データ構造とアルゴリズム

03 アルゴリズムの性能と計算複雑度

宮本 裕一郎

miyamoto **あっと** sophia.ac.jp

上智大学 理工学部 情報理工学科

目次

アルゴリズムの性能

- アルゴリズムの良さを測る尺度は，
  - 速さ，
  - 使用メモリ，
  - わかりやすさ，

  など，いろいろある．
- これらの尺度の重要度は文脈で異なるが，この講義では
  - 速さがもっとも重要であり，
  - 使用メモリ量がそれに次ぐ

  とする．

アルゴリズムの速さの評価

- アルゴリズムの速さの評価法には，
  - 実験的（経験的）方法:
    - 実際にプログラムを作り，コンピューターで実行し時間を測る方法，
  - 理論的（解析的）方法:
    - 何らかの解析でアルゴリズムの手間を見積もる方法，

  などがある．
- 実験的方法の長所は，実践的であることである．
- 実験的方法の短所は，
  - プログラミングする労力がかかること，
  - プログラマーの腕に依存すること，
  - コンピューターの性能にも依存すること，
  - プログラミング言語にも依存すること，

  などである．
- 本講義では，アルゴリズムの分野で基本とされる理論的方法を学ぶ．
- ところで，アルゴリズムの手間とは何だろうか？

アルゴリズムの手間の定義

定義 (（狭義の）アルゴリズム【再掲】)

Turing 機械（Turing machine）で実行可能な手続きをアルゴリズムという．

定義 (（狭義の）アルゴリズムの手間)

Turing 機械でアルゴリズムを実行した際の計算状況（状態）の数をアルゴリズムの手間という．

- 本講義では Turing 機械などの計算モデルの説明は扱わない．よって，より簡便に以下の通り定義する．

定義 (アルゴリズムの手間)

アルゴリズムで行われる基本演算の回数をアルゴリズムの手間とする．

- 早速，`CircleMethod` で基本演算が行われる回数を見積もってみよう．

CircleMethod の基本演算回数を数えてみる

- CircleMethod の基本演算回数は，入力のチーム数によって異なる．
  n チームの入力における代入演算の回数を n の関数で表してみる．ただし n は偶数とする．
- 代入演算は，Step 1 で n 回，
  Step 2 で $(n-1) \times (n + (n-2) + 1 + (n-1))$ 回行われる．
  よって代入演算は合計で $3n^2 - 4n + 2$ 回行われる．

CircleMethod$(t_1, t_2, \ldots, t_n)$: 【再掲】

Step 1 それぞれの $i \in \{1, 2, \ldots, n\}$ に関して，c_{i-1} に t_i を代入する．

Step 2 それぞれの $d \in (1, 2, \ldots, n-1)$ に関して，以下を繰り返す．

- Step 2-1 それぞれの $i \in \{0, 1, \ldots, n-1\}$ に関して，$p(c_i, d)$ に c_{n-i-1} を代入する．
- Step 2-2 それぞれの $i \in \{1, \ldots, n-2\}$ に関して，c'_i に c_{i-1} を代入する．
- Step 2-3 c'_0 に c_{n-2} を代入する．
- Step 2-4 それぞれの $i \in \{0, \ldots, n-2\}$ に関して，c_i に c'_i を代入する．

Step 3 それぞれの $t \in T$, $d \in \{1, 2, \ldots, n-1\}$ に関して，$p(t, d)$ を出力する．

Maximum の基本演算回数を数えてみる

- 次に，最大値を出力するアルゴリズム Maximum の基本演算回数を数えてみる．
- Maximum の基本演算は主に代入と比較である．
- 入力数値列の要素が n 個の場合における
  比較演算・代入演算の回数を n の関数で表してみる．
- 比較演算は，Step 2 で $n-1$ 回行われる．
- 代入演算は，Step 1 で 1 回，
  Step 2 で……，**入力データによって異なる**！

Maximum$(\{x_1, x_2, \ldots, x_n\})$: 【再掲】

Step 1 y に x_1 を代入する．

Step 2 それぞれの $i \in \{2, 3, \ldots, n\}$ に関して以下を行う．

　　Step 2-1 もし $y < x_i$ ならば，y に x_i を代入する．

Step 3 y を出力して終了する．

アルゴリズムの性能と計算複雑度

おまけの付録

アルゴリズムの速さの理論的評価

- アルゴリズムにおける基本演算回数を見積もる．
- ただし，一般に入力が大きいときの基本演算回数を見積もりたいので，**入力の大きさの関数**で表す．
- そして，入力の大きさだけでは基本演算回数を決定できない場合には，以下の 2 通りが考えられる．
  - ありそうな入力を想定し，基本演算回数の期待値を見積もる．
    ただし，
    - 長所として，実践的？であるが，
    - 短所として，ありそうな入力って何？

    ということがある．
  - 最悪の入力を想定し，基本演算回数を見積もる．
    ただし，
    - 長所として，入力に依存しないが，
    - 短所として，最悪の場合が稀だったら杞憂に終わるのでは？

    ということがある．
- 最も基本演算回数が多くなる入力（最悪の入力）を想定して，基本演算回数を入力の大きさの関数で表す．これを**最悪値解析**という．
- ところで，問題の入力の大きさとは？

計算問題の入力の大きさ

- 計算問題の入力の大きさは，**入力文字列の長さ**である．
- そして，多くの文脈では普通のコンピューターで計算する場合を想定している．
- よって「計算問題の入力を 2 進数表記したときの**ビット数**」が入力の大きさと考えて差し支えない．
- 厳密にビット数を数えるのは容易ではない．しかし，後の議論により，ある程度アバウトに数えて差し支えない．
- この講義では，説明の簡単のため，計算問題の設定（すなわち入力と出力の関係）とともに問題の入力の大きさも与えられているとする．
  - 総当り戦スケジュール表作成問題の入力の大きさはチーム数とする．
  - 自然数の乗算の入力の大きさは，大きい方の数字の桁数とする．
  - 最大公約数の入力の大きさは，大きい方の数字の桁数とする．
  - 安定マッチング問題の入力の大きさは男性（あるいは女性）の人数を n とすると n^2 に比例する．しかし，解析を容易にするために，計算の手間を n の関数で表すことも多い．

Maximum の基本演算回数を再び数えてみる

- 最大値を出力するアルゴリズム Maximum の基本演算回数の最悪値を見積もる．
- 入力数値列の要素が n 個の場合における比較演算・代入演算の最悪値（最大値）を n の関数で表してみる．
- 比較演算は，Step 2 で $n-1$ 回行われる．
- 代入演算は，Step 1 で 1 回，Step 2 で「最悪」$n-1$ 回，合計で n 回である．

Maximum$(\{x_1, x_2, \ldots, x_n\})$: 【再掲】

Step 1 y に x_1 を代入する．

Step 2 それぞれの $i \in \{2, 3, \ldots, n\}$ に関して以下を行う．

　　Step 2-1 もし $y < x_i$ ならば，y に x_i を代入する．

Step 3 y を出力して終了する．

漸近的解析へ

- 本講義では，ほとんどの場合，最悪値解析を採用する．
- アルゴリズム `Maximum` の実行時間は，比較演算にかかる時間を a，代入演算にかかる時間を b とすると，最悪の場合でも

$$a(n-1)+bn$$

  と見積もれる．
- この見積方法で Gale-Shapley アルゴリズムも見積もれるであろうか？

  難点 基本演算の種類が増えると，演算回数を表す式が複雑になる．
  疑問 そもそも，そんなに細かく見積もる必要があるのか？
  細かく見積もって，役に立つのか？
  - ハードウェアによって，それぞれの基本演算にかかる時間は異なるのが一般的である．そこまで対応しきれない．
  - そもそも，プログラミングせずに知りたいのは
    「おおよそ 1 秒くらい？それとも 1 日くらい？」
    という大雑把な見積もりである．
- 基本演算回数を表す式を大雑把に，しかし誰が見積もっても同じになるように表したくなる．次に，漸近的解析法を紹介する．

O 記法

- 漸近的解析のため，以下に O 記法を定義する．

定義 (O 記法)

$f\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$, $g\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ とする．このとき，

$$\exists \alpha, \beta \in \mathbb{R}_{>0},\ \forall n \in \mathbb{N},\ f(n) \leq \alpha g(n) + \beta$$

ならば「$f(n) = \mathrm{O}(g(n))$ である[1]」という．これを O 記法（big O notation）という．なお「エフエヌはオーダージーエヌである」と発音する．

例

- $3n^2 - 4n + 2 = \mathrm{O}(n^2)$
- a, b を正の実数定数とすると, $a(n-1) + bn = \mathrm{O}(n)$
- $3n^2 - 4n + 2 \neq \mathrm{O}(n)$ （「$3n^2 - 4n + 2 = \mathrm{O}(n)$ ではない」のつもり．）

[1] $f(n) \in \mathrm{O}(g(n))$ と記す流儀もある．

O 記法の直感的理解

- 例から想像がつく通り，直感的には，関数の上界を大雑把に（しかし誰が計算しても同じ結果になるという意味で厳密に）表現するのが O 記法である．
- 「$f(n)$ は n が十分に大きい場合には高々$g(n)$ くらいですよ」という気持ちである．
- より正確には，n が十分大きいときの挙動（すなわち漸近的挙動）を表していると言って良い．
- 上界の表現なので，一意ではない．
- アルゴリズムの文脈では，よく「オーダー」とよばれる．
- $f(n)$ が与えられるたびに定義から計算するのは大変なので，有用な定理を以降少し紹介する．

オーダーの推移性

定理 (オーダーの推移性)

$f(n) = O(g(n)),\ g(n) = O(h(n)) \Longrightarrow f(n) = O(h(n))$

証明.

- **定義より** $\exists \alpha, \beta \in \mathbb{R}_{>0},\ \forall n \in \mathbb{N},\ f(n) \leq \alpha g(n) + \beta$ **かつ** $\exists \alpha', \beta' \in \mathbb{R}_{>0},\ \forall n \in \mathbb{N},\ g(n) \leq \alpha' h(n) + \beta'$ **である.**
- **よって** $\forall n \in \mathbb{N},\ f(n) \leq \alpha\alpha' h(n) + \alpha\beta' + \beta$ **である.**

□

例

$n^2 = O(n^3),\ n^3 = O(n^4)$ **である. そしてもちろん** $n^2 = O(n^4)$ **である.**

オーダーの加法性

定理 (オーダーの加法性)

関数 $f_i \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ $(i \in \{1, \ldots, k\})$ **と関数** $h \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ **があり，** $f_i(n) = O(h(n))$ $(i \in \{1, \ldots, k\})$ **であるとする．このとき** $f_1(n) + f_2(n) + \cdots + f_k(n) = O(h(n))$ **である．**

証明.

- **定義より** $\exists \alpha_i, \beta_i \in \mathbb{R}_{>0},\ \forall n \in \mathbb{N},\ f_i(n) \leq \alpha_i h(n) + \beta_i$ $(i \in \{1, \ldots, k\})$ **である．**
- **よって** $\forall n \in \mathbb{N},\ f_1(n) + f_2(n) + \cdots + f_k(n) \leq (\alpha_1 + \alpha_2 + \cdots + \alpha_k)h(n) + (\beta_1 + \beta_2 + \cdots + \beta_k)$ **である．**

□

例

$5n^2 = \mathrm{O}(n^3),\ 4n^3 = \mathrm{O}(n^3)$ **である．そして** $5n^2 + 4n^3 = \mathrm{O}(n^3)$ **である．**

多項式のオーダー

定理 (多項式のオーダー)

$p \in \mathbb{Z}_{\geq 0}$ に対して $a_0, a_1, \ldots, a_p$ を実数定数とし，特に $a_p > 0$ とする．このとき $a_p n^p + a_{p-1} n^{p-1} + \cdots + a_1 n^1 + a_0 n^0 = O(n^p)$ である．

すなわち多項式のオーダーは，最大次数項のみを取り出しその係数を 1 としたもので良い．

証明.

$I^+ = \{i \in \{0, \ldots, p\} \mid a_i \geq 0\}$ とする．明らかに

$a_p n^p + a_{p-1} n^{p-1} + \cdots + a_1 n^1 + a_0 n^0 = \mathrm{O}\left(\sum_{i \in I^+} a_i n^i\right)$ であり，

$a_i n^i = \mathrm{O}(n^p)$ $(i \in I^+)$ である．よってオーダーの加法性の定理より $\sum_{i \in I^+} a_i n^i = \mathrm{O}(n^p)$ である．さらにオーダーの推移性の定理を適用することにより $a_p n^p + a_{p-1} n^{p-1} + \cdots + a_1 n^1 + a_0 n^0 = \mathrm{O}(n^p)$ である． □

対数，指数のオーダー

定理 (対数のオーダー)

任意の実数定数 $b > 1,\ x > 0$ に関して，$\log_b n = O(n^x)$ である．
また，任意の実数定数 $a > 1,\ b > 1$ に関して $\log_b n = O(\log_a n)$ である．

- この定理の前者は，対数関数が多項式関数で上から抑えられることを意味する．
- この定理の後者は，O 記法において対数の底は意味を成さない（すなわちどれでも同じ）ことを意味する．
- よって O 記法においては，対数の底は省略するのが普通である．

定理 (指数のオーダー)

任意の実数定数 $r > 1,\ d > 0$ に関して，$n^d = O(r^n)$ である．

- この定理は，多項式関数が指数関数で上から抑えられることを意味する．
- いずれも証明は省略する．

オーダーに関する注意

- オーダーの定義より $n^2 = \mathrm{O}(n^{100})$ は正しい．
  - しかし分かる範囲でなるべくきついものを書くのが普通である．アルゴリズムの速さの保証に使うならば，ゆるすぎるものを書いても意味が無い．
  - オーダーの見積りが非常に困難な場合でも後世で改善される余地が残る定義となっている．
- O 記法の類似品で下界を見積もる Ω 記法がある．
  - $g(n) = \mathrm{O}(f(n)) \Longrightarrow f(n) = \Omega(g(n))$
- さらに $f(n) = \mathrm{O}(g(n))$ かつ $f(n) = \Omega(g(n))$ のとき $f(n) = \Theta(g(n))$ であるという．
  - この場合は「両者はほとんど同じである」という気持ちである．
- これらの表記を用いて（基本演算回数などの）関数を解析することを漸近的解析という．[2]

[2] O 記法などの定義は，理論計算機科学とは独立に開発された．しかし現在最も頻繁にこの表記を用いているのは理論計算機科学である（と思う）．有用な定理が成り立つように良く定義されている．ε - δ 論法と似ている．

計算複雑度と直感的認識

- アルゴリズムで使われる基本演算の回数を入力の大きさで表した関数のオーダーを**時間複雑度**（time complexity）（あるいは時間計算量）という．
- アルゴリズムで使われる記憶領域（bit 数）を入力の大きさで表した関数のオーダーを**空間複雑度**（space complexity）（あるいは空間計算量）という．
- 時間複雑度と空間複雑度を合わせたものを**計算複雑度**（computational complexity）（あるいは計算量）という．
- 時間複雑度に関する直感的認識の例を以下に挙げる．
  - $O(\log n)$: 非常に速い，例: 2 分探索など
  - $O(n)$: 速い，例: 深さ優先探索など
  - $O(n \log n)$: 十分速い，例: マージソートなど
  - $O(n^2)$: 微妙，例: バブルソートなど
  - $O(n^3)$: 遅い，例: Floyd-Warshall アルゴリズムなど
  - $O(2^n)$: 論外，例: 列挙型アルゴリズムなど

演習問題: 基本的なオーダーの計算

問題 以下の関数 $f_1(n)$, $f_2(n)$, $f_3(n)$ を O 記法で表してみよう！

- $f_1(n) = 2.7n^3 + n^2 + 8$
- $f_2(n) = 2.8n\sqrt{n} + 1.8n \log n$
- $f_3(n) = 2^n + 8n^4$

解答例

- $f_1(n) = 2.7n^3 + n^2 + 8 = O(n^3)$（∵ 多項式のオーダーの定理）
- $f_2(n) = 2.8n\sqrt{n} + 1.8n \log n = O(n\sqrt{n}) = O(n^{\frac{3}{2}})$（対数のオーダーの定理より $\log n = O(n^{\frac{1}{2}}) = O(\sqrt{n})$ であり，あとは加法性と併せて考える）
- $f_3(n) = 2^n + 8n^4 = O(2^n)$（指数のオーダーの定理より $n^4 = O(2^n)$ であり，あとは加法性と併せて考える）

演習問題: ちょっと難しいオーダーの比較

- $f(n) = \mathrm{O}(g(n))$ のとき，
  $f(n)$ と $g(n)$ の関係を不等号を使って，$f(n) \leq g(n)$ と書くことにする．
  以下の関数 $f_1(n), \ldots, f_5(n)$ の関係を不等号を使って表してみよう！
  - $f_1(n) = 10^n$
  - $f_2(n) = n^{\frac{1}{3}}$
  - $f_3(n) = n^n$
  - $f_4(n) = \log_2 n$
  - $f_5(n) = 2^{\sqrt{\log_2 n}}$

解答例: ちょっと難しいオーダーの比較

対数，多項式，指数のオーダーの関係より $\log_2 n \leq n^{\frac{1}{3}} \leq 10^n \leq n^n$ は明らかである．問題は $2^{\sqrt{\log_2 n}}$ がどこに入るのかである．ここで（例えば底が 2 の）対数をとって比較してみる．（底が 2 の）対数は単調増加なので大小関係が保存されるからである．すると，

$\log_2 f_5(n) = \log_2 2^{\sqrt{\log_2 n}} = \sqrt{\log_2 n} \log_2 2 = \sqrt{\log_2 n}$，

$\log_2 f_2(n) = \log_2 n^{\frac{1}{3}} = \frac{1}{3} \log_2 n$，$\log_2 f_4(n) = \log_2 \log_2 n$

$$\therefore \log_2 n \leq 2^{\sqrt{\log_2 n}} \leq n^{\frac{1}{3}} \leq 10^n \leq n^n$$

である．（$z = \log_2 n$ などと置き換えると，よりわかりやすい．）

演習問題: オーダーの真偽

問題 $f(n) = \mathrm{O}(g(n))$ **であるとする．以下の式はそれぞれ真か偽か？理由も一緒に考えてみよう！**

- $\log_2 f(n) = \mathrm{O}(\log_2 g(n))$
- $2^{f(n)} = \mathrm{O}(2^{g(n)})$
- $(f(n))^2 = \mathrm{O}((g(n))^2)$

解答例

- $\log_2 f(n) = \mathrm{O}(\log_2 g(n))$ **は真である．**
  - **証明略**
- $2^{f(n)} = \mathrm{O}(2^{g(n)})$ **は偽である．**
  - **例えば，** $f(n) = 10n,\ g(n) = n$ **とすると明らかに** $f(n) = \mathrm{O}(g(n))$ **であるが，** $2^{10n} = 1024^n$ **は** $\mathrm{O}(2^n)$ **ではない．**
- $(f(n))^2 = \mathrm{O}((g(n))^2)$ **は真である．**
  - **証明略**

演習問題: 単純な素数列挙アルゴリズムの時間複雑度

問題 n 以下の素数を単純に列挙するアルゴリズム `EnumeratePrime` の時間複雑度を示せ．ただし入力の大きさは n とする．

解答例 `PrimeSimple`(x) の時間複雑度は，(剰余を基本演算とするならば) $\mathrm{O}(x)$ である．

よって `EnumeratePrime` の時間複雑度は $\sum_{x=2}^{n} \mathrm{O}(x) = \mathrm{O}(n^2)$ である．

`PrimeSimple`(x): 【再掲】

Step 1 それぞれの $y \in \{2, \ldots, x-1\}$ に関して，以下を繰り返す．

Step 1-1 もし $x \mod y = 0$ ならば，`False` を出力し終了する．

Step 2 `True` を出力し終了する．

`EnumeratePrime`(n): 【再掲】

Step 1 P を空集合とする．

Step 2 それぞれの $x \in \{2, \ldots, n\}$ に関して，以下を繰り返す．

Step 2-1 もし `PrimeSimple`(x) の出力が `True` ならば，P に x を加える．

Step 3 P を出力して終了する．

演習問題: 平方根を用いた素数列挙アルゴリズムの時間複雑度

問題 n 以下の素数を列挙する，平方根を使ったアルゴリズム EnumeratePrimeSqrt の時間複雑度を示せ．ただし入力の大きさは n とする．

解答例 PrimeSqrt の時間複雑度は $O(\sqrt{x})$ である[3]．EnumeratePrimeSqrt の時間複雑度は $\sum_{x=2}^{n} O(\sqrt{x}) = O(n\sqrt{n}) = O(n^{\frac{3}{2}})$ である．

PrimeSqrt(x): 【再掲】

Step 1 それぞれの $y \in \{2, \ldots, \lfloor \sqrt{x} \rfloor\}$ に関して，以下を繰り返す．

　Step 1-1 もし $x \mod y = 0$ ならば，False を出力し終了する．

Step 2 True を出力し終了する．

EnumeratePrimeSqrt(n): 【再掲】

Step 1 P を空集合とする．

Step 2 それぞれの $x \in \{2, \ldots, n\}$ に関して，以下を繰り返す．

　Step 2-1 もし PrimeSqrt(x) の出力が True ならば，P に x を加える．

Step 3 P を出力して終了する．

---

[3] 平方根の計算を基本演算としなくても，2 分探索の利用などを考慮すると，時間複雑度は変わらない．

演習問題: エラトステネスの篩の時間複雑度

問題 エラトステネスの篩 Eratosthenes の時間複雑度を示せ．ただし入力の大きさは n とする．

解答例 単純に考えてみる．まず，Step 3-1-2 を $O(n)$ 回行っている．そして，各整数 i の倍数は $O(n)$ 個なので，「q_i に False を代入する」操作は全体で $O(n^2)$ 回である．よって，Eratosthenes の時間複雑度は $O(n^2)$ である．これは間違いではない．

Eratosthenes(n):【再掲】

Step 1 P を空集合とする．

Step 2 それぞれの $i \in \{2,3,\ldots,n\}$ に関して，q_i を True とする．

Step 3 それぞれの $i \in (2,3,\ldots,n)$ に関して，以下を繰り返す．

- Step 3-1 もし q_i が True ならば，以下を行う．
  - Step 3-1-1 P に i を加える．
  - Step 3-1-2 i の倍数であるようなそれぞれの j（ただし n 以下）に関して，q_j を False とする．

Step 4 P を出力し終了する．

エラトステネスの篩の時間複雑度の解答例の続き

解答例 2 もう少し丁寧に考えてみる．

- 2 の倍数は高々$n/2$ 個，3 の倍数は高々$n/3$ 個である．
- よって，`Eratosthenes` の時間複雑度は
  $\sum_{k=2}^{n} \frac{n}{k} = \mathrm{O}(n \log n)$ である．
- このように局所的に見積もるだけではなく，大局的に見積もることも時には重要である．

解答例 3 さらに，よく考えてみると……

- Step 3-1-2 は整数 i が素数でなければ実行されない．
- よって，素数定理を適用することを考えると……

演習問題: ガラスのビンの耐久試験

- ガラスのビンを高いところから落として耐久試験を行う．
- ガラスのビンを落としても割れないギリギリの高さ（耐久限界）を知りたい．
- 高さは，1 から n の n 種類に限られているとする．
- そして基本演算は
  「高さ $x \in \{1, 2, \ldots, n\}$ からガラスの瓶を落とすこと」
  であるとする．
  1. ガラスのビンが 2 つある．高々$f(n)$ 回の試行で耐久限界を知る方法（アルゴリズム）を示したい．$f(n)$ のオーダーが n よりも真に小さい（すなわち $\lim_{n\to\infty} \frac{f(n)}{n} = 0$ となる）アルゴリズムを示してみよう！
  2. ガラスのビンが k 個ある．高々$f_k(n)$ 回の試行で耐久限界を知る方法（アルゴリズム）を示したい．ここで $f_k(n)$ のオーダーが $f_{k-1}(n)$ よりも真に小さい（すなわち $\forall k \in \mathbb{N},\ \lim_{n\to\infty} \frac{f_k(n)}{f_{k-1}(n)} = 0$ となる）アルゴリズムを示してみよう！

ガラスのビンの耐久試験の解答例

- （割っても良い）ガラスのビンが２つある場合には，$f(n) = O(\sqrt{n}) = O(n^{\frac{1}{2}})$ のアルゴリズムを作ることができる．
  - 例えば，1 つ目のビンで高さ $\lceil n^{\frac{1}{2}} \rceil, 2\lceil n^{\frac{1}{2}} \rceil, 3\lceil n^{\frac{1}{2}} \rceil, \ldots$ を下から試し，$i\lceil n^{\frac{1}{2}} \rceil$ で 1 つ目のビンが割れたら，2 つ目のビンでは $(i-1)\lceil n^{\frac{1}{2}} \rceil + 1$ から $i\lceil n^{\frac{1}{2}} \rceil$ までを 1 段ずつ試せばよい．1 つ目のビンの試行回数は高々$n^{\frac{1}{2}}$ 回，2 つ目のビンの試行回数も高々$n^{\frac{1}{2}}$ 回なので $f(n) \leq n^{\frac{1}{2}} + n^{\frac{1}{2}} = O(n^{\frac{1}{2}})$
- 同様に，(割っても良い）ガラスのビンが k 個ある場合には，$f(n) = O(n^{\frac{1}{k}})$ のアルゴリズムを作ることができる[4]．

[4] k が十分に多くなると 2 分探索との関連が気になってくるが，それは若干高度な話題なので各自の興味に任せる．

演習問題: オーダーの計算

以下の 1. 2. 3. **のそれぞれに関して，**

ア $f(n) = \mathrm{O}(g(n))$ **であるが** $g(n) \neq \mathrm{O}(f(n))$

イ $g(n) = \mathrm{O}(f(n))$ **であるが** $f(n) \neq \mathrm{O}(g(n))$

ウ $f(n) = \Theta(g(n))$**，すなわち** $f(n) = \mathrm{O}(g(n))$ **かつ** $g(n) = \mathrm{O}(f(n))$

のいずれであるか空欄にア，イ，ウのいずれかを入れよ．ただし，間違いである場合には減点する．

1. $f(n) = n^{\frac{1}{2}}$, $g(n) = n^{\frac{2}{3}}$ **ならば** $\square$ **である．**
2. $f(n) = 100n + \log n$**，** $g(n) = n + (\log n)^2$ **ならば** $\square$ **である．**
3. $f(n) = n^{1.01}$**，** $g(n) = n(\log n)^2$ **ならば** $\square$ **である．**

（2015 **年度期末試験問題より）**

アルゴリズムの性能と計算複雑度

おまけの付録

さらなる勉強のために

- **このスライドの** O **記法の定義は** [Korte and Vygen, 2012] **からの引用である．**
- **演習問題の，ちょっと難しいオーダーの比較，オーダーの真偽，ガラスの瓶の耐久試験は** [Kleinberg and Tardos, 2005] **からの引用である．**

参考文献

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009).
Introduction to Algorithms.
MIT Press.

[Kleinberg and Tardos, 2005] Kleinberg, J. and Tardos, E. (2005).
Algorithm Design.
Addison-Wesley.

[Korte and Vygen, 2012] Korte, B. and Vygen, J. (2012).
Combinatorial Optimization: Theory and Algorithms, volume 21 of *Algorithms and Combinatorics*.
Springer-Verlag, 5th edition.

[Sedgewick and Wayne, 2011] Sedgewick, R. and Wayne, K. (2011).
Algorithms.
Addison-Wesley.

おまけの付録: O 記法の定義の違い

- アルゴリズムの教科書をいくつか見ると，O 記法の定義がそれぞれ若干異なっていることに気づく．
- 例えば，世界的に有名と思われるアルゴリズムの教科書 [Cormen et al., 2009, Kleinberg and Tardos, 2005, Sedgewick and Wayne, 2011] では O 記法を以下の通り定義している．

定義 (O 記法)

$f\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$, $g\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ とする．このとき，

$$\exists n_0, c \in \mathbb{R}_{>0},\ \forall n \geq n_0,\ f(n) \leq cg(n)$$

ならば「$f(n) = \mathrm{O}(g(n))$ である」という．

- なお，上記の定義において不等号に等号が含まれるか否かも教科書によって微妙に異なる．
- 定義の些細な違いは重要ではない．真に重要なことは「どの定義でもこの講義で紹介した定理は同様に成り立つ」という事実である．
- 定義があるから定理があるとは限らない．有用な定理を成り立たせるために，うまく定義を作るということもあり得る．O 記法においては「大雑把な見積りだけれど，誰でも同じ見積りになる表現にしたい」という思想が根底にある（と思う）．