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
  - 正しく動く： 想定したように動く
  - 速く動く：プログラムは速いほど良い！
  - 省資源：メモリや電気代
  - 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：
  - プログラムの良し悪しを測りたい
  - 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

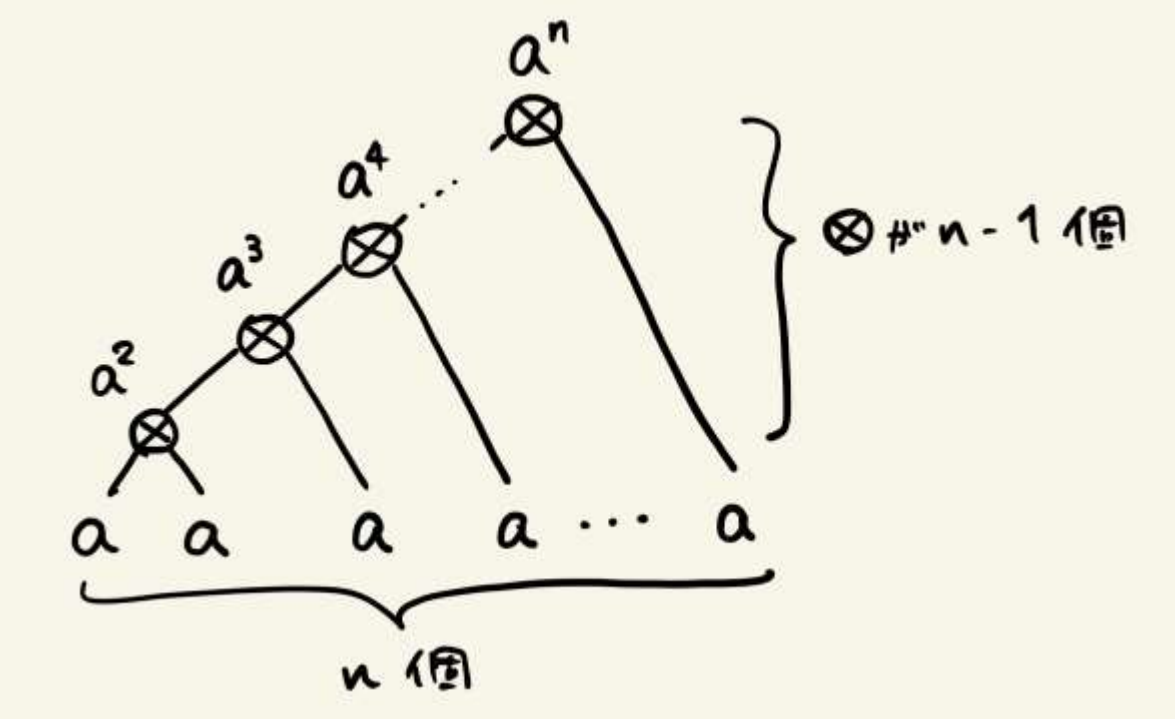
- 多くのプログラムは「データ」を扱う
  - データは繰り返し使用するもの
  - 使用の仕方が予め決められているわけではない
- アルゴリズムがうまく動くためには、データをどのようにもっておくか（＝データ構造）が重要
  - 名前を、入力順に格納？ アイウエオ順？
- データ構造はアルゴリズムと切り離せないもの
  - お互いの良さに影響を与え合う

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？

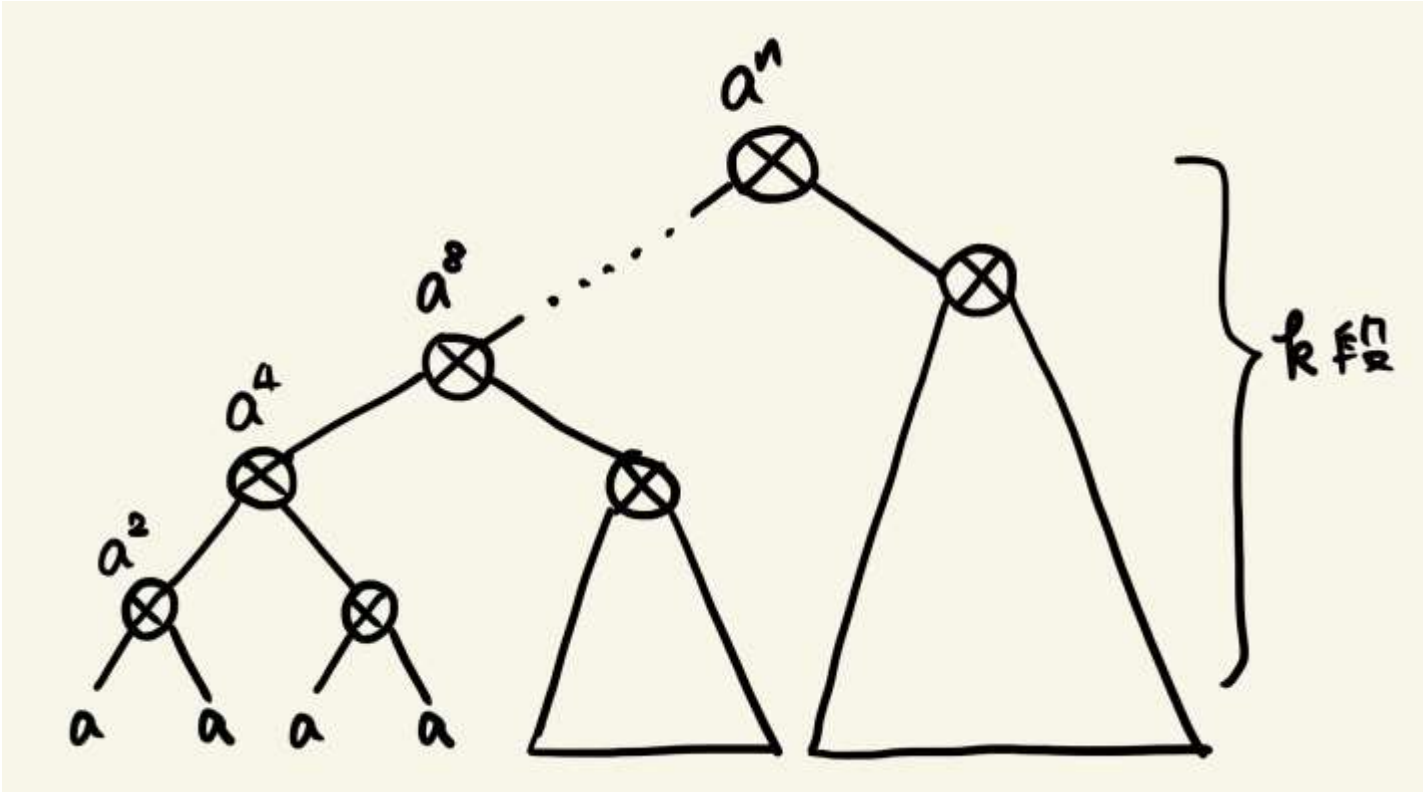
指数演算のアルゴリズム①:
単純な掛け算の繰り返しだと線形時間が必要

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



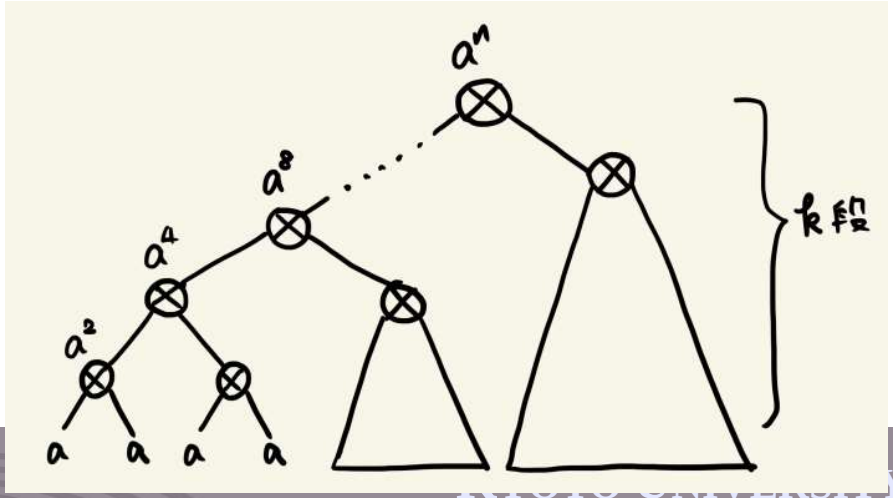
指数演算のアルゴリズム②:
ちょっと工夫すると対数時間で計算可能

- $k = \log_2 n$ 回の掛け算でできる
  - 仮定： $n = 2^k$



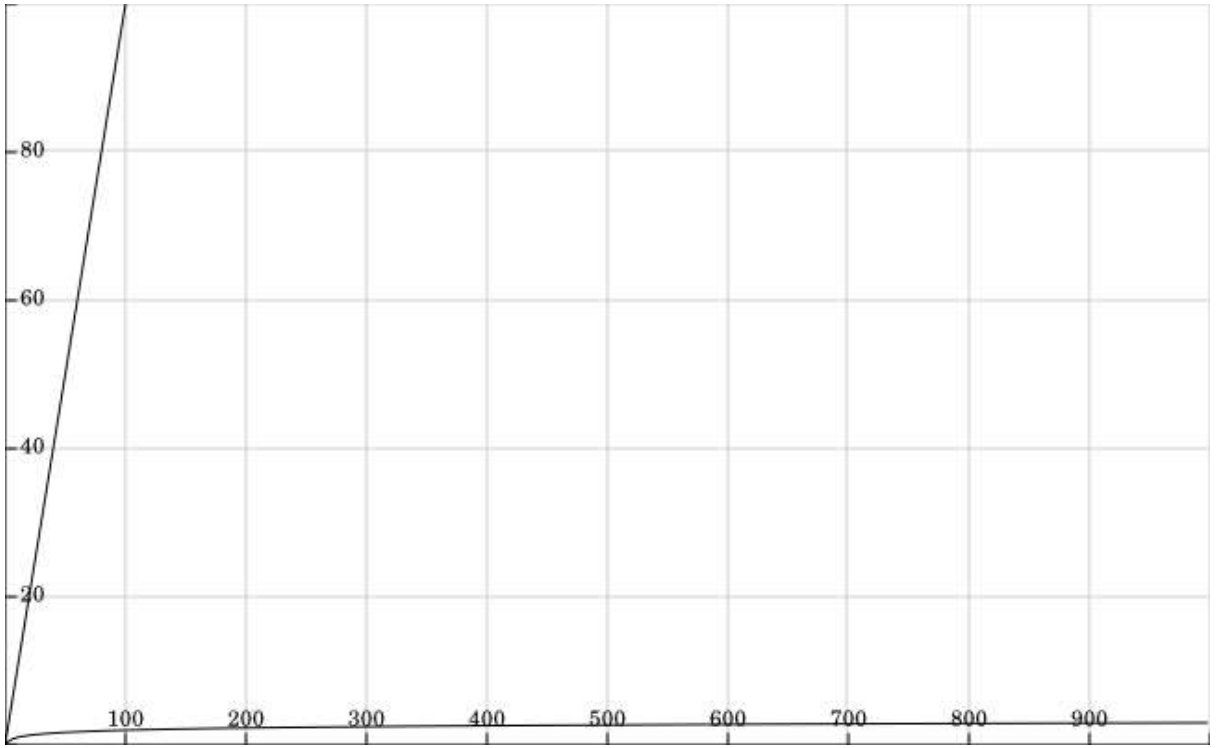
指数演算のアルゴリズム②:
ちょっと工夫すると対数時間で計算可能

- 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てうまい構造（＝データ構造）を作ることで、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回分の計算時間) × S 回

  ② (データ構造の構築にかかる計算時間) +
  (データ構造を利用した1回分の計算時間) × 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 $\{(n_i, p_i)\}_{i=1,\ldots,n}$ 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えるシステムを考える
  - S 人分の問い合わせ $n_{k_1}, n_{k_2}, \ldots, n_{k_S}$ 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

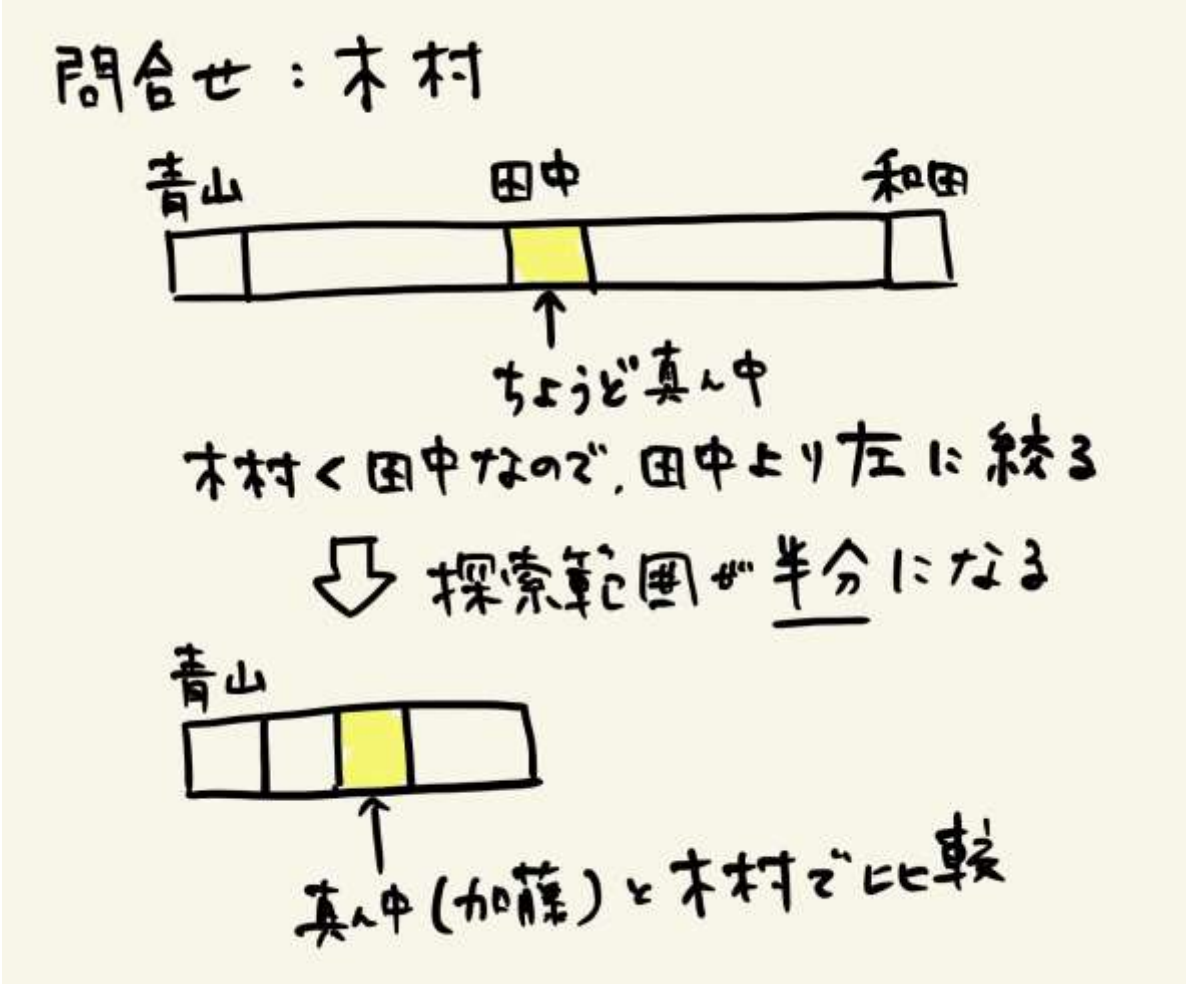
- 名簿の並び順が登録順（でたらめ）の場合を考える
- アルゴリズムとしては、前から順に探していく
- この場合、各問い合わせで、最悪n回のチェックが必要
  - 名簿上の位置（ページ）を指定してチェックすることは単位時間でできるものとする
- 合計 約 nS 回のチェックが（最悪ケースで）必要となる

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

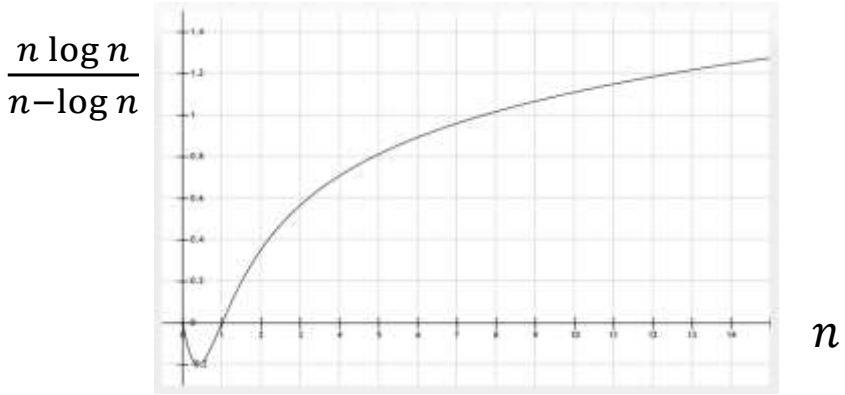
ちょっと頭を使ったアルゴリズム：辞書順に並んでいる場合は$S \log n$回のチェックで可能

- 計$S \log n$回のチェックで発見可能



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 nS 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



アルゴリズムの評価基準

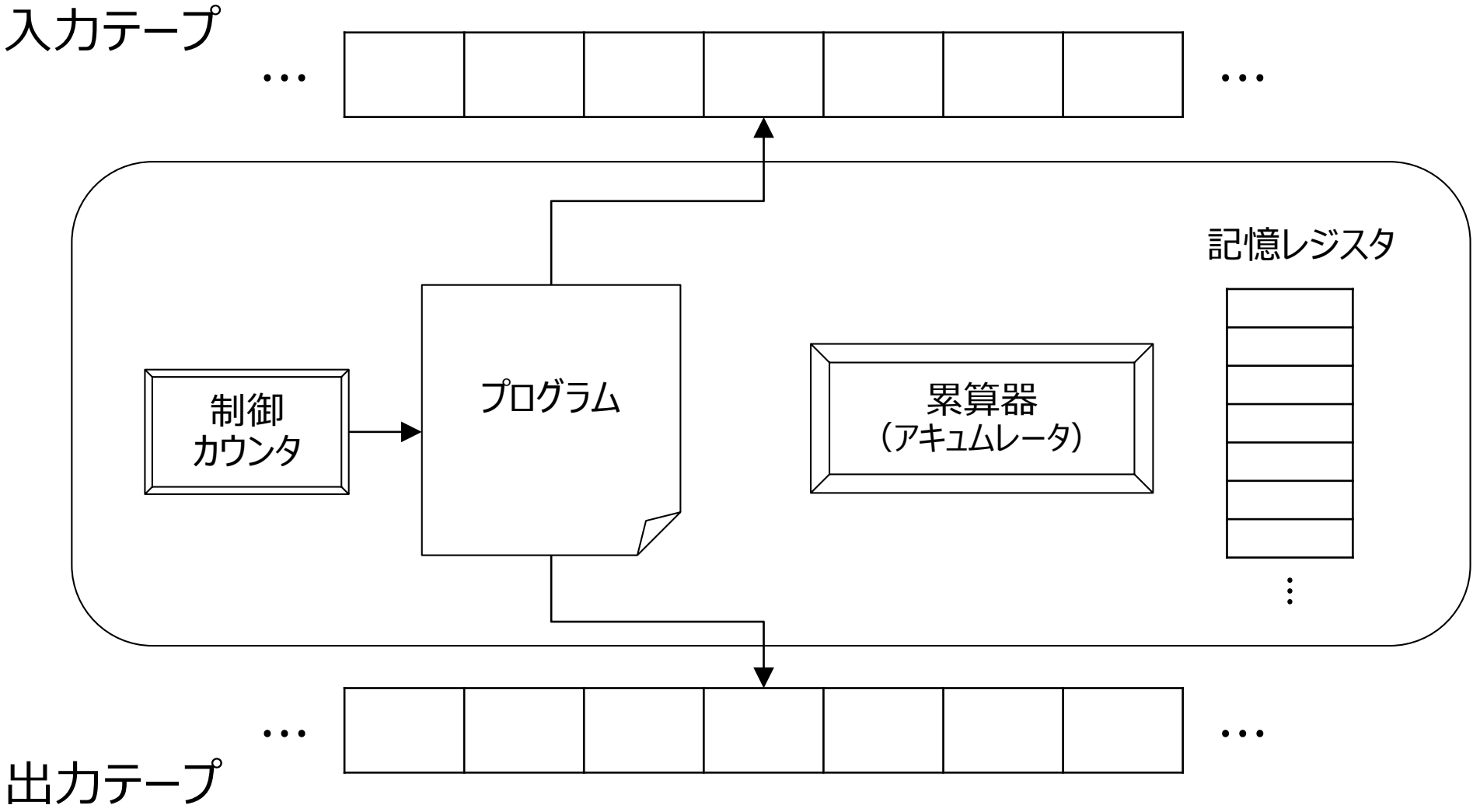
計算機の理論モデル:
計算の効率性を評価するための抽象的な計算機

- 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

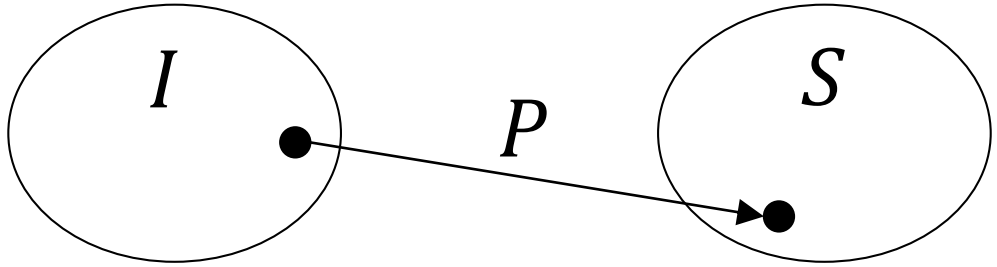
- ランダムアクセス機械：
  - 入力テープ、出力テープ、制御カウンタ、プログラム、累算器（アキュムレータ）、記憶レジスタからなる
  - 入・出力テープは無限の長さがあると仮定
  - 累算器、レジスタには任意の桁数のデータが入り、単位時間でアクセス可能と仮定
- 命令（単位時間で実行できると仮定）
  - 読み取り／書き込み（入力・出力テープを１つ進める）
  - 四則演算、ジャンプ、条件分岐、停止

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



「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$P: I \to S$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(?)

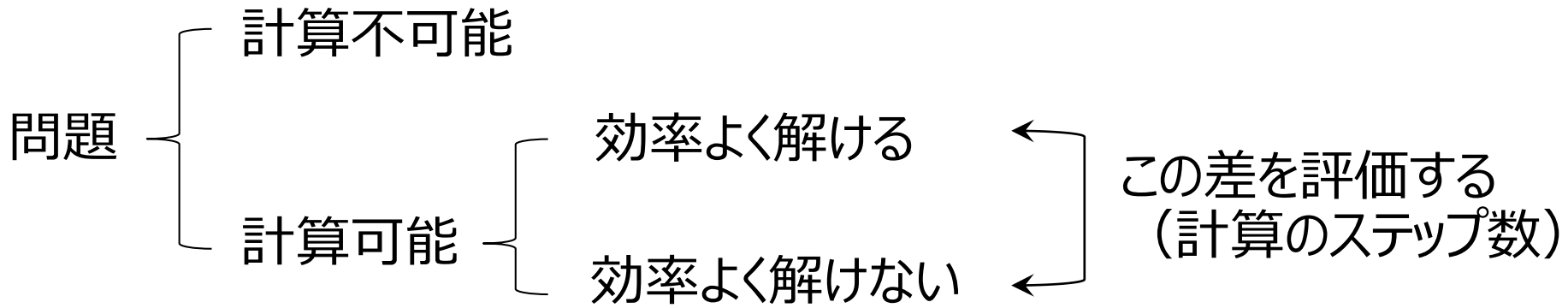
計算可能性:
「プログラムの停止問題」は計算不可能な問題

- 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 （⇔ 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$a_1, a_2, \ldots, a_n$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

    ※ $\{a_i\}_{i=1,\ldots,n}$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

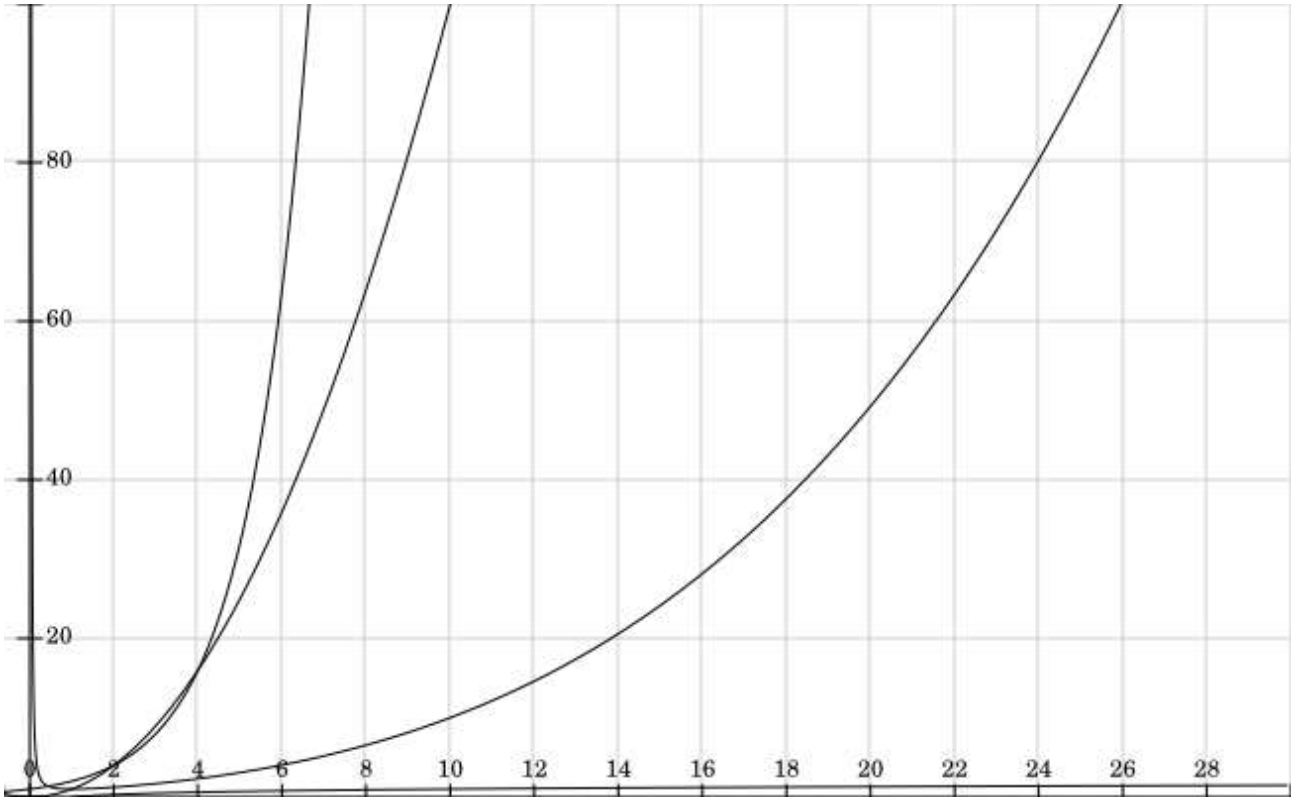
- 関数の上界：$T(n) = O(f(n))$
  - ある正整数n_0とcが存在して、任意の$n \geq n_0$に対し$T(n) \leq c\ f(n)$が成立すること
    - 例： $4n + 4 \leq 5n\ (n \geq 4)$: $c = 5, n_0 = 4$とする
- 関数の下界：$T(n) = \Omega(f(n))$
  - $T(n) \geq c\ f(n)$ （厳密には「あるcが存在し無限個のnに対し」
- 上界と下界の一致：$T(n) = \Theta(f(n))$
  - c_1とc_2が存在して、$c_1\ f(n) \leq T(n) \leq c_2\ f(n)$
  - 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「係数を無視して一番大きいものをとる」

- 直観的には係数を無視して一番大きいものをとればよい
  - $T(n) = 3n^2 + n^{\log n} + 5n \log n + 2^n$ ならば $O(2^n)$



例：多項式計算

- 問題：多項式の評価
  - 入力：$a_0, a_1, \ldots, a_n, x$
  - 出力：$a_n x^n + \cdots + a_1 x + a_0$
- 方法：
  - 単純な方法：
    $a_k x^k = a_k \times x \times \cdots \times x$ ($k+1$回の掛け算)
    - $O(\sum_{k=1}^{n} k + n) = O(n^2)$
  - Horner法：
    $(((\cdots(a_n x + a_{n-1})x - a_{n-2})x - a_{n-3})\cdots)x + a_0$
    - $O(n)$

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：

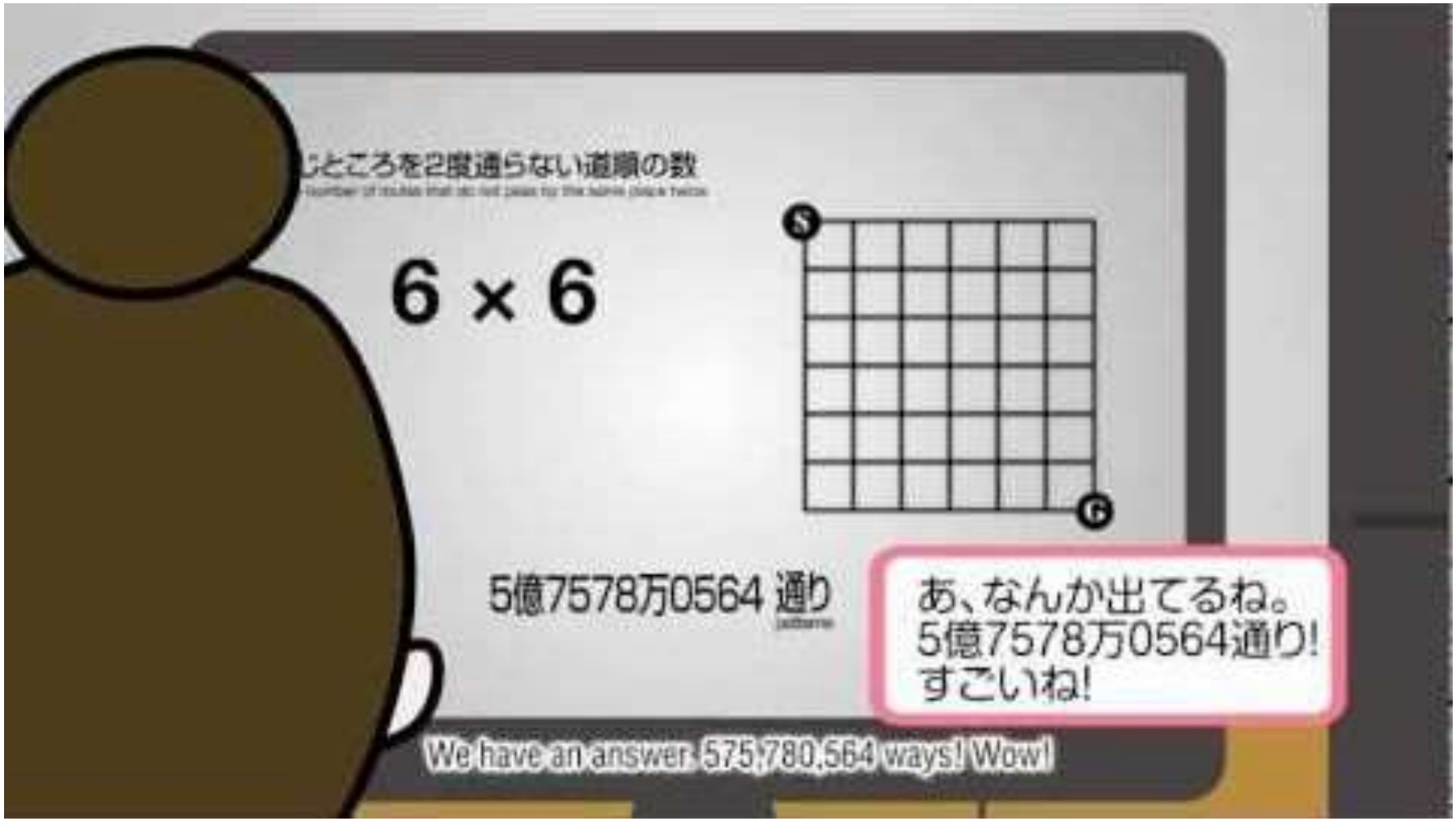
$O(\ )$	$\boldsymbol{n = 10}$	$n = 30$	$n = 60$
n	0.00001s	0.00003s	0.00006s
n^2	0.0001s	0.0009s	0.0036s
n^3	0.001s	0.027s	0.216s
2^n	0.001s	1074s	10^{12}s
3^n	0.06s	2×10^8s	4×10^{22}s

- 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



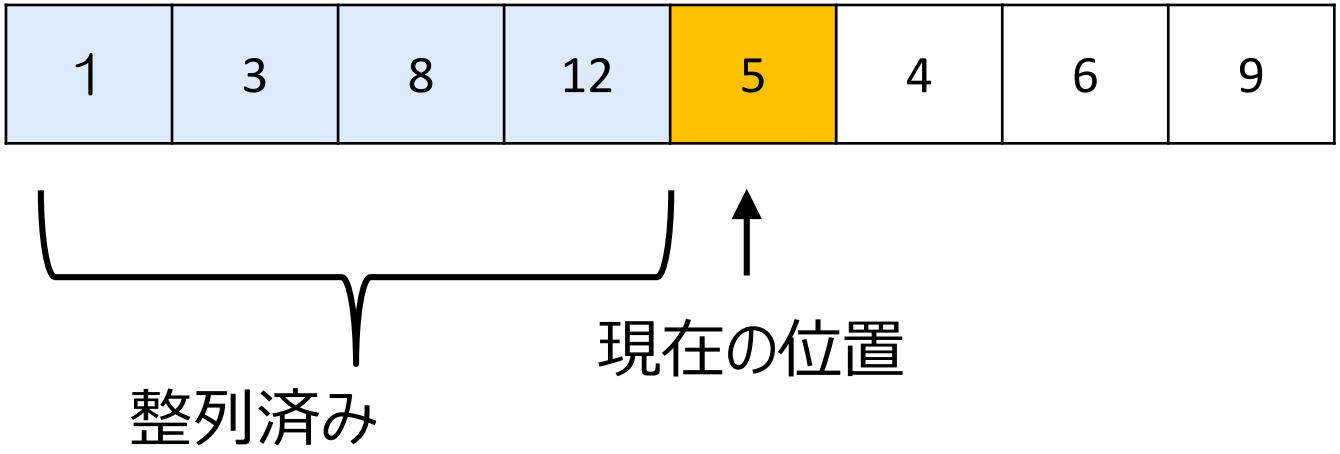
整列（ソート）のアルゴリズム

整列問題（ソート）：要素を小さい順に並び替える問題

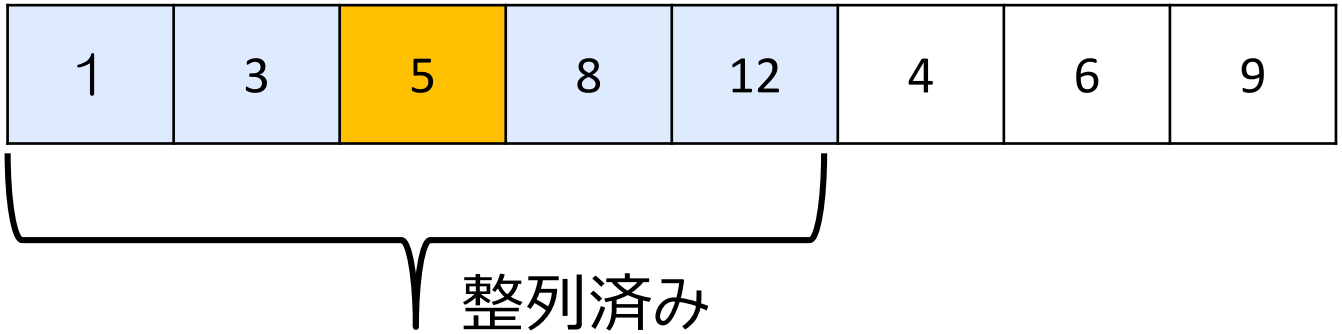
- 整列問題
  - 入力：n個の数 $a_1, a_2, \ldots, a_n$ が入った配列
  - 出力： $a_{1'} \leqq a_{2'} \leqq \ldots \leqq a_{n'}$ を満たす、入力列の置換
- 例：入力 $(4, 5, 2, 1) \rightarrow$ 出力 $(1, 2, 4, 5)$

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 $j = 1,2,..,n$ まで行うと
  $\Sigma_{j=1,...,n} O(j) = O(n^2)$ 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

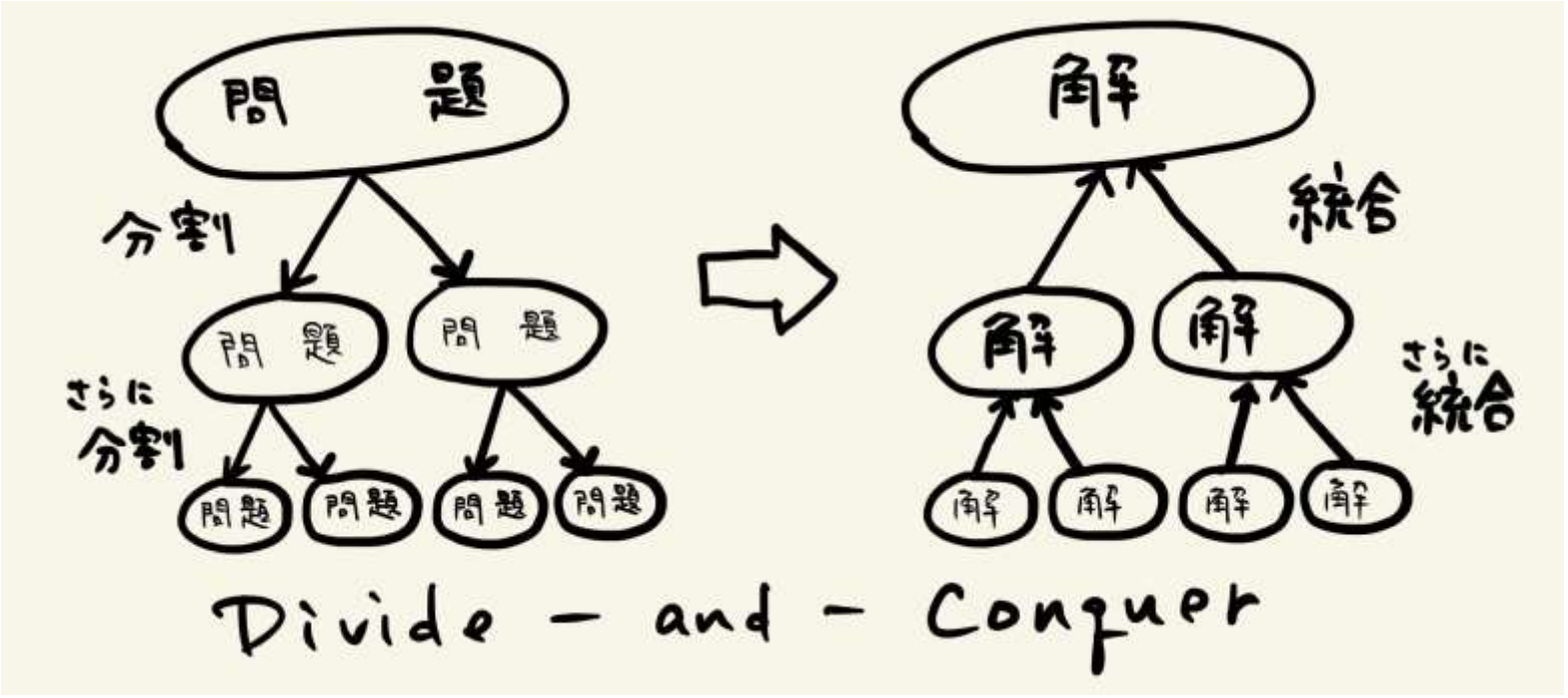
分割統治法

分割統治法：
ｱﾙｺﾞﾘｽﾞﾑ設計指針の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：
ｱﾙｺﾞﾘｽﾞﾑ設計指針の1つで、問題を小問題に分割して解く

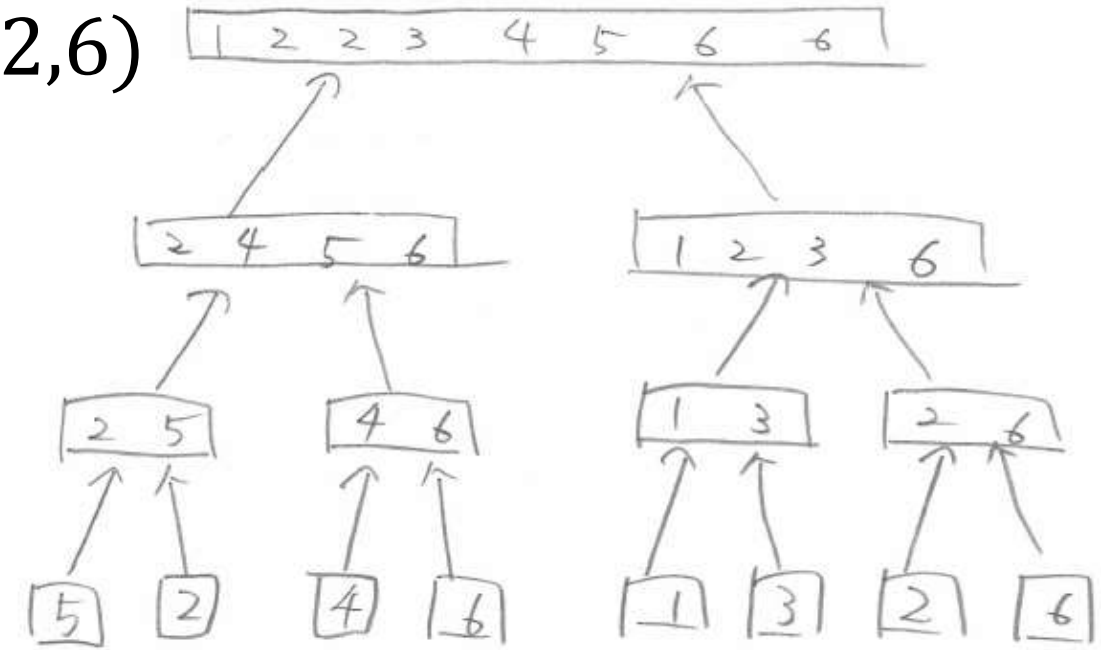
- 分割統治法：
  - 元の問題を、同じ構造をもった小さな問題に分割
  - 小さな問題の解を統合して元の問題の解を得る



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)

入力 (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$O(n\log n)$
  - 実用的には次に紹介するクイックソートが速い
- 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：

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

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

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

$$= c2^k + \underbrace{cn + \cdots + cn}_{k} < n\log n$$

分類定理（簡易版）：計算量の再帰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

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

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

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 $\mathrm{QuickSort}(A, p, r)$

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. $q \leftarrow \mathrm{Partition}(A, p, r)$ ：分割点qをみつけて分割
2. $\mathrm{QuickSort}(A, p, q)$
3. $\mathrm{QuickSort}(A, q+1, r)$

} 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 $\mathrm{Partition}(A, p, r)$：分割点$q$をみつけて分割

- $A[p]$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

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

- $q \leftarrow \mathrm{Partition}(A, p, r)$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

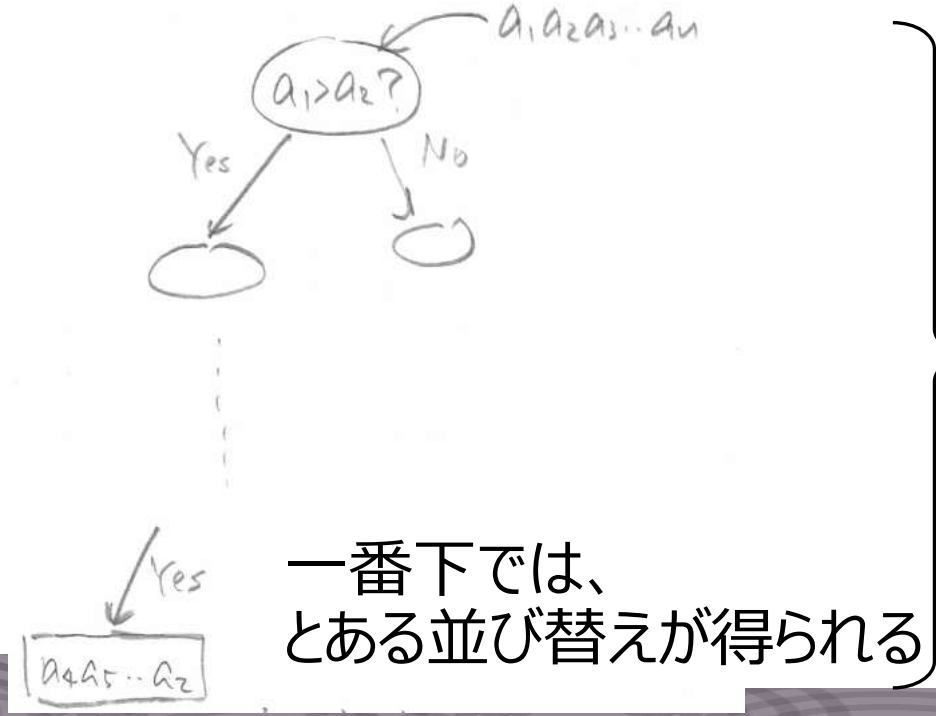
最悪$O(n)$の順序統計量アルゴリズム：うまく「だいたい真ん中」をとってくる

- $\mathrm{Order}(A, k)$：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をみつける $\mathrm{Order}(A, \lfloor n/10 \rfloor)$
  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$\mathrm{Order}(S_1, k)$、$|S_1| \leq k \leq |S_1| + |S_2|$ならば$m$は目的の要素、$k > |S_1| + |S_2|$ならば$\mathrm{Order}\left(S_3, k - (|S_1| + |S_2|)\right)$

ソートの計算量下界

ソートの計算量の下界：いかなるソートアルゴリズムでも O(n log n) が限界

- がんばってもO(n log n)より良いアルゴリズムは作れない
- n個の要素が全て異なるものとすると、ソート後に得られる列の可能性はn!通り
- ソートは2つの数字の比較を何かの順で繰り返すことで動く



一番下では、
とある並び替えが得られる

この高さがどう頑張っても
O(n log n)になる

ソートの計算量の下界の証明：並び替えの回数は必ず、最悪 O(n log n) 必要

- 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、$2^h \geqq n!$でないといけない
- 対数をとると
  $h \geqq \log n! \geqq \log (n/e)^n = n \log n - n \log e = O(n \log n)$
  - なお、Stirlingの公式 $n! \geqq \sqrt{(2\pi n)}\,(n/e)^n \geqq (n/e)^n$

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

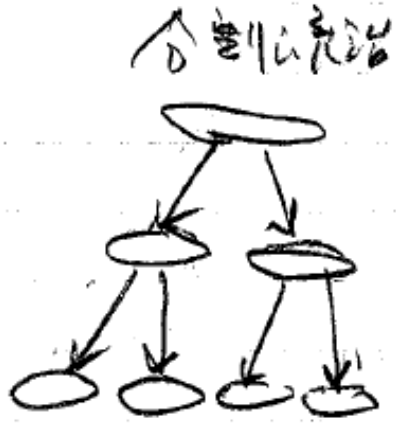
- 動的計画法は分割統治法と同じく、問題を再帰的に分割する方法、ただし：
  - 分割統治： トップダウン
  - 動的計画： ボトムアップ
- 動的計画法の大まかなステップ：
  1. 問題の構造を再帰的に捉える
  2. 解を再帰的に構成する
  3. ボトムアップで解を計算する

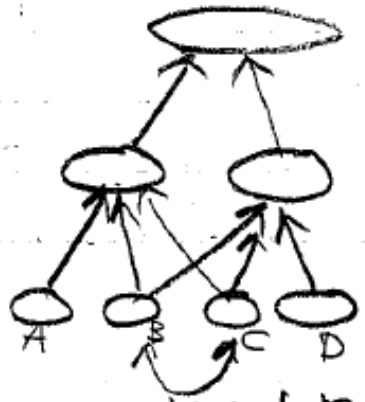
（1. と 2. は「ここは分割統治と同じ」）

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うる
- 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

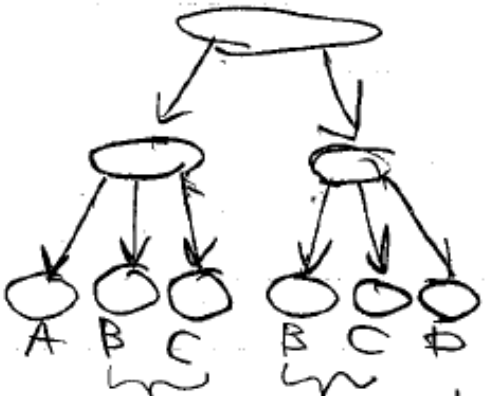




ボトムアップな感じで
BCCを使いまわせる

部分問題を共有してよい

これを分割統治でやると



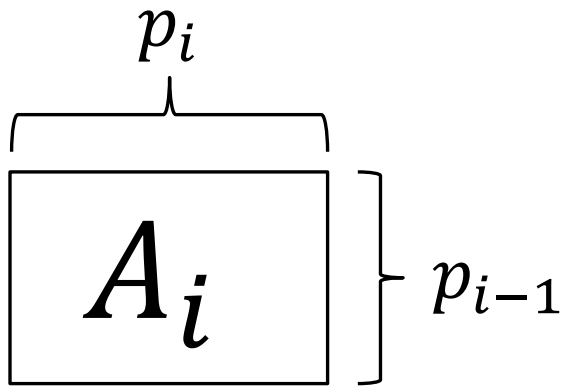
同じ問題を何度も再帰的に

逆にいえば

この部分問題の重複生成がDPのカギ

動的計画法の例：複数の行列積の計算

- 入力： 行列 $A_1, A_2, \ldots, A_n$
- 出力： 積 $A_1 A_2 \cdots A_n (= A_{1,\ldots,n})$
- 掛け算できる前提
  - $A_i A_{i+1}$の計算は$O(p_{i-1}\, p_i\, p_{i+1})$かかる
- $A_1 : 10 \times 100, A_2 : 100 \times 5, A_1 : 5 \times 50$ とすると
  $\big((A_1 A_2) A_3\big) = 7500, \big(A_1 (A_2 A_3)\big) = 75000$

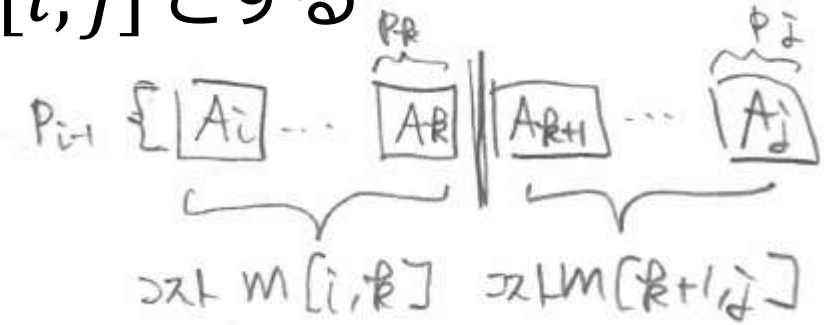


再帰式の構成：観察

- $A_1 A_2 \cdots A_n (= A_{1,\ldots,n})$を$k$番目で分割するとする
- $A_{1,\ldots,k}$と$A_{k+1,\ldots,n}$を別々に計算して最後に統合
  $O(p_0\ p_k\ p_n)$
- もっとも計算コストの小さい解でk番目の分割が最後にくるとすると、$A_{1,\ldots,k}$と$A_{k+1,\ldots,n}$の分割もコストが最小のはず
  - そうでなければ、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

再帰的な解構成：行列積をより小さな行列積の積として計算

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



- 再帰式：

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

- トップダウン計算（分割統治）だと指数的な計算量

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

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に

- 再帰式の適用により最小のm[1, n]が求まる
- 一方、ボトムアップだと：
  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：実際の掛け算の順番を得るには各再帰式での最良のkを記憶しておく

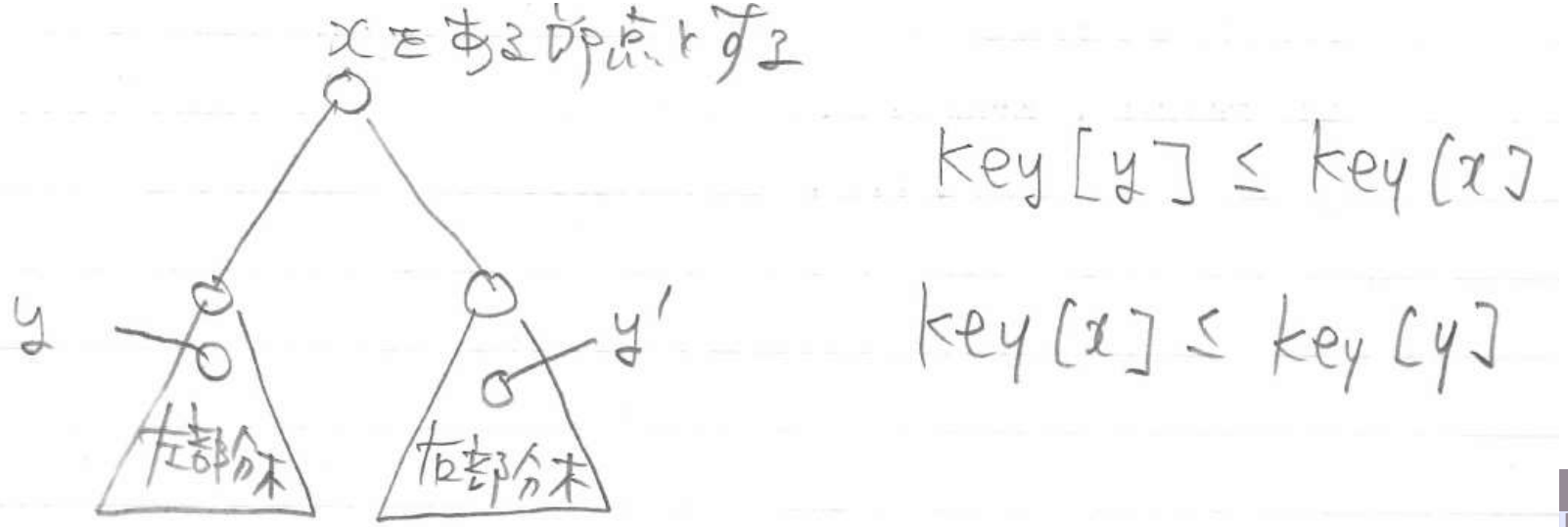
探索

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

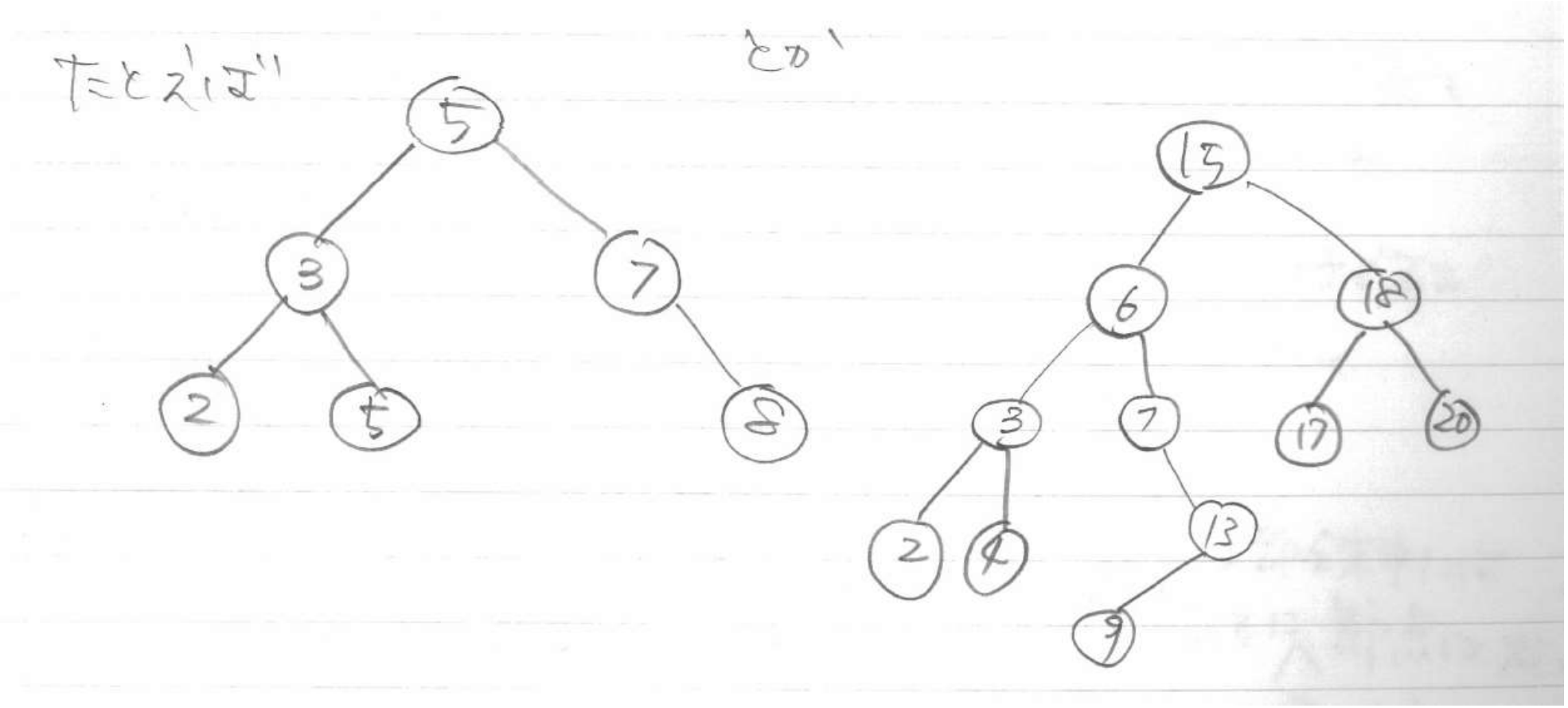
二分探索木:
データ検索のための樹状データ構造

- 各節点が key（キー）、left（左の子）、right（右の子）、p（親）とデータをもつ
- キーには順序が付けられる
  - ２つの節点 x, y に対して key[x] = key[y], key[x] < key[y], key[x] > key[y] のいずれかが成立
- 以下の条件を満たす



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

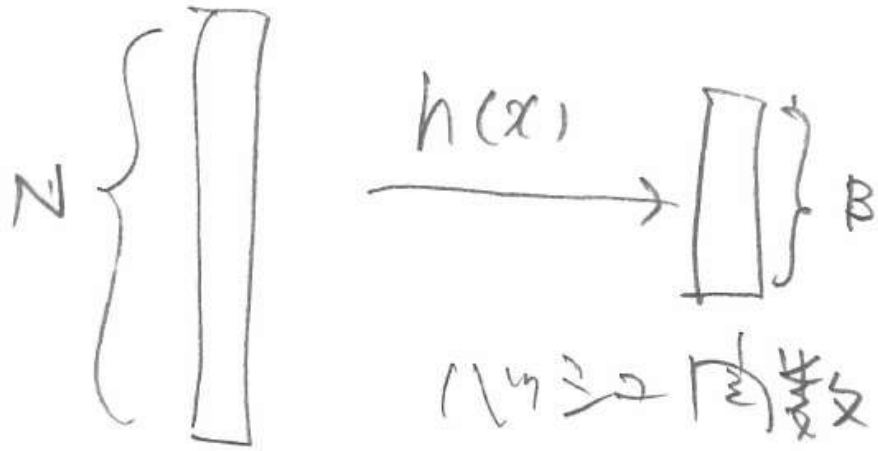


ハッシュ表:
O(1)で探索するためのデータ構造

- 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～N)
    - キーがアルファベット６文字なら $N=26^6 \simeq 3\times10^8$
  - 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とすればM«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,…,N}→ {1,2,…,B}



- key 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、なるべく均等に格納するものがよい
  - 例：$x = a_1a_2\ldots a_6$ （アルファベット6文字）に対するハッシュ関数 $h(x) = \Sigma_{i=1,\ldots,6}\, c(a_i) \bmod B$: 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
  - $x =$ すし, $y =$ 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

- 配列にデータを直接格納せず、リストを格納
  - $M > B$ でもよい
- 計算量：
  - 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
- ハッシュがランダムならば$\ell \approx \frac{M}{B}$, これを定数になるようにすれば$O(1)$

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$h_0, h_1, h_2, \ldots$を用意して、h_iが衝突したら、次のh_{i+1}を調べる
  - 例：$h_i = h(x) + i \bmod B$

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$v_1, v_2, \ldots, v_k$で$(v_1, v_2) \in E$ （長さ$k-1$）
  - $v_1, v_2, \ldots, v_k$がすべて異なるときパスは単純であるという
  - $v_1, v_2, \ldots, v_{k-1}$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： $\mathrm{O}(|A||N(v)|)$
- ハッシュを使って：$\mathrm{O}(|N(v)|)$

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みつける

- 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みつけたい