精読『アルゴリズムイントロダクション 第4版』

第1回

Panot

最終更新：April 15, 2023

精読『アルゴリズムイントロダクション 第4版』
第1回
Panot
最終更新：April 15, 2023

- 今回のイベントは読書会発表兼講義という形です。
- 自分のアルゴリズムの復習を兼ねて、アルゴリズムをちゃんと勉強したい方のためにできるだけわかりやすく説明するつもりです。
- この世界的アルゴリズムの聖書と言われている教科書の理解のためになれれば幸いです
- 質問や指摘はチャットで自由に書いて、僕は適当に拾って答えます。
- 前提知識として、初歩的なプログラミングと、数列の和やべき乗や対数ができること
- アルゴリズムの C++実装例とこのスライドを github で配布する予定です。

今日の範囲

第１章：計算におけるアルゴリズムの役割
第２章：さあ、始めよう
第３章：実行時間の特徴づけ

- 今日の範囲は第 1 章から第 3 章です
- 第 1 章と第 2 章の内容はほとんど同じですが、第 3 章の内容はコアが同じですが、もっとわかりやすく更新されています。
- 章のタイトルも「Growth of Functions」「関数の増加」から「Characterizing Running Time」「実行時間の特徴づけ」に変わっています。

今日の内容

- アルゴリズムとは？
- なぜアルゴリズムの理解が必要なのか？
- アルゴリズム解析の概要
  - 挿入ソート
  - マージソート
- 「オーダ」
  - 直感
  - 定義

- 第１章：「アルゴリズムとは？」「なぜアルゴリズムの理解が必要なのか」。
- 第２章：「アルゴリズム解析の概要」ソーティング問題を例にあげ、その問題を解くアルゴリズムに「Insertion sort - 挿入ソート」と「Merge sort - マージソート」をあげます。
- 第３章：いわゆる「関数のオーダ」、「ビッグオー」など聞いたことがある方もいらっしゃると思いますが、その直感とちゃんとした定義について話します。

Section 1

第１章：計算におけるアルゴリズムの役割

アルゴリズムとは？

- 明確に定義された手続き
- 値または値の集合を**入力**として取る
- ある値または値の集合を**出力**して生成する

アルゴリズムとは？

- 明確に定義された手続き
- 値または値の集合を**入力**として取る
- ある値または値の集合を**出力**して生成する

- 基本的な操作の組み合わせだけで書かれた、間違いようのない手続きです。
- 計算機はバカだから、基本的な操作しか理解できず、明確に書かなければなりません。
- アルゴリズムには入力があって、入力に対して出力を生成する。
- 数学の関数のイメージをしてもいいと思いますが、乱択アルゴリズムになるとややこしいです。それでも結局乱択アルゴリズムは多くの場合擬似乱数発生器の疑似乱数を使うので、そのシードも入力の一部として考えてもいいです。

アルゴリズムとは？

何に使われている？

- **計算問題**を解く道具として
- **正当な**アルゴリズム
- すべての**問題のインスタンス**に対して
  - 常に**停止**する
  - 出力が**正しい**

- 何に使われているか？
- 計算問題とは、ある入力に対してどのような出力が欲しいかを指定する問題です。
- アルゴリズムはそれを実現させる道具。
- ある計算問題に対して正当なアルゴリズムは、すべての問題のインスタンスに対して、常に停止して、出力が正しい。
- 「常に停止する」の必要性は、無限時間計算したら正解を出せる手続きもある。イメージとしては極限の収束など。

ソーティング問題

入力 n 個の数の列 $\langle a_1, a_2, \ldots, a_n \rangle$
出力 $a'_1 \leq a'_2 \leq \ldots \leq a'_n$ を満たす入力列の置換（並べ換え）$\langle a'_1, a'_2, \ldots, a'_n \rangle$

問題のインスタンスの例 $\langle 31, 41, 59, 26, 41, 58 \rangle$
正しい出力 $\langle 26, 31, 41, 41, 58, 59 \rangle$

ソーティング問題

入力 n 個の数の列 $\langle a_1, a_2, \ldots, a_n \rangle$
出力 $a'_1 \leq a'_2 \leq \ldots \leq a'_n$ を満たす入力列の置換（並べ換え）$\langle a'_1, a'_2, \ldots, a'_n \rangle$

問題のインスタンスの例 $\langle 31, 41, 59, 26, 41, 58 \rangle$
正しい出力 $\langle 26, 31, 41, 41, 58, 59 \rangle$

- 計算問題は一般的にかかれているのに対して、問題のインスタンスとは一つの具体例

章末問題 1-1 – 実行時間の比較

なぜアルゴリズムの理解が必要なのか？

各関数 $f(n)$ と時間 t に対して、アルゴリズムが問題を解くのに $f(n)$ マイクロ秒かかるとき、t 時間で解くことができる最大の問題サイズ n を求めよ。

	1 秒	1 分	1 時間	1 日	1 ヶ月	1 年	1 世紀
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

2023-04-15

精読『アルゴリズムイントロダクション 第４版』
└─第１章：計算におけるアルゴリズムの役割
└─章末問題 1-1 – 実行時間の比較

章末問題 1-1 – 実行時間の比較
なぜアルゴリズムの理解が必要なのか？

各関数 $f(n)$ と時間 t に対して、アルゴリズムが問題を解くのに $f(n)$ マイクロ秒かかるとき、t 時間で解くことができる最大の問題サイズ n を求めよ。

	１秒	１分	１時間	１日	１ヶ月	１年	１世紀
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

- なぜアルゴリズムの理解が必要なのか？　という問いに対するいい演習はこの問題です
- 要するに、あるアルゴリズムが与えられたとき、そのアルゴリズムの効率はどれぐらい実行時間に影響するか。
- たとえば１分待てる場合、そのアルゴリズムはどれぐらい大きい入力に対して出力できるか
- これは理論上、実際の計算機だと他の要因もあるが

章末問題 1-1 – 実行時間の比較

なぜアルゴリズムの理解が必要なのか？

各関数 $f(n)$ と時間 t に対して、アルゴリズムが問題を解くのに $f(n)$ マイクロ秒かかるとき、t 時間で解くことができる最大の問題サイズ n を求めよ。

	1 秒	1 分	1 時間	1 日	1 ヶ月	1 年	1 世紀
$\lg n$	10^{301}	10^{18061}	$10^{1083707}$	$10^{26008991}$	$10^{780269748}$	$10^{9363236985}$	$10^{936323698513}$
$\sqrt{n}$	10^6	3.6×10^9	1.30×10^{13}	7.46×10^{15}	6.72×10^{18}	9.67×10^{20}	9.67×10^{22}
n	10^3	6×10^4	3.6×10^6	8.64×10^7	2.59×10^9	3.11×10^{10}	3.11×10^{12}
$n \lg n$	140	4895	204094	3.94×10^6	9.77×10^7	1.04×10^9	8.56×10^{10}
n^2	31	244	1897	9295	50911	176363	1.76×10^6
n^3	10	39	153	442	1373	3144	14597
2^n	9	15	21	26	31	34	41
$n!$	6	8	9	11	12	13	15

Section 2

第２章：さあ、始めよう

挿入ソート
直感

- 少数の要素を効率よくソートする
- 手札のトランプカードをソートするイメージ
- 手続き
  - 左手を空にし、テーブルの上にカードを裏向きに置く
  - テーブルから１枚ずつカードを取って、左手の正しい位置に**挿入**していく
  - 正しい位置を探すのに、手札の右側から順に比較していく
- どの時点でも左手の手札はソート済み

挿入ソート

擬似コード

INSERTION-SORT(A, n)

```
for i = 2 to n
  key = A[i]
  // A[i] をソート済みの列 A[1:i-1] に挿入する
  j = i - 1
  while j > 0 かつ A[j] > key
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = key
```

挿入ソート

擬似コード

INSERTION-SORT(A, n)

```
1  for i = 2 to n
2    key = A[i]
3    // A[i] をソート済みの列 A[1:i-1] に挿入する
4    j = i - 1
5    while j > 0 かつ A[j] > key
6      A[j + 1] = A[j]
7      j = j - 1
8    A[j + 1] = key
```

- この教科書は「1-オリジン」
- sort-in-place: その場でソート：入力の配列がソートされ、新しい配列ができない
- 1: i は今配列のどの番号を挿入しようとしているか。$[1 : i - 1]$ はソート済み
- 2: key は今挿入しようとしている「キー」
- 4: j は挿入する場所を探すインデックス
- 5: j を $[1 : i - 1]$ 内で右から左まで挿入する場所を探す
- 6: 挿入する場所ではないとき、配列内容をずらす
- 7: j を左に移す
- 8: while ロープが終わったのは挿入する場所を見つけたこと、キーを挿入する

挿入ソート
実行例

INSERTION-SORT(A, n)

```
for i = 2 to n
  key = A[i]
  // A[i] をソート済みの列 A[1:i-1] に挿入する
  j = i - 1
  while j > 0 かつ A[j] > key
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = key
```

入力：$A = [5, 2, 4, 6], n = 4$

挿入ソート

実行例

INSERTION-SORT(A, n)

```
for i = 2 to n
  key = A[i]
  // A[i] をソート済みの列 A[1:i-1] に挿入する
  j = i - 1
  while j > 0 かつ A[j] > key
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = key
```

状態：$A = [5, 2, 4, 6], i = 2, key = 2, j = 1$ while 条件が満たされる

挿入ソート

実行例

INSERTION-SORT(A, n)

```
for i = 2 to n
  key = A[i]
  // A[i] をソート済みの列 A[1:i-1] に挿入する
  j = i - 1
  while j > 0 かつ A[j] > key
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = key
```

状態：$A = [5, 5, 4, 6], i = 2, key = 2, j = 0$ while 条件が満たされず

挿入ソート

実行例

INSERTION-SORT(A, n)

```
for i = 2 to n
  key = A[i]
  // A[i] をソート済みの列 A[1:i-1] に挿入する
  j = i - 1
  while j > 0 かつ A[j] > key
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = key
```

状態：$A = [2, 5, 4, 6], i = 2, key = 2, j = 0$

挿入ソート

実行例

INSERTION-SORT(A, n)

```
for i = 2 to n
  key = A[i]
  // A[i] をソート済みの列 A[1:i-1] に挿入する
  j = i - 1
  while j > 0 かつ A[j] > key
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = key
```

状態：$A = [2, 5, 4, 6], i = 3, key = 4, j = 2$ while 条件が満たされる

挿入ソート

実行例

INSERTION-SORT(A, n)

```
for i = 2 to n
  key = A[i]
  // A[i] をソート済みの列 A[1:i-1] に挿入する
  j = i - 1
  while j > 0 かつ A[j] > key
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = key
```

状態：$A = [2, 5, 5, 6], i = 3, key = 4, j = 1$ while 条件が満たされず

挿入ソート

実行例

INSERTION-SORT(A, n)

```
for i = 2 to n
  key = A[i]
  // A[i] をソート済みの列 A[1:i-1] に挿入する
  j = i - 1
  while j > 0 かつ A[j] > key
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = key
```

状態：$A = [2, 4, 5, 6], i = 3, key = 4, j = 1$

挿入ソート

実行例

INSERTION-SORT(A, n)

```
for i = 2 to n
  key = A[i]
  // A[i] をソート済みの列 A[1:i-1] に挿入する
  j = i - 1
  while j > 0 かつ A[j] > key
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = key
```

状態：$A = [2, 4, 5, 6], i = 4, key = 6, j = 3$ while 条件が満たされず

ループ不変式

定義

loop invariant – ループ不変式 は反復ごとに変わらない性質のこと。アルゴリズム、特にループの正当性を解析するのに役に立つ。

挿入ソートのループ不変式：$A[i:i-1]$ は元々 $A[i:i-1]$ にある要素であり、ソート済みである

ループ不変式

条件

初期条件 ループの実行開始直前ではループ不変式は真である。

ループ内条件 ループ反復の実行開始直前ではループ不変式が真ならば、次のループ反復の実行開始直前でも真である。

終了条件 ループが停止する。停止したとき、アルゴリズムの正当性の証明に繋がる有力な性質がループ不変式とループ停止理由から得られる。

初期条件	ループの実行開始直前ではループ不変式は真である。
ループ内条件	ループ反復の実行開始直前ではループ不変式が真ならば、次のループ反復の実行開始直前でも真である。
終了条件	ループが停止する。停止したとき、アルゴリズムの正当性の証明に繋がる有力な性質がループ不変式とループ停止理由から得られる。

- 数学的帰納法っぽさがある。

アルゴリズムの解析

- アルゴリズムが必要とする**リソースを予測**する。
- リソース：**実行時間**、メモリ、通信バンド幅、エネルギー
- ランダムアクセスマシン（RAM）
  - 単一プロセッサ
  - 命令実行時間一定
  - メモリアクセス時間一定

アルゴリズムの解析

挿入ソートの例

- 入力の大きさ：配列の長さ
- 解析するリソース：実行時間 $T(n)$
- k 行目の１回実行時間を c_k とする。
- 回数を計算して、総合実行時間を計算する。
- while ループの実行回数は挿入する場所によるため、各 for ループの反復によって異なる。そのため、各 for ループ反復の i に対して、while ループの実行回数を t_i とする。

アルゴリズムの解析

挿入ソートの例

- 入力の大きさ：配列の長さ
- 解析するリソース：実行時間 $T(n)$
- k 行目の1回実行時間を c_k とする。
- 回数を計算して、総合実行時間を計算する。
- while ループの実行回数は挿入する場所によるため、各 for ループの反復によって異なる。そのため、各 for ループ反復の i に対して、while ループの実行回数を t_i とする。

- 実行時間は入力の長さに依存している。
- 実行例を想像してみてください：挿入したい場所を右から順番に探します。見つかったら挿入して while ループから出ます。
- なので、挿入場所の発見が早い、つまり元々正しい場所にある場合は while ループがすぐ終わるし、
- 挿入場所の発見が遅い、つまり一番左まで探さなければならない、つまり元々は逆順に配列されていたら while ループが長いです。

挿入ソート

実行時間解析

INSERTION-SORT(A, n)

		コスト	実行回数
1	`for i = 2 to n`	c_1	n
2	`  key = A[i]`	c_2	$n-1$
3	`  // コメント`	0	$n-1$
4	`  j = i - 1`	c_4	$n-1$
5	`  while j > 0 かつ A[j] > key`	c_5	$\sum_{i=2}^{n} t_i$
6	`    A[j + 1] = A[j]`	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7	`    j = j - 1`	c_7	$\sum_{i=2}^{n} (t_i - 1)$
8	`  A[j + 1] = key`	c_8	$n-1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{i=2}^{n} t_i + (c_6 + c_7) \sum_{i=2}^{n} (t_i - 1)$$

挿入ソート

実行時間解析

最良の場合：元々整列されている $t_i = 1$

$$
\begin{aligned}
T(n) &= c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 (n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8) n - (c_2 + c_4 + c_5 + c_8) \\
&= an + b
\end{aligned}
$$

実行時間は入力の大きさの線形関数である。

挿入ソート

実行時間解析

最悪の場合：元々は逆順に整列されている $t_i = i$

$$
\begin{aligned}
T(n) = \;& c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + (c_6 + c_7) \left(\frac{n(n-1)}{2} \right) \\
= \;& \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\
& - (c_2 + c_4 + c_5 + c_8) \\
= \;& an^2 + bn + c
\end{aligned}
$$

実行時間は入力の大きさの線形関数である。

最悪時の解析

- 通常は**最悪時**の解析を行う
- 乱択アルゴリズムなどは**平均時**の解析を行う

最悪時の解析

- 通常は**最悪時**の解析を行う
- 乱択アルゴリズムなどは**平均時**の解析を行う

- 最悪時の理由は、解析結果は上界となる、アルゴリズムによって再悪事がよく出る、平均時も最悪時と変わらないことが多い。

「オーダ」

- 実行時間は、入力の大きさ n に関する項の和のとき、一番増加オーダが高いもので特徴づけられる。
- $T(n) = 148n^2 + 4n \lg n + 43\sqrt{n} + 345$ を $T(n) = \Theta(n^2)$ と特徴づけられる。

2023-04-15

精読『アルゴリズムイントロダクション 第 4 版』
└─第 2 章：さあ、始めよう
└─「オーダ」

「オーダ」

- 実行時間は、入力の大きさ n に関する項の和のとき、一番増加オーダが高いもので特徴づけられる。
- $T(n) = 148n^2 + 4n \lg n + 43\sqrt{n} + 345$ を $T(n) = \Theta(n^2)$ と特徴づけられる。

- オーダ n 二乗
- これは「ビッグ・オー」じゃなくて、「ビッグ・シータ」
- オーダ記法はいくつかあって（この教科書では５つ紹介）、それぞれ定義があって、第３章で紹介する。
- それまでインフォーマルに使っている。

「オーダ」

- $T_1(n) = n^2$ と $T_2(n) = 1000n \lg n$ を比べよう。

「オーダ」

- $T_1(n) = n^2$ と $T_2(n) = 1000n \lg n$ を比べよう。
- T_1 はいずれ T_2 を追い越す。具体的にこの場合 13746 で追い越す。

「オーダ」

- $T_1(n) = n^2$ と $T_2(n) = 1000n \lg n$ を比べよう。
- T_1 はいずれ T_2 を追い越す。具体的にこの場合 13746 で追い越す。

- なぜ最大項だけを気にするのか？
- 両方とも極限をとれば無限大に増加するが、最大項はいくら係数が小さくても、必ず他の項を追い越す。
- もちろん、係数を小さくする努力もすべきだが、オーダを小さくする努力のほうが勝る

分割統治法

- 挿入ソート：逐次添加法（incremental）
- マージソート：分割統治法（divide-and-conquer）

  分割　問題をいくつかの同じ問題のより小さいインスタンスである部分問題に分割。

  統治　部分問題を再帰的に解く。ただし、部分問題のサイズが十分小さいときは直接的な方法で解く。

  統合　部分問題の解を組み合わせて元の問題の解を得る。

分割統治法

- 挿入ソート：逐次添加法（incremental）
- マージソート：分割統治法（divide-and-conquer）

分割 問題をいくつかの同じ問題のより小さいインスタンスである部分問題に分割。

統治 部分問題を再帰的に解く。ただし、部分問題のサイズが十分小さいときは直接的な方法で解く。

統合 部分問題の解を組み合わせて元の問題の解を得る。

- 逐次添加法は１個ずつ見る
- 分割統治法は小さい問題に分割して、解決して、統合する
- ポイントは「同じ問題のより小さいインスタンス」と「サイズが十分小さいときは直接的な方法で解く」ところ

マージソート

分割 ソートすべき長さ n の列を $n/2$ の部分列に分割する。

統治 マージソートを用いて２つの部分列を再帰的にソートする。

統合 ２つのソートされた部分列をマージしてソートされた解を作る。

分割　ソートすべき長さ n の列を $n/2$ の部分列に分割する。
統治　マージソートを用いて２つの部分列を再帰的にソートする。
統合　２つのソートされた部分列をマージしてソートされた解を作る。

- このアルゴリズムのキモは**統合**の部分にある。

マージソート

マージの直感： ソート済みの 2 つの部分列の先頭を比べつつ、小さい方を順に元の列の解を作っていく。

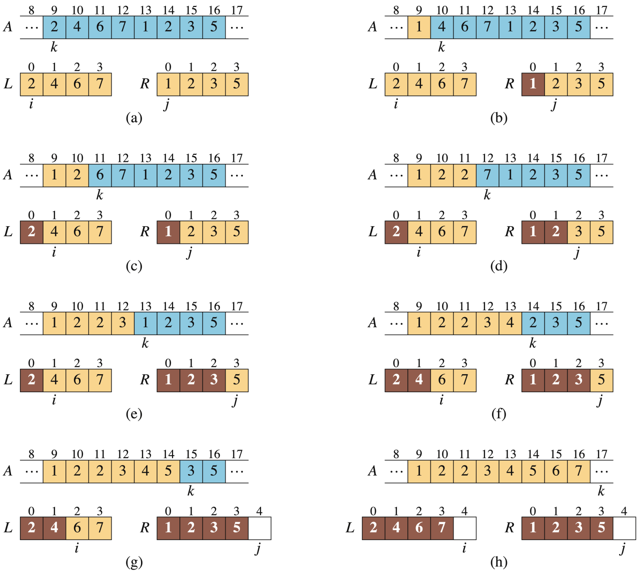
マージソート

マージの擬似コード

MERGE(A, p, q, r)

```
n_L = q - p + 1                 // A[p:q] の長さ
n_R = r - q                     // A[q + 1:r] の長さ
L[0:n_L - 1] と R[0:n_R - 1] を新しい配列とする
for i = 0 to n_L - 1            // A[p:q] を L[0:n_L - 1] へコピー
  L[i] = A[p + i]
for j = 0 to n_R - 1            // A[q + 1:r] を R[0:n_R - 1] へコピー
  R[j] = A[q + j + 1]
i = 0
j = 0
k = p
while i < n_L かつ j < n_R  // L と R ともマージされていない要素があるかぎり、最小の要素を A に挿入
  if L[i] <= R[j]
    A[k] = L[i]
    i = i + 1
  else A[k] = R[j]
    j = j + 1
  k = k + 1
while i < n_L                   // L に要素が残っている場合、A に挿入
  A[k] = L[i]
  i = i + 1
  k = k + 1
while j < n_R
  A[k] = R[j]                   // R に要素が残っている場合、A に挿入
  j = j + 1
  k + k + 1
```

マージソート



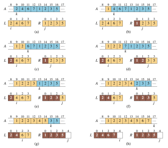
A … 2 4 6 7 1 2 3 5 … k
L 2 4 6 7 i R 1 2 3 5 j
(a)
A … 1 4 6 7 1 2 3 5 … k
L 2 4 6 7 i R 1 2 3 5 j
(b)
A … 1 2 6 7 1 2 3 5 … k
L 2 4 6 7 i R 1 2 3 5 j
(c)
A … 1 2 2 7 1 2 3 5 … k
L 2 4 6 7 i R 1 2 3 5 j
(d)
A … 1 2 2 3 1 2 3 5 … k
L 2 4 6 7 i R 1 2 3 5 j
(e)
A … 1 2 2 3 4 2 3 5 … k
L 2 4 6 7 i R 1 2 3 5 j
(f)
A … 1 2 2 3 4 5 3 5 … k
L 2 4 6 7 i R 1 2 3 5 j
(g)
A … 1 2 2 3 4 5 6 7 … k
L 2 4 6 7 i R 1 2 3 5 j
(h)

2023-04-15

精読『アルゴリズムイントロダクション 第４版』
└─第２章：さあ、始めよう

└─マージソート

マージソート



- このマージの手続きは、２分割された配列が与えられ、それぞれの分割された配列はソート済みである。これはマージの入力です。
- アルゴリズムの７行目まで、それぞれを別々の配列に入れて、１０行目まで、準備が終わり、図 a になります。
- １２行目から１８行目までの while ループはアルゴリズムのキモです。i と j が指している数を比べて、小さい方を配列 A に入れていく
- 図 a のところは、i が指している 2 と j が指している 1 を比べた。1 が小さいので 1 を配列 A の k 番目に入れる。小さい方の指している j を 1 増やして、A のインデックス k を 1 増やして、この反復が終わる。
- こうやっていくと、j が配列の大きさから出てしまったところに、このループが終わる。
- 終わったところは、図 g になりますね。
- あとは、残っているものを配列 A に入れてマージが終了します。

マージソート

マージの解析

MERGE(A, p, q, r)

```
n_L = q - p + 1
n_R = r - q
L[0:n_L - 1] と R[0:n_R - 1] を新しい配列とする      // Θ (n)
for i = 0 to n_L - 1                       // Θ (n)
  L[i] = A[p + i]
for j = 0 to n_R - 1                       // Θ (n)
  R[j] = A[q + j + 1]
i = 0
j = 0
k = p
while i < n_L かつ j < n_R                  // Θ (n)
  if L[i] <= R[j]
    A[k] = L[i]
    i = i + 1
  else A[k] = R[j]
    j = j + 1
  k = k + 1
while i < n_L                              // Θ (n)
  A[k] = L[i]
  i = i + 1
  k = k + 1
while j < n_R                              // Θ (n)
  A[k] = R[j]
  j = j + 1
  k + k + 1
```

マージソート

擬似コード

MERGE-SORT(A, p, r)

```
if p >= r                          // 要素が 0 または 1 個
  return
q = floor((p + r)/2)               // A[p:r] の真ん中
MERGE-SORT(A, p, q)                // A[p:q] を再帰的にソートする
MERGE-SORT(A, q + 1, r)            // A[q + 1:r] を再帰的にソートする
// それぞれソート済みの A[p:q] と A[q + 1:r] を A[p:r] にマージする
MERGE(A, p, q, r)
```

2023-04-15

精読『アルゴリズムイントロダクション 第４版』
└─第２章：さあ、始めよう
└─マージソート

マージソート

擬似コード

MERGE-SORT(A, p, r)

```
if p >= r                    // 要素が 0 または 1 個
  return
q = floor((p + r)/2)         // A[p:r] の真ん中
MERGE-SORT(A, p, q)          // A[p:q] を再帰的にソートする
MERGE-SORT(A, q + 1, r)      // A[q + 1:r] を再帰的にソートする
// それぞれソート済みの A[p:q] と A[q + 1:r] を A[p:r] にマージする
MERGE(A, p, q, r)
```

- まずベースケースを処理する。要素が１以下の場合、配列がソート済みなので終了。
- 中間点を算出する。
- 分割された配列をそれぞれ再帰的にソートする。
- ソートが終わったらマージ作業をする。

分割統治法

分割 問題をいくつかの同じ問題のより小さいインスタンスである部分問題に分割。

統治 部分問題を再帰的に解く。ただし、部分問題のサイズが十分小さいときは直接的な方法で解く。

統合 部分問題の解を組み合わせて元の問題の解を得る。

分割統治法

分割　問題をいくつかの同じ問題のより小さいインスタンスである部分問題に分割。
統治　部分問題を再帰的に解く。ただし、部分問題のサイズが十分小さいときは直接的な方法で解く。
統合　部分問題の解を組み合わせて元の問題の解を得る。

- では、分割統治法の解析に入りたいと思います。
- まず、分割統治法はどのようなパラダイムか、もう一回復習します。

分割統治法

問題のインスタンスの大きさが n のとき

- **分割** 問題を a 個の同じ問題の大きさ n/b インスタンスである部分問題に分割。
- **統治** 部分問題を再帰的に解く。ただし、部分問題のサイズ n' が $n < n_0$ のときは直接的な方法で解く。
- **統合** 部分問題の解を組み合わせて元の問題の解を得る。

分割統治法

問題のインスタンスの大きさが n のとき

- 分割 問題を a 個の同じ問題の大きさ n/b インスタンスである部分問題に分割。
- 統治 部分問題を再帰的に解く。ただし、部分問題のサイズ n' が $n < n_0$ のときは直接的な方法で解く。
- 統合 部分問題の解を組み合わせて元の問題の解を得る。

- では、もっと具体的に

分割統治法

問題のインスタンスの大きさが n のとき、実行時間が $T(n)$

- **分割** 問題を a 個の同じ問題の大きさ n/b インスタンスである部分問題に分割。$D(n)$
- **統治** 部分問題を再帰的に解く aT(n/b)。ただし、部分問題のサイズ n' が $n' < n_0$ のときは直接的な方法で解く $\Theta(1)$。
- **統合** 部分問題の解を組み合わせて元の問題の解を得る。$C(n)$

$T(n)$ がこのような漸化式になる。

$$T(n) = \begin{cases} \Theta(1) & n < n_0 \text{のとき} \\ D(n) + aT(n/b) + C(n) & \text{それ以外のとき} \end{cases}$$

分割統治法

問題のインスタンスの大きさが n のとき、実行時間が $T(n)$

- **分割** 問題を a 個の同じ問題の大きさ n/b インスタンスである部分問題に分割。$D(n)$
- **統治** 部分問題を再帰的に解く $aT(n/b)$。ただし、部分問題のサイズ n' が $n' < n_0$ のときは直接的な方法で解く $\Theta(1)$。
- **統合** 部分問題の解を組み合わせて元の問題の解を得る。$C(n)$

$T(n)$ がこのような漸化式になる。

$$T(n) = \begin{cases} \Theta(1) & n < n_0 \text{のとき} \\ D(n) + aT(n/b) + C(n) & \text{それ以外のとき} \end{cases}$$

- それぞれの要素の実行時間はこのように数式化できる。

分割統治法

マージソートの場合

問題のインスタンスの大きさが n のとき、実行時間が $T(n)$

- **分割** 問題を a 個の同じ問題の大きさ n/b インスタンスである部分問題に分割。$D(n) = \Theta(1)$
- **統治** 部分問題を再帰的に解く $\mathrm{aT(n/b)} = \mathrm{2T(n/2)}$。ただし、部分問題のサイズ n' が $n < n_0$ のときは直接的な方法で解く $\Theta(1)$。
- **統合** 部分問題の解を組み合わせて元の問題の解を得る。$C(n) = \Theta(n)$

$T(n)$ がこのような漸化式になる。

$$T(n) = \begin{cases} c_1 & n = 1\text{のとき} \\ 2T(n/2) + c_2 n & n > 1\text{のとき} \end{cases}$$

分割統治法

マージソートの場合

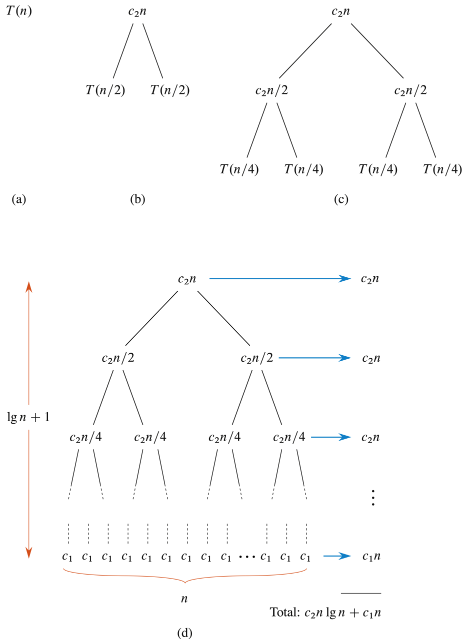
問題のインスタンスの大きさが n のとき、実行時間が $T(n)$

- **分割** 問題を a 個の同じ問題の大きさ n/b インスタンスである部分問題に分割。$D(n) = \Theta(1)$
- **統治** 部分問題を再帰的に解く $aT(n/b) = 2T(n/2)$。ただし、部分問題のサイズ n' が $n < n_0$ のときは直接的な方法で解く $\Theta(1)$。
- **統合** 部分問題の解を組み合わせて元の問題の解を得る。$C(n) = \Theta(n)$

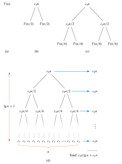
$T(n)$ がこのような漸化式になる。

$$T(n) = \begin{cases} c_1 & n = 1のとき \\ 2T(n/2) + c_2 n & n > 1のとき \end{cases}$$

- マージソートの場合はこうなります。
- この漸化式を数学的に解くマスター法は「第４章：分割統治」で紹介されるが、ここではまだやりません。



$T(n)$
(a)
$c_2 n$
$T(n/2)$ $T(n/2)$
(b)
$c_2 n$
$c_2 n/2$ $c_2 n/2$
$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$
(c)
$c_2 n$ → $c_2 n$
$c_2 n/2$ $c_2 n/2$ → $c_2 n$
$c_2 n/4$ $c_2 n/4$ $c_2 n/4$ $c_2 n/4$ → $c_2 n$
$\lg n + 1$
c_1 c_1 c_1 c_1 c_1 c_1 c_1 c_1 c_1 $\cdots$ c_1 c_1 c_1 → $c_1 n$
n
Total: $c_2 n \lg n + c_1 n$
(d)



- イメージとしてはこれですね。
- a は元の問題です。
- 分割して、b になりますが、分割するのに $c_2 n$ 時間を要します。そして、分割された小さい問題は $T(n/2)$ 時間です。
- これを繰り返して、ベースケースまで分割すると、図 d のようになります。
- 各層の分割は合わせて $c_2 n$ 時間、分割する回数は $\lg n$ なので、全部分割に要する時間は $c_2 n \lg n$ です。
- そして、ベースケースはそれぞれ c_1 時間を必要として、すべて n 個なので、$c_1 n$ 時間を必要とします。
- 合わせて $c_2 n \lg n + c_1 n$ ですが、特徴づけは $\Theta(n \lg n)$ です。

Section 3

第３章：実行時間の特徴づけ

漸近記法

直感

- 無限大に増加する関数の、増加の「速度」を測りたい。
- 関数 $F(n)$ が n に関する項の和のとき、一番増加オーダが高いもので特徴づけられる。
- 例：$F(n) = 7n^3 + 100n^2 - 20n + 6$ が n^3 で特徴づけられる。
- $F(n)$ は漸近的に n^3 の比例に増加する。
- $F(n)$ は漸近的に n^4 より増加が遅い。
- $F(n)$ は漸近的に n^2 より増加が速い。

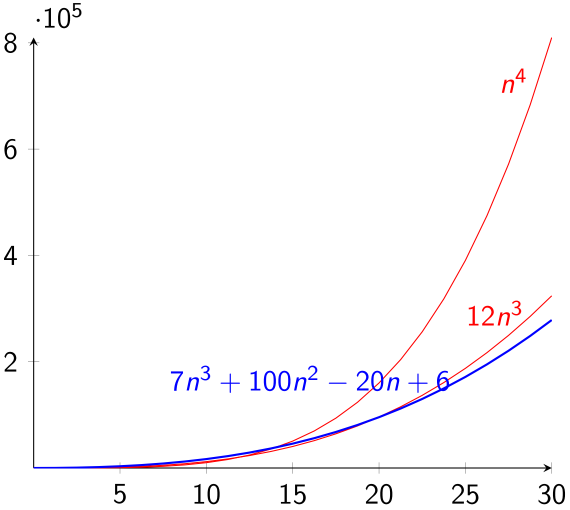
漸近記法

直感

- 例：$F(n) = 7n^3 + 100n^2 - 20n + 6$
- O **記法**は増加速度の**上界**を表現する。
  $F(n)$ の増加速度は n^3 や n^4 の増加速度より速くないため、$F(n)$ は $O(n^3)$ であり $O(n^4)$ である。
- Ω **記法**は増加速度の**下界**を表現する。
  $F(n)$ の増加速度は n^3 や n^2 の増加速度より遅くないため、$F(n)$ は $\Omega(n^3)$ であり $\Omega(n^2)$ である。
- Θ **記法**は増加速度の**タイトな限界**を表現する。
  $F(n)$ の増加速度は n^3 の増加速度と漸近的に比例しているため、$F(n)$ は $\Theta(n^3)$ である。

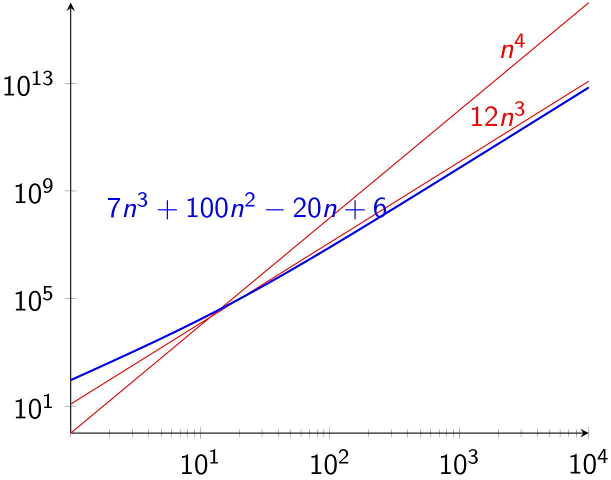
漸近記法

O-直感



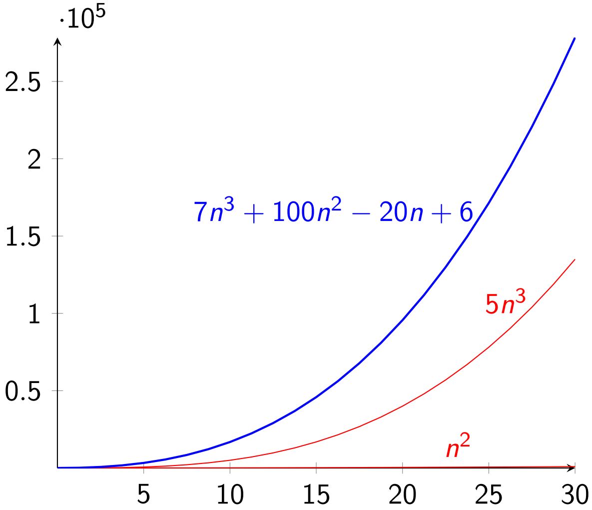
漸近記法

O-直感



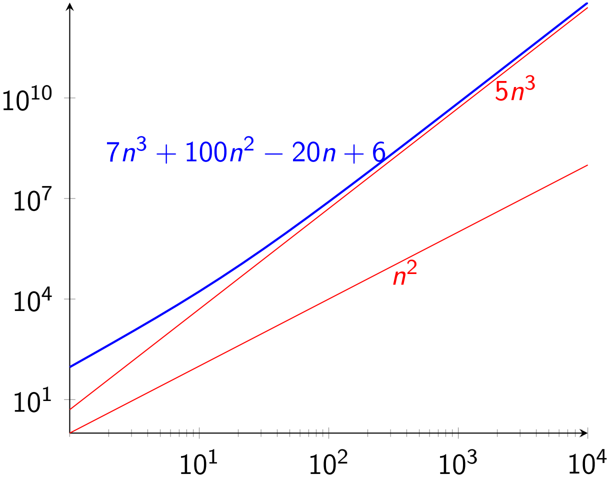
漸近記法

Ω-直感



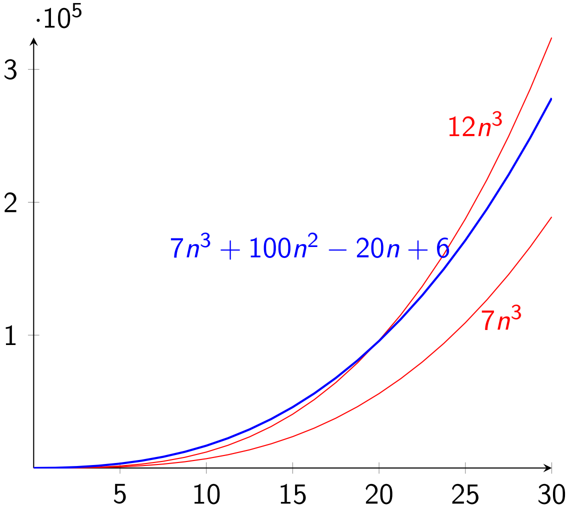
漸近記法

Ω-直感



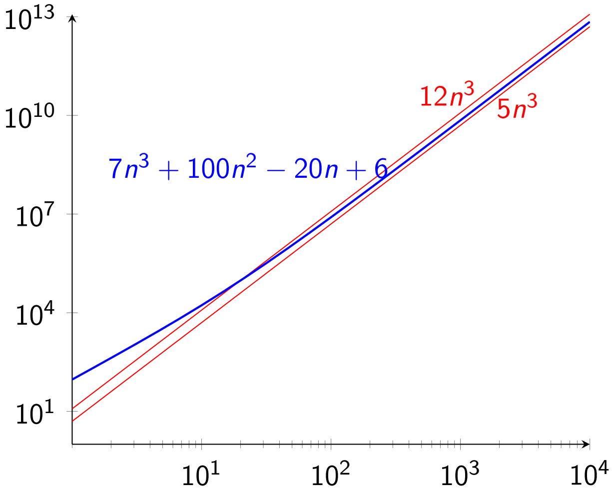
漸近記法

Θ-直感

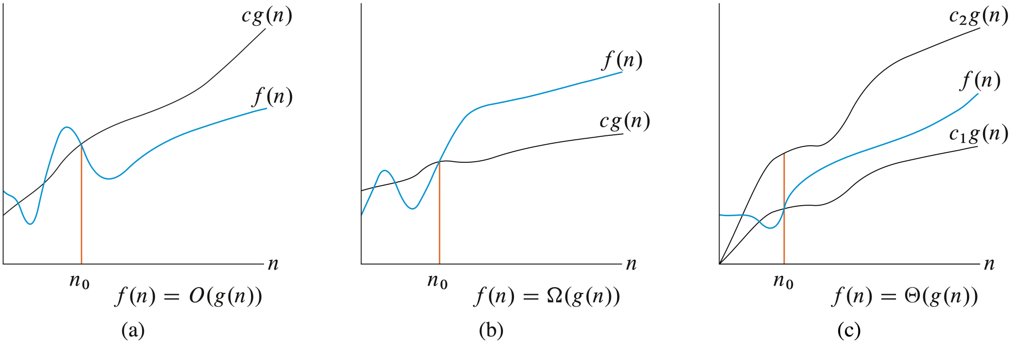


漸近記法

Θ-直感



漸近記法



$cg(n)$
$f(n)$
n
n_0
$f(n) = O(g(n))$
(a)
$f(n)$
$cg(n)$
n
n_0
$f(n) = \Omega(g(n))$
(b)
$c_2 g(n)$
$f(n)$
$c_1 g(n)$
n
n_0
$f(n) = \Theta(g(n))$
(c)

漸近記法

定義

Definition (O 記法 – 漸近的上界)

任意な関数 $g(n)$ に対して、関数の集合 $O(g(n))$ はこのように定義する：

$$O(g(n)) = \{f(n) : \text{ある正の定数 } c, n_0 \text{ が存在して、すべての } n \geq n_0 \text{ に対して } 0 \leq f(n) \leq cg(n)\}$$

$f(n) \in O(g(n))$ を表したいとき、$f(n) = O(g(n))$ と書いてもよいとする。

Definition (Ω 記法 – 漸近的下界)

任意な関数 $g(n)$ に対して、関数の集合 $\Omega(g(n))$ はこのように定義する：

$$\Omega(g(n)) = \{f(n) : \text{ある正の定数 } c, n_0 \text{ が存在して、すべての } n \geq n_0 \text{ に対して } 0 \leq cg(n) \leq f(n)\}$$

$f(n) \in \Omega(g(n))$ を表したいとき、$f(n) = \Omega(g(n))$ と書いてもよいとする。

漸近記法

定義

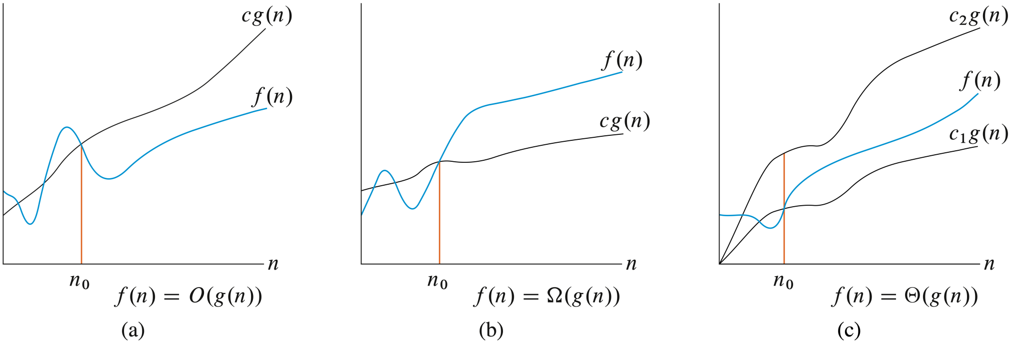
Definition (Θ 記法 – 漸近的にタイトな限界)

任意な関数 $g(n)$ に対して、関数の集合 $\Theta(g(n))$ はこのように定義する：

$$\Theta(g(n)) = \{f(n) : \text{ある正の定数 } c_1, c_2, n_0 \text{ が存在して、すべての } n \geq n_0 \text{ に対して}$$
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$f(n) \in \Theta(g(n))$ を表したいとき、$f(n) = \Theta(g(n))$ と書いてもよいとする。

漸近記法

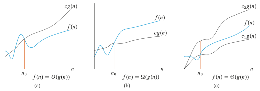


$cg(n)$
$f(n)$
n
n_0
$f(n) = O(g(n))$
(a)
$f(n)$
$cg(n)$
n
n_0
$f(n) = \Omega(g(n))$
(b)
$c_2 g(n)$
$f(n)$
$c_1 g(n)$
n
n_0
$f(n) = \Theta(g(n))$
(c)

2023-04-15

精読『アルゴリズムイントロダクション 第4版』
└─第３章：実行時間の特徴づけ
└─漸近記法

漸近記法



- この章では漸近記法に関する定理とよく使われる公式などまとめられている
- 詳しくは教科書を見てください