

<https://bit.ly/2HgJzwa>

KYOTO UNIVERSITY

情報科学基礎論 ～アルゴリズムとデータ構造～

集合知システム分野
鹿島久嗣

DEPARTMENT OF INTELLIGENCE SCIENCE
AND TECHNOLOGY

参考書:

■ 基本 :

- 杉原厚吉「データ構造とアルゴリズム」
(共立出版)



■ より高度な内容 :

- Cormen, Leiserson, Rivest, Stein 「Introduction to Algorithms (アルゴリズムイントロダクション)」

アルゴリズムとデータ構造とは

動機:

「良い」プログラムを書くためには？

- プログラムの良し悪し
 - ー 正しく動く： 想定したように動く
 - ー 速く動く： プログラムは速いほど良い！
 - ー 省資源： メモリや電気代
 - ー 例： お店の顧客管理
- 特定のプログラム言語やハードウェアとはなるべく独立に：
 - ー プログラムの良し悪しを測りたい
 - ー ひいては良いプログラムを作りたい

アルゴリズム:

与えられた問題を解くための有限の手続き

■ アルゴリズム (algorithm) とは

– プログラム言語やハードウェア (CPU、メモリ) とは別に、どのような手続きを表現しようとするかという「問題の解き方」

– もうすこし厳密にいうと、「与えられた問題を解くための機械的操作からなる、有限の手続き」

- 機械的操作：四則演算やジャンプなど

- かならず有限ステップで終わるべし



- 手続き (procedure)：有限ステップでの終了が保証されない

データ構造:

データを管理し、アルゴリズムを効率化する

- 多くのプログラムは「データ」を扱う
 - データは繰り返し使用するもの
 - 使用の仕方が予め決められているわけではない
- アルゴリズムがうまく動くためには、データをどのようにもっておくか（＝データ構造）が重要
 - 名前を、入力順に格納？ アイウエオ順？
- データ構造はアルゴリズムと切り離せないもの
 - お互いの良さに影響を与え合う

アルゴリズムの例: 指数演算のアルゴリズム

■ 問題

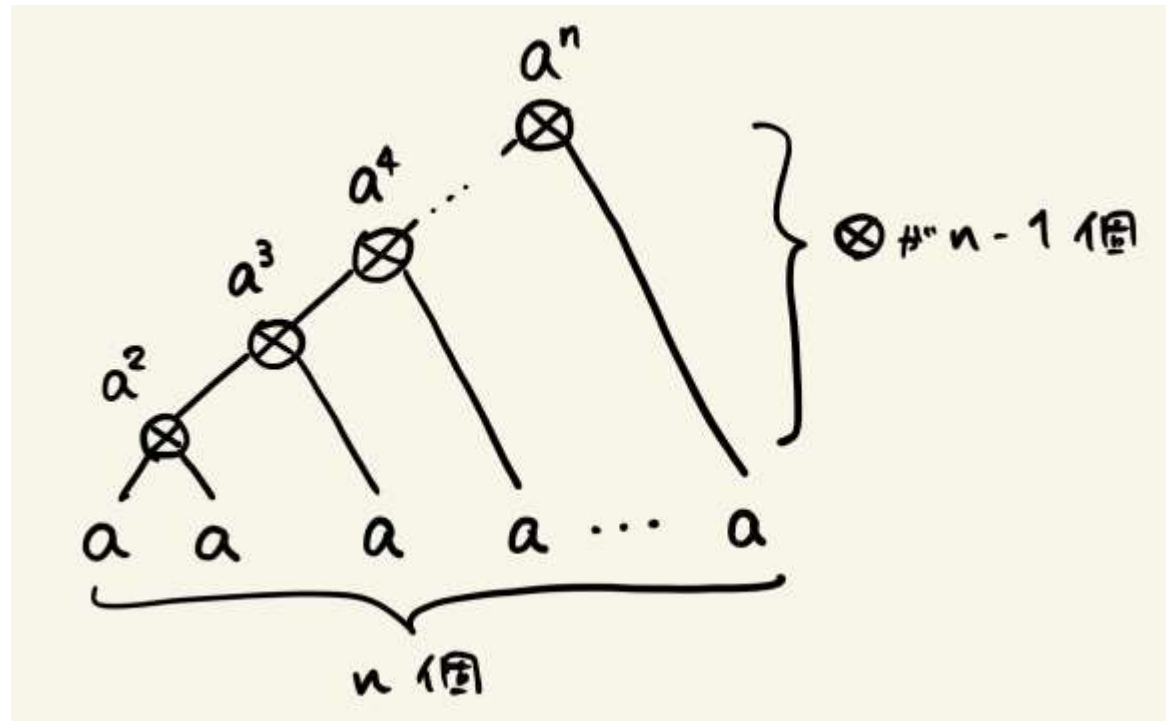
- 入力 : 2つの正整数 a と n
- 出力 : a^n
- 仮定 : 許されるのは四則演算のみとする
(いきなり n 乗するのはダメ)

■ 四則演算が何回必要か？

指数演算のアルゴリズム①:

単純な掛け算の繰り返したと線形時間が必要

- $a^n = ((\dots ((a \times a) \times a) \times \dots) \times a)$ で計算
- $n - 1$ 回の掛け算でできる

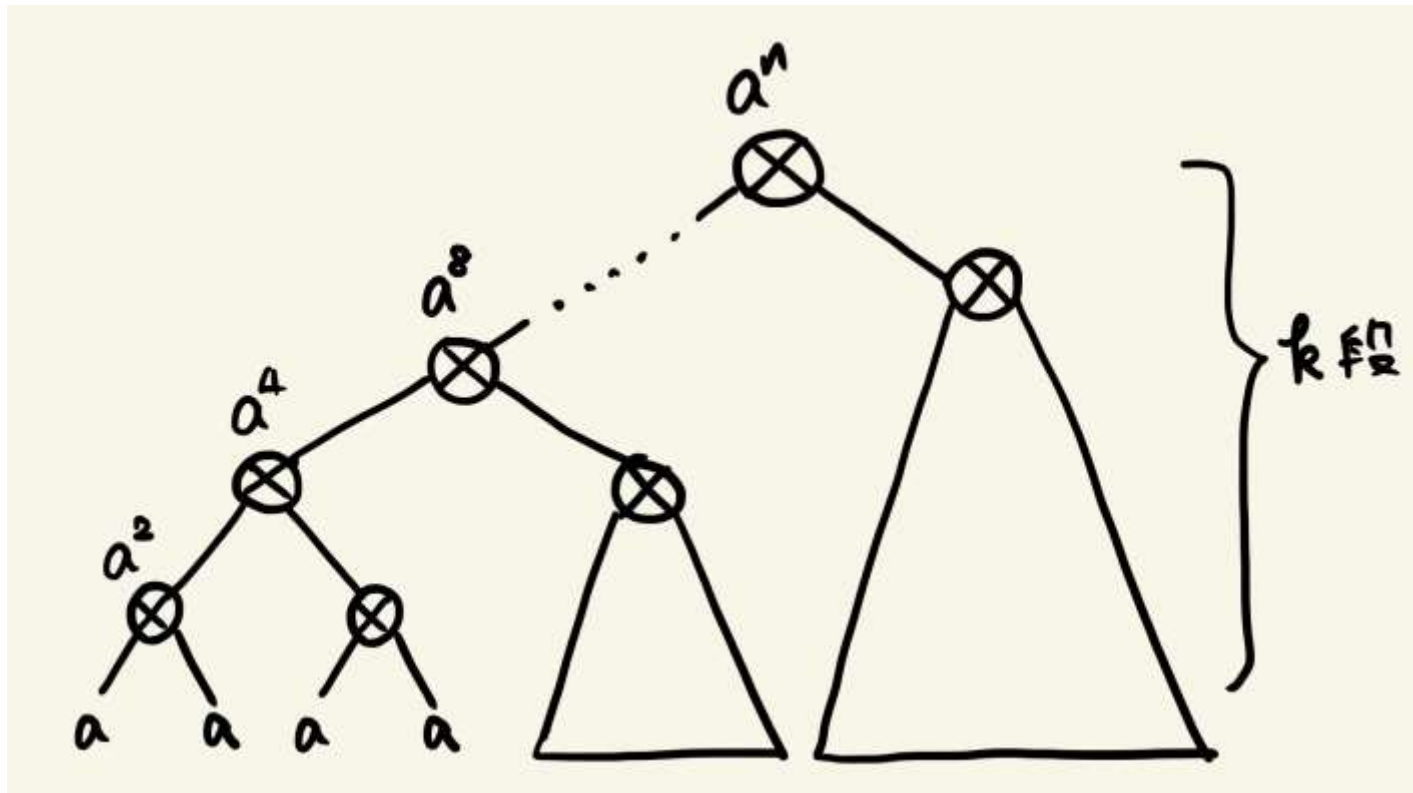


指数演算のアルゴリズム②:

ちょっと工夫すると対数時間で計算可能

- $k = \log_2 n$ 回の掛け算でできる

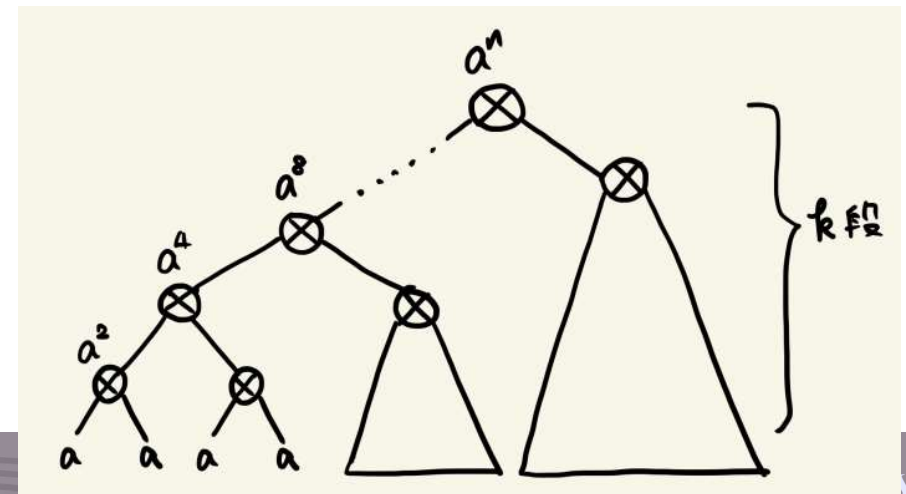
—仮定: $n = 2^k$



指数演算のアルゴリズム②:

ちょっと工夫すると対数時間で計算可能

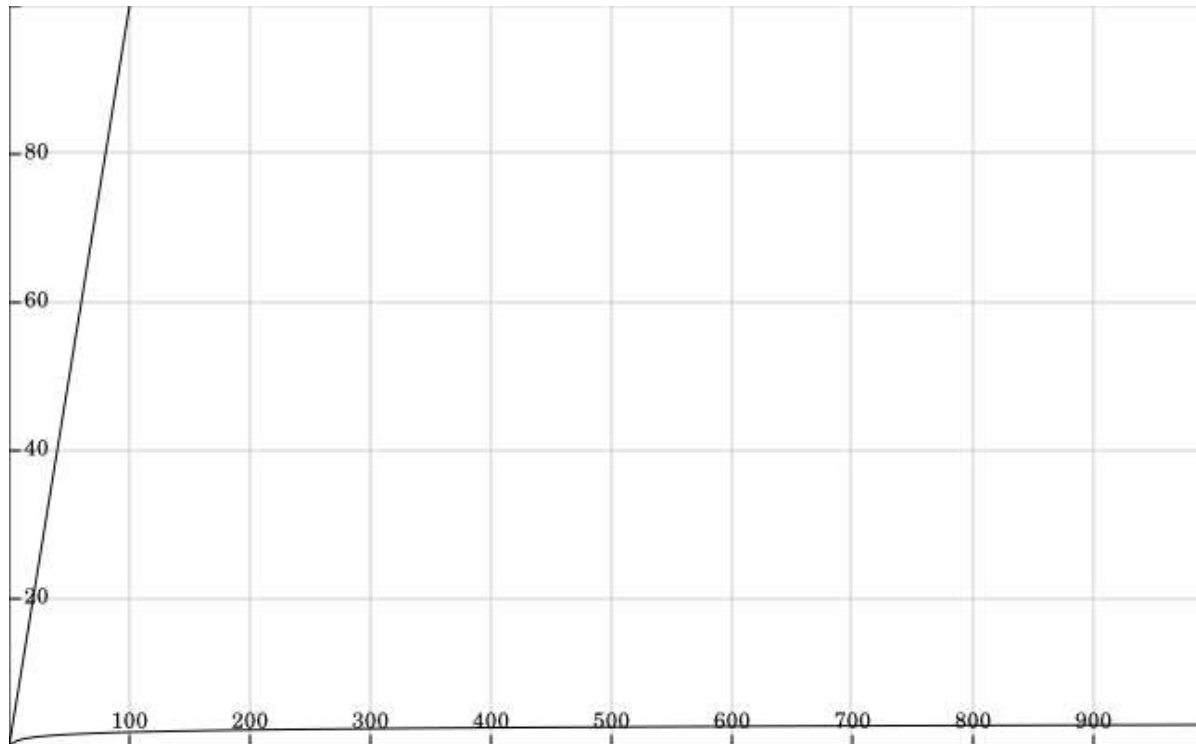
- なお、 $n \neq 2^k$ の場合も $3\log_2 n$ 回の演算で可能
 - n を2進表現する ($\log_2 n$ 回の割り算)
 - 例: $n = 22 = 10110$
 - 1が立っている桁数に対し2の冪を求める ($\log_2 n$ 回の掛け算)
 - すべて足す ($\log_2 n$ 回の足し算)



アルゴリズムの重要性:

アルゴリズムの工夫で計算効率に大きな差が生じる

- $n = 1024 = 2^{10}$ のとき、掛け算の回数は
 - ① 1023回 (大体 n 回)
 - ② 10回 (大体 $\log n$ 回)



データ構造の例:

データに対して繰り返し操作を行う場合に有効

- 前のアルゴリズムの例では1回の計算のみを対象としていた
- データに対して繰り返し計算を行う場合には、予めデータを処理しやすい構造（＝データ構造）を作ることによって、その後の計算を高速に行えるようになる（ことがある）
- 例えば、 S 回計算するとして
 - ① $(1\text{回分の計算時間}) \times S \text{ 回}$
 - ② $(\text{データ構造の構築にかかる計算時間}) + (\text{データ構造を利用した1回分の計算時間}) \times S$で① $>$ ② となる場合にはデータ構造を考えることが有効

具体的な問題例:

店舗における顧客情報管理システム

- n 人の顧客情報 $\{(n_i, p_i)\}_{i=1, \dots, n}$ が載った名簿を考える
 - n_i : 名前、 p_i : 情報
 - 例 : (元田中 将大, mmototanaka@kyoto-u.ac.jp)
- 客が来るたびに名前を聞いて入力すると、その人の情報が得られるシステムを考える
 - s 人分の問い合わせ $n_{k_1}, n_{k_2}, \dots, n_{k_s}$ が順に与えられる
 - それぞれに対して p_{k_j} を返す

単純なアルゴリズム：

並び順がでたらめな場合は最悪で約 nS 回のチェックが必要

- 名簿の並び順が登録順（でたらめ）の場合を考える
- アルゴリズムとしては、前から順に探していく
- この場合、各問い合わせで、最悪 n 回のチェックが必要
 - 名簿上の位置（ページ）を指定してチェックすることは単位時間でできるものとする
- 合計 約 nS 回のチェックが（最悪ケースで）必要となる

ちょっと頭を使ったアルゴリズム：

辞書順に並んでいる場合は $S \log n$ 回のチェックで可能

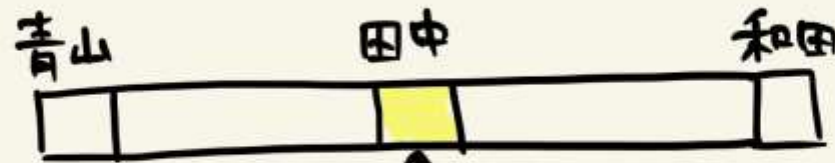
- 予め名簿を辞書順に並べておくとする
- 問い合わせ名を名簿の「ほぼ真ん中」の人の名前と比較
 - 前者が辞書順で前ならば、目的の人は名簿の前半分にいるはず
 - 今後は前半分だけを調べればよい
 - こんどは前半分の「ほぼ真ん中」の人と比較
 - ...
 - 計 $S \log n$ 回のチェックで可能

ちょっと頭を使ったアルゴリズム：

辞書順に並んでいる場合は $S \log n$ 回のチェックで可能

- 計 $S \log n$ 回のチェックで発見可能

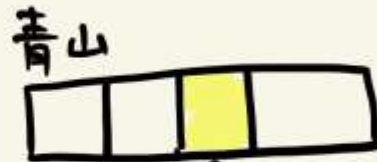
問合せ：木村



ちょうど真ん中

木村 < 田中 なので、田中より左に絞る

↓ 探索範囲が 半分 になる



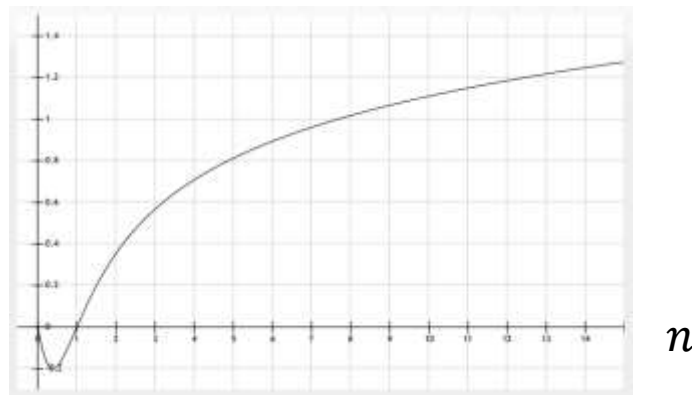
真ん中 (加藤) と木村で比較

データ構造の重要性:

データを正しく持つことで計算コストが大きく削減される

- 「データをどのように管理するか」= データ構造
- 賢いほうのアルゴリズムの恩恵にあずかるにはデータが予め整列（ソート）されている必要がある
 - 一般に整列は $n \log n$ 回 に比例する演算回数が必要
- よって ① $n S$ と ② $n \log n + S \log n$ の比較
 - S が大きくなると②の方がお得になってくる

$$\frac{n \log n}{n - \log n}$$



アルゴリズムの評価基準

計算機の理論モデル:

計算の効率性を評価するための抽象的な計算機

- 我々は特定の言語やハードウェアに依存しない議論がしたい
- 計算のステップ数を数えるためには計算機のモデルを決めておく必要がある
- 計算機の理論モデル
 - (1930s) Turing機械、 λ 計算、Postシステム、帰納的関数、ランダムアクセス機械 (RAM) など様々なモデルが提案される
 - これらによって計算モデルと計算可能性の概念が議論
 - 上記のモデルはすべて同等 → 安定な概念

ランダムアクセス機械:

現在のコンピュータに近い、計算機の理論モデル

■ランダムアクセス機械：

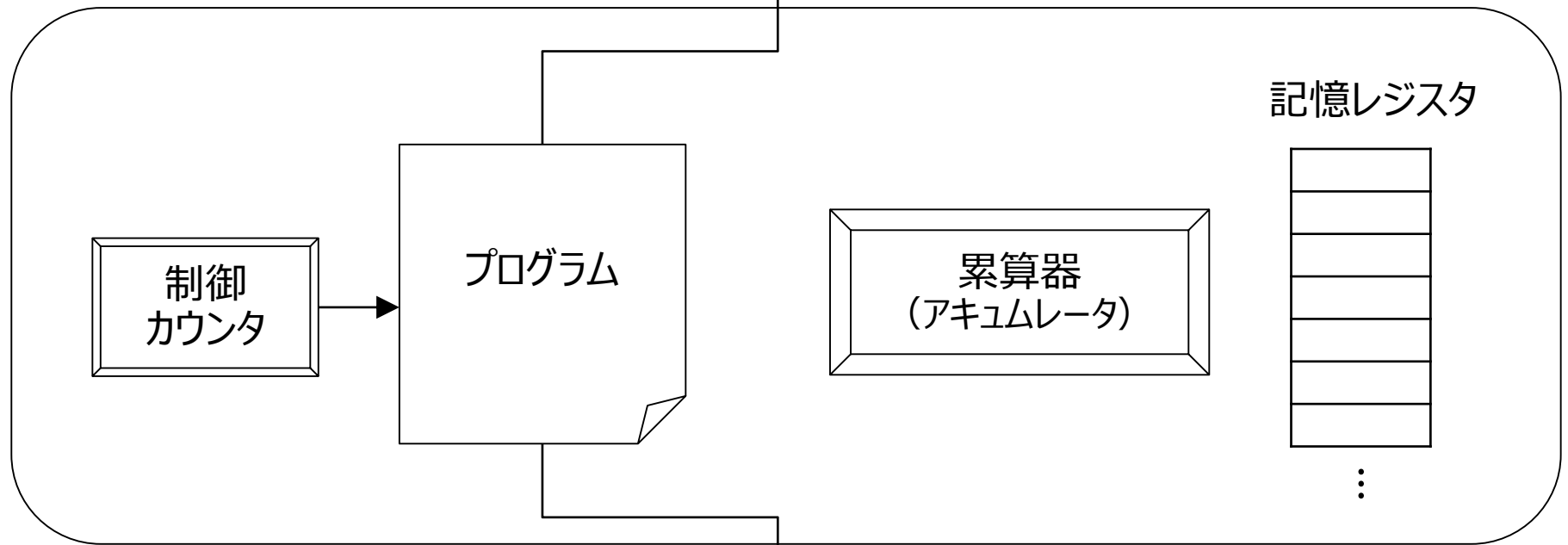
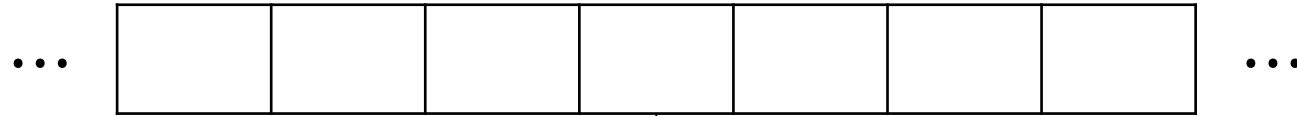
- 入力テープ、出力テープ、制御カウンタ、プログラム、累算器（アキュムレータ）、記憶レジスタからなる
- 入・出力テープは無限の長さがあると仮定
- 累算器、レジスタには任意の桁数のデータが入り、単位時間でアクセス可能と仮定

■命令（単位時間で実行できると仮定）

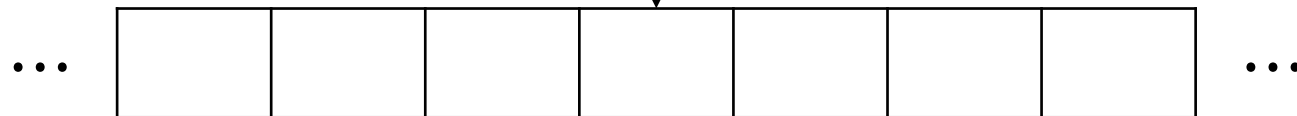
- 読み取り／書き込み（入力・出力テープを1つ進める）
- 四則演算、ジャンプ、条件分岐、停止

ランダムアクセス機械: 現在のコンピュータに近い、計算機の理論モデル

入力テープ



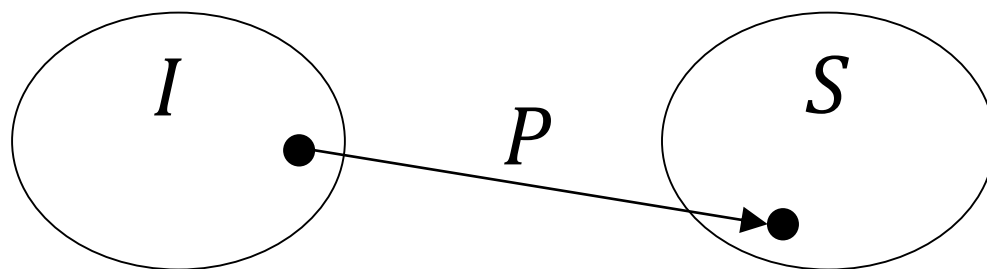
出力テープ



「問題」の定義:

入力（問題）と出力（解）の対応関係

- 問題：問題例集合 I と解集合 S の対応関係 $P: I \rightarrow S$ が決まっているもの



—例：素数判定問題

- 問題例：正整数 n （ I は正整数全体）
- 解： n が素数なら Yes, そうでないなら No
（ $S = \{\text{Yes}, \text{No}\}$ ）
- n が決まれば Yes か No かは決まっている（決定問題）
⇔ 探索問題（例：代数方程式）

「アルゴリズム」の定義:

「問題」を解くための有限の手続き

- 計算モデルと問題に対して定義される

- 定義：アルゴリズム

- 問題Aに対する計算モデルCでのアルゴリズムとは、有限長の（Cで許された）機械的命令の系列であり、Aのどんな入力に対しても有限ステップで正しい出力をするもの

- 問題Aの一部の問題例が解けるとかではダメ

計算可能性:

世の中にはアルゴリズムが存在しない問題が存在する

- 問題Aが計算モデルCで計算可能：
Aに対するCのアルゴリズムが存在すること
- 計算可能な問題があるなら、計算不可能な問題もある
- 計算不可能な問題の一例：プログラムの停止問題
 - 入力：プログラム P、データ x
 - 出力：計算(P, x)が有限ステップで終了するならYes
そうでなければNo
- データxはプログラムでもよいことを考えると、この計算不可能性はプログラムの自動的なデバッグは難しいことを示唆している(?)

計算可能性:

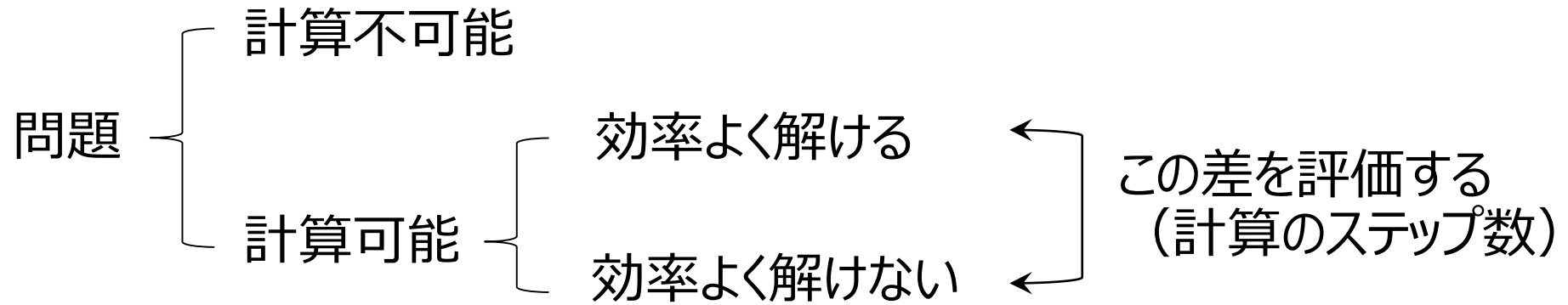
「プログラムの停止問題」は計算不可能な問題

- 計算不可能な問題の一例：プログラムの停止問題
 - 入力：プログラム P 、データ x
 - 出力：計算 (P, x) が有限ステップで終了するならYes
そうでなければNo
- 証明の概略：この問題を解けるプログラム Q が存在するとして矛盾を示す
 1. Q を使って新しいプログラム Q' をつくる
 - Q' はプログラム P を入力として、 (P, P) が停止するなら停止しない、停止しないなら停止するようなプログラム
 2. Q' に Q' を適用して矛盾を示す（ (Q', Q') を考える）

アルゴリズムの性能評価:

通常は最悪な入力に対するアルゴリズムの計算量を考える

■ 計算可能な問題に対するアルゴリズムの評価



■ 計算量 (computational complexity)

— 時間量 (time complexity) : 計算時間の評価

— 空間量 (space complexity) : 使用メモリ量の評価

■ 最悪計算量 : サイズ n の全ての問題の入力の中で最悪のものに対する計算量 (\Leftrightarrow 平均計算量)

最悪計算量と平均計算量： 探索問題の場合の例

- （前述の）名簿から名前を見つける探索問題
 - 入力： $n + 1$ 個の正整数 a_1, a_2, \dots, a_n と k
 - 出力： $a_i = k$ となる i が存在すれば i ；なければ No
- 前から順に探すアルゴリズムを考える
 - 最悪ケースでは n 回の比較が必要
 - 平均ケースでは約 $\frac{n}{2}$ 回
 - ※ $\{a_i\}_{i=1, \dots, n}$ の要素がすべて異なり、 $n + 1$ 通りの場合が同等の確率で起こると仮定すると
 - これらは異なるといってよいだろうか？

オーダー評価:

アルゴリズムの計算量はオーダーで評価する

- 多くの場合（特に理論的には）最悪ケースで考える
- （前述の）名簿を前から順に探すアルゴリズムでは、最悪ケースでは n 回のチェックが必要
 - n 回の比較、 n 回のカウンタ移動、 n 回のデータ読み取りを考えると、 $3n$ 回の操作が必要
 - 加えて、出力と停止を加えると計 $3n + 2$ ステップ？
 - 本質的には n が重要 → オーダー記法で $O(n)$ と書く
- オーダー記法： n が大きくなったときの振舞いを評価する
 - n が大きくなったとき、 $n, 3n, 3n + 2$ の違いは、 n^2 に比べると極めて小さい

関数の上界・下界: オーダー記法

- 関数の上界 : $T(n) = O(f(n))$
 - ある正整数 n_0 と c が存在して、任意の $n \geq n_0$ に対し $T(n) \leq c f(n)$ が成立すること
 - 例 : $4n + 4 \leq 5n$ ($n \geq 4$): $c = 5, n_0 = 4$ とする
- 関数の下界 : $T(n) = \Omega(f(n))$
 - $T(n) \geq c f(n)$ (厳密には「ある c が存在し無限個の n に対し」)
- 上界と下界の一致 : $T(n) = \Theta(f(n))$
 - c_1 と c_2 が存在して、 $c_1 f(n) \leq T(n) \leq c_2 f(n)$
 - 例 : $T(n) = 3n^2 + 3n + 10n \log n = \Theta(n^2)$

オーダー評価:

アルゴリズムの計算量はオーダーで評価する

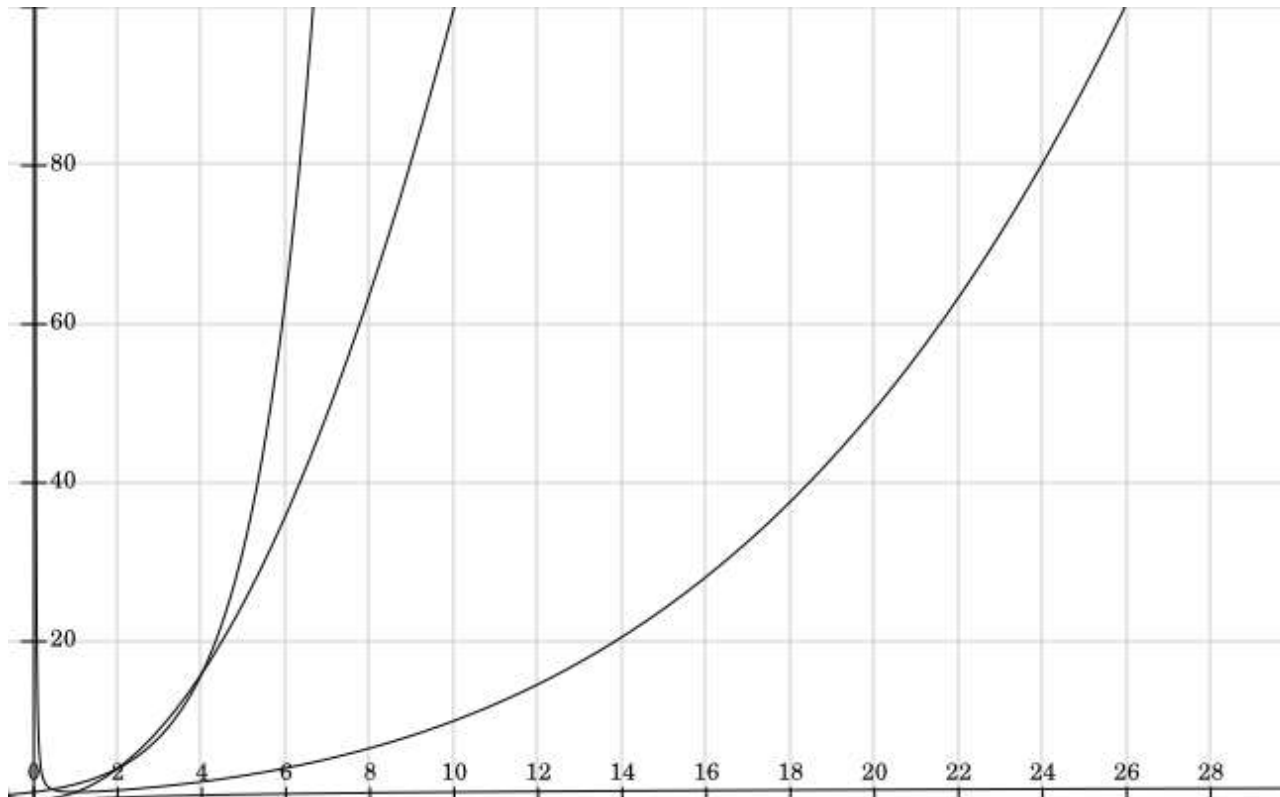
- 以降、特になにも言わない場合は最悪ケースで考える
- (前述の) 名簿を前から順に探すアルゴリズムでは、最悪ケースでは n 回のチェックが必要
 - n 回の比較、 n 回のカウンタ移動、 n 回のデータ読み取りを考えると、 $3n$ 回の操作が必要
 - 加えて、出力と停止を加えると計 $3n + 2$ ステップ?
 - 本質的には n が重要 → オーダー記法で $O(n)$ と書く
- オーダー記法: n が大きくなったときの振舞いを評価する
 - n が大きくなったとき、 $n, 3n, 3n + 2$ の違いは、 n^2 に比べると極めて小さい

オーダー記法の性質：

ざっくり言えば「係数を見捨てて一番大きいものを選ぶ」

- 直観的には係数を見捨てて一番大きいものを選べばよい

$$T(n) = 3n^2 + n^{\log n} + 5n \log n + 2^n \text{ ならば } O(2^n)$$



例： 多項式計算

■ 問題：多項式の評価

– 入力： a_0, a_1, \dots, a_n, x

– 出力： $a_n x^n + \dots + a_1 x + a_0$

■ 方法：

– 単純な方法：

$a_k x^k = a_k \times x \times \dots \times x$ ($k + 1$ 回の掛け算)

– $O(\sum_{k=1}^n k + n) = O(n^2)$

– Horner法：

$((\dots (a_n x + a_{n-1})x - a_{n-2})x - a_{n-3}) \dots)x + a_0$

– $O(n)$

計算量のクラス：

多項式時間で解けるものが効率的に解ける問題

- 一秒間に100万回の演算ができるとすると：

$O()$	$n = 10$	$n = 30$	$n = 60$
n	0.000001s	0.000003s	0.000006s
n^2	0.00001s	0.00009s	0.0036s
n^3	0.001s	0.027s	0.216s
2^n	0.001s	1074s	10^{12} s
3^n	0.06s	2×10^8 s	4×10^{22} s

- スパコン「京」は1秒に8162兆回（100億倍= 10^{10} ）

計算量のクラス：

多項式時間で解けるものが効率的に解ける問題とする

- P：多項式時間アルゴリズムを持つ問題のクラス
- NP完全問題、NP困難問題など（おそらく）多項式時間では解けない問題のクラス
 - ただし、実用上現れる重要な問題に、このクラスに属するものが多い

数え上げお姉さん： 単純な解法が破たんする例



整列（ソート）のアルゴリズム

整列問題（ソート）：

要素を小さい順に並び替える問題

■ 整列問題

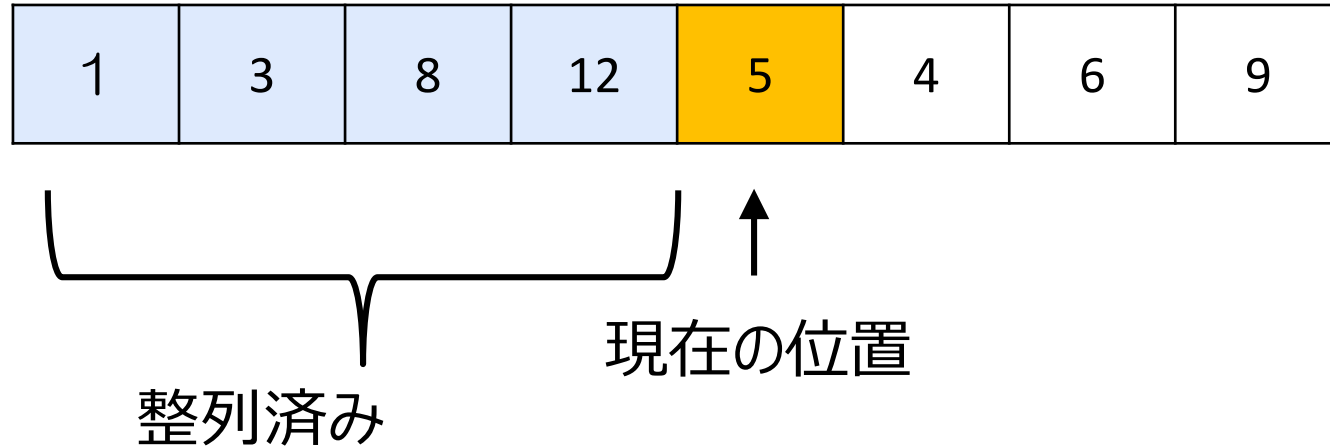
– 入力： n 個の数 a_1, a_2, \dots, a_n が入った配列

– 出力： $a_1' \leq a_2' \leq \dots \leq a_n'$ を満たす、入力列の置換

■ 例： 入力 $(4, 5, 2, 1) \rightarrow$ 出力 $(1, 2, 4, 5)$

単純なソートアルゴリズム： 左から順に整列する

- 現在の位置よりも左はすでに整列済みとする



- 現在の位置から左に見ていき、順序が保たれるところまで移動する



単純なソートアルゴリズムの計算量：

計算効率はいそれほど良くないが省スペースで実行可能

- 「現在の位置から左に見ていき、順序が保たれるところまで移動する」アルゴリズム
- 「」の操作には、現在の位置を j とすると $O(j)$ 回の交換が必要
- これを $j = 1, 2, \dots, n$ まで行くと
 $\sum_{j=1, \dots, n} O(j) = O(n^2)$ なので
あまり効率はよくない（良いアルゴリズムは $O(n \log n)$ ）
- ただし、「その場でのソート」が可能なので省スペース
 - 入力配列以外に定数個の領域しか使用しない

分割統治法

分割統治法：

アルゴリズム設計指針の1つで、問題を小問題に分割して解く

- 特定の問題に対するアドホックな個別の解法ではなく、多くの問題に適用可能なアルゴリズムの一般的な設計指針

- 分割統治法、動的計画法、...

- 分割統治法：

- 元の問題を、同じ構造をもった小さな問題に分割

- 小さな問題の解を統合して元の問題の解を得る

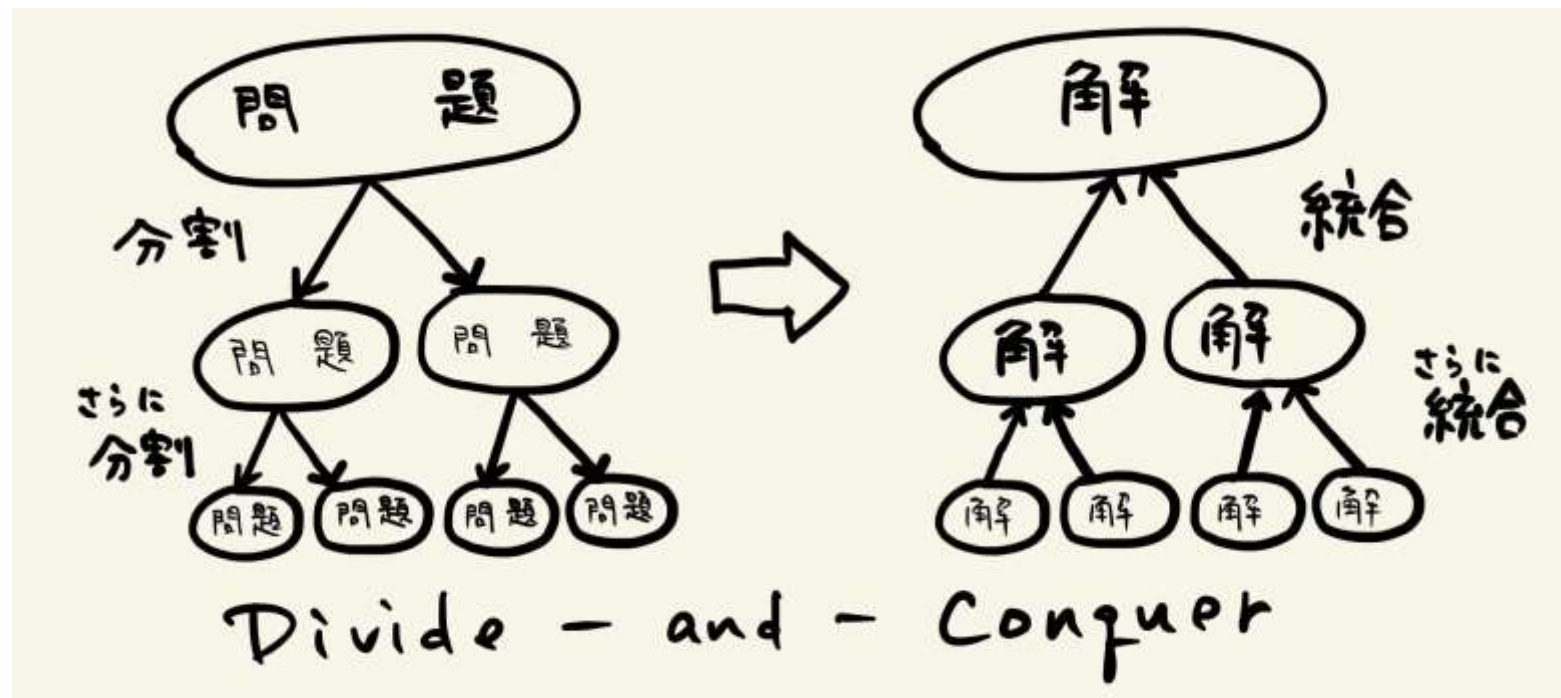
分割統治法：

アルゴリズム設計指針の1つで、問題を小問題に分割して解く

■ 分割統治法（Divide-and-conquer）：

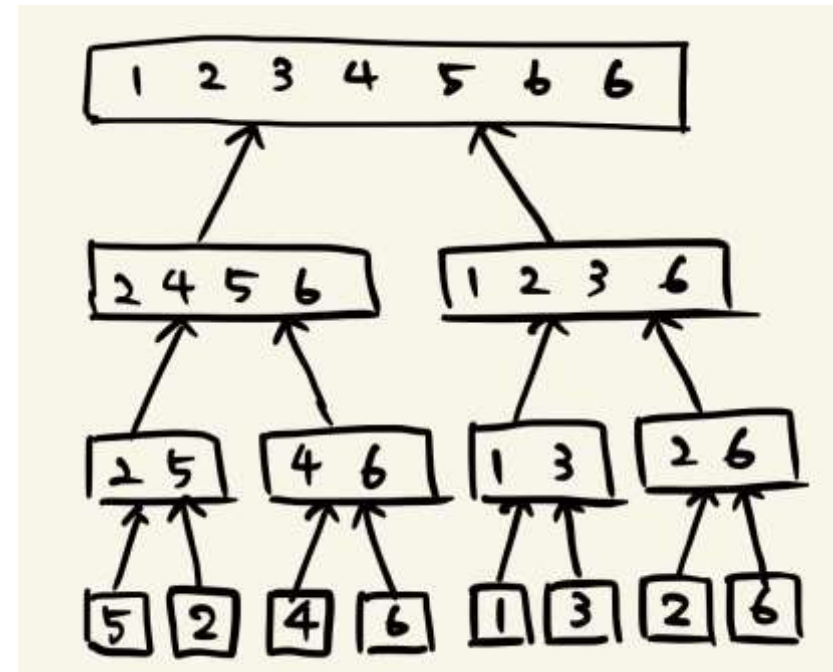
— 分割：元の問題を、同じ構造をもった小さな問題に分割

— 統合：小さな問題の解を統合して元の問題の解を得る



分割統治法の例： マージソート

- 入力された配列を前後に分割し、それぞれに対してマージソートを適用する
 - 再帰的に行うことで、サイズ1の配列まで到達する
 - 逆向きに統合して解を構成する
- 例：配列 (5,2,4,6,1,3,2,6)



マージソート：

マージソートの計算量は $O(n \log n)$

- $n = 2^k$ として $O(n \log n)$

- 実用的には後に紹介するクイックソートのほうが速い

- 計算量評価の再帰式：

$$T(n) = \begin{cases} O(1) & (n = 1) \\ 2T(n/2) + O(n) & (n \geq 2) \end{cases} = O(n \log n)$$

再帰 統合

マージソート：

マージソートの計算量は $O(n \log n)$

■ 計算量評価の再帰式：

$$T(n) = \begin{cases} O(1) & (n = 1) \\ 2T(n/2) + O(n) & (n \geq 2) \end{cases}$$

■ $T(n) = 2T(n/2) + cn = 2 \left(T\left(\frac{n}{2^2}\right) + c \frac{n}{2} \right) + cn$

$$= 2 \left(2 \left(\dots \left(2 \left(\underbrace{T\left(\frac{n}{2^k}\right)}_c + c \frac{n}{2^{k-1}} \right) + c \frac{n}{2^{k-2}} \right) \dots \right) + c \frac{n}{2} \right) + cn$$

$$= c2^k + \underbrace{cn + \dots + cn}_k < n \log n.$$

分類定理（簡易版）：

計算量の漸化式から計算量を導く

- $T(n)$ の漸化式から $T(n)$ のオーダーを導く
- 定理：大きさ n の問題を、大きさ $\frac{n}{b}$ の問題 a 個に分割した

$$- T(n) = \begin{cases} c & (n = 1) \\ aT\left(\frac{n}{b}\right) + cn & (n \geq 2) \end{cases}$$

$$- \text{このとき} : T(n) = \begin{cases} O(n) & (a < b) \\ O(n \log n) & (a = b) \\ O(n^{\log_b a}) & (a > b) \end{cases}$$

クイックソート：

分割統治法にもとづく高速なアルゴリズム

- 最もよく用いられる、分割統治に基づくソートアルゴリズム
- 平均計算量 $O(n \log n)$ 、最悪では $O(n^2)$

— 実用的には速い

— その場でのソートが可能

- アルゴリズム $\text{QuickSort}(A, p, r)$

p : 配列中でソートする部分の先頭
 r : 配列中でソートする部分の末尾

1. $q \leftarrow \text{Partition}(A, p, r)$: 分割点 q をみつけて分割
 2. $\text{QuickSort}(A, p, q)$
 3. $\text{QuickSort}(A, q + 1, r)$
- } 分割したそれぞれについて
クイックソートを適用

クイックソートの分割関数 $\text{Partition}(A, p, r)$:
基準となる要素（枢軸）との大小比較で2グループに分割

- $A[p]$ を枢軸(pivot)として選択
- $A[p:r]$ を $A[p]$ 未満の要素と、 $A[p]$ 以上の要素に分割
 - $A[p]$ 未満の要素が新たに $A[p:q]$ となる
 - $A[p]$ 以上の要素が新たに $A[q + 1:r]$ となる
 - 2つのインデックス i, j を使って配列 $A[p:r]$ を操作 :
 - $j = r$ から左に走査して枢軸未満の要素を発見
 - $i = p$ から右に走査して枢軸以上の要素を発見
 - 両者を入れ替える
 - これを両者が出会うまで繰り返す ($O(r - p)$)

クイックソートの計算量：
「平均で」 $O(n \log n)$ を実現できる

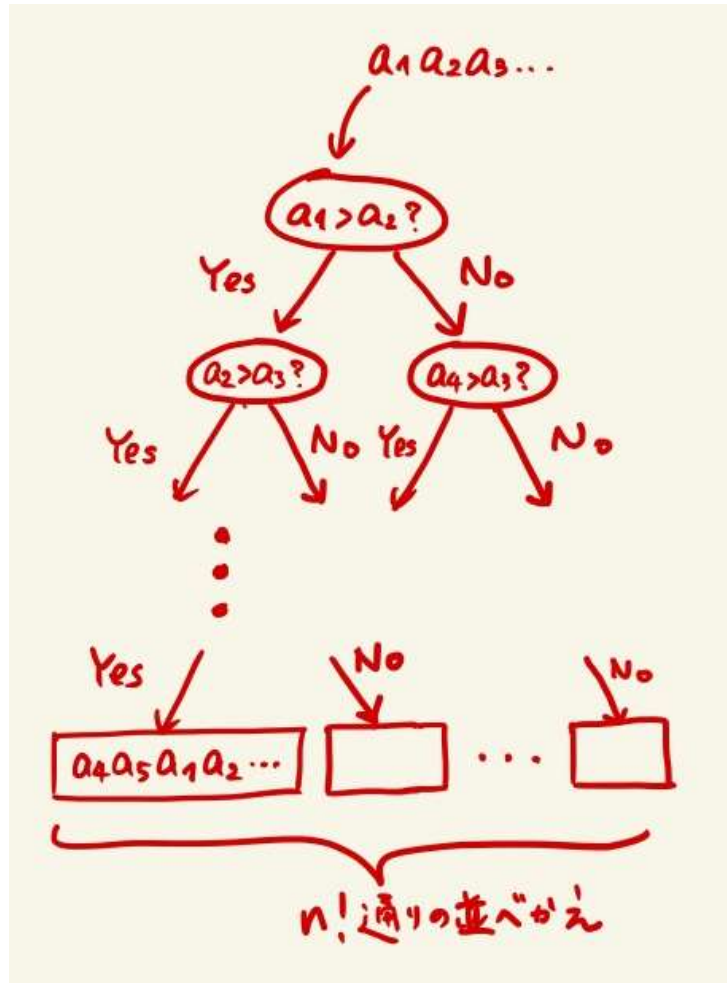
- 最悪の場合：
 n 個の要素が $n - 1$ 個と1個に分割されたとすると $O(n^2)$
 - 1回の分割でサイズが定数個しか減らない場合
- 最良の場合：
 n 個の要素が $\frac{n}{2}$ 個2つに分割されたとすると $O(n \log n)$
 - 分割定理で $a = b$ の場合
 - 定数分の1のサイズに分割される場合
- 最悪の場合を避けるために：ランダムに枢軸を選択
 - 問題例には依存しない平均計算量を達成できる

ソートの計算量の下界：

$O(n \log n)$ より小さい計算量は達成できない

- ソートのアルゴリズムは最悪計算量 $O(n \log n)$ が必要
- n 個の要素はすべて異なるとすると、ソート後に得られる列の可能性は $n!$ 通り
- ソートは2つの数の比較を繰り返すことで動く
- ソートの流れを2分木で書く：
 - 各頂点で2つの数を比較して分岐
 - 葉は、ある特定の並べ替えに対応
 - 全ての並べ替えが可能であるために葉が $n!$ 個は必要
- その実現には少なくとも木の高さが $O(n \log n)$ 必要

ソートの計算量の下界： $O(n \log n)$ より小さい計算量は達成できない



この高さがどう頑張っても
 $O(n \log n)$ になることを示す

一番下では、とある並び替えが得られる

ソートの計算量の下界：

$O(n \log n)$ より小さい計算量は達成できない

- 全ての可能な並び替えが得られるためには、最下段の要素が少なくとも $n!$ は必要
- 図の高さがキッチリ h （完全2分木）とすると、最下段の要素（葉）の数は 2^h
 - 逆に 2^h 個の葉をもつ木で最も低いのが完全2分木
- よって、 $2^h \geq n!$ でないといけない
- 対数をとると
$$h \geq \log n! \geq \log(n/e)^n = n \log n - n \log e = O(n \log n)$$
 - なお、Stirlingの公式 $n! \geq \sqrt{2\pi n}(n/e)^n \geq (n/e)^n$

順序統計量：

大きい方から k 番目の要素は線形時間で発見可能

- 順序統計量：大きい方から k 番目の要素
- ソートを使えば $O(n \log n)$
- 工夫すれば $O(n)$ で可能
 - 平均的に $O(n)$ で見つける方法
 - 最悪ケースで $O(n)$ で見つける方法

平均 $O(n)$ の順序統計量アルゴリズム： クイックソートと同じ考え方で可能

- $q \leftarrow \text{Partition}(A, p, r)$ を実行した結果：
 1. $k \leq q$ であれば、求める要素は $A[p:q]$ にある
 2. $k > q$ であれば、求める要素は $A[q+1:r]$ にある—再帰的にPartitionを呼ぶことで範囲を限定していく
- 平均的には問題サイズは半々になっていく：

$$T(n) = T\left(\frac{n}{2}\right) + O(n) = O(n)$$

クイックソートでは $2T\left(\frac{n}{2}\right)$

分割

最悪 $O(n)$ の順序統計量アルゴリズム： うまく「だいたい真ん中」をとってくる

- $\text{Order}(A, k)$: 全要素 A の大きい方から k 番目の要素を見つける
 1. A を5個ずつのグループに分け、それぞれをソートして、中央値（3番目の値）を見つけ、これらを集めて T とする
 2. T の中央値 m を見つける $\text{Order}(A, \lfloor n/10 \rfloor)$
 3. A を m より大きいもの（ S_1 ）、同じもの（ S_2 ）、小さいもの（ S_3 ）に分割する
 4. $k \leq |S_1|$ ならば $\text{Order}(S_1, k)$ 、 $|S_1| \leq k \leq |S_1| + |S_2|$ ならば m は目的の要素、 $k > |S_1| + |S_2|$ ならば $\text{Order}(S_3, k - (|S_1| + |S_2|))$

動的計画法

動的計画法：

ボトムアップの再帰によるアルゴリズム設計指針

- 動的計画法は分割統治法と同じく、問題を再帰的に分割する方法、ただし：
 - 分割統治： トップダウン
 - 動的計画： ボトムアップ
- 動的計画法の大まかなステップ：
 1. 問題の構造を再帰的に捉える
 2. 解を再帰的に構成する
 3. ボトムアップで解を計算する

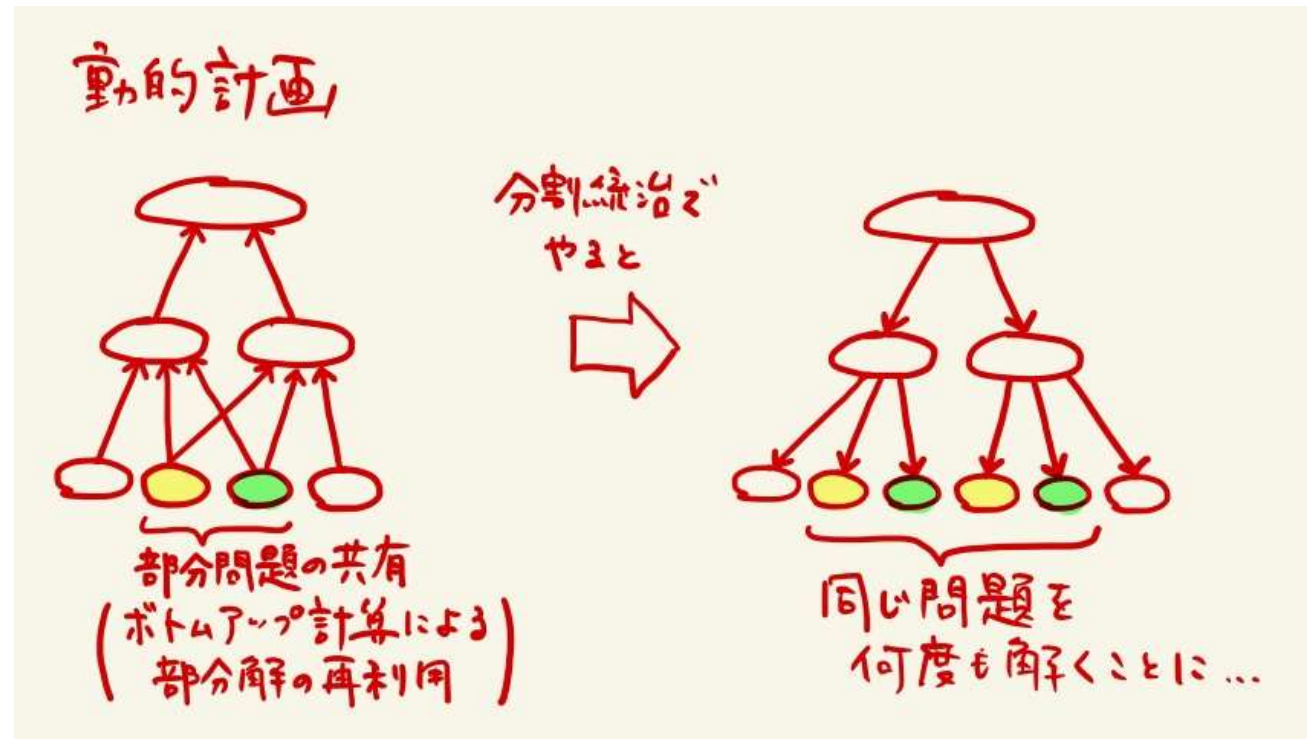
} ここは分割統治と同じ

動的計画法のポイント： 解の使いまわしによる効率化

- 分割された問題が重複している場合に差が生じる
 - トップダウンでは同じ問題を何度も解くことになる
 - ボトムアップでは解の使いまわしが可能
 - 両者に指数的な差が生じる
- 逆にいえば、部分問題が重複していることが動的計画法のカギ

動的計画法のポイント： 解の使いまわしによる効率化

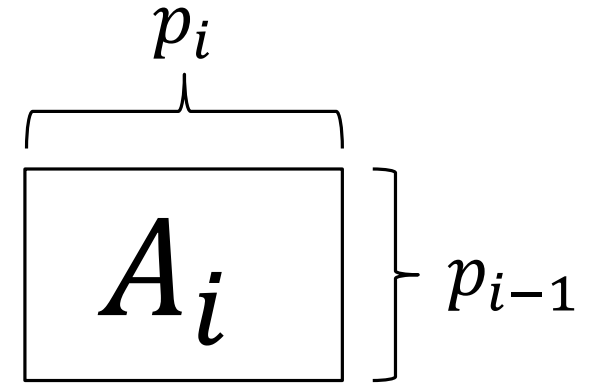
- 分割された問題が重複している場合に差が生じる



- 部分問題が重複していることが動的計画法のカギ

動的計画法の例： 複数の行列積の計算

- 入力： 行列 A_1, A_2, \dots, A_n
- 出力： 積 $A_1 A_2 \cdots A_n (= A_{1,\dots,n})$
- 掛け算できる前提



- $A_i A_{i+1}$ の計算は $O(p_{i-1} p_i p_{i+1})$ にかかる
- $A_1: 10 \times 100, A_2: 100 \times 5, A_1: 5 \times 50$ とすると
 $((A_1 A_2) A_3) = 7500, (A_1 (A_2 A_3)) = 75000$

解の構成における観察： 最適解は部分最適解からつくられる

- $A_1 A_2 \cdots A_n (= A_{1,\dots,n})$ の計算において、 k 番目の分割が最後にくるとする
 - $A_{1,\dots,k}$ と $A_{k+1,\dots,n}$ を別々に計算して最後に統合
 - 最後の統合コストは $O(p_0 p_k p_n)$
- これが最適解であるなら、 $A_{1,\dots,k}$ と $A_{k+1,\dots,n}$ の計算コストも最小のはず（部分問題の最適解のはず）
 - そうでなければ、これをコスト最小のものに置き換えれば全体のコストが下がるはず

最小コストについて成り立つ再帰式：

部分行列積の最適解を組合わせて大きな最適解をつくる

- $A_{i,\dots,j}$ を計算する最小のコストを $m[i, j]$ とする

- $m[i, j]$ は再帰的に表現できる：

$$m[i, j] = \begin{cases} 0 & (i = j) \\ \min_{i \leq k < j} m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j & (i \neq j) \end{cases}$$

- トップダウン計算（分割統治）だと指数的な計算量になる

$$T(n) = \sum_{k=1}^{n-1} (T(k) + T(n - k) + c) = 2 \sum_{k=1}^{n-1} T(k) + cn$$

動的計画法の計算量：

ボトムアップ計算により多項式時間で解が求まる

- 再帰式の適用により最小の $m[1, n]$ を求める

- ボトムアップで計算：

1. $i = j$ の場合について計算（全部ゼロ）

2. $i = j - 1$ の場合について計算

3. $i = j - 2$ の場合について計算

4. ...

— 上記が n ステップ、それぞれで $O(n)$ 個の再帰式評価、それぞれの評価に $O(n)$ 必要なので、全部で $O(n^3)$ の計算量

- バックトラック：各再帰式での最良の k を記憶しておくことで実際の掛け算の順番を得る

探索

探索問題:

データ集合から所望のデータを見つけてくる問題

- データ集合から所望のデータを見つけてくる問題
 - データは「キー」と「データ（の内容）」からなる
 - あるキーが与えられたとき、それに一致するキーをもつデータを見つけてくる
- 二分探索木 や ハッシュ によって実現可能

2 分探索木：

データ集合から所望の要素を見つけるデータ構造

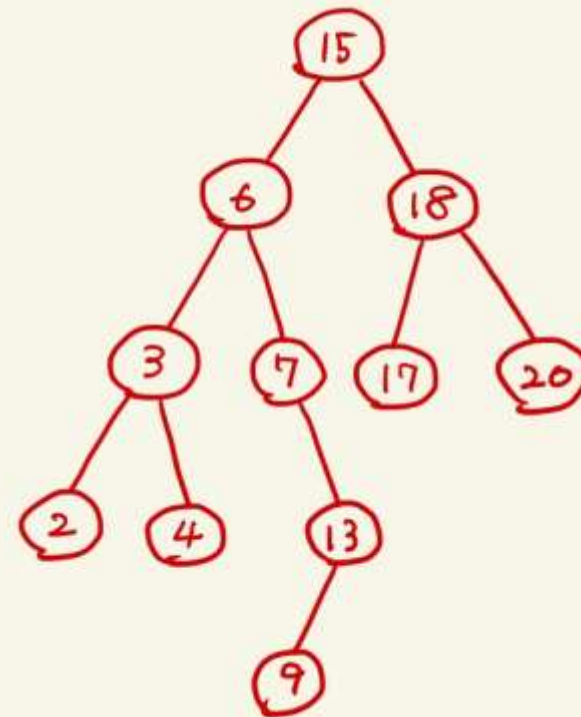
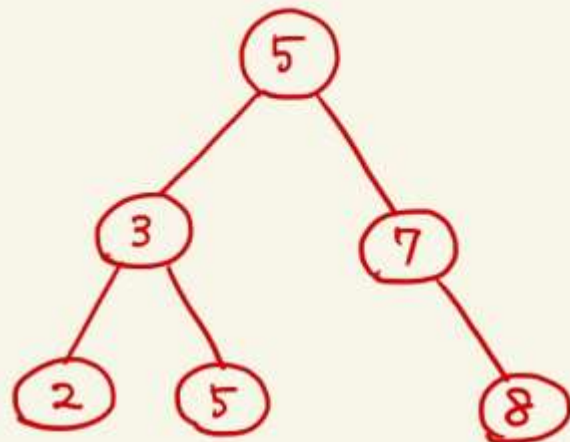
■ 2 分探索木：

- 各節点が key , left (左の子), right (右の子), p (親) をそれぞれ最大 1 つもつ二分木
- キーには順序がつけられる； 2 つの節点 x, y に対して
$$\text{key}(x) = \text{key}(y), \text{key}(x) > \text{key}(y),$$
$$\text{key}(x) < \text{key}(y)$$
のいずれかが成り立つ
- キーは以下の条件を満たす
 - y : x の左側の子を根とする部分木に含まれる頂点
 - y' : x の右側の子を根とする部分木に含まれる頂点
 - $\text{key}(y) \leq \text{key}(x) \leq \text{key}(y')$

二分探索木の探索:

$O(\log n)$ で発見可能

- 各節点での大小比較により、 $O(\log n)$ で発見できる
- 下記の例で確認できる

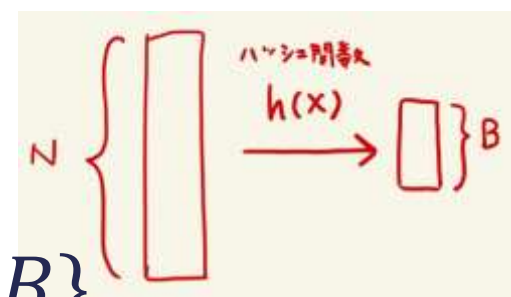


ハッシュ表:

$O(1)$ で探索するためのデータ構造

- データ集合から、あるキーをもつデータを $O(1)$ で発見する
- 単純な実現：
 - キーに対して自然数を割り当てる($1 \sim N$)
 - キーがアルファベット6文字なら $N = 26^6 \simeq 3 \times 10^8$
 - サイズの配列 N を準備する
 - キーを自然数に変換して、配列のその位置に格納する
- 問題点：
 - 長さ N の配列を使うと大きすぎる
 - M 個のデータを格納するのに、 $M \ll N$ なのでムダが多い

ハッシュ関数: ハッシュ表を省スペースで実現する



- ハッシュ関数 $h(x): \{1, 2, \dots, N\} \rightarrow \{1, 2, \dots, B\}$
- キー x を持つデータを $h(x)$ の位置に格納する
 - 異なる x が同じ位置に格納されうるので、衝突したら
(例えば) 次の場所に格納
- $h(x)$ のデザインは色々考えられるが、なるべく均等に格納されるものがよい
 - 例: $x = a_1 a_2 \dots a_6$ (アルファベット6文字) に対する
ハッシュ関数 $h(x) = \sum_{i=1,2,\dots,6} c(a_i) \bmod B$: c は文字コード

ハッシュの衝突：

内部ハッシュと外部ハッシュによって衝突を回避

- 異なるキー x と y に対して $h(x) = h(y)$ となることがある
 - $x = \text{すし}, y = \text{しす}$
- 衝突の回避法：内部ハッシュと外部ハッシュ
 - 外部ハッシュ：衝突回避のためにリストを格納
 - 内部ハッシュ：衝突時に別のハッシュ関数を用いる

外部ハッシュ： 衝突回避のためにリストを格納する

- 配列にデータを直接格納せず、 $h(x)$ の位置にリスト（へのポインタ）を格納
 - $M > B$ でもよい
- 計算量：ハッシュが一樣ならば平均的に $O(1)$
 - ハッシュ関数を用いて配列にアクセスするところまで $O(1)$
 - そこから先の計算量はリストの長さ ℓ に依存
 - ハッシュが一樣ならば $\ell \approx \frac{M}{B}$,
これを定数とみなせば平均的に $O(1)$

内部ハッシュ：

衝突したときのために複数のハッシュ関数を用意

- 配列にデータを直接格納

- $M \leq B$ である必要

- ハッシュ関数の列 h_0, h_1, h_2, \dots を用意して、 h_i が衝突したら、次の h_{i+1} を調べる

- 例： $h_i = h(x) + i \bmod B$

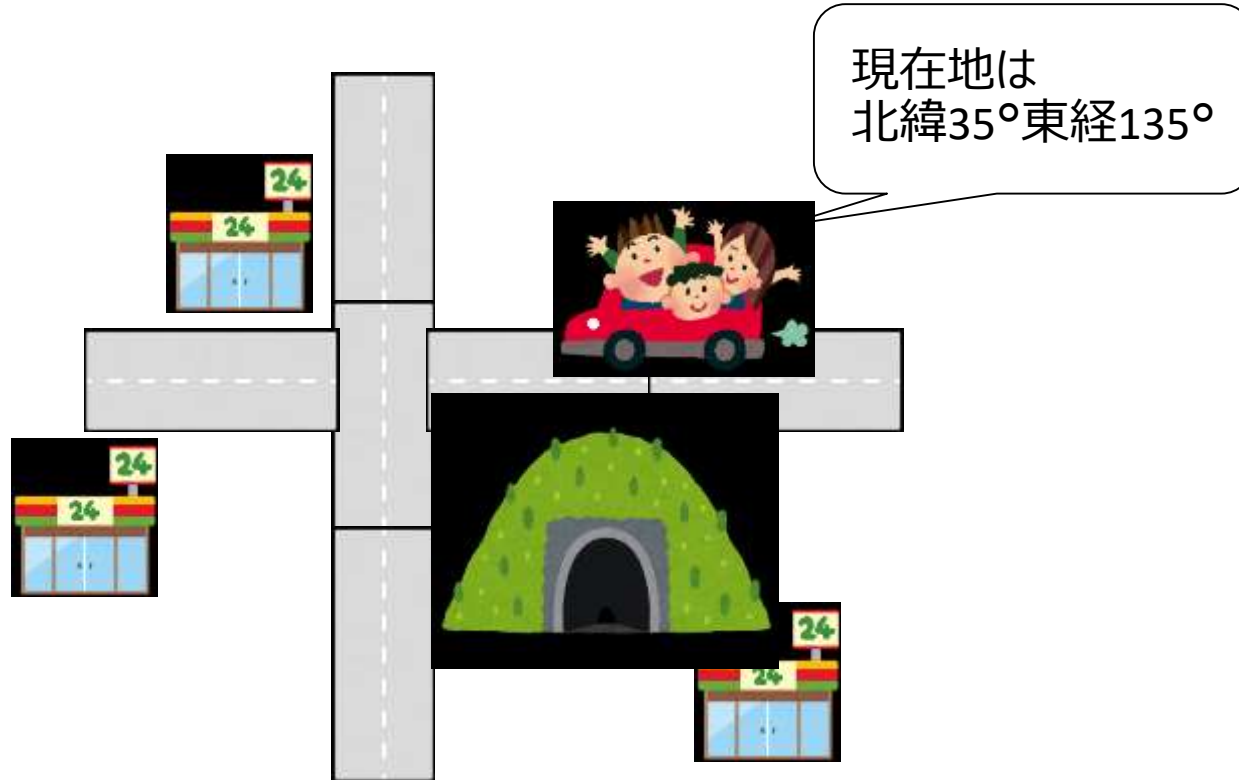
近傍探索問題

近傍探索： 類似データを探す問題

- これまで考えてきた探索問題は、質問と同一のキーをもつデータを探す問題
- 近傍探索：質問に類似したデータを探す
 - 最近傍探索：最も類似したデータを探す
 - もしくは一定度以上類似したデータを探す
- たとえば：
 - 近隣施設の検索（地理情報システム）
 - 文字認識（パターン認識）

最近傍探索の例： 近隣施設の検索

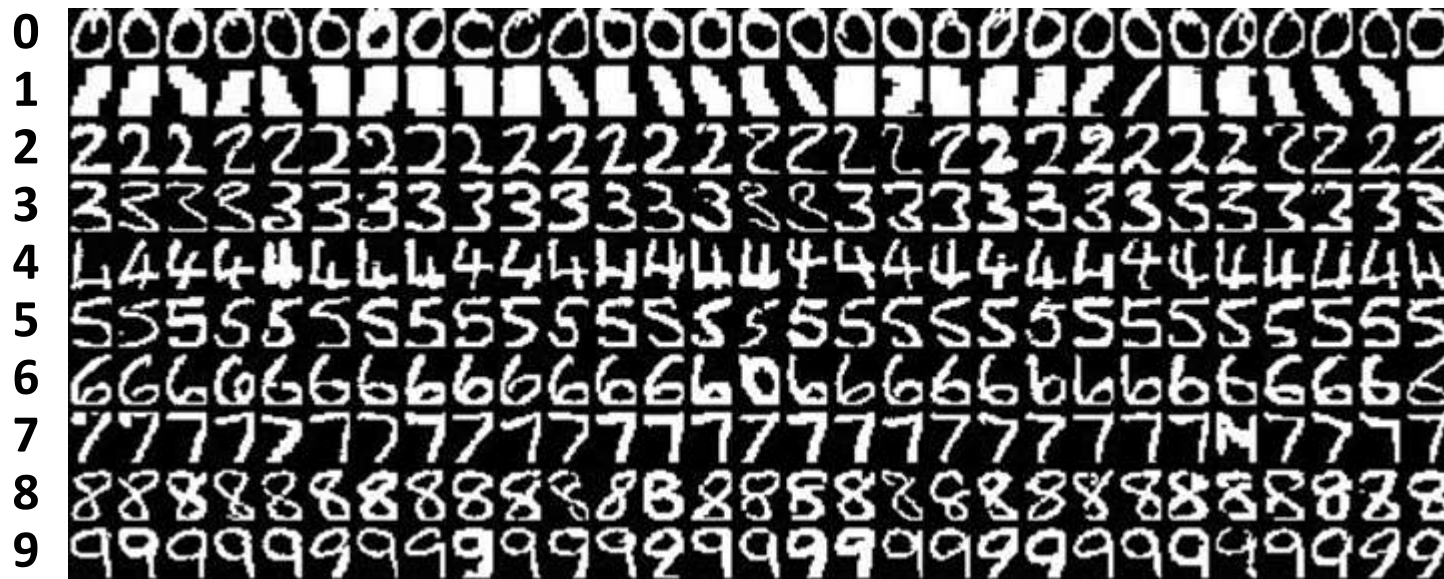
- 「OK、G☺☺gle。最寄りのコンビニはどこ？」



手書き文字認識：

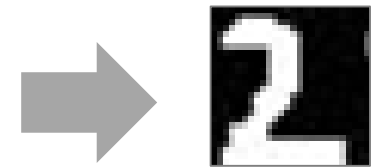
過去の手書き文字データをもとに認識する

- 手書き数字認識：手書きの数字を読み込み 0 ～ 9 の数字を認識する
- 正解の分かっている手書き数字データをもとに、新たな手書き数字を認識する



収集した手書き文字データ

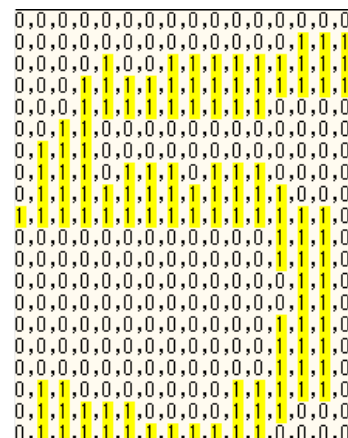
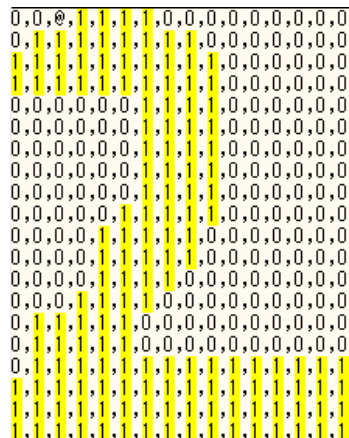
これは何？



手書き数字の表現:

手書き数字を多次元のベクトルとして表現

- 各数字を $16 \times 20 = 320$ 次元の2値ベクトルとして表現

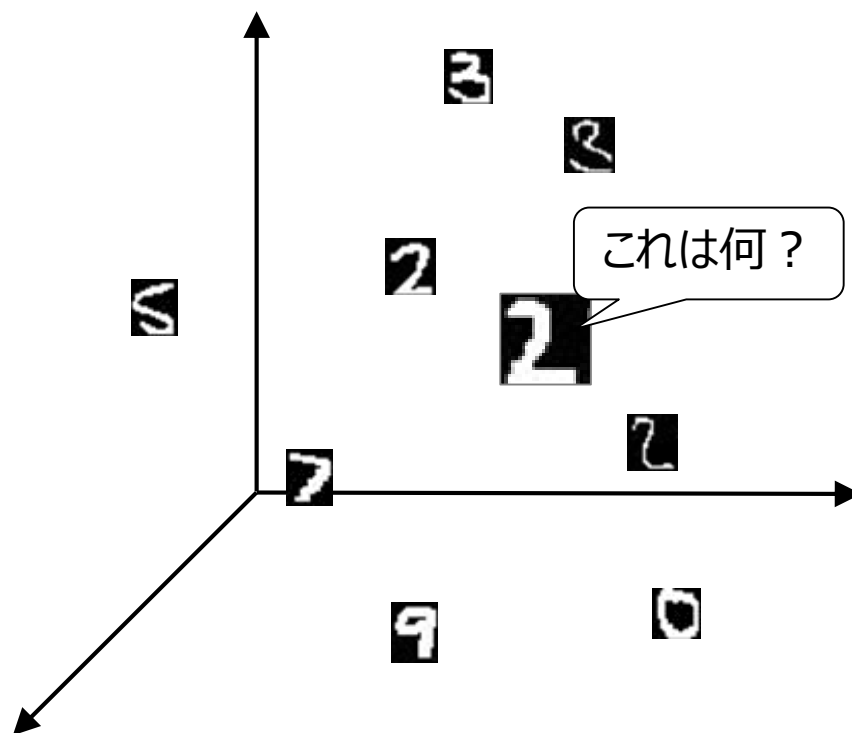


- ピクセルの濃淡がある場合、実数値ベクトル
- あるいは局所的なパターンの出現で特徴ベクトルを構成する

手書き数字の認識：

過去の例の中から対象に最も近いものを見つける

- 対象の手書き文字 **2** に最も近いものを多次元空間内で発見する



ボロノイ図・ドロネー図

ボロノイ図・ドロネー図：

空間全体を n 個の点のいずれに近いかで分割する

- 空間（通常2次元）全体を点集合 $S = \{P_1, P_2, \dots, P_n\}$ （母点とよぶ）のどれに近いかで分割したもの
- P_i のボロノイ領域： $d(P, P_i) < d(P, P_j) \ (i \neq j)$ となる P 全体（ d は空間上の距離）
 - ボロノイ辺：2つのボロノイ領域を隔てる境界辺
 - ボロノイ点：3つ以上のボロノイ領域の境界が共有する点
- ドロネー図：ボロノイ領域が隣り合う母点同志を線分で結んでできる図形

ボロノイ図を使った最近傍点探索：

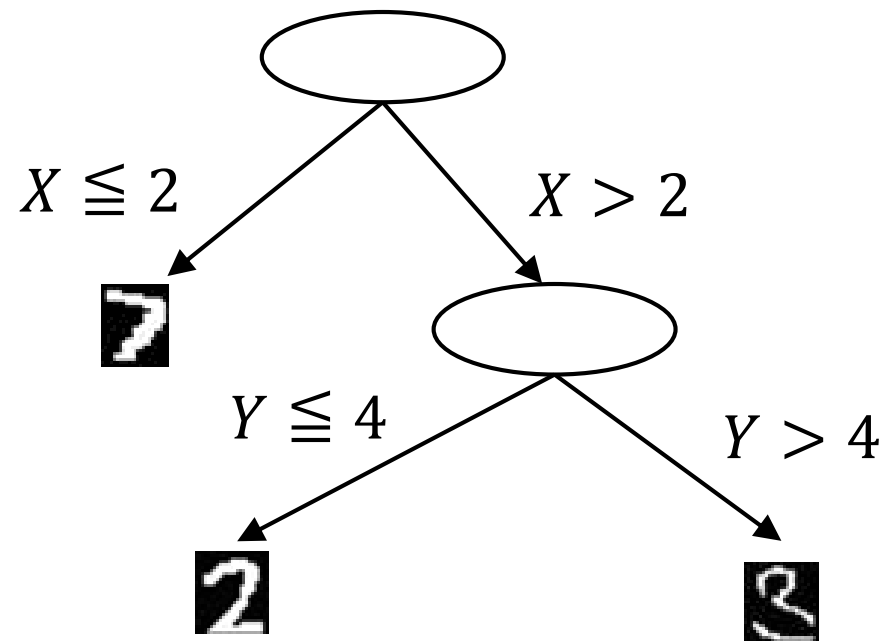
空間全体を n 個の点のいずれに近いかで分割する

- アルゴリズム： 検索点 P に対する S の最近傍点を出力
 1. 任意の母点 P_i を選ぶ
 2. P_i に隣接する母点の中で P に最も近い点 P_j を見つける
 - 隣接：ボロノイ領域が隣り合う（ドロネー辺がある）
 3. $d(P, P_i) < d(P, P_j)$ なら P_i を最近傍点として出力し終了
 4. そうでなければ $P_i \leftarrow P_j$ としてステップ2へ
- 点が概ね一様に分布しているとして $O(\sqrt{n})$

k -d木

2分木による近傍探索： 空間探索のための探索木

- 空間を探索するための探索木をつくる
- 空間全体の領域分割を各次元における 2 分割を繰り返すことで行う



k -d木 :

空間探索のための探索木

- k -d木 : k 次元の空間を探索するための2分探索木
- k -d木において、各々の分割はデータ点において行われる

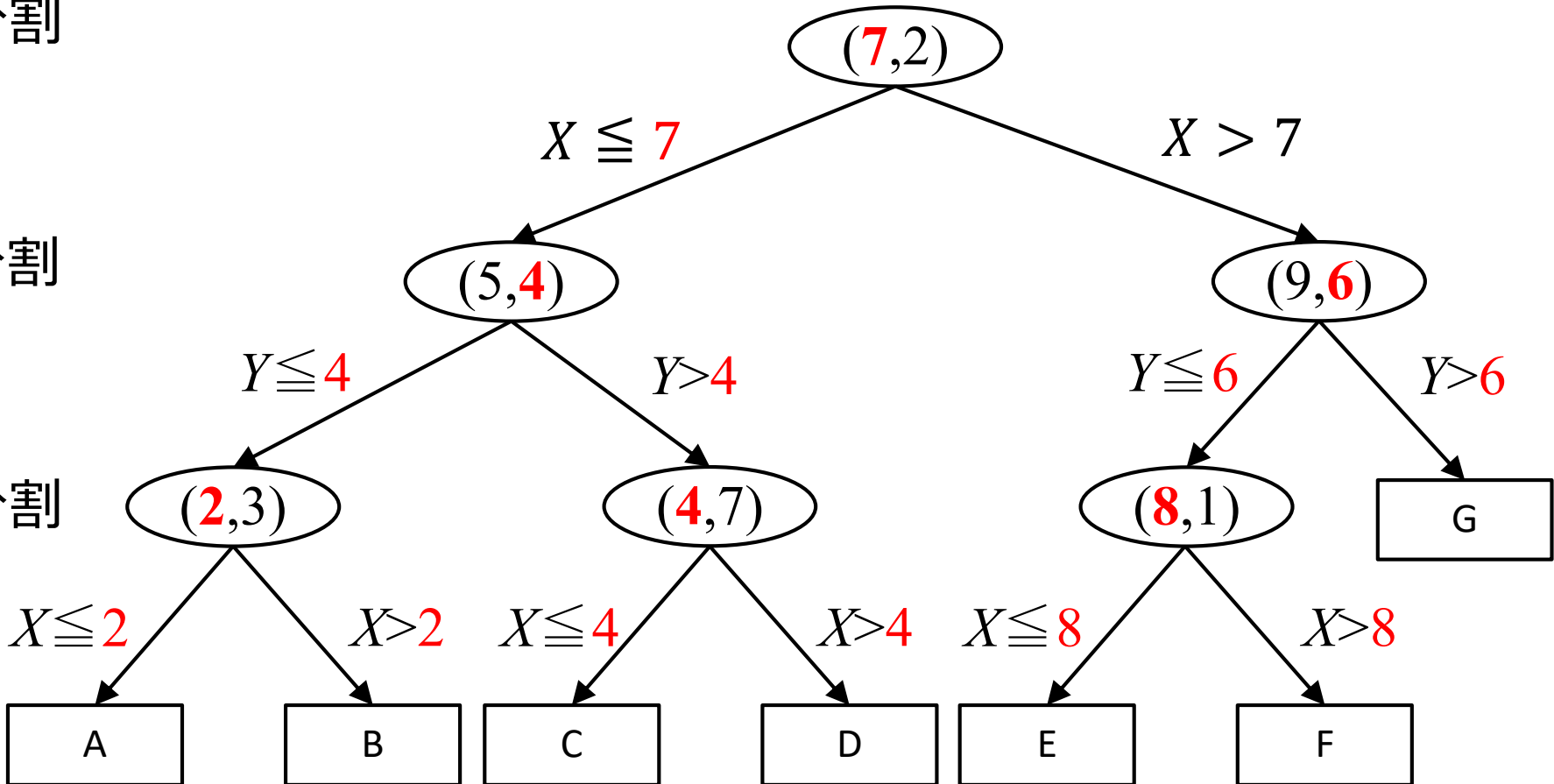
k -d木の例： 空間をデータ点で領域に分割

- 例：2次元データ $(2,3)$, $(5,4)$, $(9,6)$, $(4,7)$, $(8,1)$, $(7,2)$

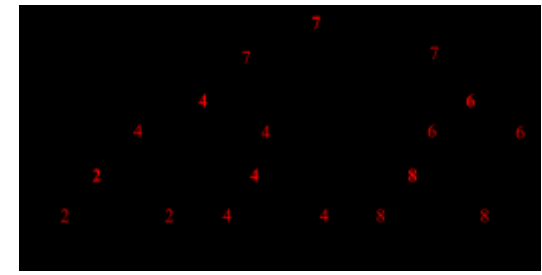
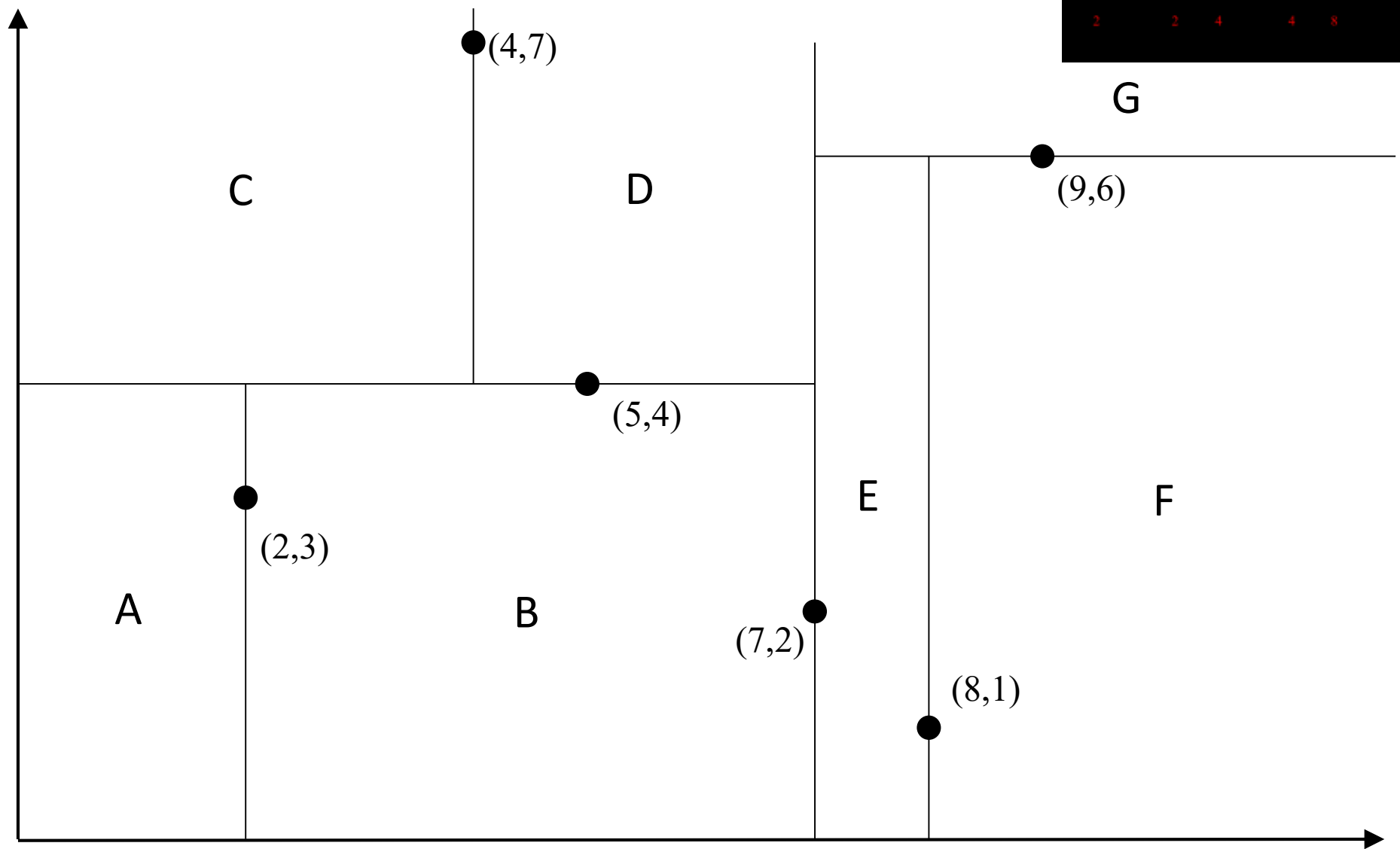
X 軸で分割

Y 軸で分割

X 軸で分割



k -d木の例： 空間をデータ点で領域に分割



k -d木の構成：

分割軸を決めて中央値で分割する

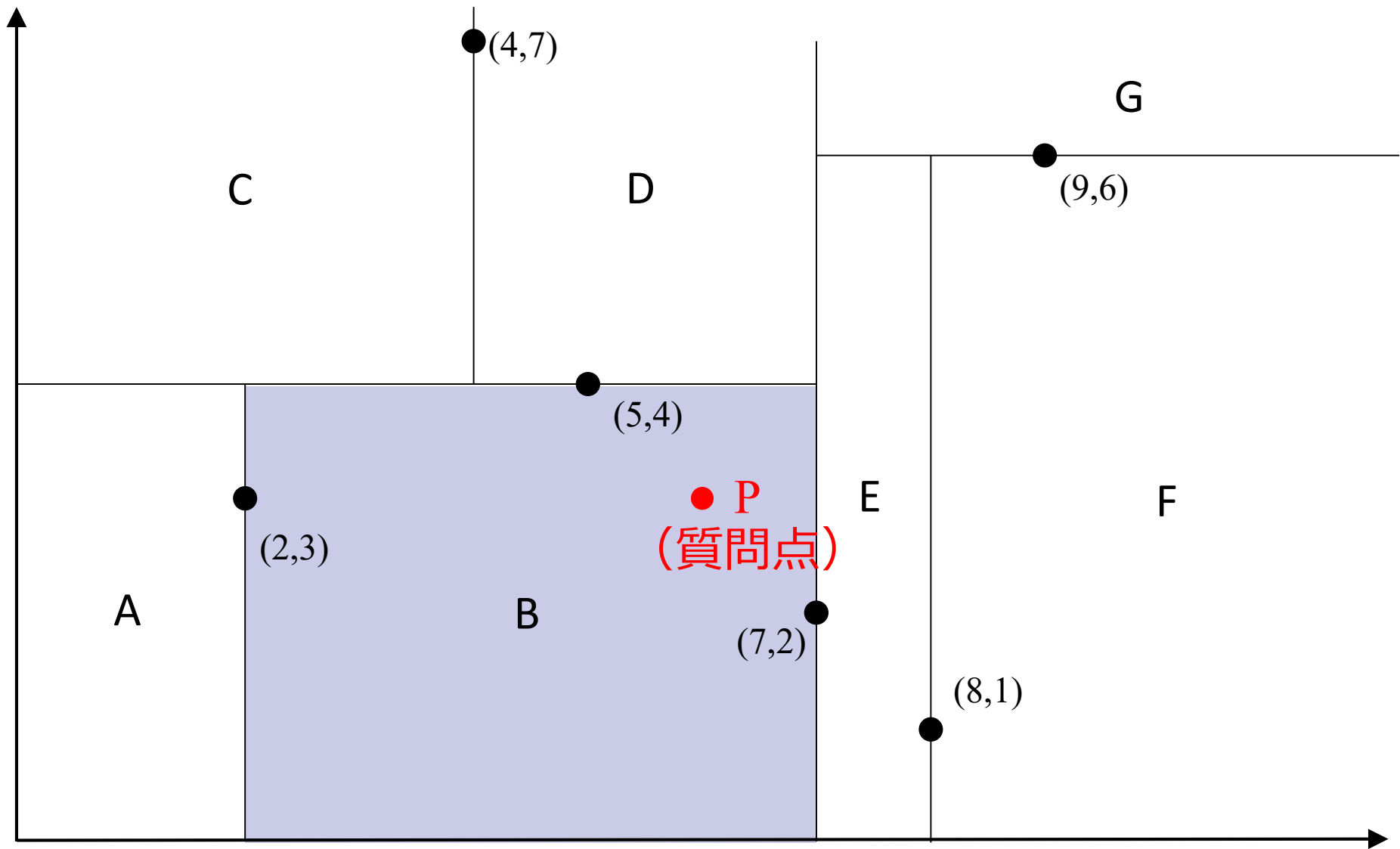
■ 構成法（のひとつ）

- 軸を適当な順で選ぶ（たとえば分散が最大の軸）
- 選んだ軸について、領域に含まれるデータの（その軸についての）中央値となるデータで分割
- 再帰的に分割を繰り返して、これ以上分割できなくなったときに終了

■ 探索木は平衡木が望ましい

- 中央値で分割を繰り返すので、高さはおおむね $\log n$

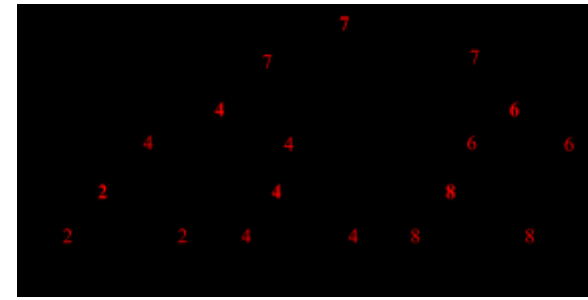
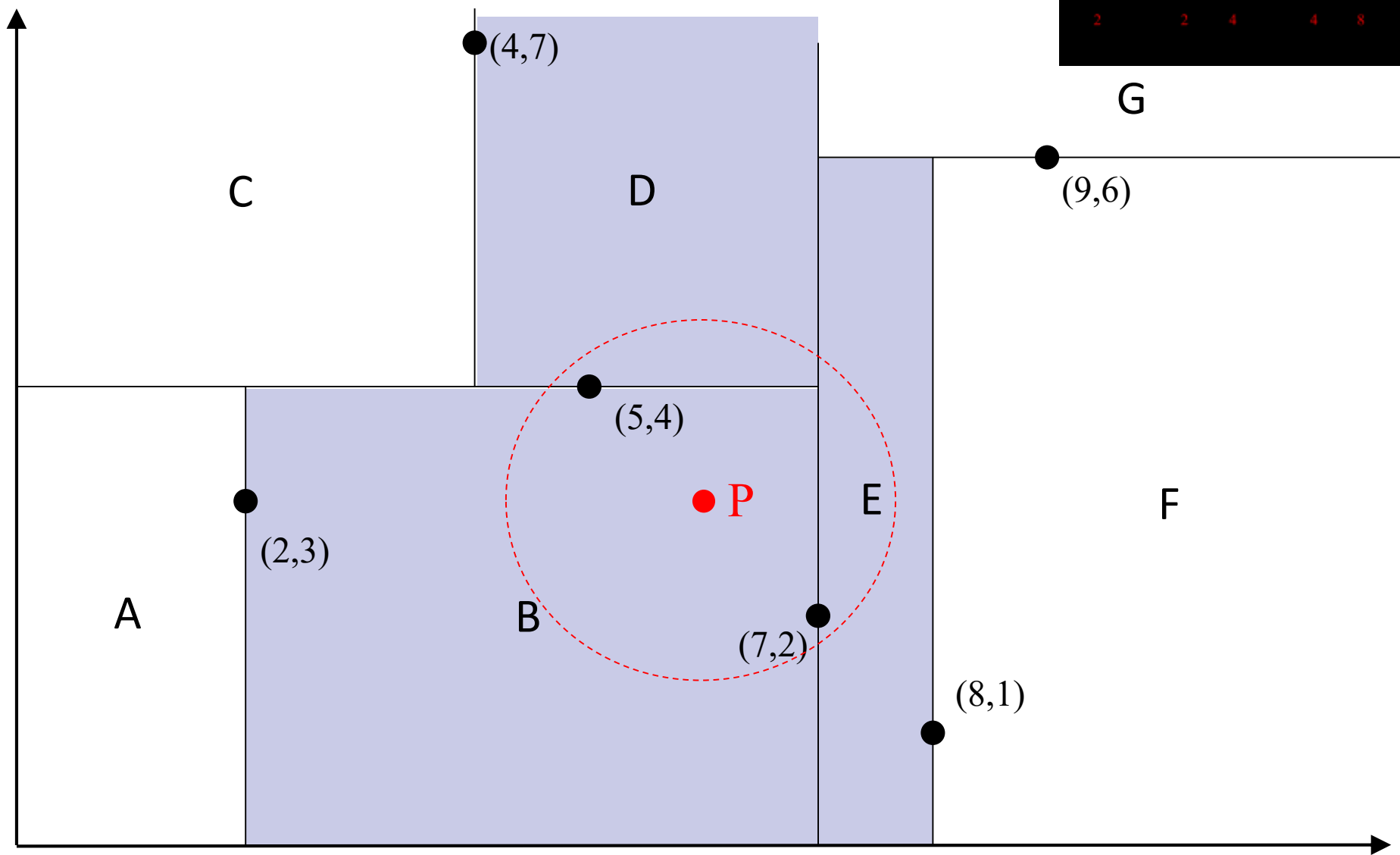
k -d木による質問点の探索：
探索木を下ることで質問点を含む領域を $O(\log n)$ で発見



k -d木による近傍点の探索： 枝刈りによって効率的に探索を行う

- 質問点 P を含む半径 r の領域にデータがあるかを調べたい
- 基本方針：質問点 P を含む半径 r の領域と、分割された各領域に重なりがあるかをチェックしながら k -d木を下る
- k -d木の各分岐において：質問点 P を含む半径 r の領域に...
 - 分岐点が含まれれば解として記録しておく
 - 分岐先の領域が重なるなら、そちらの探索を続行する
 - 両方の分岐先の領域と重なるなら、両方探索する
 - 重ならない領域については、以降の分岐の探索は打ち切れる
- 最悪の場合、すべての分岐をチェックする必要がある

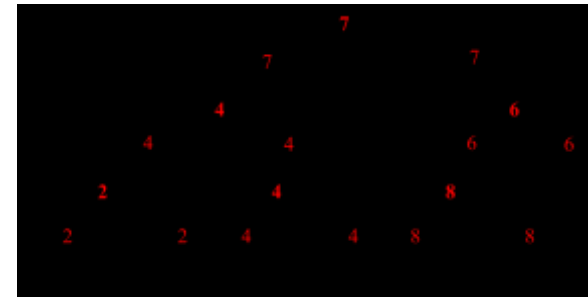
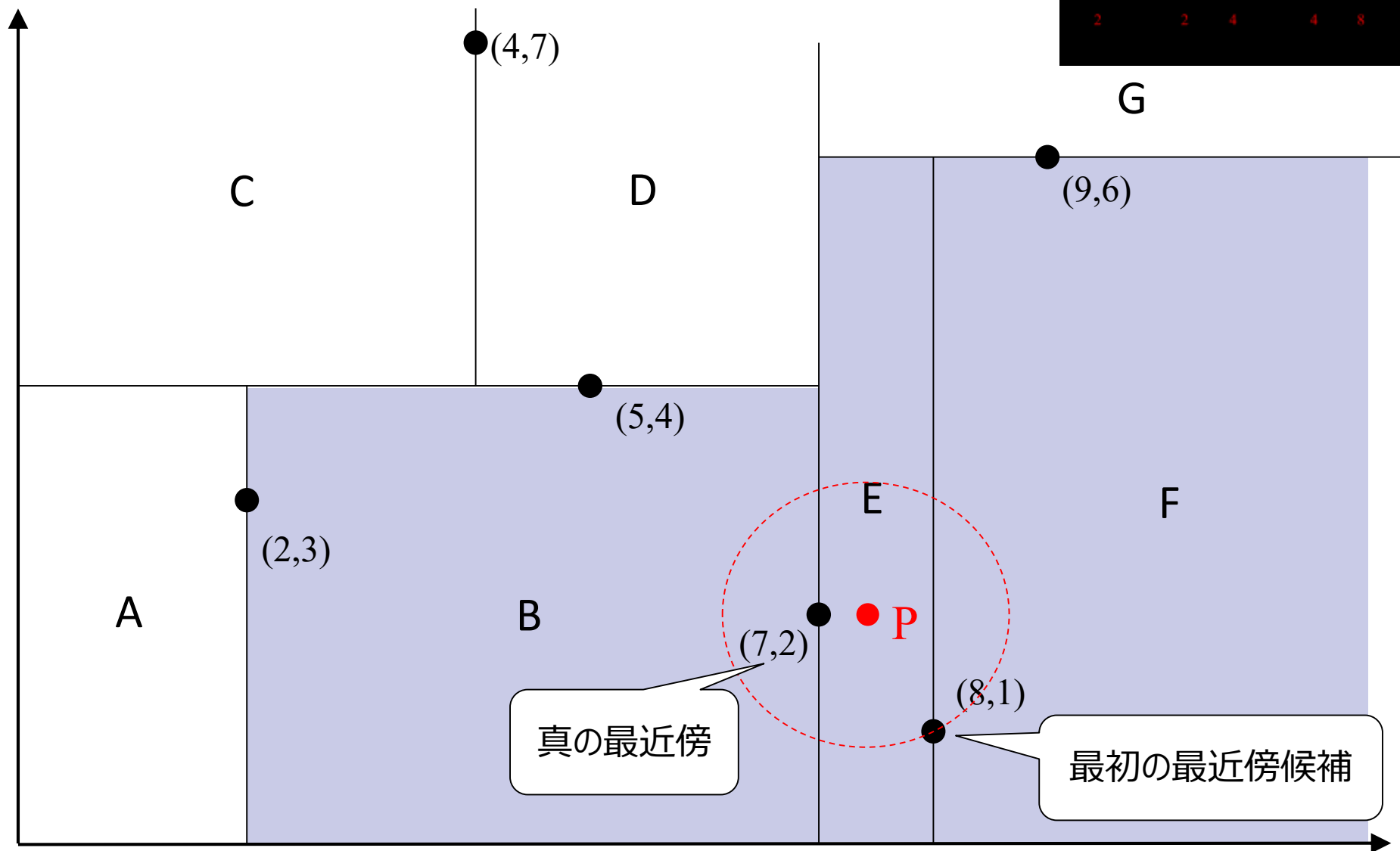
k -d木による近傍点の探索： 枝刈りによって効率的に探索を行う



k -d木による最近傍点の探索： 近傍点探索を少し変更

- P に近い点が見つかる度に、暫定的な最近傍点 P' と距離 r を更新
 - 初めは P を含む領域を見つけ最も葉に近い分割点を初期 P' とする
 - P から半径 r 以内により近い点がなければ P' が真の最近傍点
- もう少し効率のよい方法：深さ優先探索
 - 現在の領域から木を上に向かっていきながら分岐点をチェック
 - 分岐点の反対側の領域が、 P から半径 r 以内の領域と被っていれば、他方の分岐先も調べる
 - より近い点を見つけたら、 P' と r を更新
 - 根において、探索すべき方向がなくなったら終了

k -d木による最近傍点の探索： 枝刈りによって効率的に探索を行う



高次元の探索：

k -d木は高次元で効率が悪いので次元削減が必要

- k -d木が有効なのは数次元程度

- データ数 $n \gg 2^k$ が望ましい

- 高次元の場合に、探索効率が悪くなる（多くの点を調べることになる）

- 次元削減によって次元を落としてから k -d木を適用する

- ランダム射影

- 主成分分析

- ...

グラフ

グラフ：

頂点を辺でつないだもの

- グラフ $G = (V, E)$: 頂点を辺 (= 枝) でつないだもの
 - V : 頂点集合 (有限集合)
 - E : 辺の集合 (V 上の2項関係 ; $E \subseteq V \times V$)
直積集合
- $e = (u, v) \in E$ に向きがあるかどうかで有向グラフ、無向グラフにわけられる
 - u, v は「隣接する」という

グラフ関連の用語定義：

部分グラフ、パス

- 部分グラフ：2つのグラフ $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ が $V_1 \subseteq V_2$ かつ $E_1 \subseteq E_2$ のとき、 G_1 は G_2 の部分グラフであるという
- パス（道）：
 - 点の系列 v_1, v_2, \dots, v_k で $(v_1, v_2) \in E$ （長さ $k - 1$ ）
 - v_1, v_2, \dots, v_k がすべて異なるときパスは単純であるという
 - v_1, v_2, \dots, v_{k-1} がすべて異なり $v_1 = v_k$ のとき閉路という
 - 有向グラフのとき：パスと有向パス（向きが揃っている）

グラフ上の探索

グラフ上の探索： グラフの頂点列挙問題

- G 上のある頂点 v_0 から開始して、 G 上を巡回してすべての頂点を列挙することを考える
 - 仮定：既に訪れた頂点に隣接する頂点を認識できる
(挙げたことにできる)
- 基本方針：これまでに挙げた頂点に隣接する頂点のうち、まだ訪問していないものひとつを選んで移動...を繰り返す

グラフ上の頂点列挙の方針： 列挙済み／未訪問の頂点の管理が肝

■考えるべきこと：

A ：すでに列挙した頂点集合の管理

B ：これから訪問すべき頂点集合の管理

1. v_0 を A と B にいれる
2. B から頂点をひとつ (v) とりだす
3. v に隣接する頂点で A に入っていないものがあれば、それらを全て A と B の両方にいれる
4. B が空なら、 A がすべての頂点集合。そうでなければ2へ。

列挙済み頂点の管理：
ハッシュを使えば効率的に

■ 考えるべきこと：

A ：すでに列挙した頂点集合の管理

B ：これから訪問すべき頂点集合の管理

- ある頂点を既に列挙したかどうかを効率よくチェックする：
 v に隣接する頂点集合 $N(v)$ のそれぞれが A に含まれているか？
- 素朴にやると： $O(|A||N(v)|)$
- ハッシュを使って： $O(|N(v)|)$

これから訪問すべき頂点の管理： キューやスタックで管理

- 考えるべきこと：

A：すでに列挙した頂点集合の管理

***B*：これから訪問すべき頂点集合の管理**

- 2通りの実現法：実現方法によって訪問順が変わる
 - － キュー：幅優先探索
 - － スタック：深さ優先探索

キューとスタック： FIFOのデータ構造

- キュー：First-in-first-out (FIFO)のデータ構造
 - 2つのポインタ：headとtail
 - 追加：tail位置に追加して、tail+1
 - 取り出し：head位置を読み出して、head+1
- スタック：Last-in-first-out (LIFO)のデータ構造
 - ポインタ：top
 - 追加：top位置に追加して、top+1
 - 取り出し：top位置を取り出し、top-1

グラフ上の最短経路問題：

始点から終点へのコスト最小のパスを見つける

■ グラフ $G = (V, E)$ において：

— 各辺 $e \in E$ に、非負の実数コスト $l(e)$ が与えられている

— 特別な頂点である始点 v_s と終点 v_t がある

■ 始点 v_s から終点 v_t へ至るパスのうち、パス上の枝のコストの和が最小になるようなパスを見つけたい

レポート課題： ソートの確認

- 講義内で紹介した3種類のソートアルゴリズムがどのように動くかを、自分で作った例を使って説明せよ。それぞれの比較回数を調べよ。
- 締切：6/4
- 提出：PDF形式でPandAで提出
- 提出物に氏名・学籍番号・メールアドレスを含めること