KYOTO UNIVERSITY

アルゴリズムとデータ構造②
～アルゴリズムの評価～

鹿島久嗣
（計算機科学コース）

DEPARTMENT OF INTELLIGENCE SCIENCE AND TECHNOLOGY

アルゴリズムの評価基準

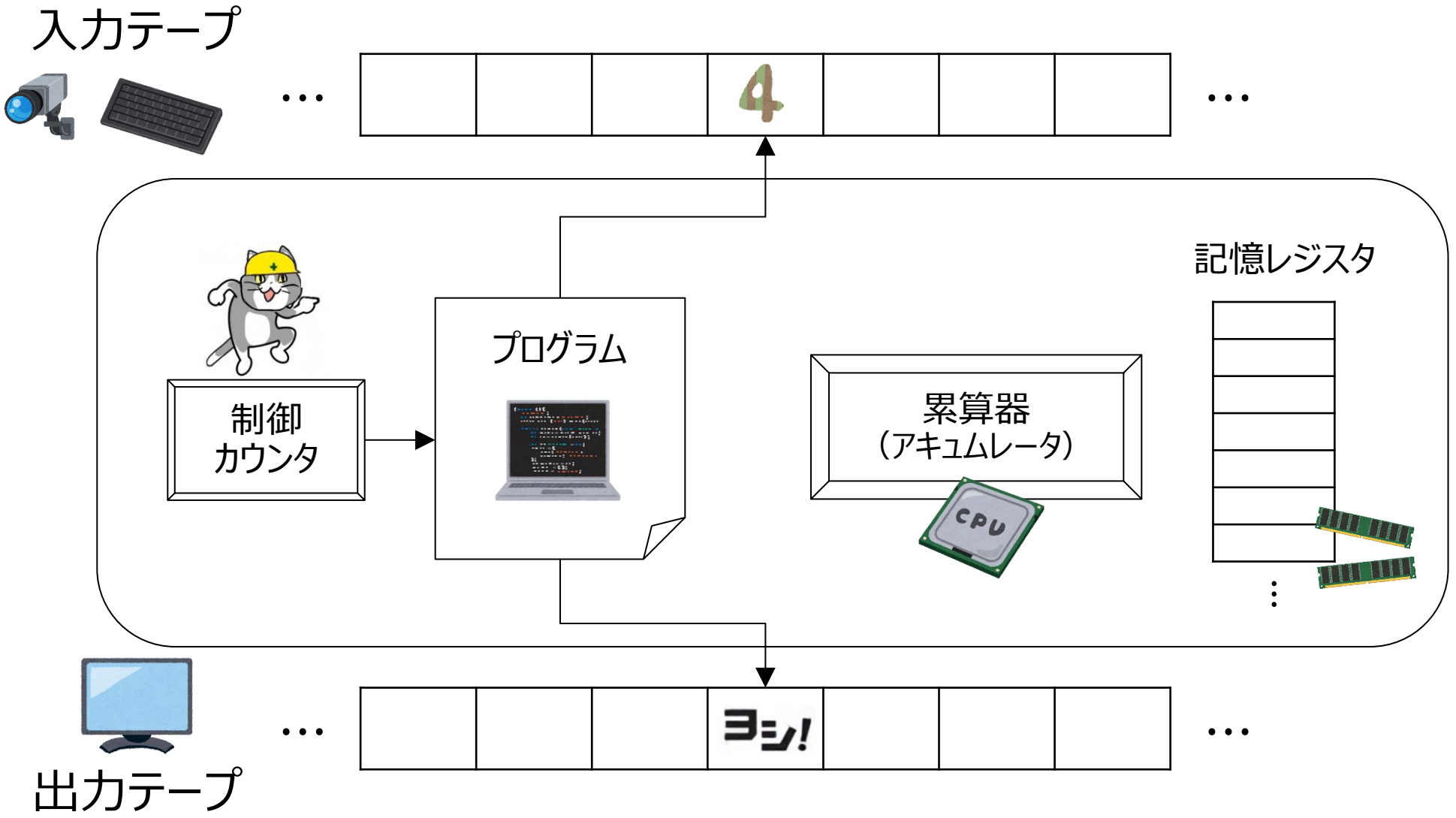
計算機の理論モデル:
計算の効率性を評価するための抽象的な計算機

- 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（＝時代が変わっても有効な議論がしたい）
- 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（RAM）：現在のコンピュータに近い、計算機の理論モデル

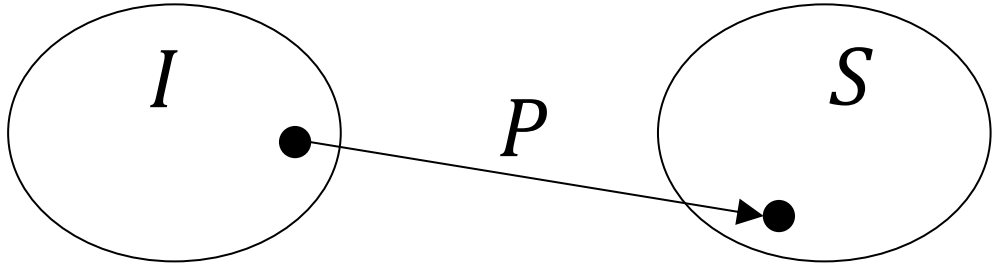
- ランダムアクセス機械：理想化された計算機
  - 入力テープ、出力テープ（モニタ出力）、制御カウンタ、プログラム、累算器（アキュムレータ；一時的な記憶）、記憶レジスタ（記憶装置）をもつ
  - 入・出力テープは無限の長さがあると仮定
  - 累算器、レジスタには任意の桁数のデータが入り、単位時間でアクセス可能と仮定
- 命令：それぞれ単位時間で実行できると仮定
  - 読み取り／書き込み（入力・出力テープを1つ進める）
  - 四則演算、ジャンプ、条件分岐、停止

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$P: I \rightarrow S$



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

「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
    - 注意：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(?)

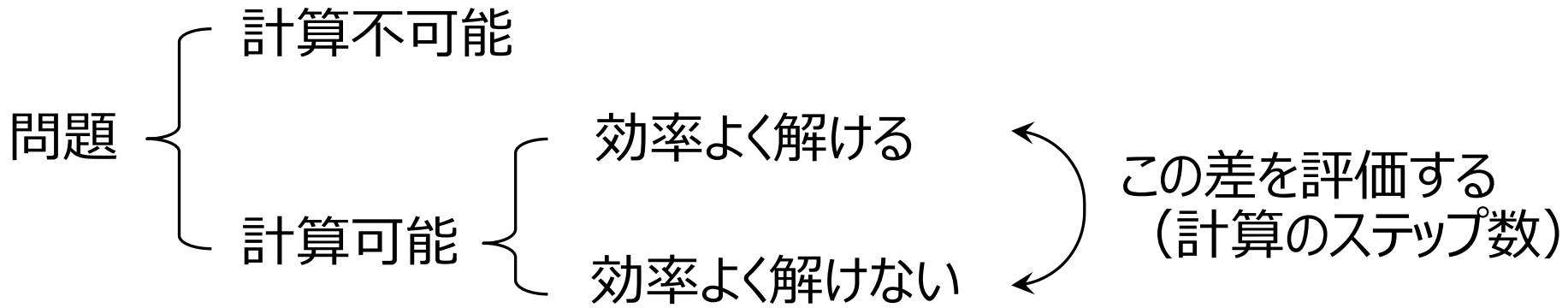
計算可能性:
「プログラムの停止問題」は計算不可能な問題

- 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′を入力して矛盾を示す（$Q'(Q') = (Q', Q')$を考える）
   - もしも(Q', Q')が停止するならば
     $\Rightarrow Q'(Q') = (Q', Q')$は停止しない（矛盾）
   - プログラムQが存在するという仮定がおかしい

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

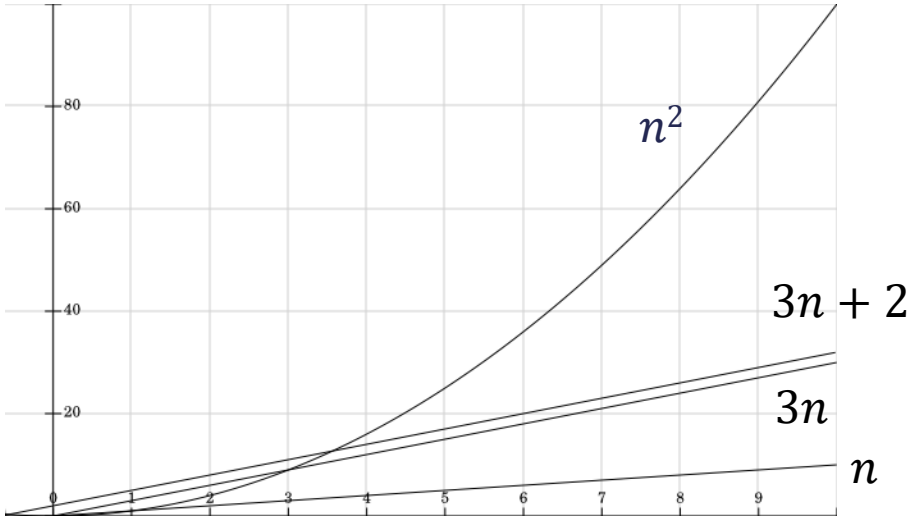
    ※ $\{a_i\}_{i=1,\ldots,n}$の要素がすべて異なり、
    $n+1$通りの場合が等しい確率で起こると仮定する
  - これらは異なるといってよいだろうか？

オーダー評価：アルゴリズムの計算量は最悪ケースのオーダーで評価する

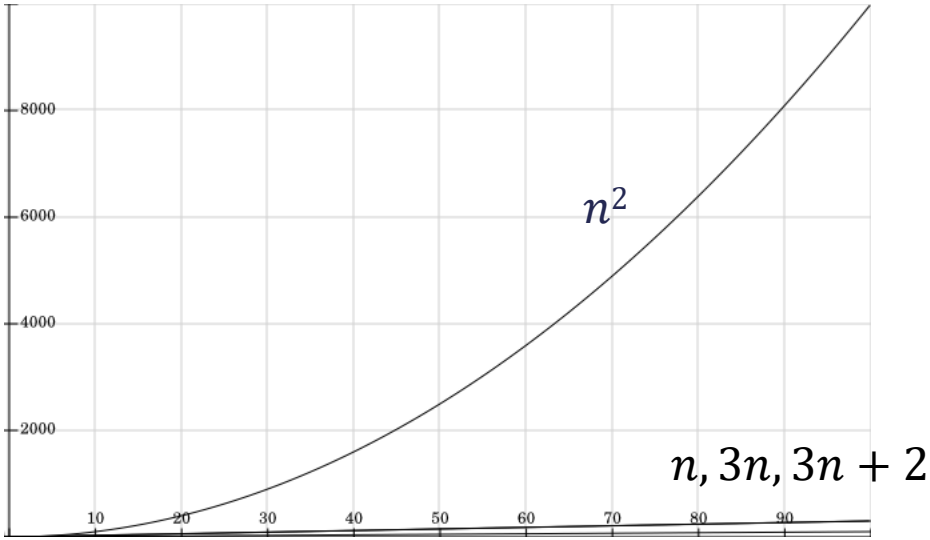
- 以降、特になにも言わない場合は最悪ケースで考える
- （前述の）名簿を前から順に探すアルゴリズムでは、最悪ケースにおいて、n回のチェックが必要
  - もし、n回の比較、n回のカウンタ移動、n回のデータ読み取りと数えるならば、合計で $3n$ 回の操作が必要
  - さらに、出力と停止を加えるなら 計 $3n + 2$ ステップに？
  - 本質的には n が重要 → オーダー記法で $O(n)$ と書く
- オーダー記法： nが大きいときの振舞いを評価する
  - nが大きくなったとき、$n, 3n, 3n + 2$ の違いは、n^2 に比べると極めて小さい

オーダー記法の性質：問題サイズ n の指数部分が支配的になる

- $n, 3n, 3n+2$ と n^2 の比較
- nが大きくなると、n^1とn^2の差しか意味をもたない
- 「遠くから見ると」定数係数にほとんど意味がなくなる



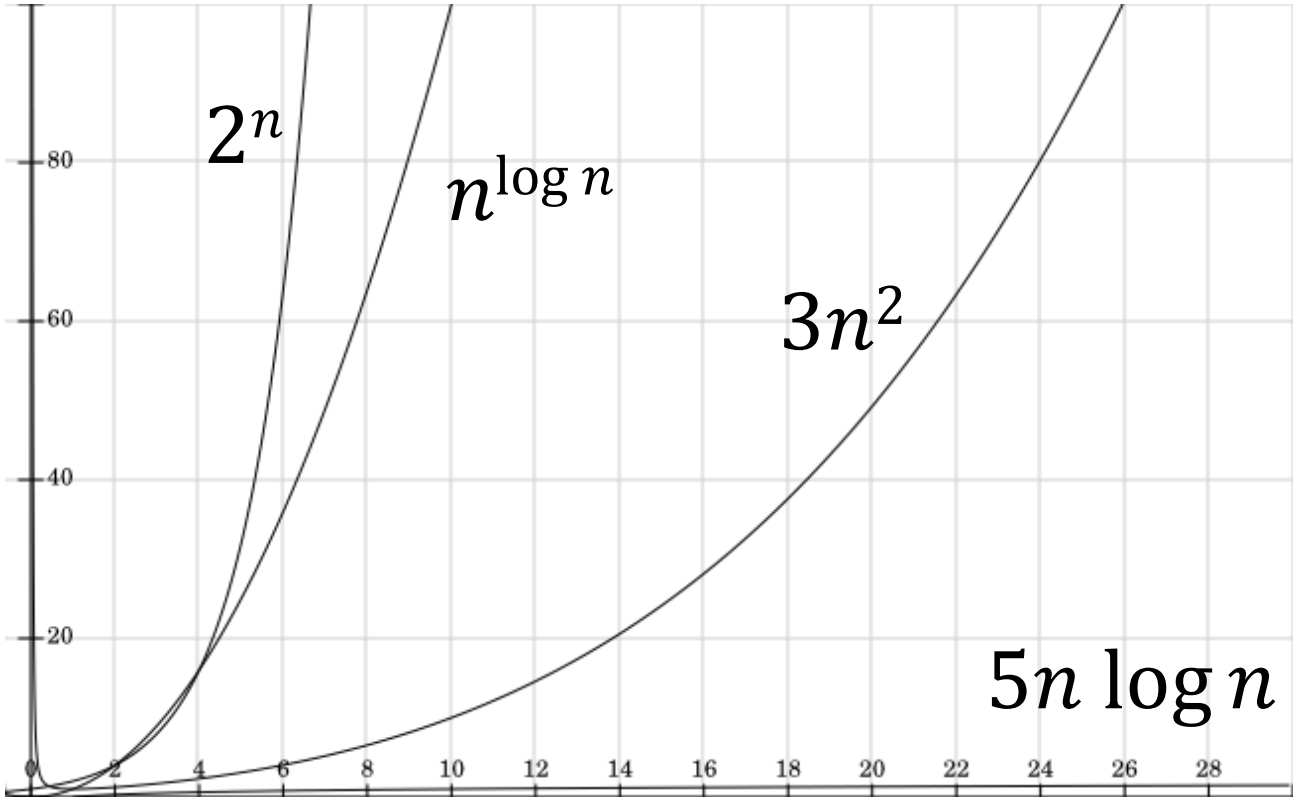
nが小さい場合



nが大きい場合

オーダー記法の評価：ざっくり言えば「係数を無視して一番大きいものをとる」

- 大雑把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 $O(2^n)$



オーダー記法の定義：計算量の上界・下界を見積もる

- 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：$T(n) = 3n^2 + 3n + 10n \log n = \Theta(n^2)$

オーダー記法の性質：２つの性質

1. $f(n) = \mathrm{O}\big(h(n)\big),\ g(n) = \mathrm{O}\big(h(n)\big)$
$\rightarrow f(n) + g(n) = O\big(h(n)\big)$

2. $f(n) = \mathrm{O}\big(g(n)\big),\ g(n) = \mathrm{O}\big(h(n)\big)$
$\rightarrow f(n) = O(h(n))$

オーダー評価の例：多項式の計算（単純法 vs. Horner法）

- 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：
  - 直接的な計算方法：
    $a_k x^k = a_k \times x \times \cdots \times x$ (k回の掛け算)
    - これは $O(\sum_{k=1}^{n} k + n) = O(n^2)$
  - Horner法：
    $(((\cdots(a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})\cdots)x + a_0$
    - これは $O(n)$

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：

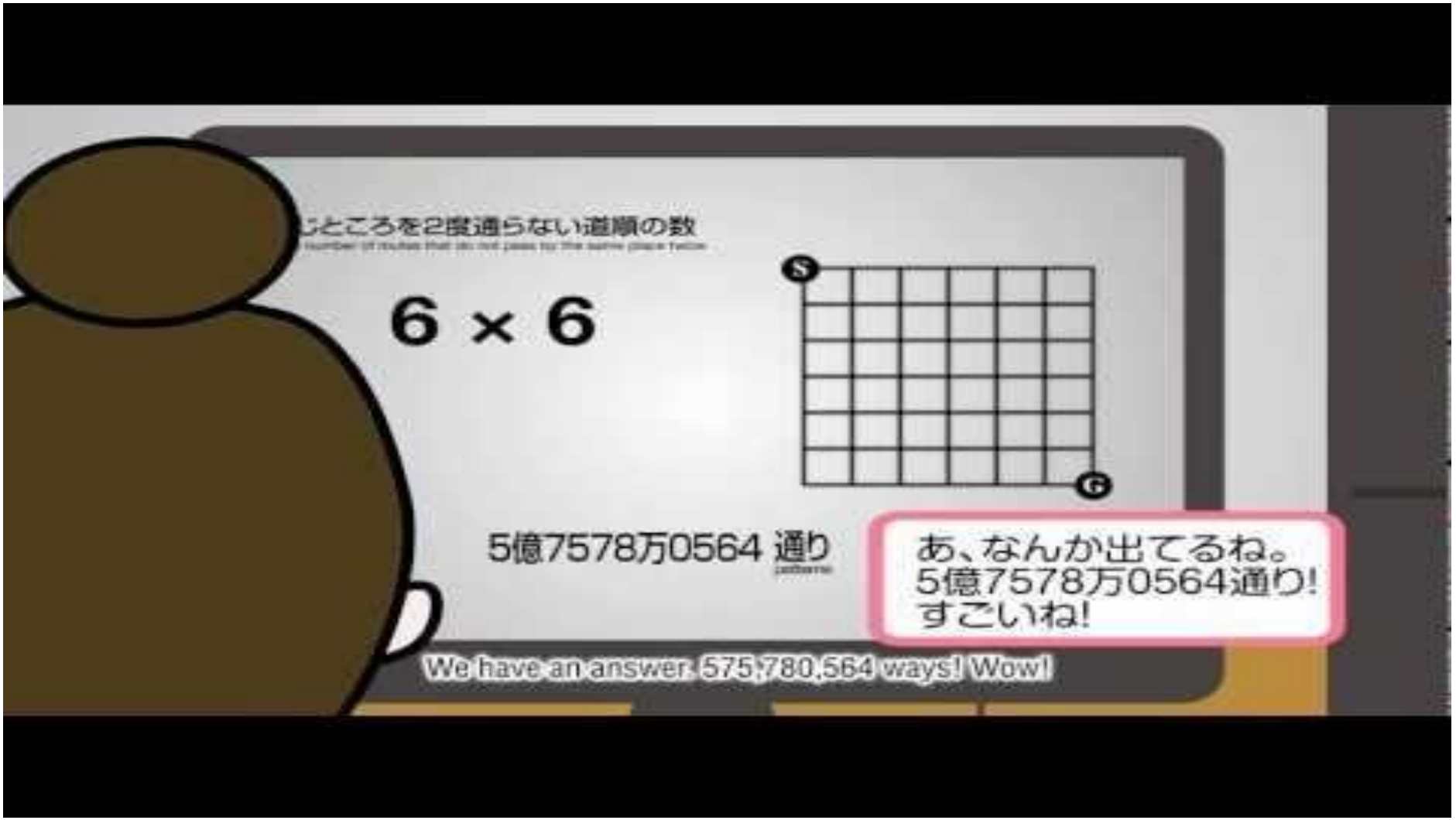
$O(\ )$	$n = 10$	$n = 30$	$n = 60$
n	0.00001s	0.00003s	0.00006s
n^2	0.0001s	0.0009s	0.0036s
n^3	0.001s	0.027s	0.216s
2^n	0.001s	1074s	10^{12}s
3^n	0.06s	2×10^8s	4×10^{22}s

- スパコン「京」は1秒間に8162兆回（100億=10^{10}倍速い）
- 「富岳」は1秒間に44京2010兆回（10兆=10^{12}倍速い）

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
  - 理論的な保証はないが「実用上」有用な様々な戦略によって多くの場合良い解が得られる方法
    - 分枝限定法、局所探索、…
  - 近似アルゴリズムのような、最適解の保証はないが、最適解からどの程度悪いかという保証がある方法

数え上げお姉さん：
単純な解法が破たんする例



リスト

集合を管理するデータ構造：データを保持するための基本データ構造

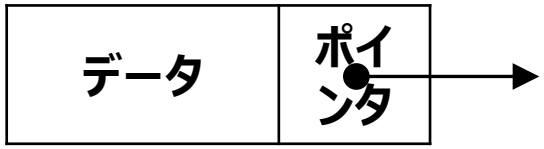
- 集合を管理するデータ構造
  - データをコンピュータのメモリにどのように保持するか
- 提供すべき機能：
  - 集合への要素の追加
  - 集合からの要素の削除
  - 集合内での要素の検索
- たとえば、配列ならば…：
  - メモリを必要分確保しておき、順次保管する
  - 所望の位置にアクセス可能だが、削除が面倒

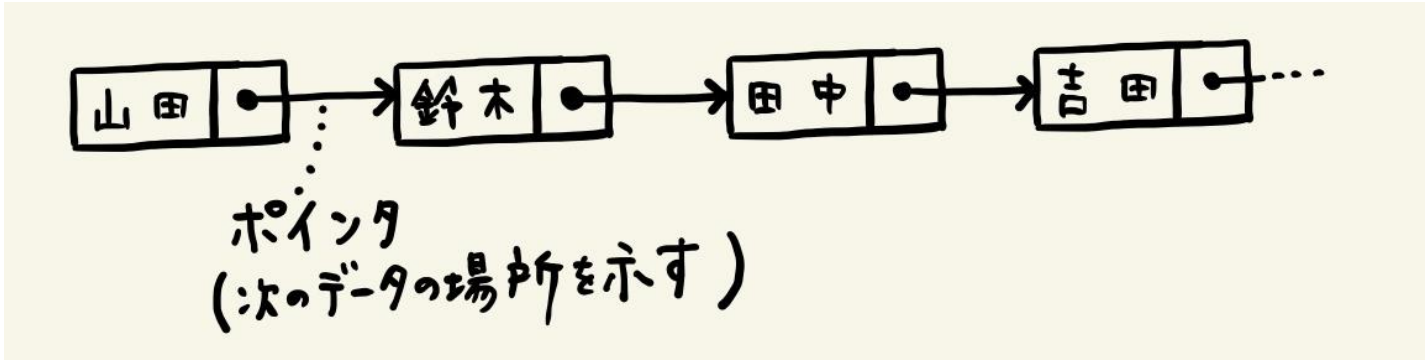
配列による集合の管理

番地	データ
1	山田
2	鈴木
3	田中
4	吉田
⋮	⋮

リスト：
集合を管理する基本データ構造

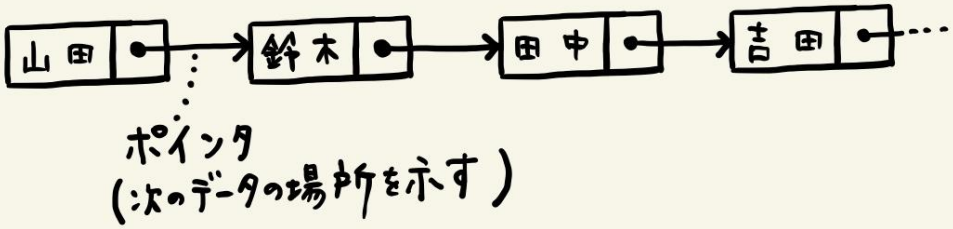
- リスト：データをポインタで一列につなげたもの
  - ポインタ：次のデータの場所（番地）を示す

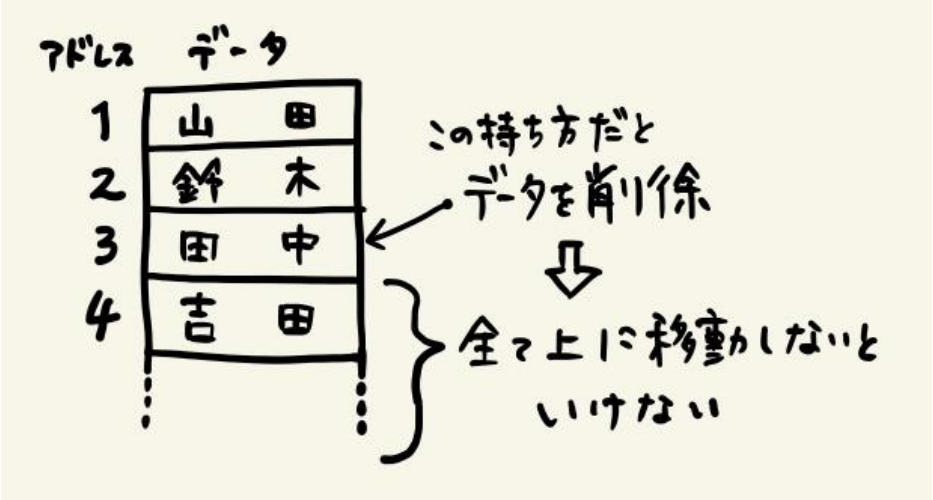




リストの利点：
データを動的に追加・削除可能

- リストは、必要に応じてメモリを確保できる
  - 後ろに繋げていけばよい
- 追加・削除が容易
  - 配列でもつと削除が大変になる

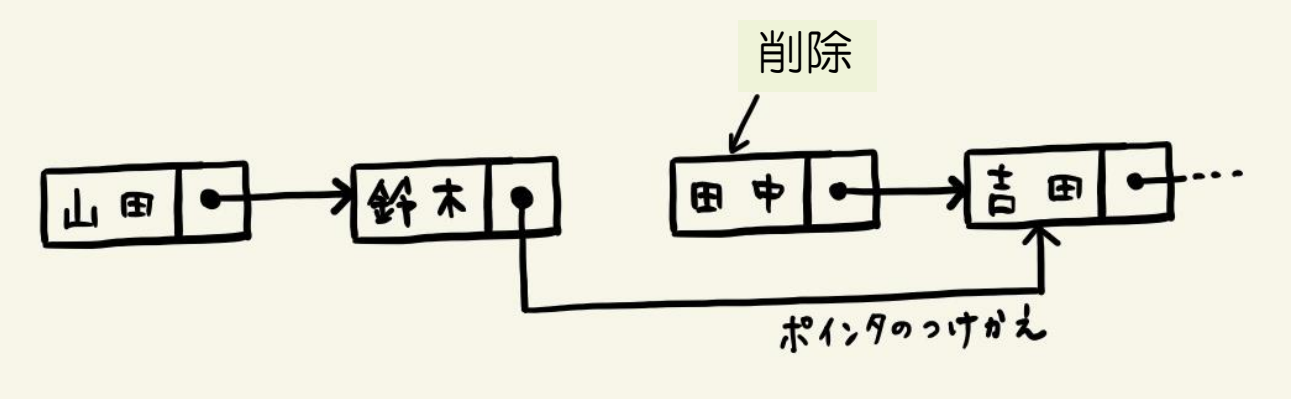




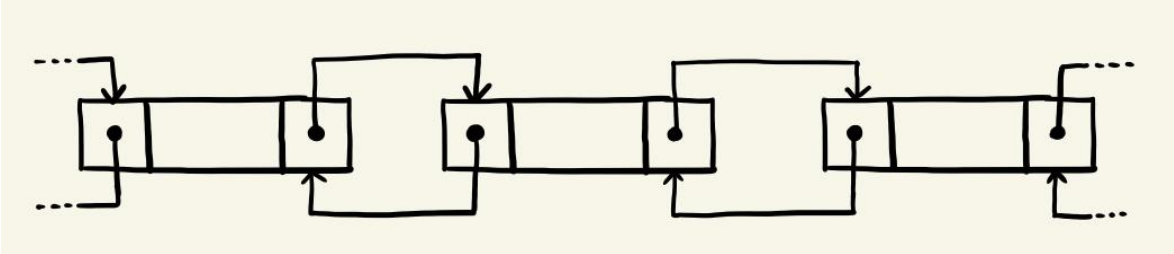
- 検索では得しない（効率的な検索には別の仕組が必要）

リストの利点：
データを動的に追加・削除可能

- 削除：ポインタの付け替えで対応

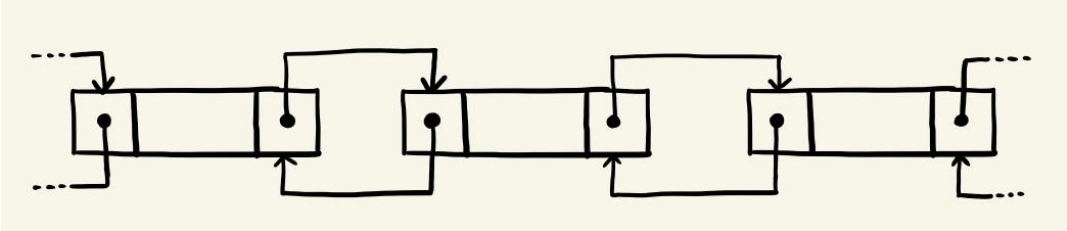


  - ポインタのつけかえには、誰が自分にポインタを指しているかを知る必要がある（単純には$O(n)$）
  - 二重線形リスト： $O(1)$ で発見可能

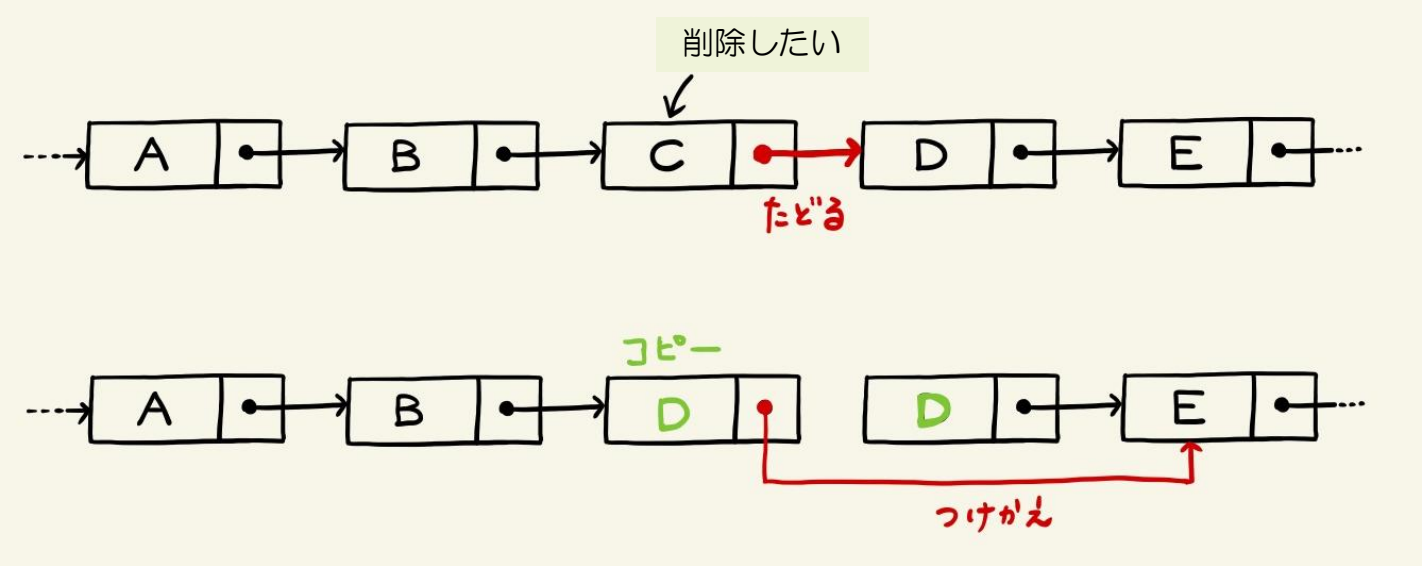


リストの利点：
データを動的に追加・削除可能

- 二重線形リストは O(1) で削除できるがポインタが2つ必要

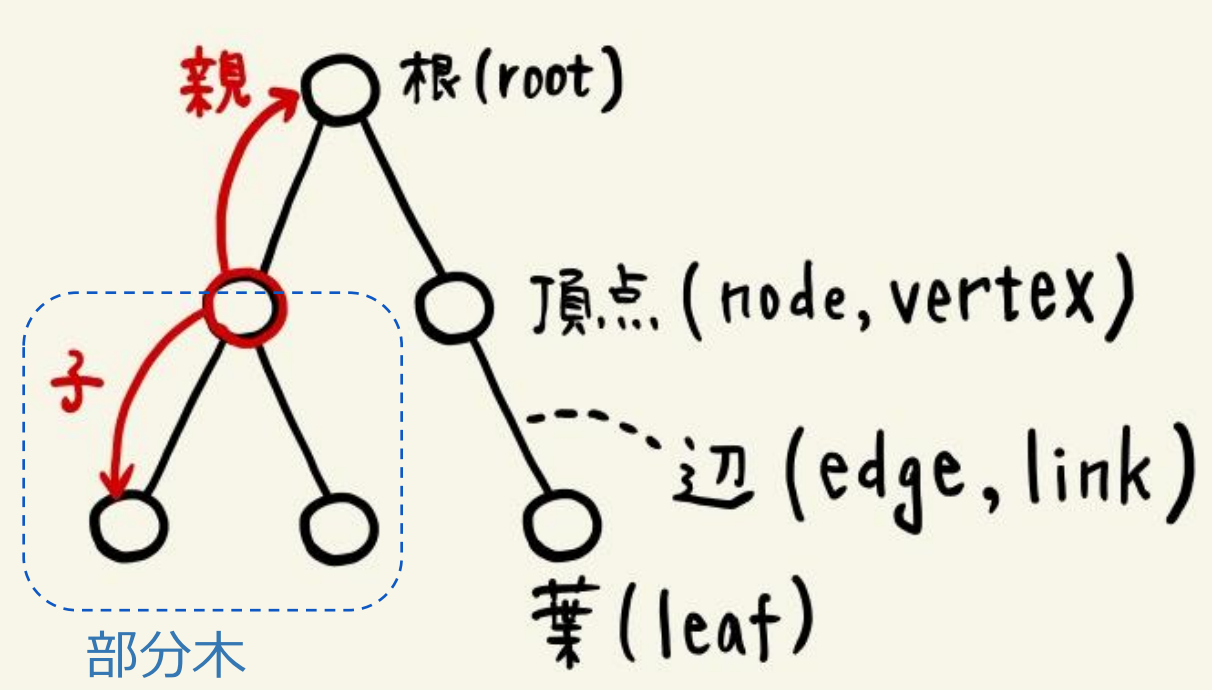


- 実は二重にしなくても可能：たどる→コピー→付け替え



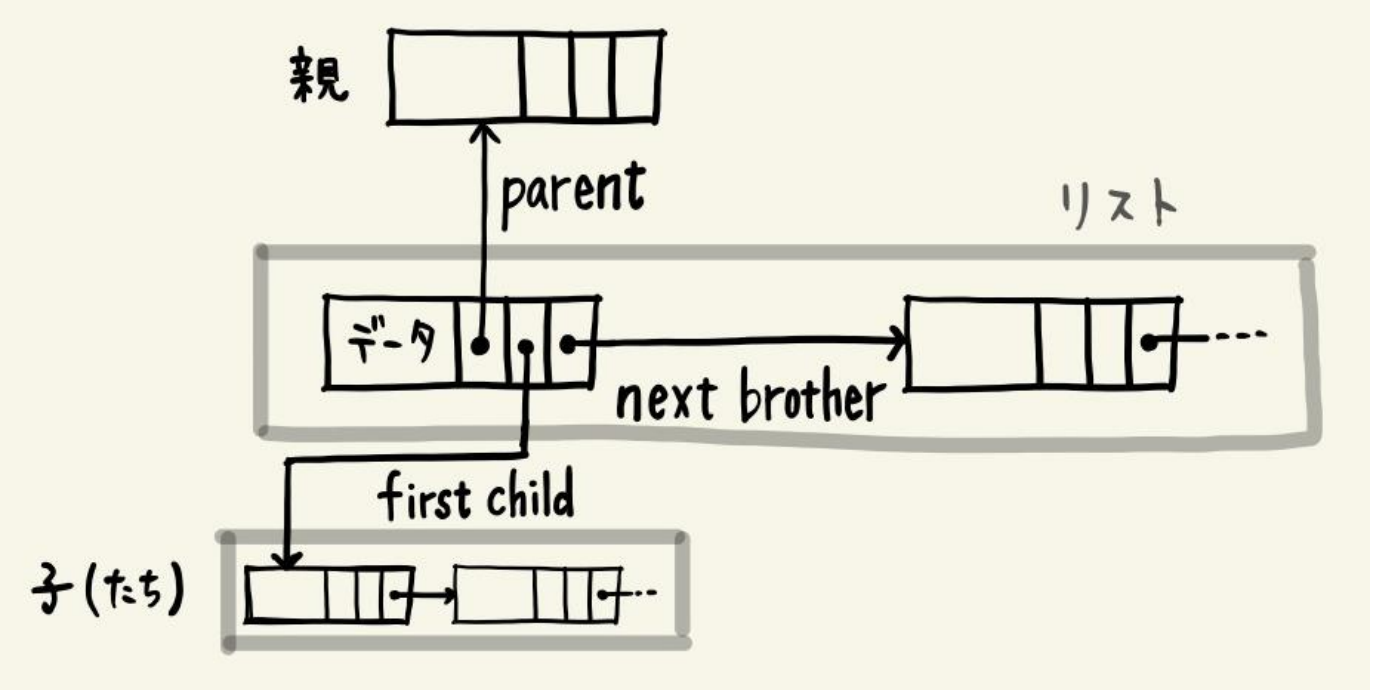
根付き木：枝分かれするリストの一般化

- 頂点集合と、それらを結ぶ辺から構成される
  - 辺に接続する頂点の片方が親でもう一方を子とする
  - 各頂点は0～複数個の子をもつ
  - 根以外の頂点は、必ずただひとつの親頂点をもつ
  - 葉：子をもたない頂点
  - 部分木：
    ある頂点以下の部分



根付き木の実現：各頂点が3個のポインタをもつ

- 各頂点は親へのポインタ、次のきょうだいへのポインタ、最初の子へのポインタをもつ
  - 全ての子へのポインタをもつかわりに最初の子だけを指す
    –各頂点は最大3個のポインタを保持



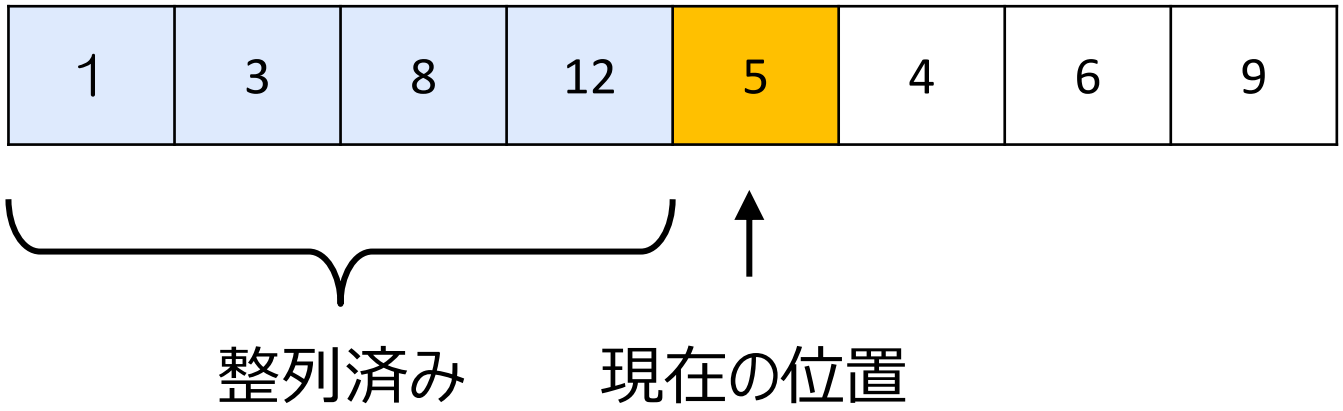
整列（ソート）のアルゴリズム

整列問題（ソート）：要素を小さい順に並び替える問題

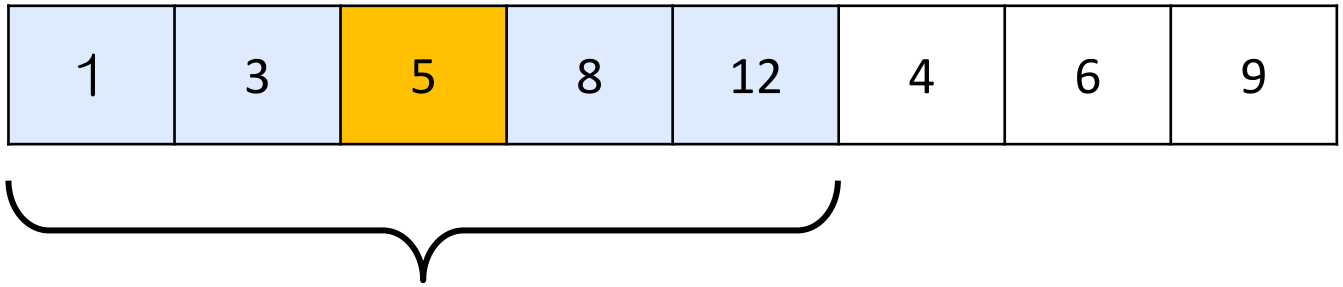
- 整列問題
  - 入力：n個の数 $a_1, a_2, \ldots, a_n$ が入った配列
  - 出力： $a_{1'} \leqq a_{2'} \leqq \ldots \leqq a_{n'}$ を満たす、入力列の置換
- 例：入力 $(4, 5, 2, 1) \rightarrow$ 出力 $(1, 2, 4, 5)$

単純なソートアルゴリズム：ソート済み領域を左から順に拡大していく

- ある時点において、現在の位置よりも左の部分は整列済みとする



- 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
- 現在の位置がjであるとすると、「・・・」の操作には$O(j)$回の比較・交換が必要
- これを $j = 1,2,..,n$ まで行うと
  $\Sigma_{j=1,...,n} O(j) = O(n^2)$ になる
- このアルゴリズムはあまり効率はよくない
  - 効率の良いアルゴリズムは$O(n \log n)$ （後述）
- 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