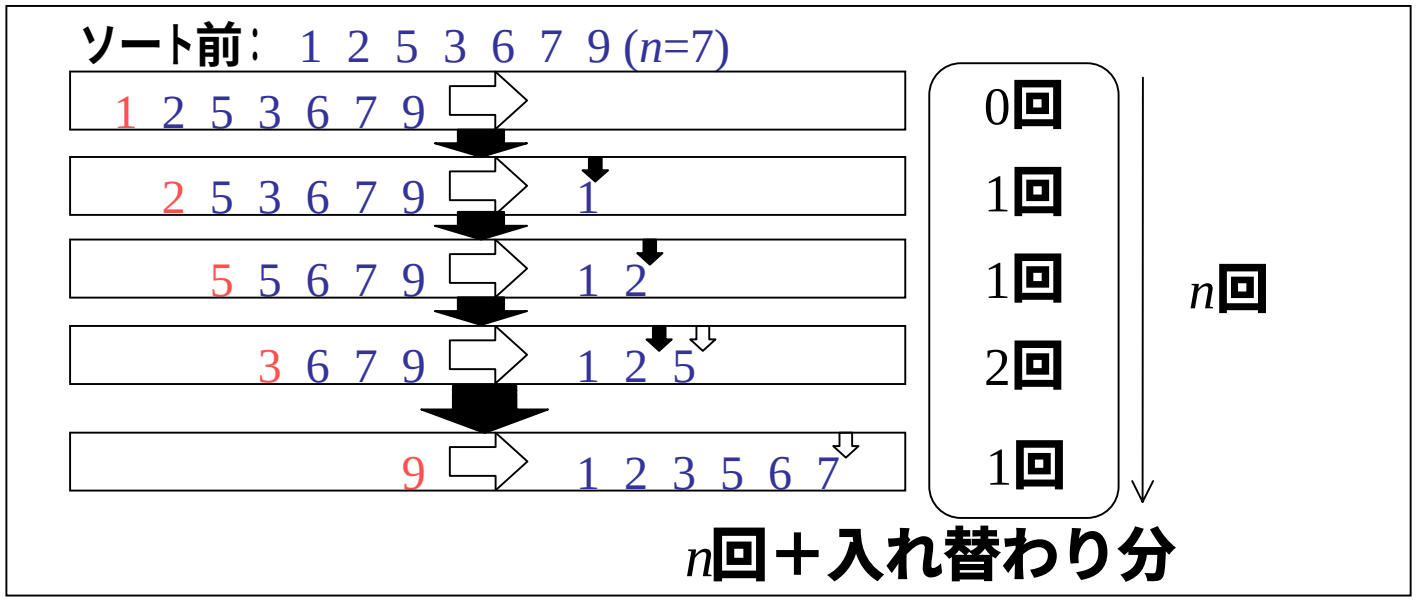
- 物理シミュレーション → 物体は，それほど入れ替わらない
  → ほとんどソート済み → Insert sort が早い
- Insert sort 通常時：




$$\frac{1}{2}\left(\frac{n}{2}\right)\times n=\frac{n^2}{4} \qquad O(n^2)$$

# Sweep and prune に最適なソート

- Insert sort ほとんどソート済みの場合：




$$O(n+\text{入れ替わり分})=O(n+\alpha n)=O(n)$$

（$\alpha$：隣との入れ替わりが起こる割合）

- 参考：ランダムに入れ替わる場合：

$$O(n+\text{入れ替わり分})=O(n+\alpha n\cdot\beta n)=O(n^2)$$

$\alpha$：入れ替わりが起こる 割合，

$\beta$：入れ替わり先の位置 の比

1 2 7 5 6 3 9

$$\beta=\frac{3}{7}$$

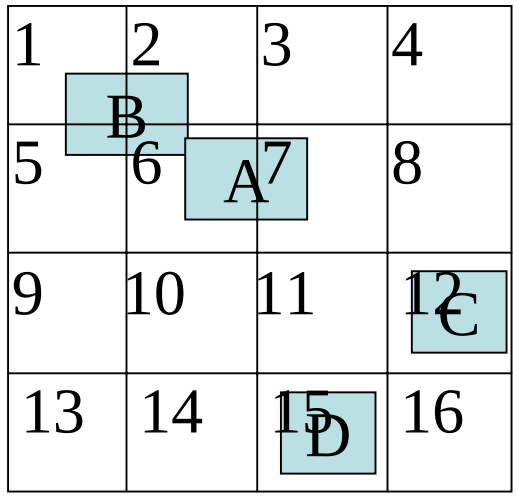
# Sweep and pruneの計算量

- 総当りの判定 → $O(n^2)$
- Sweep and prune → $O$(ソート＋$n$回チェック)
  - Quick sort を使う場合 → $O(n \log n)$
  - Insertion sort2度目以降→ $O(n)$

- Sweep and pruneとソート
  - 物体間に ＜ 演算 を導入
  - 高速なソートのアルゴリズムを利用
  - 前回の計算結果を利用

# Uniform grid と bucket sort (バケツソート)

- Uniform grid




| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | B | | | B | A<br>B | A | | | | | C | | | D | D |

1物体をグリッドに放り込むための計算量：

上下限の計算＋グリッド数・登録：$O(1)$

チェックに必要な計算量：

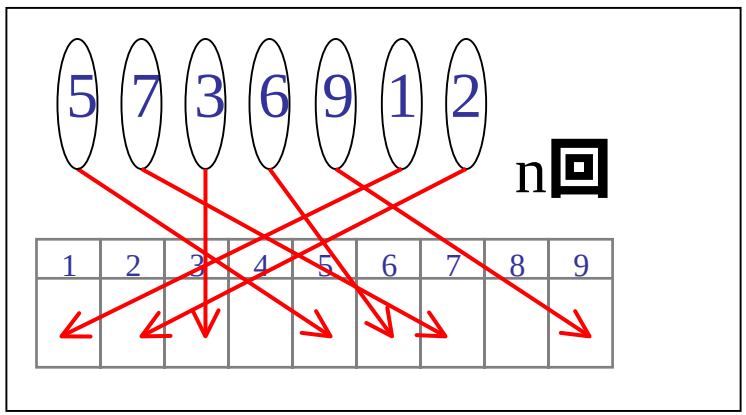
グリッド数・登録数チェック：$O(1)$　　全体で $O(n)$

# Bucket sort

ソート前： 5 7 3 6 9 1 2 (n=7)

格納領域を準備

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |




1回

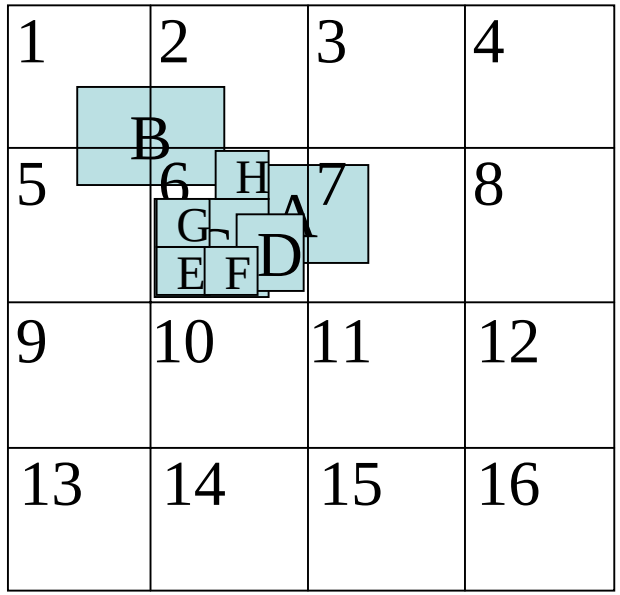
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | | 5 | 6 | 7 | | 9 |

$O(n)$

- バケツソートはなぜ早いか？
  - データの種類＝バケツの数
    - データが16bit整数なら，バケツが65536個必要

# Bucket sort の計算量

- Uniform gridは本当に早いか？




↓を作るのにかかる計算量：$O(n)$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | B | | | B | AB<br>CD<br>EF<br>GH | A | | | | | C | | | D | D |

総当りチェックなら　$O(n_1^{\,2})$

# Bucket sortの計算量2

- データの種類が∞の場合の計算量

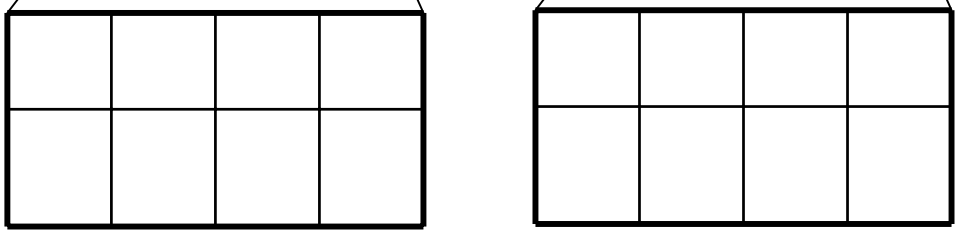
3.1 2.4 2 2.2 2.9 8 9.2 1 5

| 0～2 | 2～4 | 4～6 | 6～8 | 8～10 |
|---|---|---|---|---|
| 1 | 2 2.2 2.4 2.9 3.1 | 5 | 8 | 9.2 |

n個のデータをバケツに入れる

| 2～2.4 | 2.4～2.8 | 2.8～3.2 | 3.2～3.6 | 3.6～4 |
|---|---|---|---|---|
| 2 2.2 2.4 | | 2.9 3.1 | | |

n個のデータをバケツに入れる



n個のデータをバケツに入れる

運がよければ $\log_4 n$ 階層

全体：$O(n \log n)$

# Quick sort と bucket sort

3.1 2.4 2 2.2 2.9 8 9.2 1 5

| ～3.1 | 3.1～ |
|---|---|
| 2.4 2 2.2<br>2.9 1 | 3.1 8 9.2 5 |

| ～2 | 2～ |
|---|---|
| 1 | 2 2.4<br>2.2 2.9 |

| ～8 | 5～ |
|---|---|
| 3.1 5 | 8 9.2 |

| ～2.4 | 2.4～ |
|---|---|
| 2 2.2 | 2.4 2.9 |

n個のデータをバケツに入れる

n個のデータをバケツに入れる

n個のデータをバケツに入れる

**運がよければ** $\log_2 n$ **階層**

**全体：** $O(n \log n)$

Quick sortもbucket sort(**データの種類∞の場合**)**の一種**

# Quick sortの速さの元

- Quick sortは，
  n個の比較をするたびに，データの数が半分に
  - 半分の 半分の … n, n/2, n/4, n/8… → $\log_2 n$回

$$O(n \log n)$$

- Selection sort だと，
  - 一番小さな物を取り出す… n, n-1, n-2, n-3 … → n回

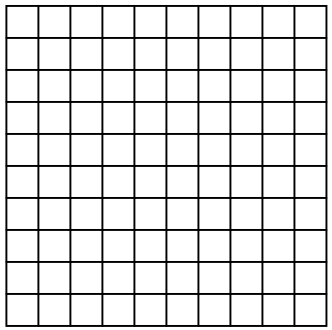
$$O(n \cdot n)$$

n個処理するたびに

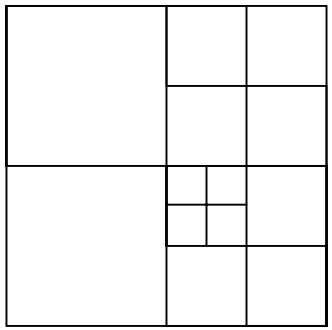
問題の大きさが○割になる → $O(n \log n)$ 複利

問題の大きさが○減る → $O(n^2)$ 単利

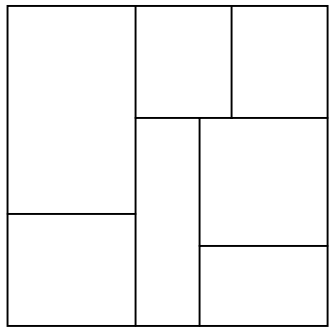
# Gridとバケツソート



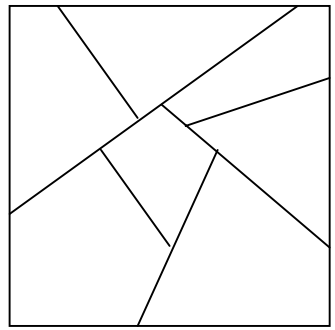
Uniform grid



Quadtree



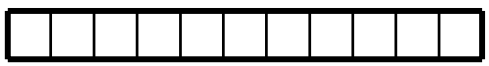
k-d tree



BSP

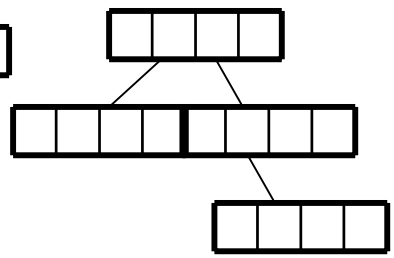
Bucket sort

**階層が高さ1に**
**なることを期待**



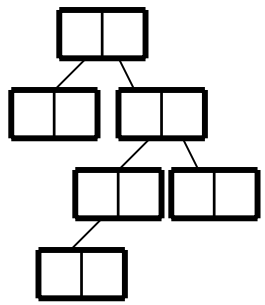
Bucket sort

**階層あり，**
**バケツ4つ**



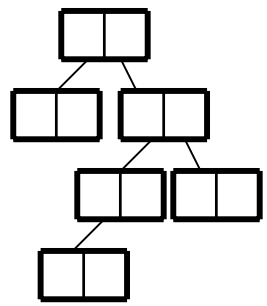
Bucket sort

**階層あり，**
**バケツk個**



Quick sort

**階層あり，**
**バケツ2つ**



# ソート以外の方法

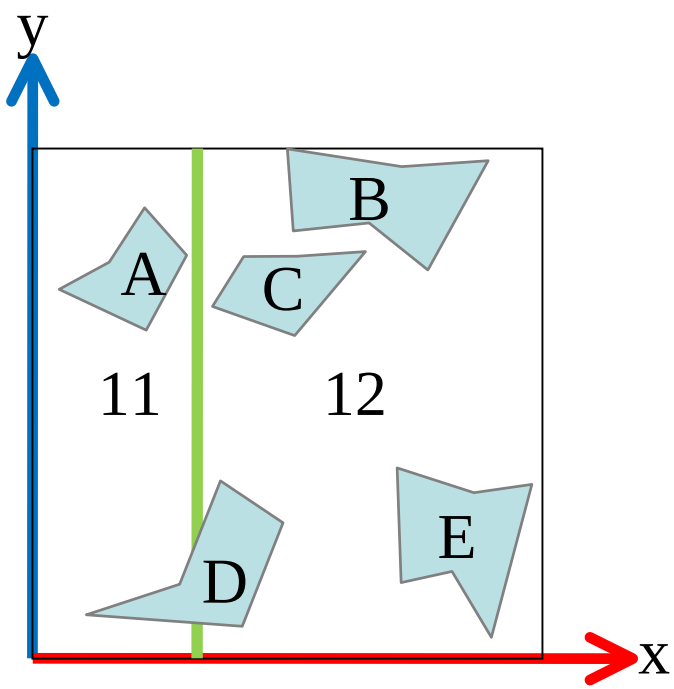
- 3次元空間の物体の位置関係→大小関係とは異なる
  ソートできるとは限らない

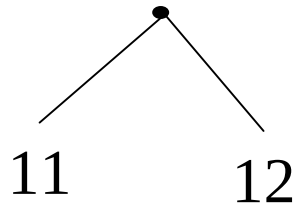
- 3次元のアルゴリズムも，
  ソート・検索と対比できる部分がある
  - BSP　(Binary space partitioning: 空間分割)
  - BVH　(Bounding Volume Hierarchy: BBoxの階層)

# kd Tree

- 軸方向に分割を繰り返してツリーを作る
- kd Treeのkは分割数を表す

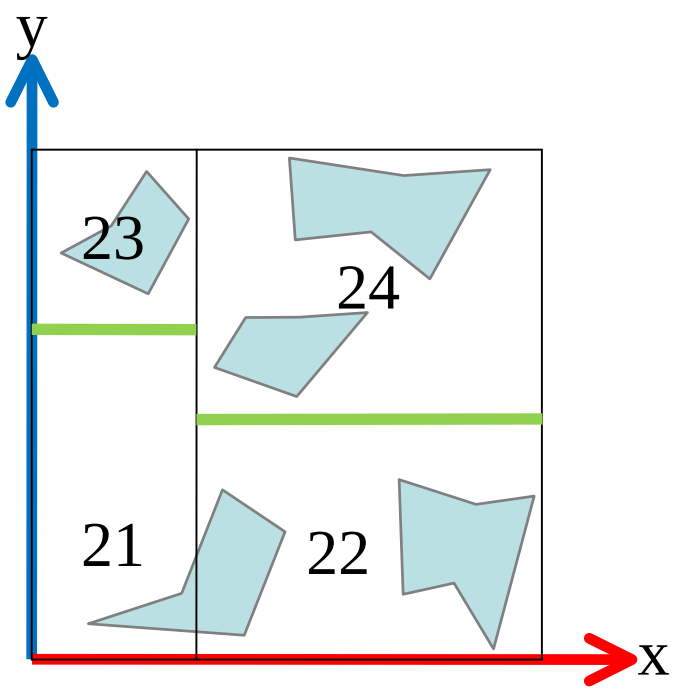
x軸方向で分割

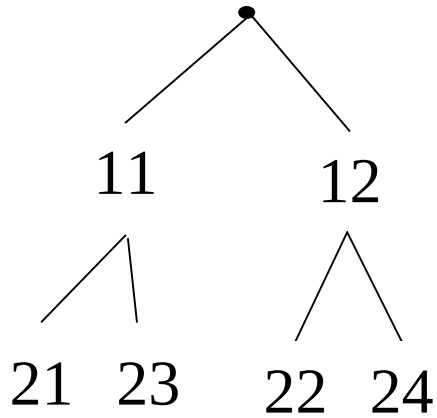


# kd Tree

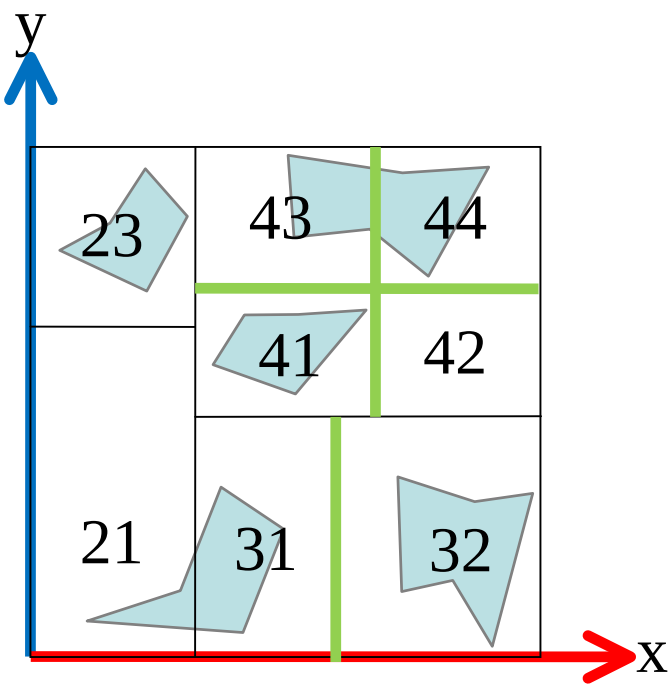
- y軸方向で分割

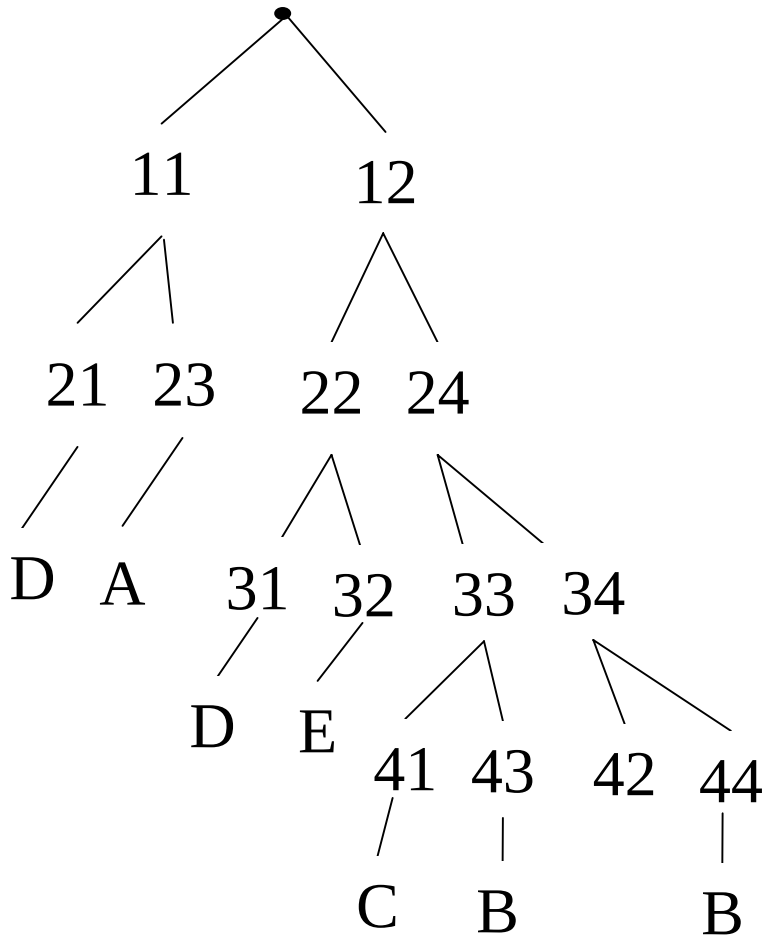


# kd Tree

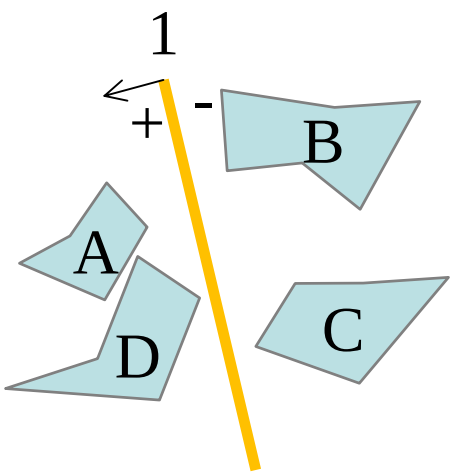
- 再度x軸方向，y軸方向に分割
- 図は軸方向に2分割しているので2d Tree
- ツリー構築は $O(n \log n)$

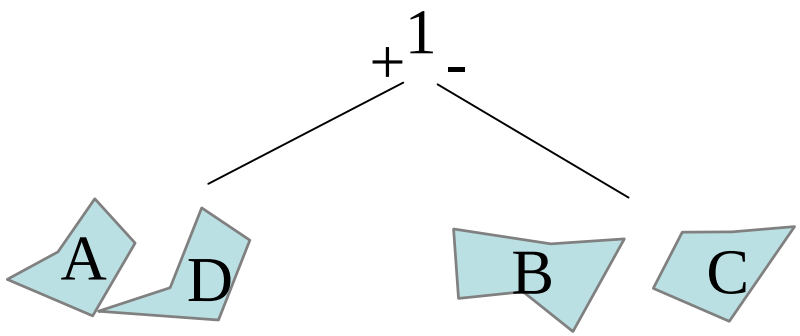


# BSP(Binary Space-Paritioning Tree)

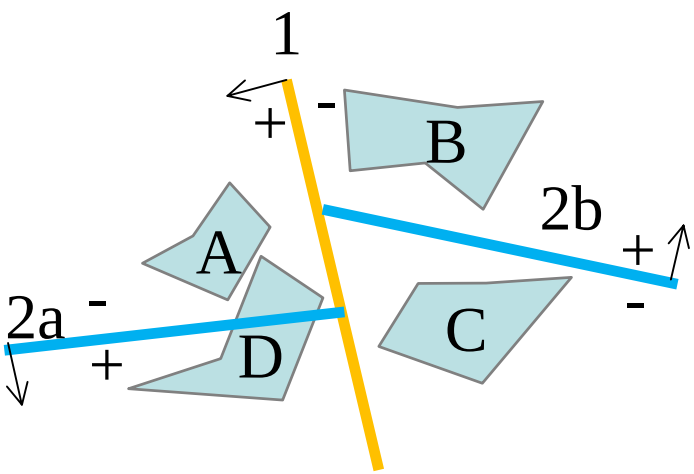
- kd Treeは座標軸方向で分割をする
- BSPは任意の方向で2分割をする
- 分割された空間を正の半空間，負の半空間と呼ぶ

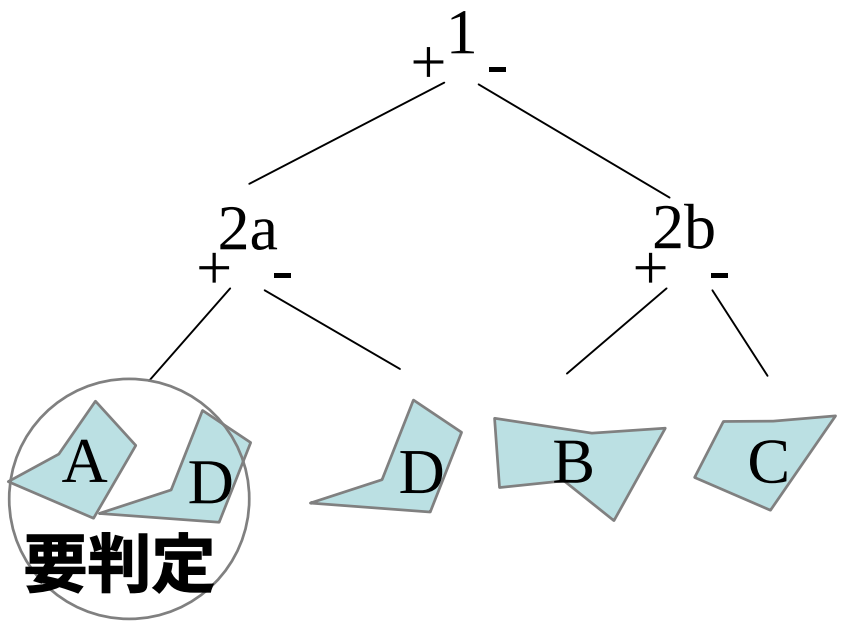


# BSPの構築

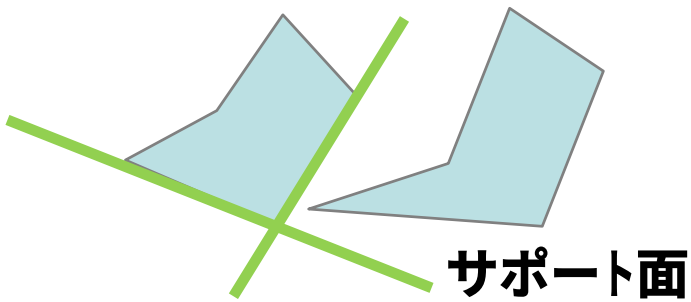
- 分割された空間をさらに2分割していくので
  2分木ができる
- BSPの構築はクイックソートと同様のO(n log n)



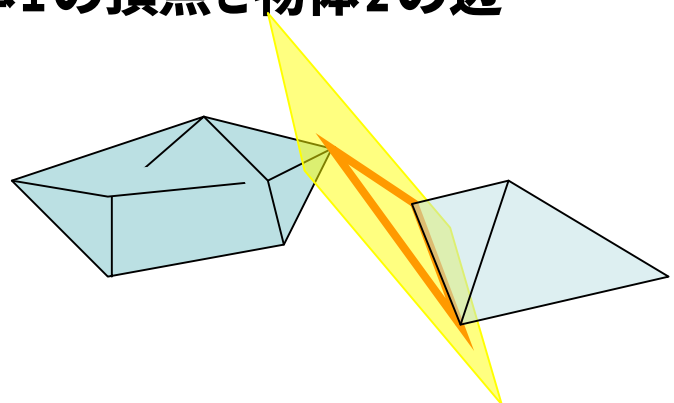



# BSP構築のための分割平面の選択

- 分割平面の選択によりツリーの構造が大きく変わる
- 分割面は無限に存在
  - 最適なものを一意に決定できない
    - 例1：中身の形状のサポート面で分割してみる

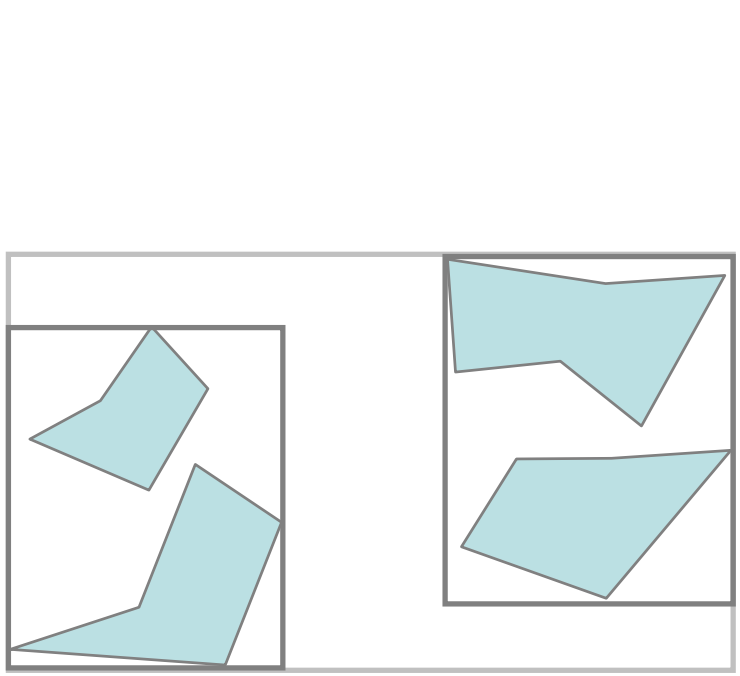


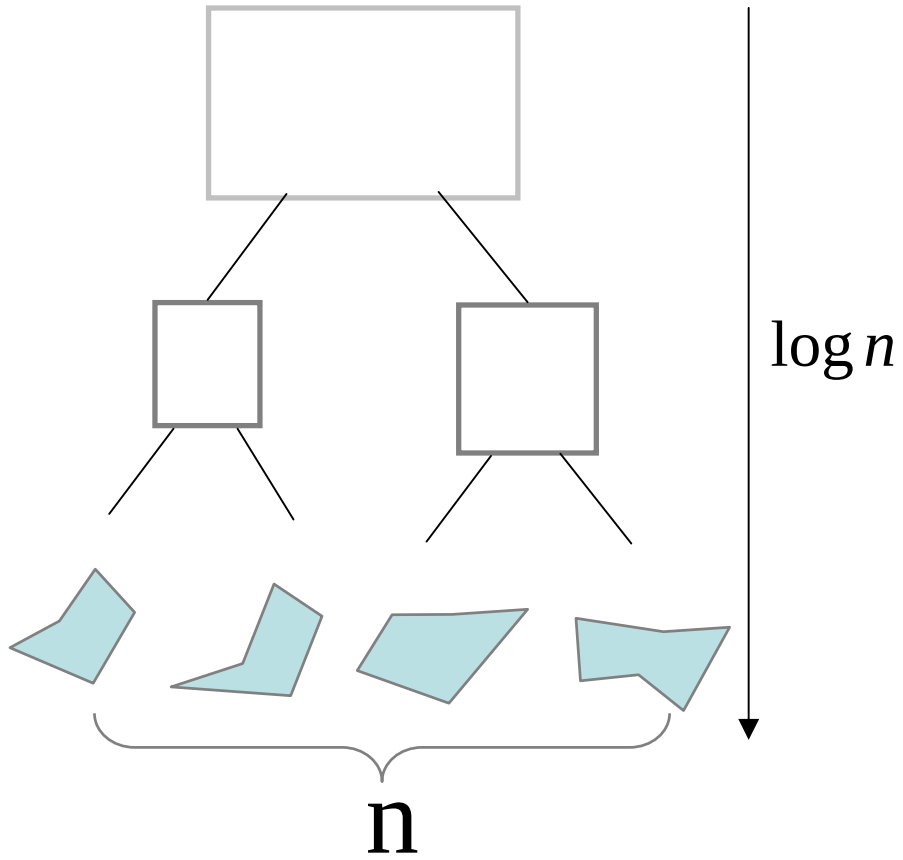

    - 例2：物体1の頂点と物体2の辺



# BVH (Bounding Volume Hierarchies)

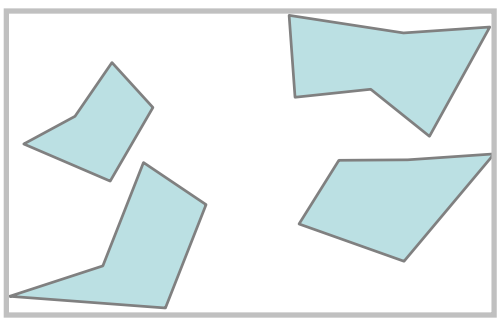
- 境界ボリュームで階層構造（ツリー）を作る
- log n回で衝突検出ができる

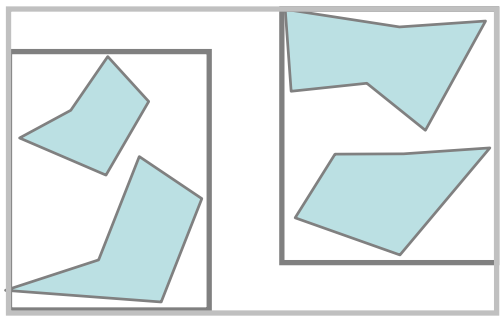


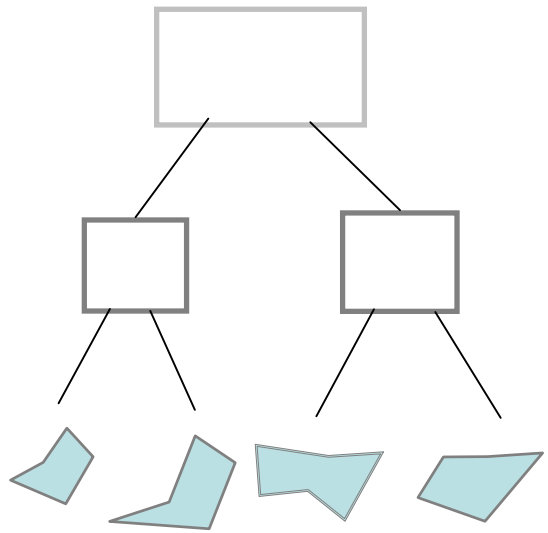


# BVHの構築　トップダウン法

- BSPを構築 → BVを作る
- $O(n \log n)$





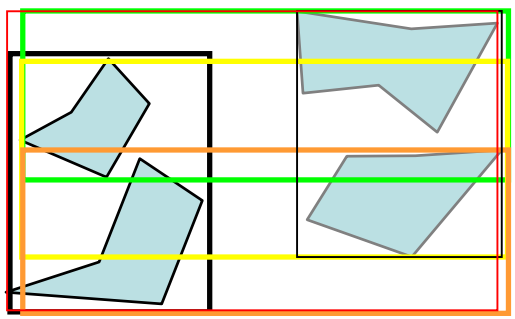


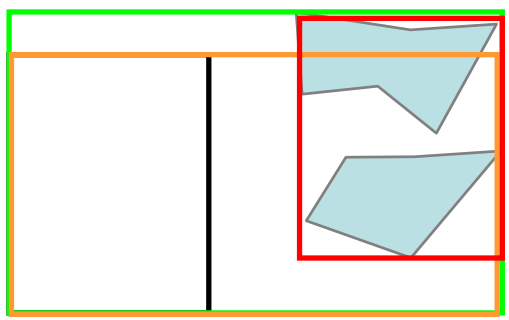
# BVHの構築　ボトムアップ法

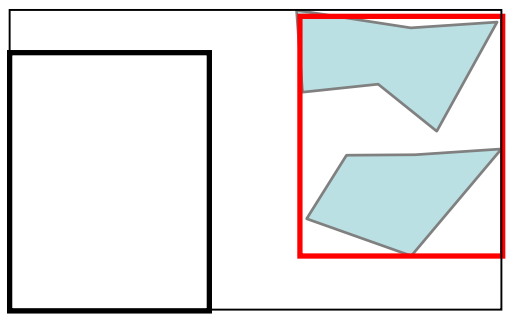
- オブジェクトのペアで作るBounding Volume(BV)が
- 最小になるようにBVを作る
- すべてのペアでBVを作る必要がある
- 最良なツリーができる

1. すべてのペアのBVを計算　　$n(n-1)/2$
2. 一番体積が小さいものを採用　　$(n-1)$ 回

全体で$O(n^3)$



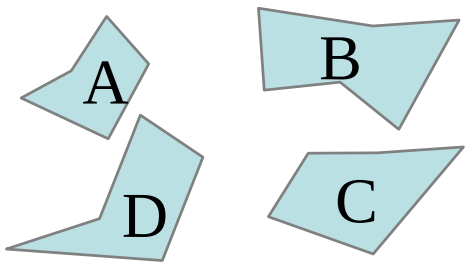


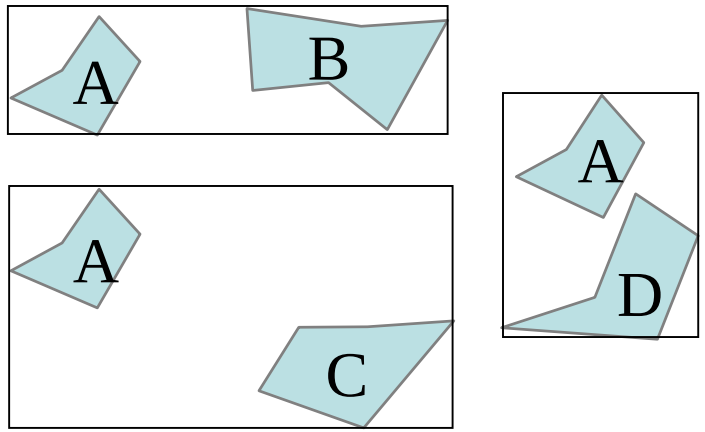


# BVHの構築　改良ボトムアップ法

- オブジェクトのペアのボリュームを列挙

Aについて




| | A-B | A-C | A-D |
|---|---|---|---|
| ボリュームの大きさ | 3 | 6 | 1 |

*n-1*回

- Bounding Volume(BV)が最小の物を（A-D）採用

B,C,Dについても同様に探す　*n-1*回

両方で$O(n^2)$

# BVHの構築　改良ボトムアップ法

- 見つかった，ペアを並べてQueueをつくり，

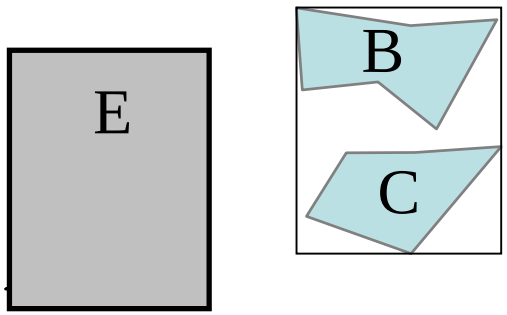
| A-D | B-C | C-B | D-A |
|---|---|---|---|
| 1 | 2 | 2 | 1 |

- Bounding Volume (BV) が
  小さい順にソート

| A-D | D-A | B-C | C-B |
|---|---|---|---|
| 1 | 1 | 2 | 2 |

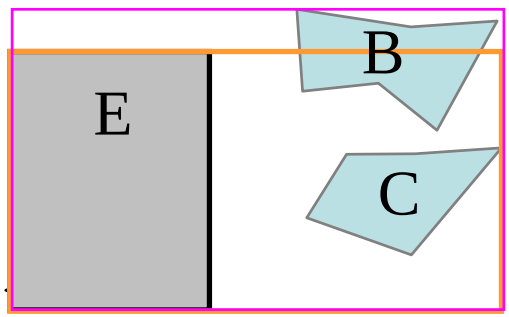
# BVHの構築　改良ボトムアップ法

- A-Dを確定 → Eとする




| A-D | D-A | B-C | C-B |
|---|---|---|---|
| 1 | 1 | 2 | 2 |

- Eと残っているノードでペアを作成→BV最小の物を追加




E-C:7, E-B: 8　　n-i-1回VB計算

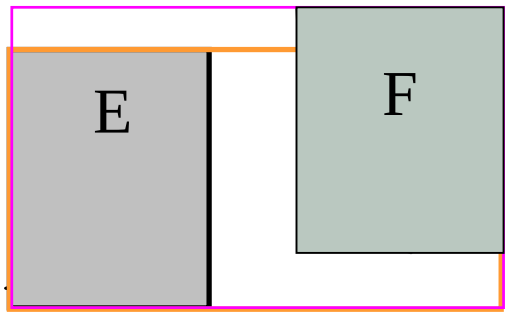
| D-A | B-C | C-B | E-C |
|---|---|---|---|
| 1 | 2 | 2 | 7 |

# BVHの構築　改良ボトムアップ法

- Dはないので，D-Aを取り去る

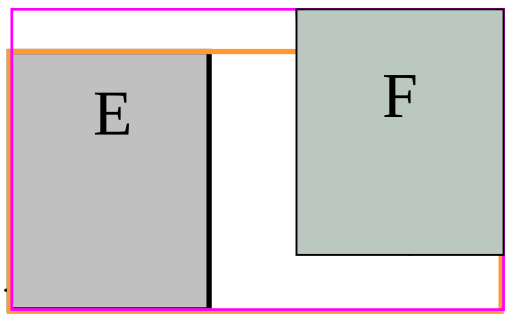
| ~~D-A~~ | B-C | C-B | E-C |
|---|---|---|---|
| 1 | 2 | 2 | 7 |

- B-Cを確定 → Fとする




| B-C | C-B | E-C |
|---|---|---|
| 2 | 2 | 7 |

  - Fと残っているノードでペアを作成→BV最小の物を追加

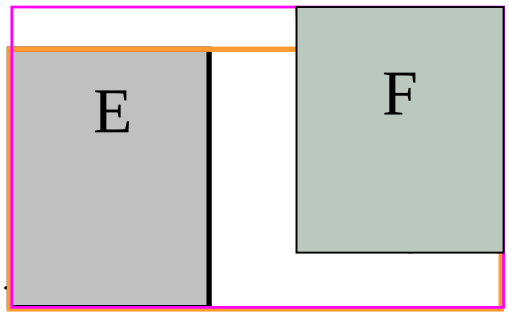



| C-B | E-C | F-E |
|---|---|---|
| 2 | 7 | 8 |

# BVHの構築　改良ボトムアップ法

- Cはないので，C-Bを取り去る
- Cはないので，E-Cを取り去る
- F-Eを確定する

| ~~C-B~~ | ~~E-C~~ | F-E |
|---|---|---|
| 2 | 7 | 8 |




| F-E |
|---|
| 8 |

- ペアのQueueが空になったら終了

- 計算量のオーダーは，全体で$O(n^2)$

# なぜボトムアップBVH構築が遅いのか？

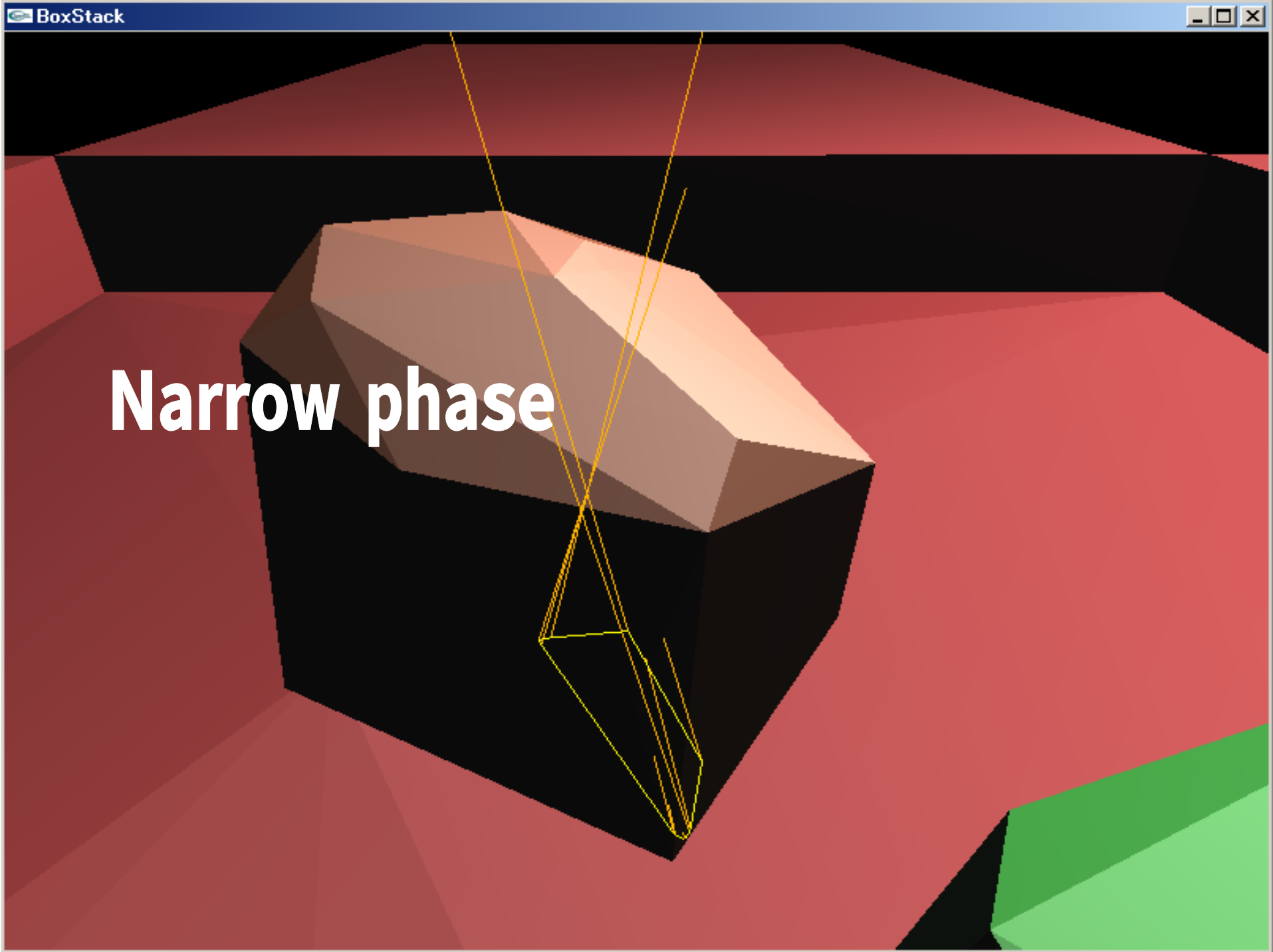
- 何も考えないと $O(n^3)$ 改良版で $O(n^2)$
- ボトムアップBVHは，計算結果を再利用しにくい
  - A-BのBV, A-CのBV が分かっても，
    B-C, B-Dについて何もいえない
- BSP / k-d tree は，結果を再利用できている
  - 最初の面より，A,Dが左，B,Cが右
  - 以降，A-B, D-C など面をまたぐものは考えないでよい

# Broad phase こんなときにはこのアルゴリズム

- Bounding Volumeの位置・大きさが

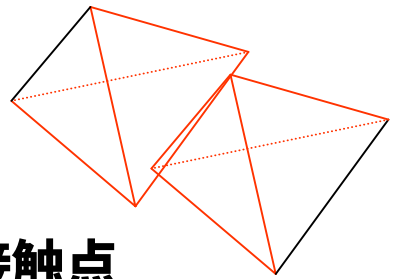

  - 剛体を構成する形状をBVH　剛体:Sweep & prune
- 物体が


- だと思いますが，実際はケースバイケースです
  - 物体の並び方，シーンのサイズ etc. で変わります

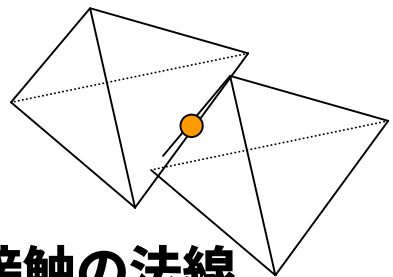

BoxStack
Narrow phase

# 物理エンジンに必要な接触情報

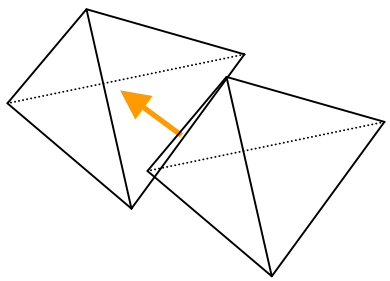
- Narrow phase：具体的な形状（ポリゴン）同士の判定
  - 接触情報の種類
    - 接触したポリゴン



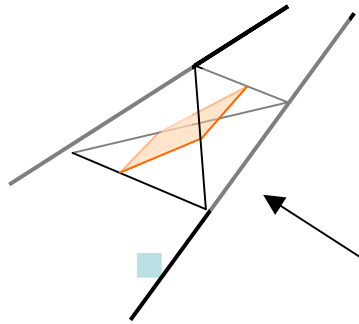
    - 接触点



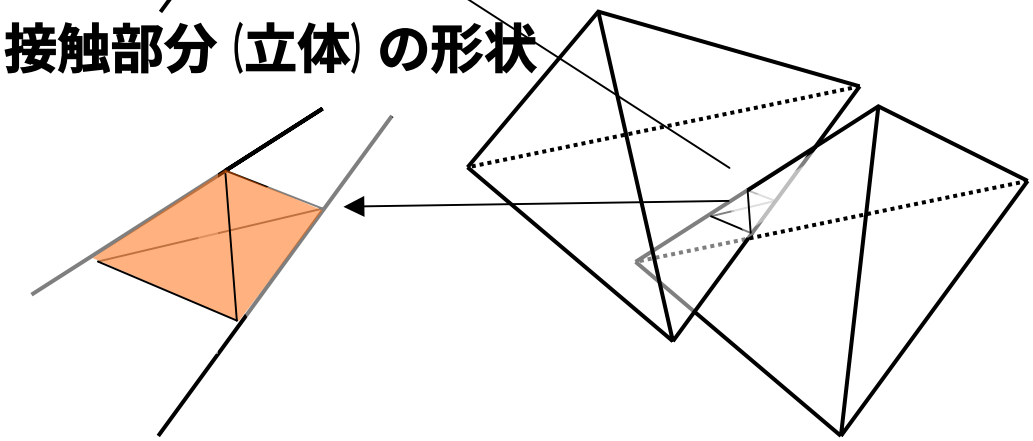
    - 接触の法線



    - 接触領域（面）の形状

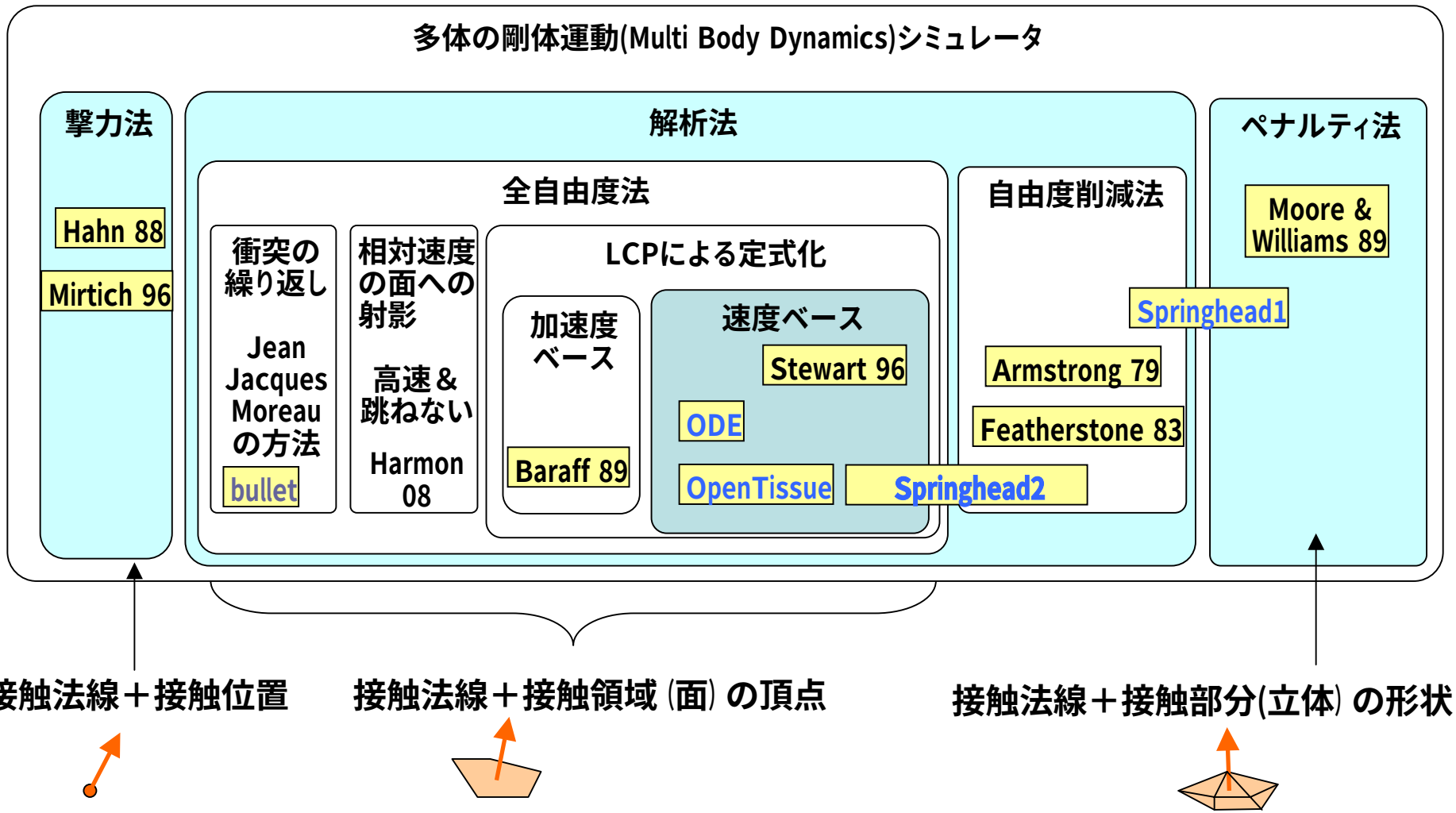


    - 接触部分（立体）の形状



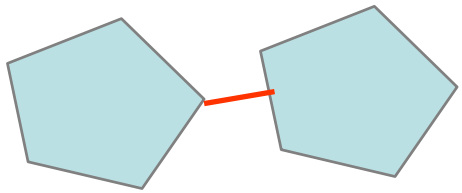
# 物理エンジンの分類と接触判定

- 物理エンジンの分類
  - 接触力計算の方法による

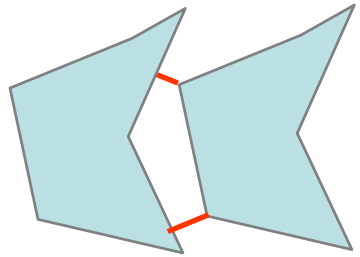


# 接触点と法線を求める

- 形状の表現
  - 形状は、 ポリゴン？ プリミティブ？ …
  - 凸多面体



凸形状

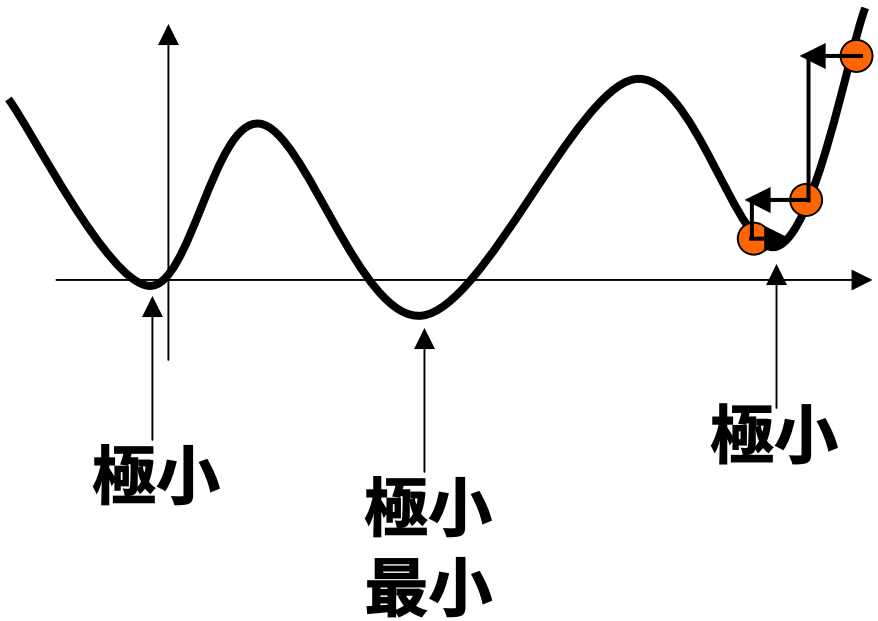


非凸形状

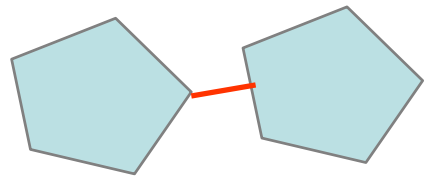
  - 凸形状 距離が極小となる点が1点 → 最近傍点が簡単に求まる
  - GJK algorithm
    - E. G. Gilbert, D. W. Johnson and S. S. Keerthi
    - A Fast Procedure for Computing the Distance between Complex Objects in Three-Dimensional Space (1988)
  - Lin-Canny algorithm というのもあります
    - Ming Lin, John Canny
    - QA fast algorithm for incremental distance calculation (1991)
    - V-Clip

# なぜ，凸形状

- 一般には，極小は簡単，
- 最小は難しい

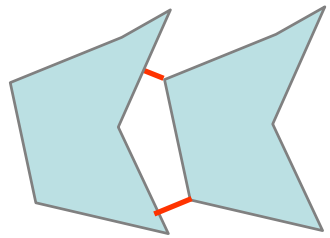



凸形状は極小が一つ
極小＝最小



凸形状

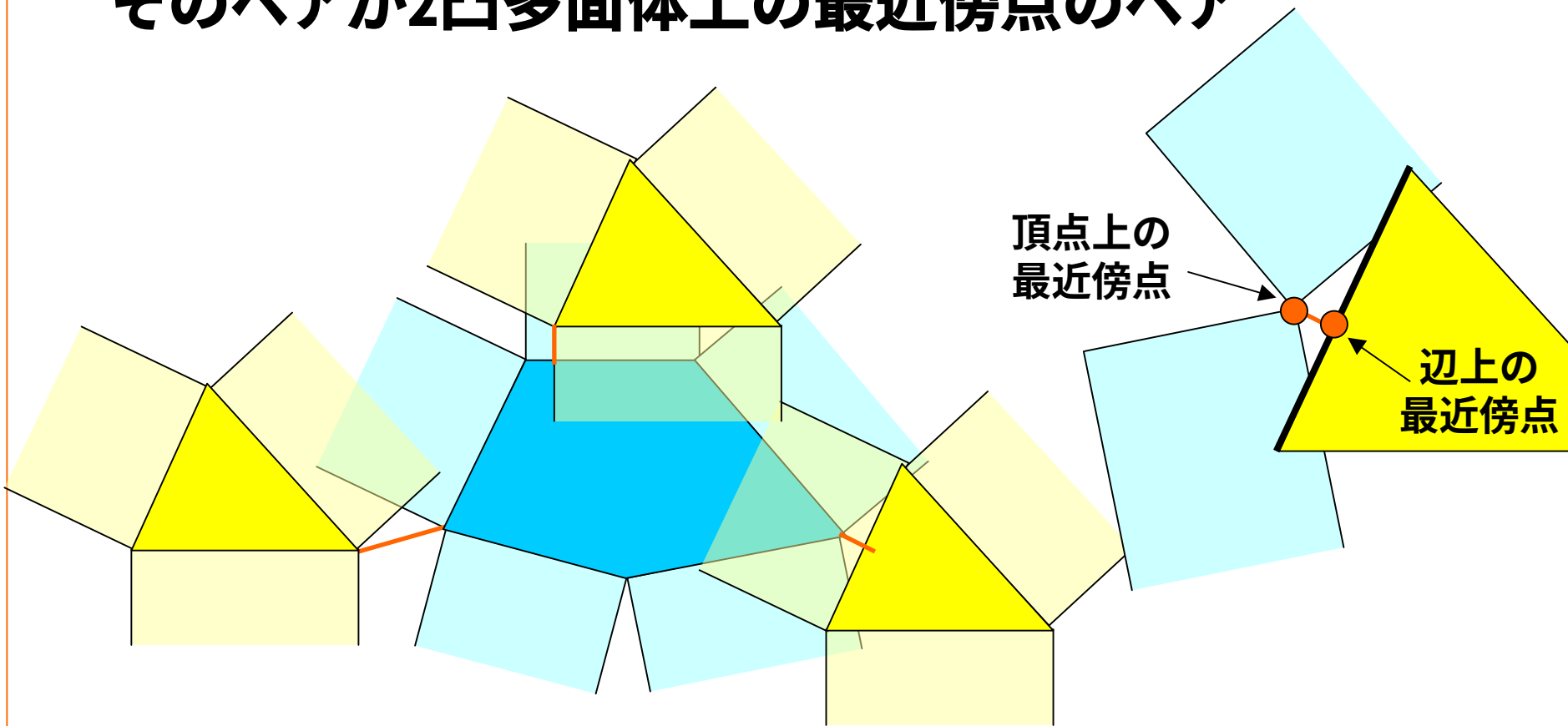
非凸形状は極小が複数



非凸形状

# Lin-Canny algorithm

- 頂点/辺/面のボロノイ領域に，もう一つの凸多面体上の頂点/辺/面上の最近傍点が含まれていれば，そのペアが2凸多面体上の最近傍点のペア

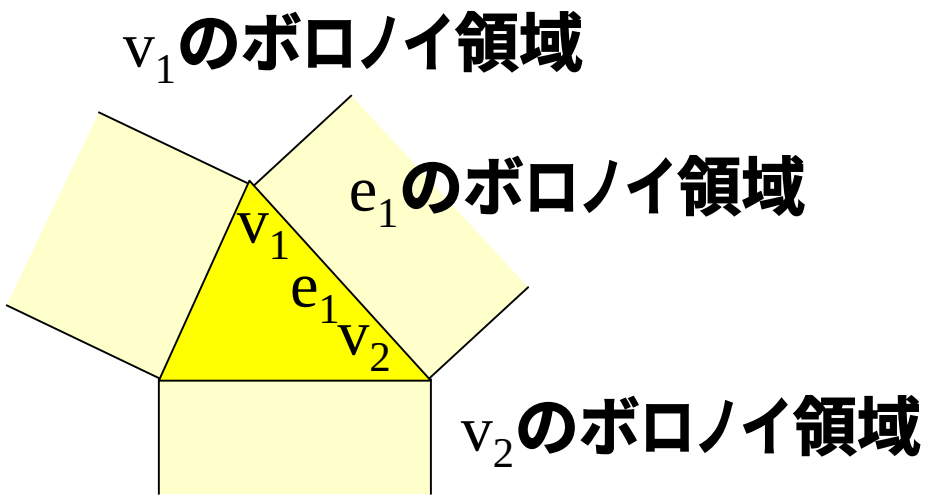



2次元版，5角形と3角形

# Lin-Canny algorithm

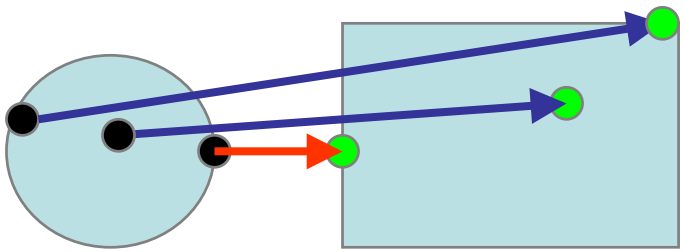
- 含まれていなければ，より近くなる隣の図形へ移動．

- ボロノイ領域
  - ある図形の勢力範囲 → その図形のボロノイ領域
    図形：頂点，辺，面

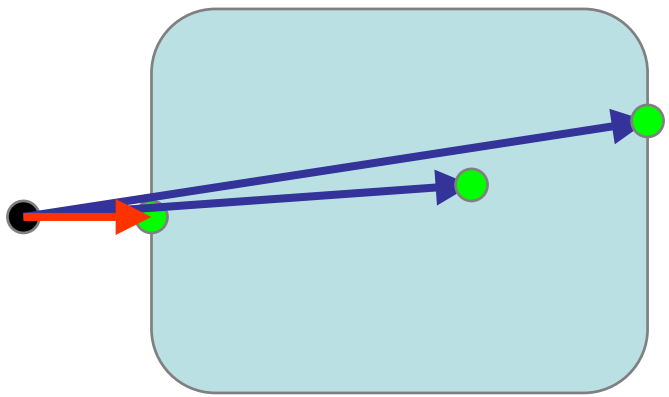


# GJK

凸形状A上の点から，凸形状B上の点へのベクトルを
原点を始点に並べる（Minkowski sum）と
ベクトルの終点の集合も凸形状になる



元の図形

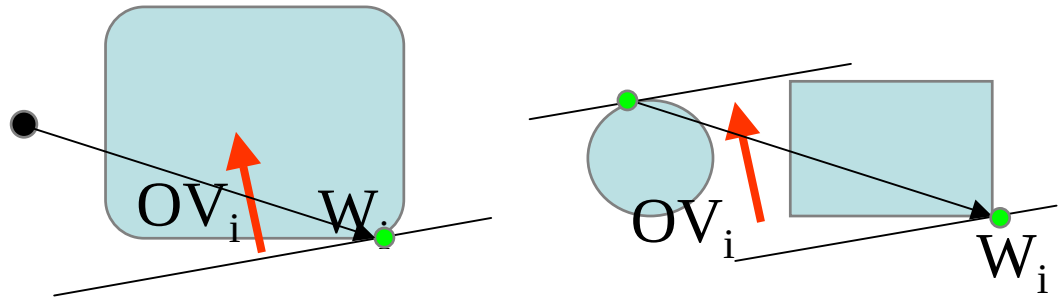


Minkowski Sum をとったもの

# GJK

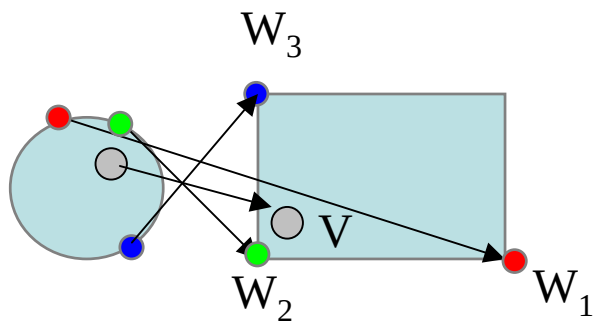
$V_0$ : **凸形状内の任意の点**

$W_i$ :$\overrightarrow{OV_i}$と$\overrightarrow{OW_i}$**の内積が最小の点**(support point)

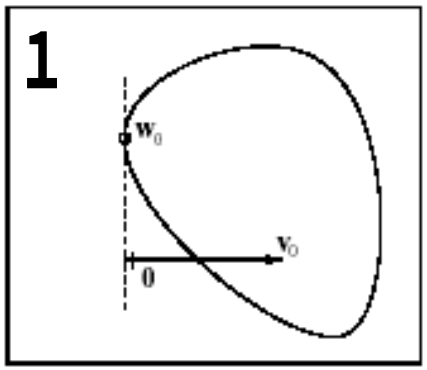


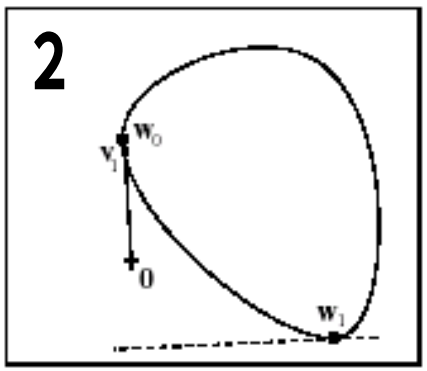

$V_i$ :**三角形**$W_{i-2}\ W_{i-1}\ W_i$**内の点で原点に一番近い点**

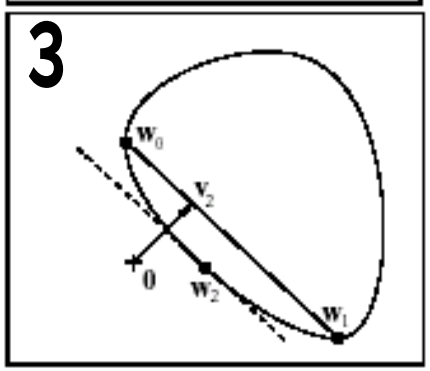
$V_i = s\ W_{i-2} + t\ W_{i-1} + (1-s-t)\ W_i$

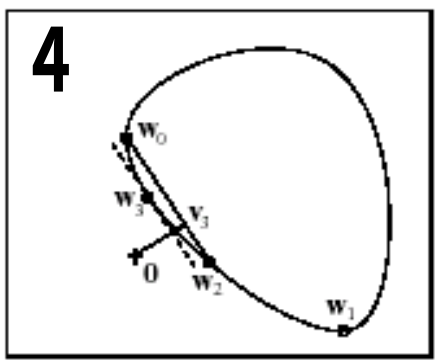



**元の図形上で、最近傍点が求まる**



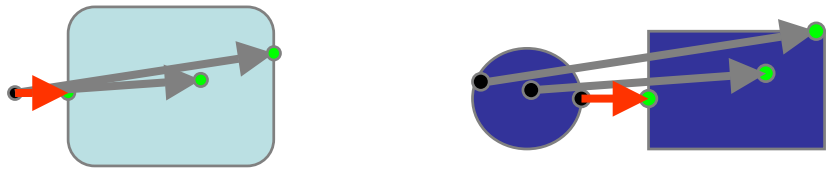



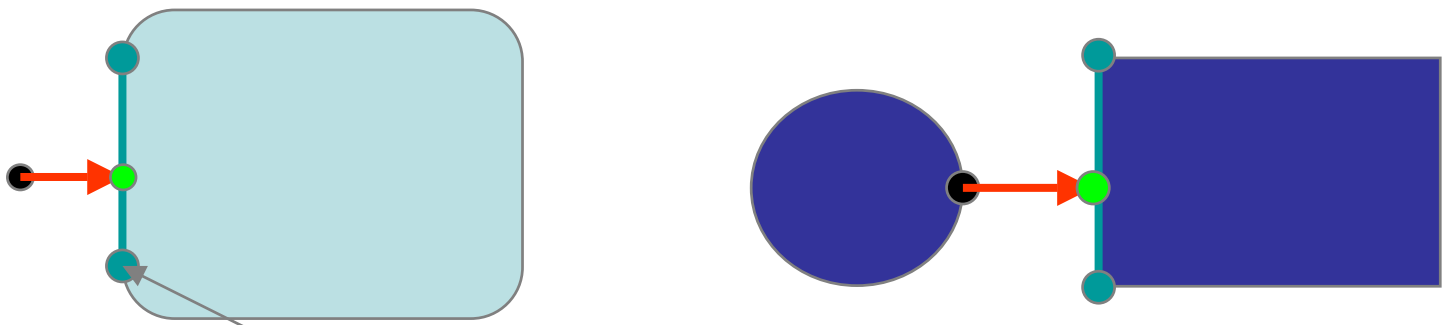



# GJKの結果の解釈

- 法線　　そのまま実世界の法線

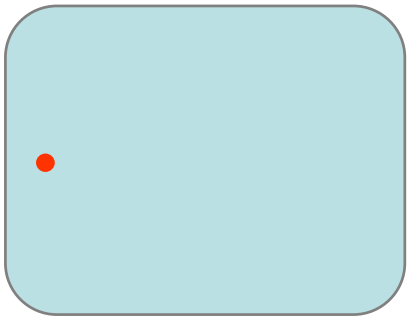


- 接触点　　Support point を内分した点
  →実世界の Support points を内分して求める.

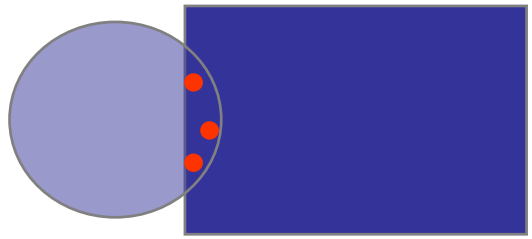


# GJKの結果の解釈2

- 重なっていると，戻せない



Minkowski Sum

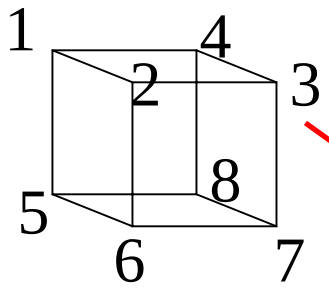


元の形状

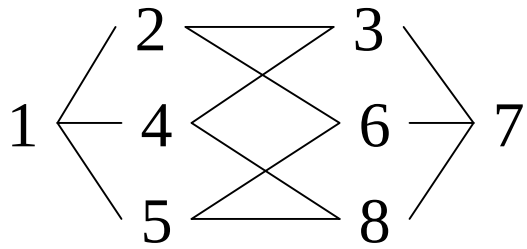
元の形状でのの3つの点は，変換した形状では1点
→どこだか分からない．

# GJKの高速化

- Support point の計算に時間がかかる




一番右下手前
の頂点は？



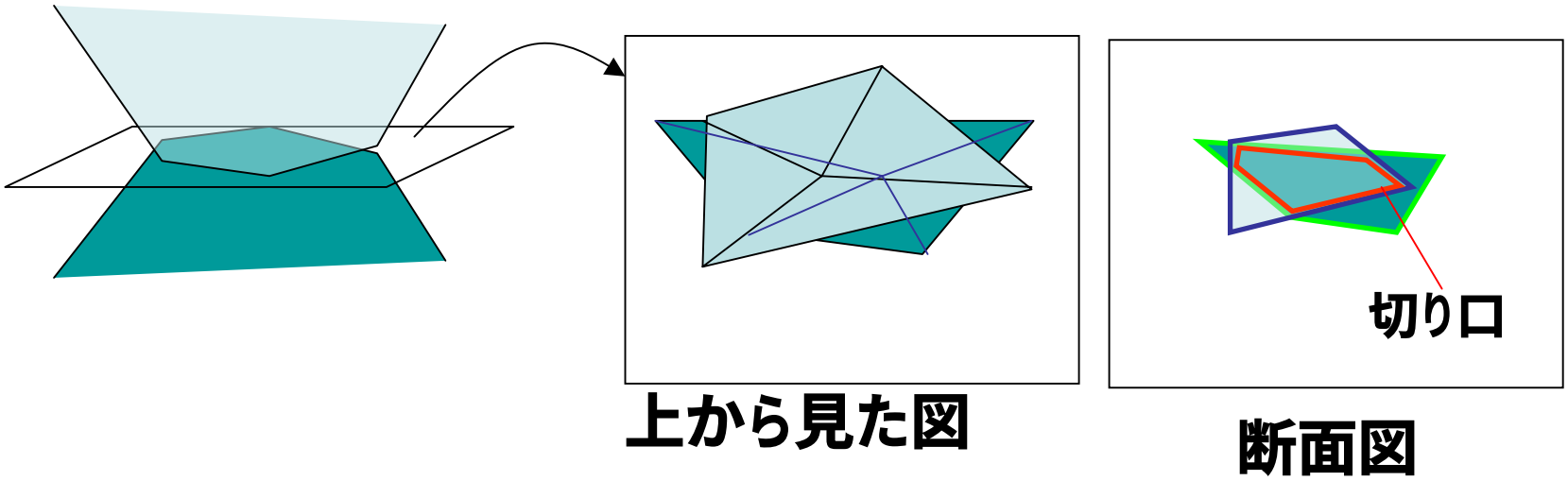

- 1-2-3-7と順にたどって探すしかないのか？

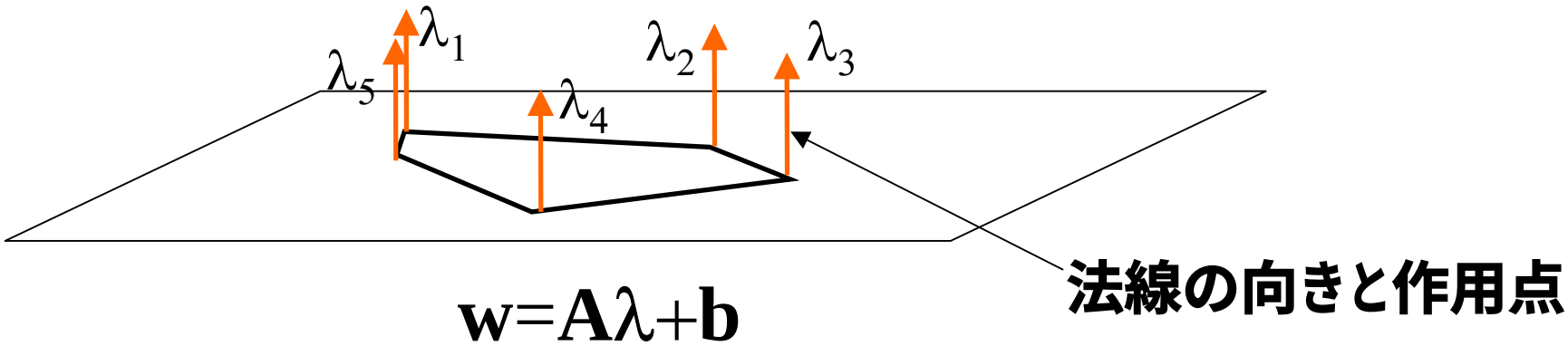
- 例えば，あらかじめベクトルの成分の＋－で頂点を分類

| | |
|---|---|
| －－－ | 5 |
| －－＋ | 1 |
| －＋－ | 6 |
| －＋＋ | 2 |
| ＋－－ | 8 |
| ＋－＋ | 4 |
| ＋＋－ | 7 |
| ＋＋＋ | 3 |

# 解析法に必要な接触情報

- 接触面の形状＝切り口

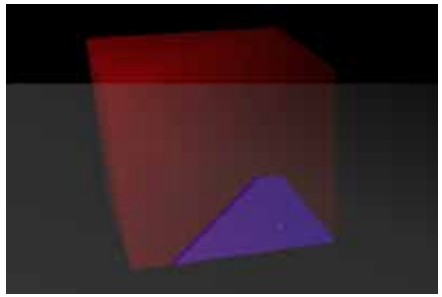


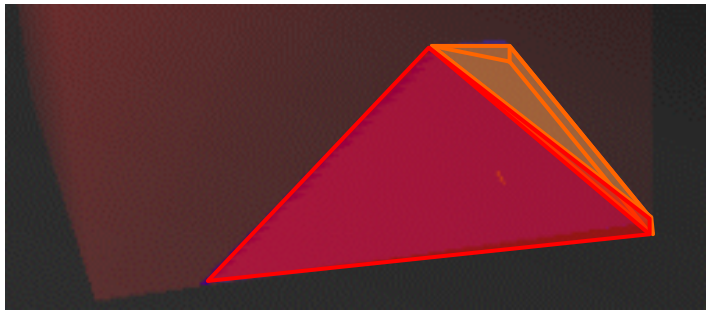



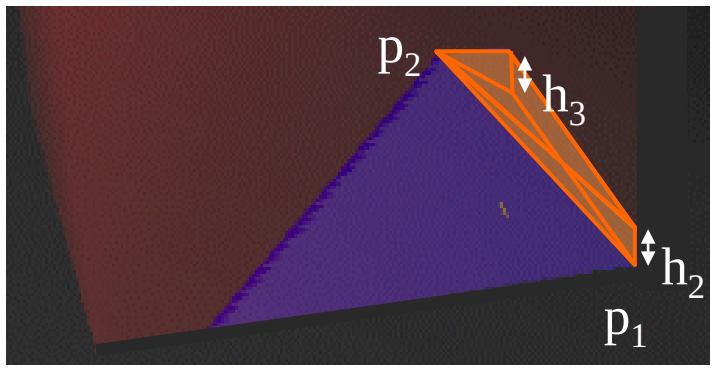

$$w = A\lambda + b$$

# ペナルティ法に必要な接触情報

- 交差部分の体積





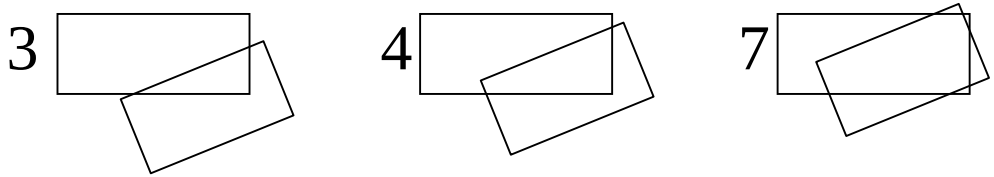



バネモデルからの力

$$\mathbf{f} = k \int_{\mathbf{p} \in tri} h_p \mathbf{n} dS$$

$$= k \frac{1}{3}(h_1 + h_2 + h_3)\ \mathbf{n}$$

- 解析法と同様，切り口の頂点でも動きますが，
  頂点の数が増えたり減ったりしたときに，跳ねます．

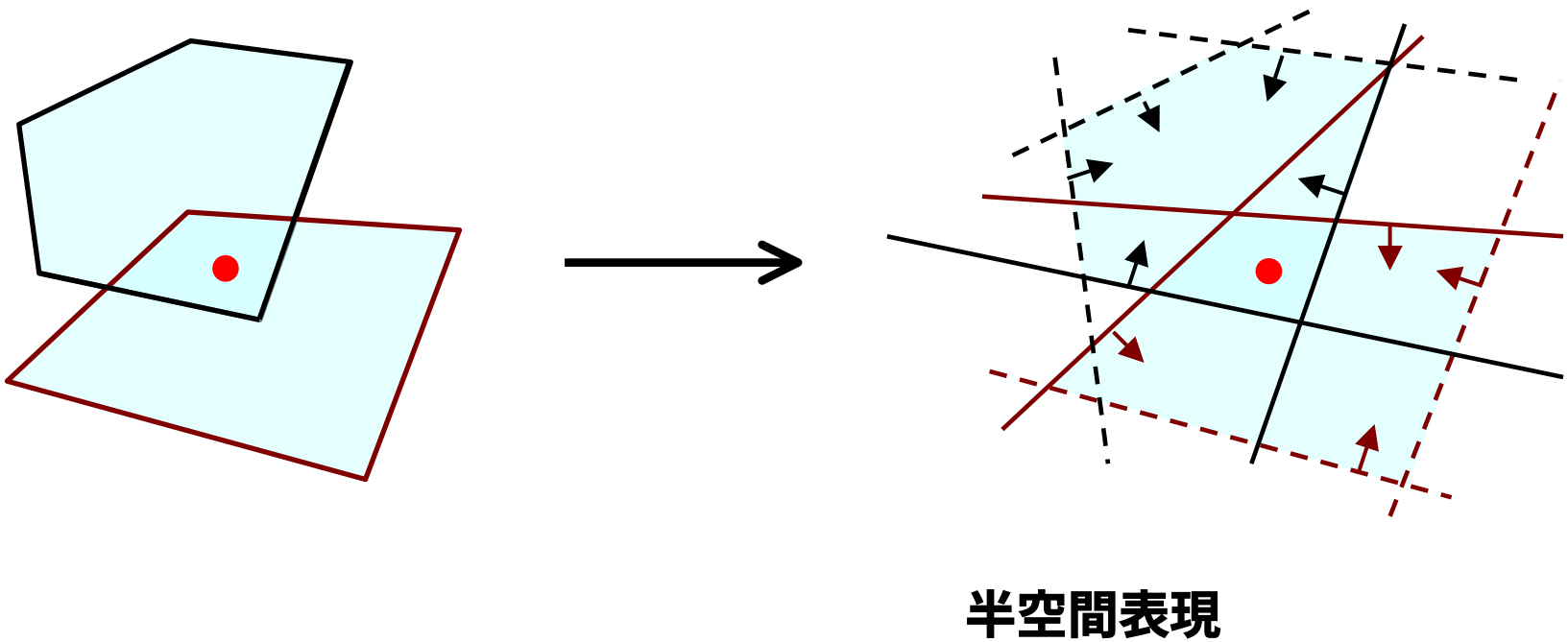


# 接触部分の形状の求め方

- 衝突部分の形状（切り口）を求める方法
  - 切り口の頂点に接触の拘束を考える
- D. E. Muller and F.P.Preparata:
  “Finding the intersection of two convex” (1978)
  - 凸形状2つの交差部分の形を求める
    - 共有点1点が必要（GJKで求められる）
  - 2次元：切り口の頂点と辺が分かる
  - 3次元：交差部分の形状の，頂点と面が分かる

# 接触部分の形状の求め方

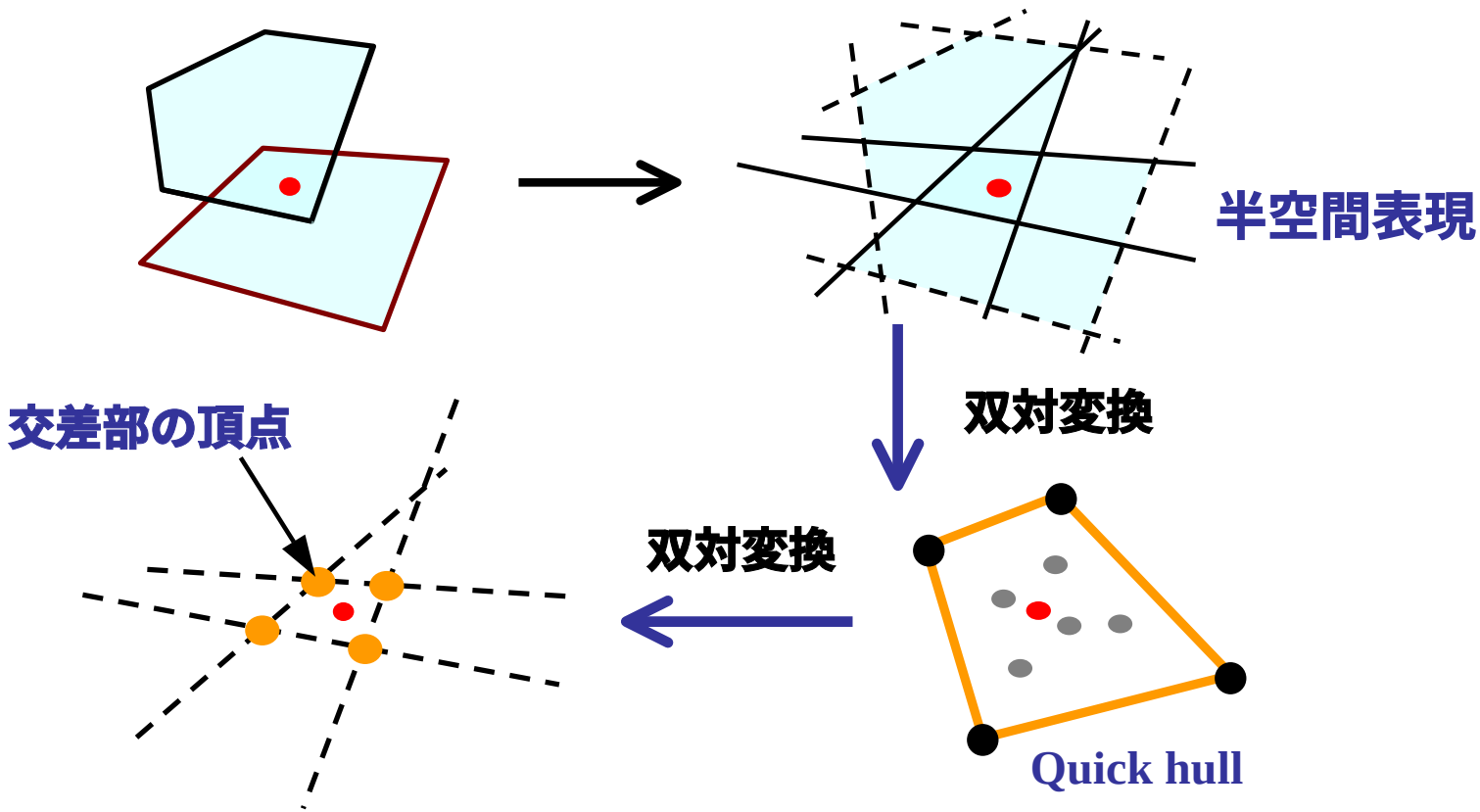
- 2つの凸形状の交差部



半空間表現

# 接触部分の形状の求め方

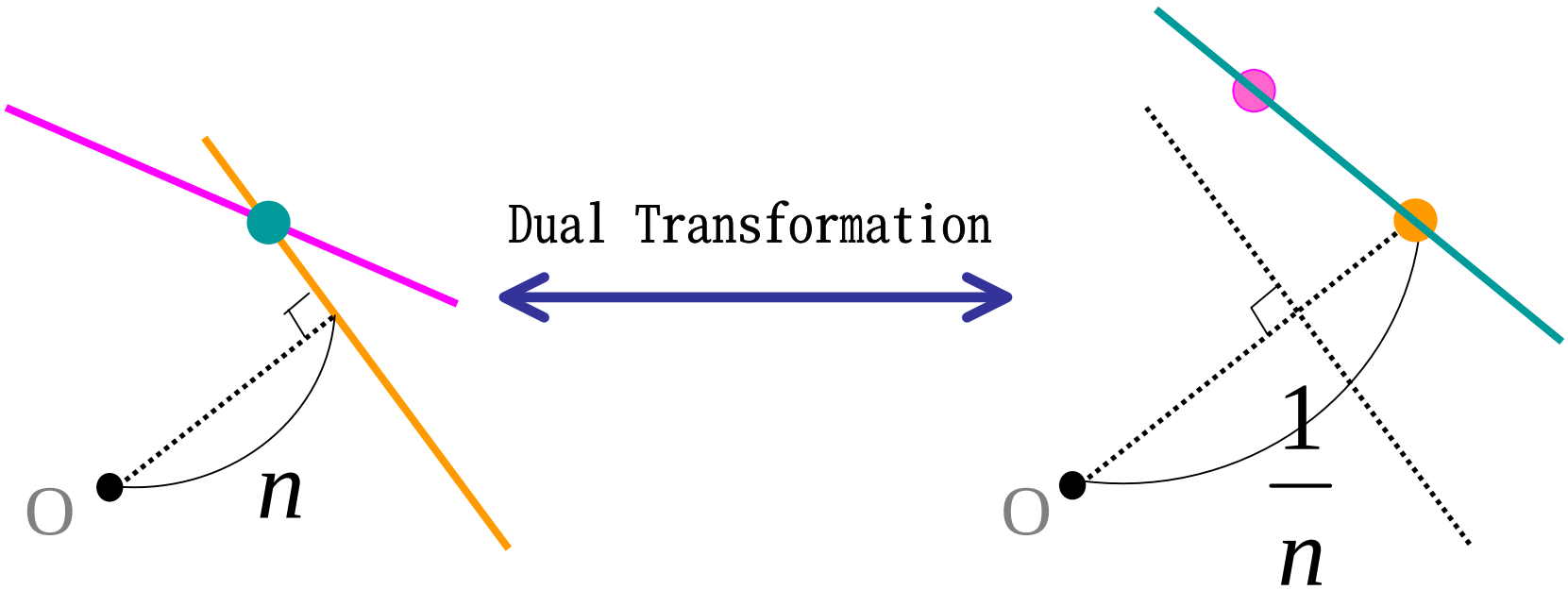
- 2つの凸形状の交差部(2)




n log n

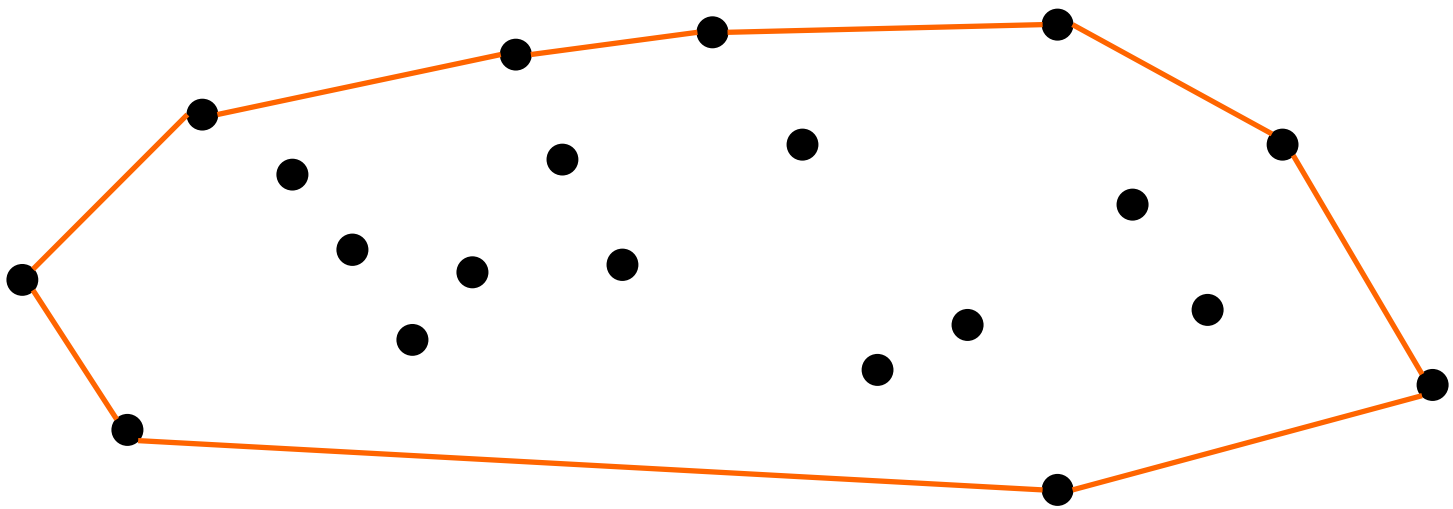
# 接触部分の形状の求め方

- (幾何学の) 双対変換
  - 双対変換は，面を点に，点を面に変換する．
  - 双対変換の双対変換は元に戻る

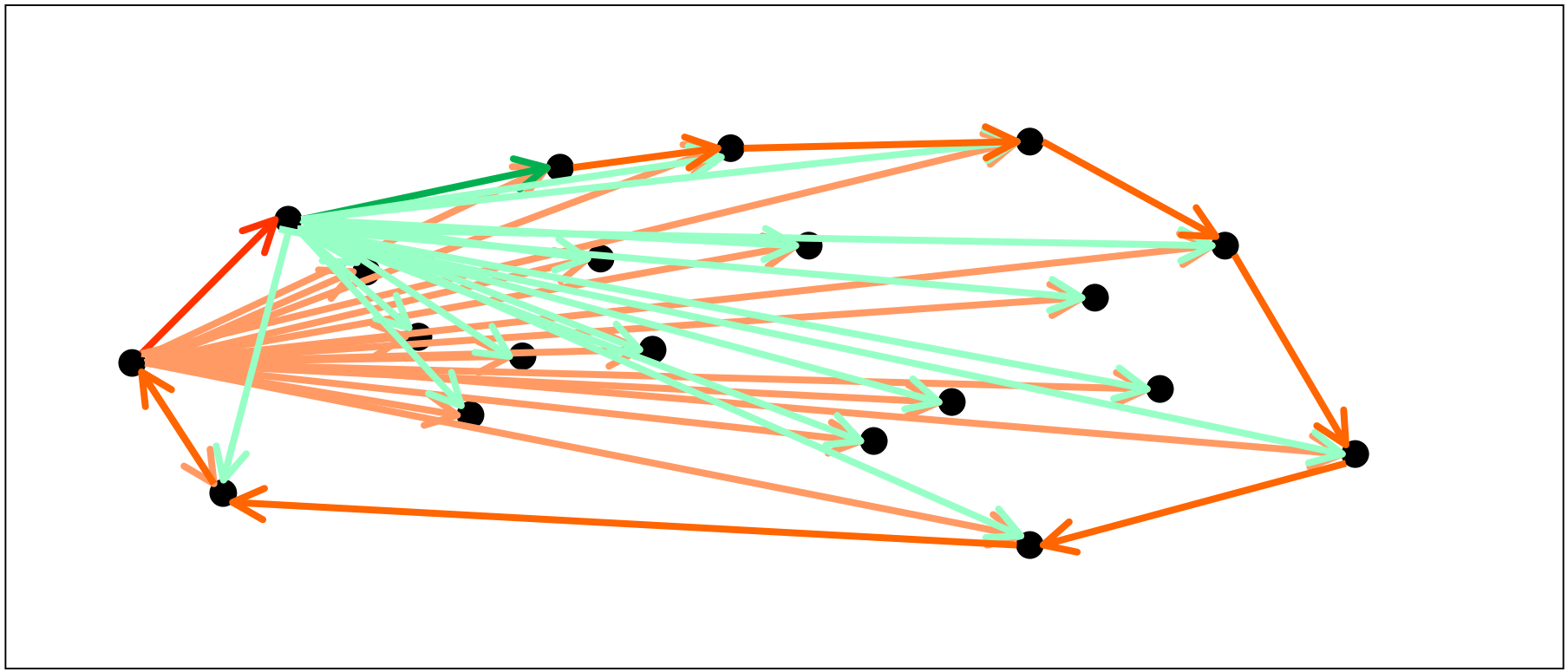


# Convex hull

- 点が与えられたときに，それらを含む，最も小さな凸形状を求める問題



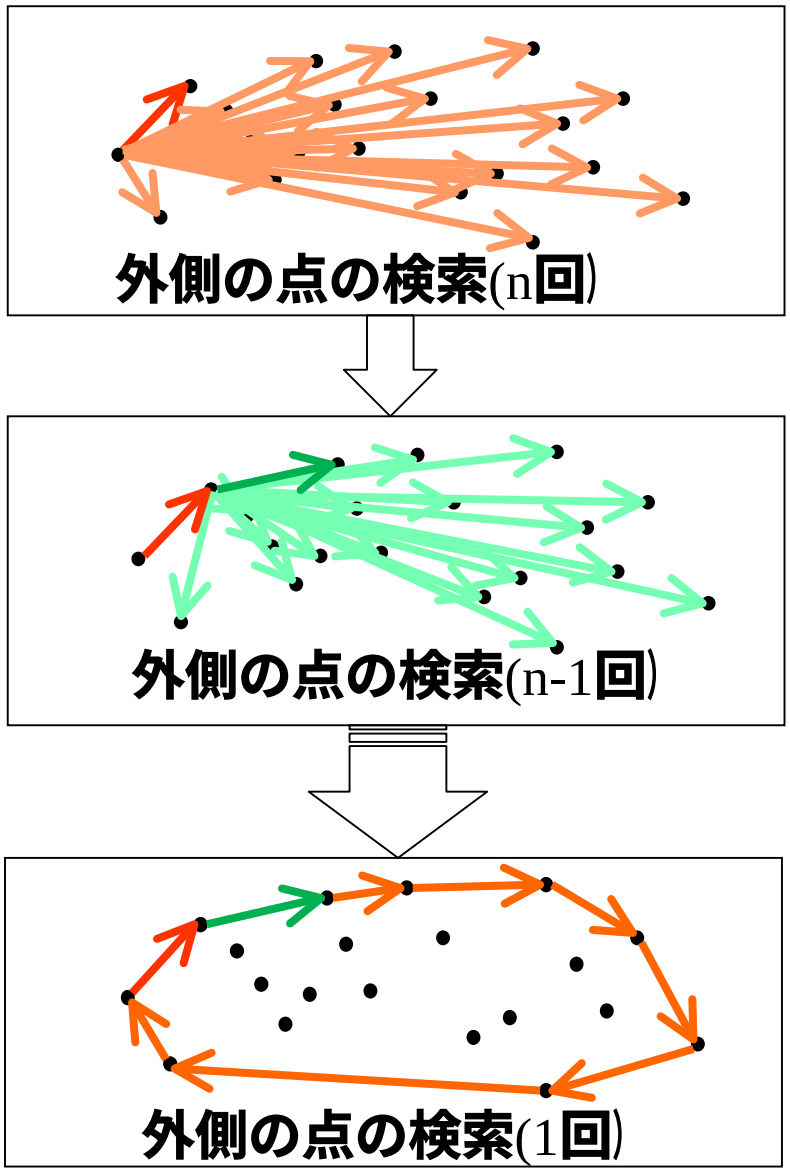
# 総当り



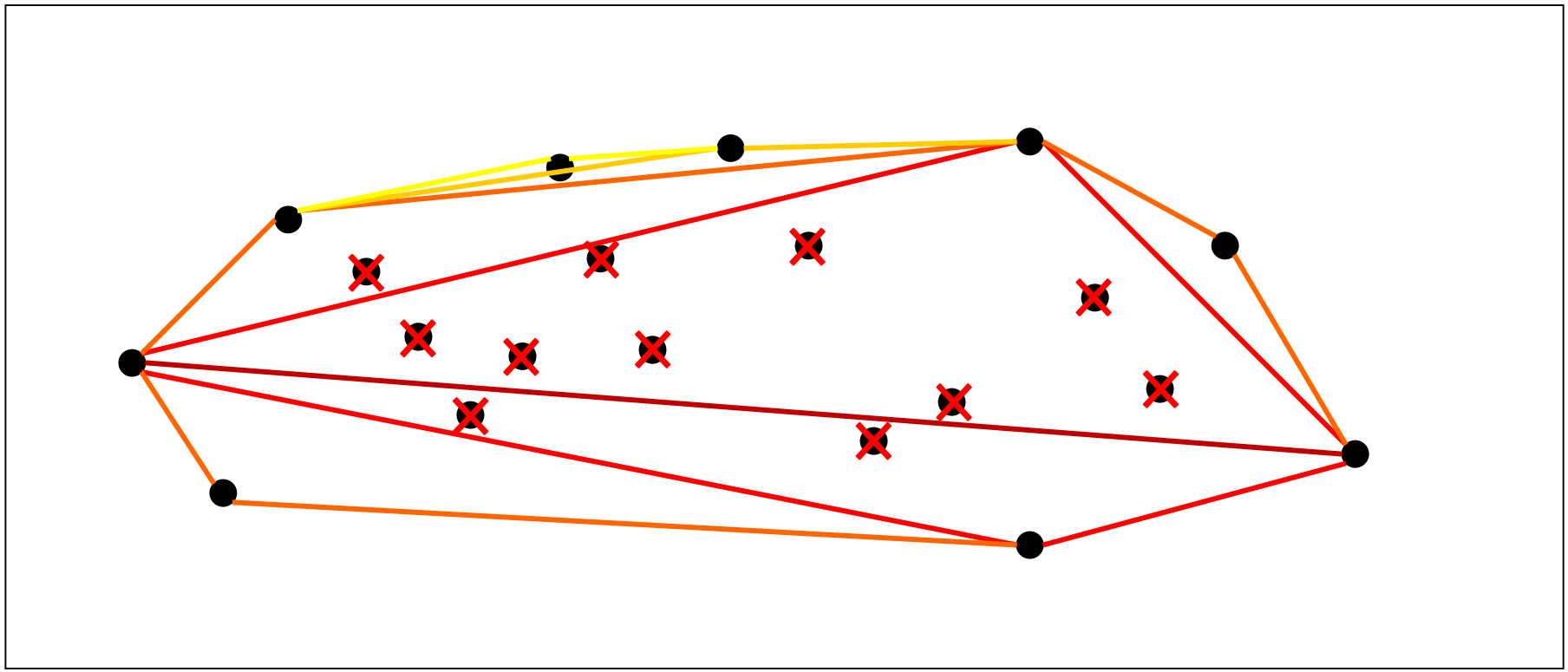
- ある点から見て一番角度の大きい点を選ぶ　n回比較
- すべての点において繰り返す　n回

全体で$O(n^2)$

# Selection sort と同じ $O(n^2)$

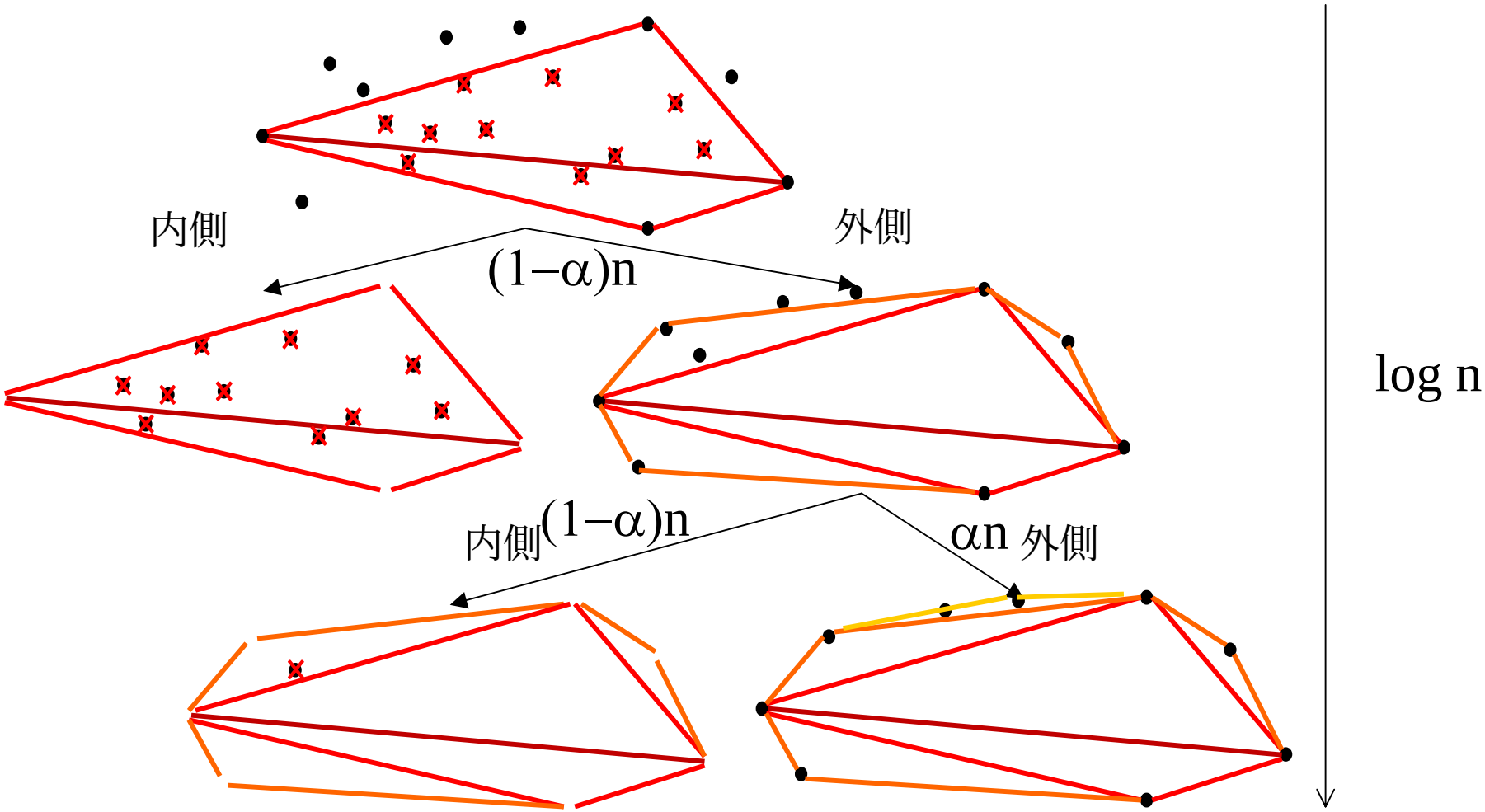


# Quickhullアルゴリズム



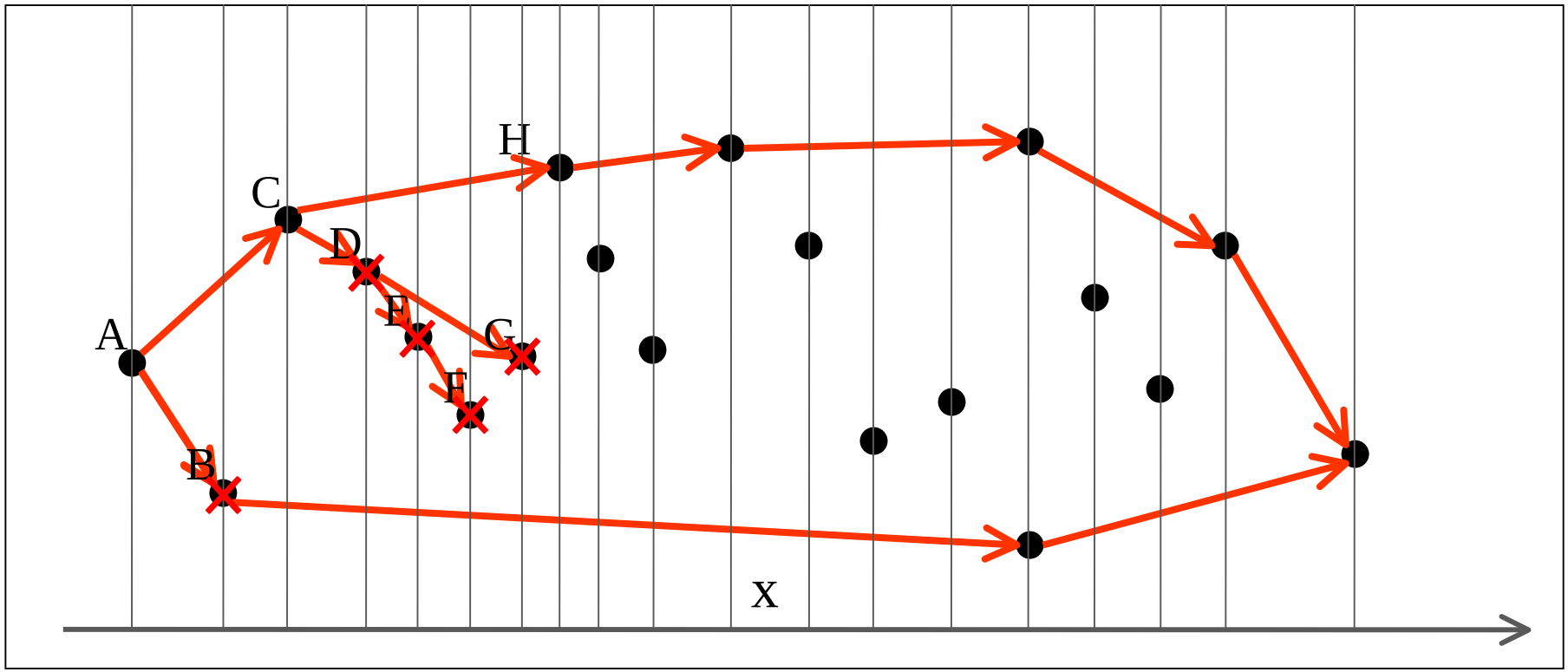
1. 端点を結ぶ — $n$回比較
2. 各辺に対して最も外側にある点を追加 — $n/2$回比較
3. 辺を分割して，内側の点を削除 — $n_2=\alpha n/2$
4. 各辺に対して最も外側にある点を追加 — $n_2$回比較
5. 点がなくなるまで3から繰り返す — $\log n$ 回

# Quick sort と似ている




O(n log n)

# Andrewのアルゴリズム

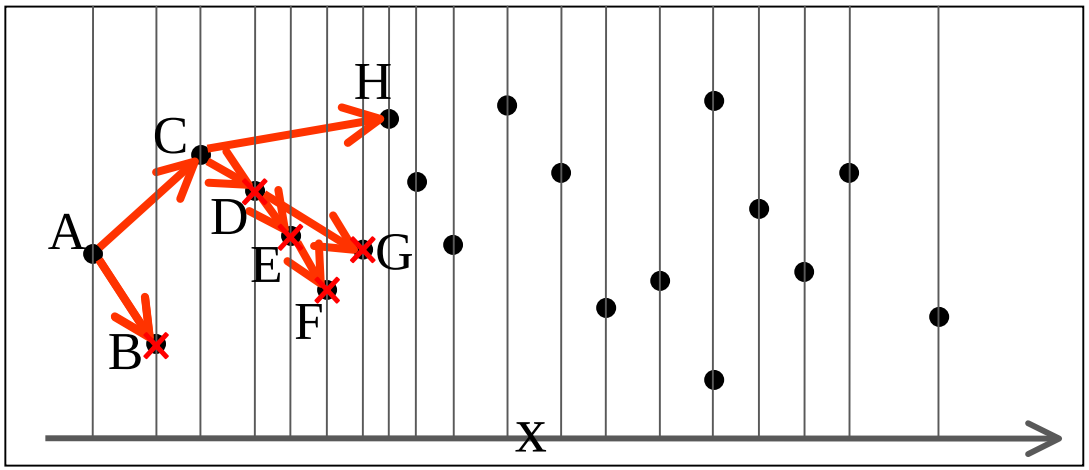


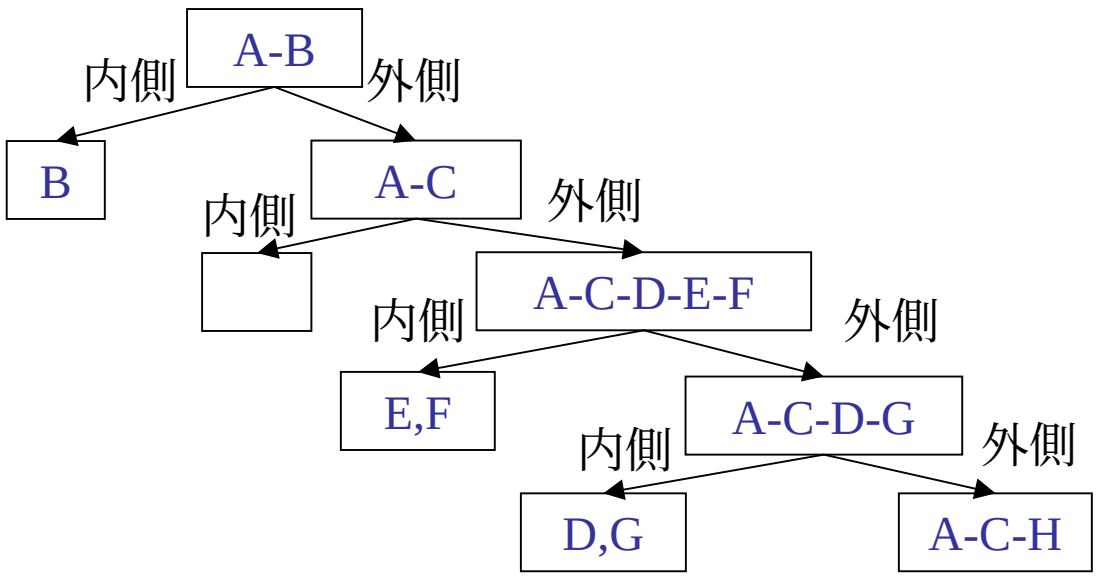

1. 左から順序付け
2. 最も左側のA-Bを最初のチェーンにする
3. 暫定的なチェーンよりも左側の点を追加

} O(n)

4. 右端の点まで繰り返す　O(log n)

# Andrewのアルゴリズム



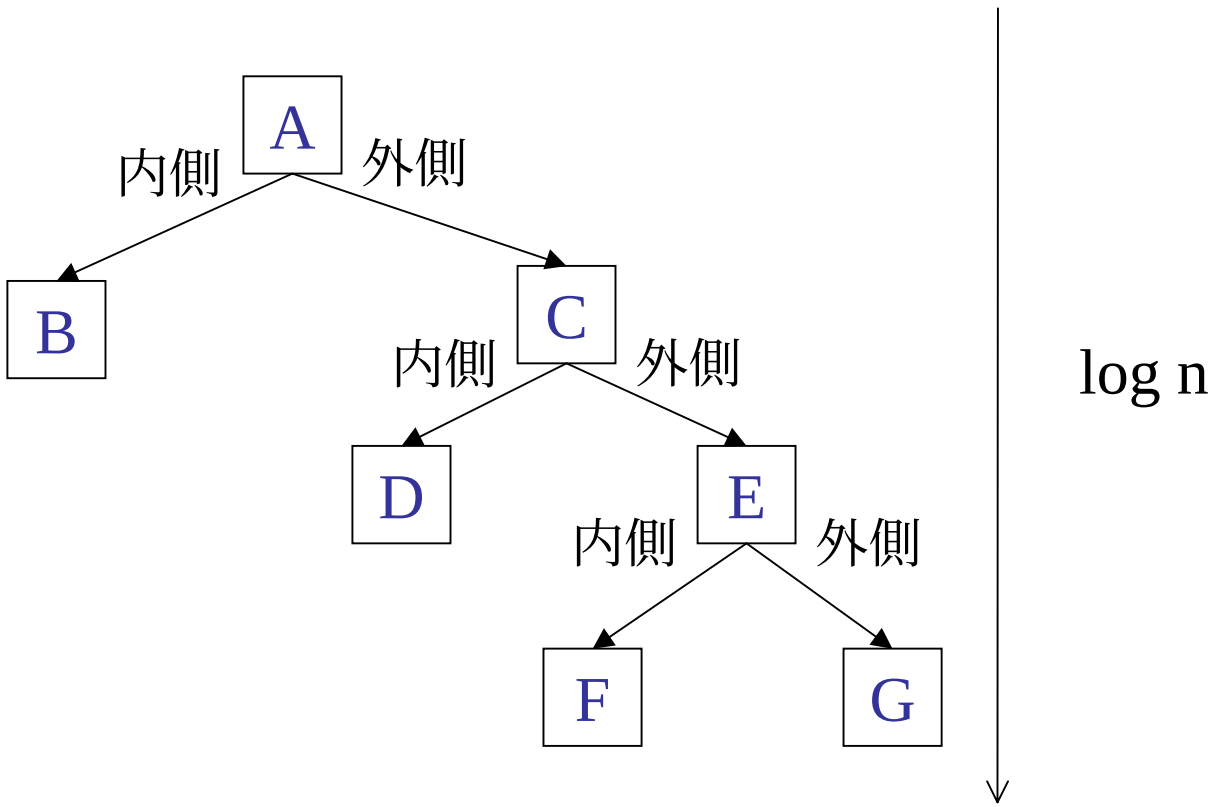




log n 回

O(n log n)

# 凸包の計算

アルゴリズムなし　　: $O(n^2)$
Andrewのアルゴリズム　: $O(n \log n)$
Quickhullアルゴリズム　: $O(n \log n)$



# 終わりに

- ソート＆検索はアルゴリズムの基本
  - log n, n log n のアルゴリズムを探そう
    - 毎回何割かの問題が片付く
      → log n 回で終わる　　複利
    - 毎回 c個の問題が片付く
      → n/c 回かかる　　単利
- 3次元空間には ＜演算がないことも多い
  - 1次元にして，＜演算を使う
    - Sweep & prune
    - k-d tree
  - 何とか，毎回何割か片付ける．
    - BSP
    - Quick hull

# 本の紹介

- F.P.プレパラータ他：　計算幾何学入門
  - Convex hull とか 双対変換とか
- 杉原厚吉：　　計算幾何工学
  - 数値計算誤差によって起こる大災害を防ぐ
- 平岡和幸他：　プログラミングのための線形代数
  - 物理シミュレーションの線形代数が難しいと感じたら
- 結城浩：　　数学ガール
  - 数式の情緒を教えてくれる本

# Q&A

**質問/コメント/感想 をお願いします**

**講演者と他の受講者の理解の**
**助けになります.**