1-7　アルゴリズム

東京大学 数理・情報教育研究センター

2021年4月6日

東京大学 数理・情報教育研究センター 小林浩二 2021 CC BY-NC-SA

概要

- データ・AI利活用に必要なアルゴリズムの基礎を学び、代表的なアルゴリズムである探索と並べ替えについて学習します。

東京大学 数理・情報教育研究センター 小林浩二 2021 CC BY-NC-SA

本教材の目次

東京大学 数理・情報教育研究センター 小林浩二 2021 CC BY-NC-SA

アルゴリズムとは

- アルゴリズムとは、「問題」を解くための手順を意味します。
  - アルゴリズムを「問題」に対する解法と理解しても良いでしょう
- 様々な「問題」が考えられます。例えば、次の様な「問題」を挙げることができます。
  - 与えられた2つの数の和を求める（2つの数の足し算を行う）
  - データの集まりの中からあるデータを探し出す
  - 数を小さい順（昇順）に並べる
  - 文章の中から特定の文字列を抜き出す
  - 数が素数かどうか調べる

 CC BY-NC-SA

アルゴリズムの例

- アルゴリズムの例を考えます。例として、次の様な「問題」を考えます。
- 問題1：「10と5という2つの数が与えられたとき、その和を求める」
- この問題に対するアルゴリズムは例えば、次の様になります。
  - アルゴリズム1：

    手順1：10+5 を計算して、その計算結果をzに代入する

    手順2：zを求めるべき結果として出力してアルゴリズムを終了する

- 同じ問題でも様々なアルゴリズムが考えられます。例えば、この問題では次の様なアルゴリズムも同じ結果となります。
  - アルゴリズム1改：

    手順1：15 を出力してアルゴリズムを終了する

 CC BY-NC-SA

アルゴリズムの例

- もう少しアルゴリズムの例を見ていきます。
- 問題2：「2つの数が与えられたとき、その和を求める」
- この問題に対するアルゴリズムは例えば次の様になります。
  - アルゴリズム2：

    手順1：1つの目の数であるxを決める（数を入力して、xに代入する）

    手順2：2つの目の数であるyを決める（数を入力して、yに代入する）

    手順3：x+y を計算して、その計算結果をzに代入する

    手順4：zを出力してアルゴリズムを終了する

 CC BY-NC-SA

アルゴリズムの例

- もう少しアルゴリズムの例を見ていきます。
- 問題3：「サイコロを5回振って、出た目の和（合計）を求める」
- この問題に対するアルゴリズムは例えば、次の様になります。
  - アルゴリズム3：

    手順1：サイコロを振って、出た目をzに代入する

    手順2：サイコロを振って、出た目をzに加える

    手順3：サイコロを振って、出た目をzに加える

    手順4：サイコロを振って、出た目をzに加える

    手順5：サイコロを振って、出た目をzに加える

    手順6：zを出力してアルゴリズムを終了する

 CC BY-NC-SA

アルゴリズムの例

- アルゴリズム3が冗長に感じる場合、次の様なアルゴリズムも考えられます。
  - アルゴリズム3改：

    手順1：z=0とし、c=0とする（サイコロを振った数を変数cで数える）

    手順2：サイコロを振って、出た目をzに加える

    手順3：cの値を1増やす

    手順4：cが5ならzを出力してアルゴリズムを終了する。そうでなければ、手順2に戻る

 CC BY-NC-SA

アルゴリズムの例

- 問題4：「偶数の目が出るまでサイコロを振って、出た目の総和を出力する」
- この問題に対するアルゴリズムは例えば次の様になります。
  - アルゴリズム4：

    手順1：z=0とする（出た目をzに加えていく）

    手順2：サイコロを振って、出た目をxに代入し、xをzに加える

    手順3：xが偶数ならzを出力してアルゴリズムを終了する。そうでなければ、手順2に戻る

 CC BY-NC-SA

アルゴリズムの例

- 問題5：「1つの整数が与えられたとき、その数が2,3,6のいずれかの倍数か判定する」
- この問題に対するアルゴリズムは例えば次の様になります。
  - アルゴリズム5：

    手順1：整数を入力して、xに代入する

    手順2：xが2で割り切れれば、手順3へ。そうでなければ手順4へ

    手順3：xが3で割り切れれば、「xは6の倍数」と出力して終了。そうでなければ「xは2の倍数」と出力して終了

    手順4：xが3で割り切れれば、「xは3の倍数」と出力して終了。そうでなければ「xは2,3,6の倍数ではない」と出力して終了

 CC BY-NC-SA

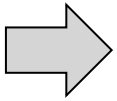
フローチャートとは

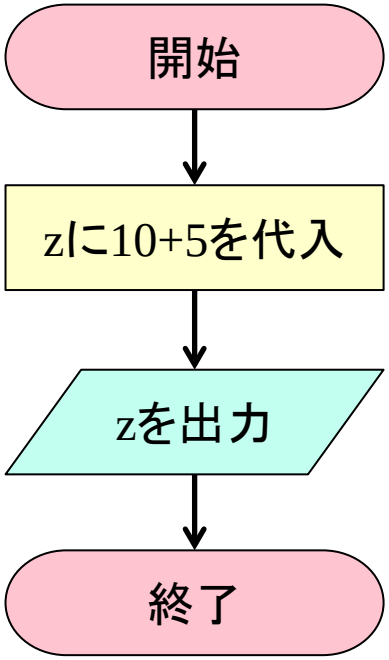
- アルゴリズムは、前頁までの様に言葉だけで記述しても構わないのですが、アルゴリズムの表現の1つとしてフローチャートと呼ばれる図（流れ図とも呼びます）を用いてアルゴリズムの手順を直感的に分かり易くするために可視化する方法があります。
- フローチャートの定義は後で説明します。まず、前述のアルゴリズムをフローチャートで表してみます。

アルゴリズム1：

手順1：10+5 を計算して、その計算結果をzに代入する

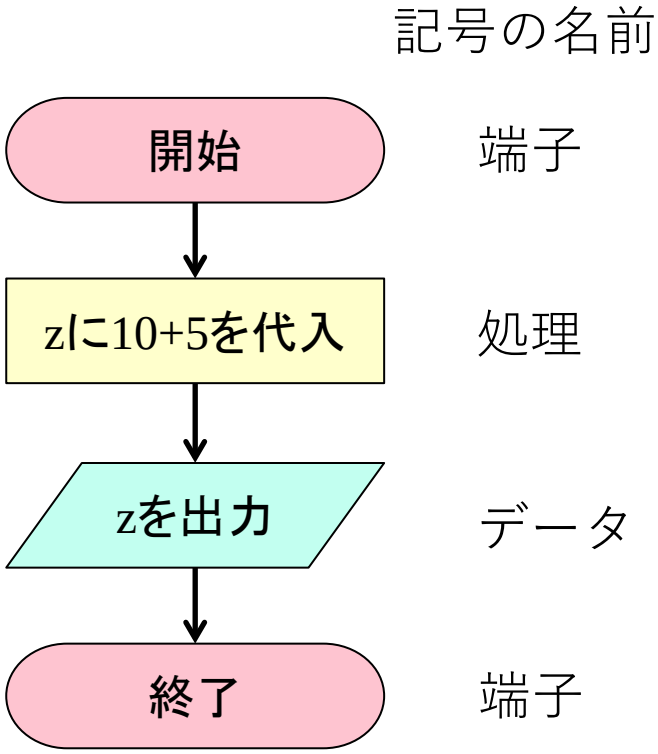
手順2：zを求めるべき結果として出力してアルゴリズムを終了する





フローチャートとは

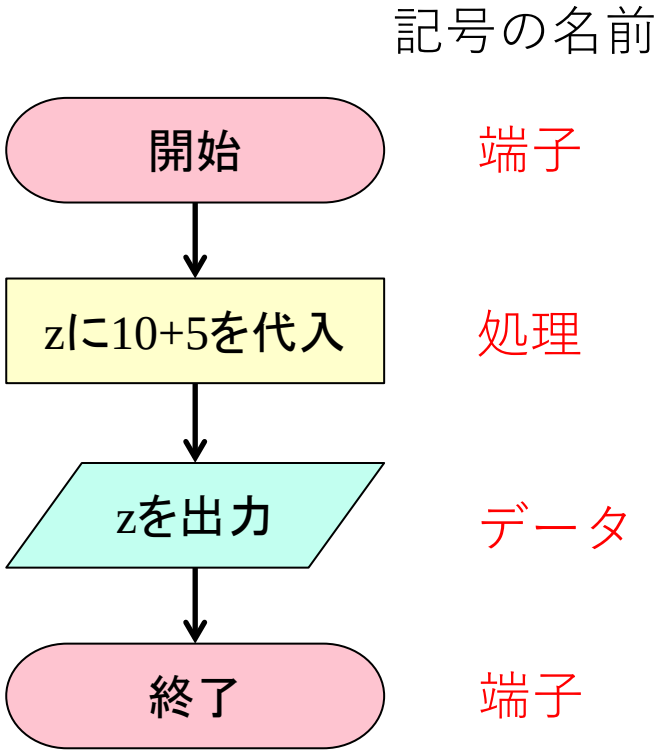
- フローチャートは、幾つかの記号を矢印で繋いだ図で表します。
  - この教材ではJISの規格にそったフローチャートを紹介します。
  - 記号の色はこの教材独自のもので一般的なものではありません。
- アルゴリズムの手続きの種類毎に用いるべき記号が定められています。
  - 様々な手続き（記号）が存在しますが、この教材では代表的なもののみ紹介します。
- 各手続きを表す記号を矢印で繋いで、手続きの順番を表します。



 CC BY-NC-SA

フローチャートとは

- 基本的な記号を説明します。
- 端子：フローチャート、すなわち、アルゴリズムの開始と終了を主に表します。
- 処理：演算や変数への代入などあらゆる処理を表します。
- データ：値（データ）の入出力を表します。右の例では、出力を表しています。



フローチャートの例

- 別のフローチャートの例です。

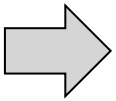
アルゴリズム2：

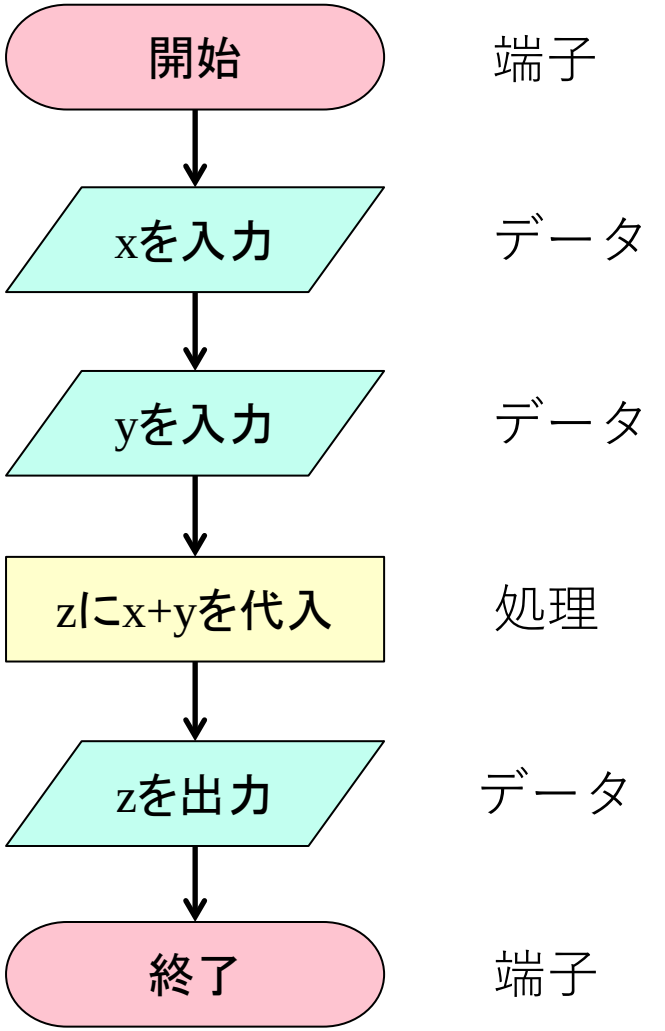
手順1：1つの目の数であるxを決める（数を入力して、xに代入する）

手順2：2つの目の数であるyを決める（数を入力して、yに代入する）

手順3：x+y を計算して、その計算結果をzに代入する

手順4：zを出力してアルゴリズムを終了する





 CC BY-NC-SA

フローチャートの例

- 別のフローチャートの例です。

アルゴリズム3：

手順1：サイコロを振って、出た目をzに代入する

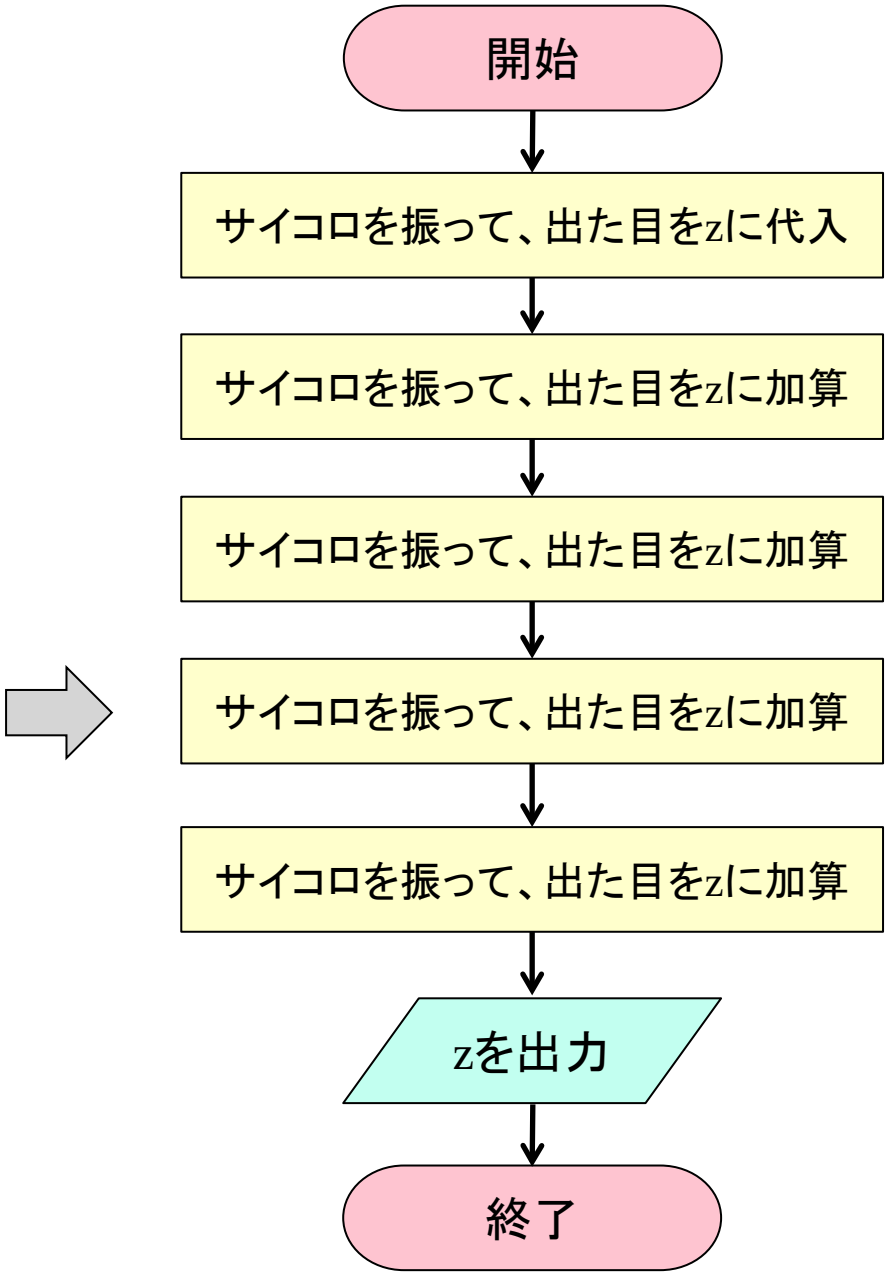
手順2：サイコロを振って、出た目をzに加える

手順3：サイコロを振って、出た目をzに加える

手順4：サイコロを振って、出た目をzに加える

手順5：サイコロを振って、出た目をzに加える

手順6：zを出力してアルゴリズムを終了する



 CC BY-NC-SA

フローチャートの例

- 別のフローチャートの例です。

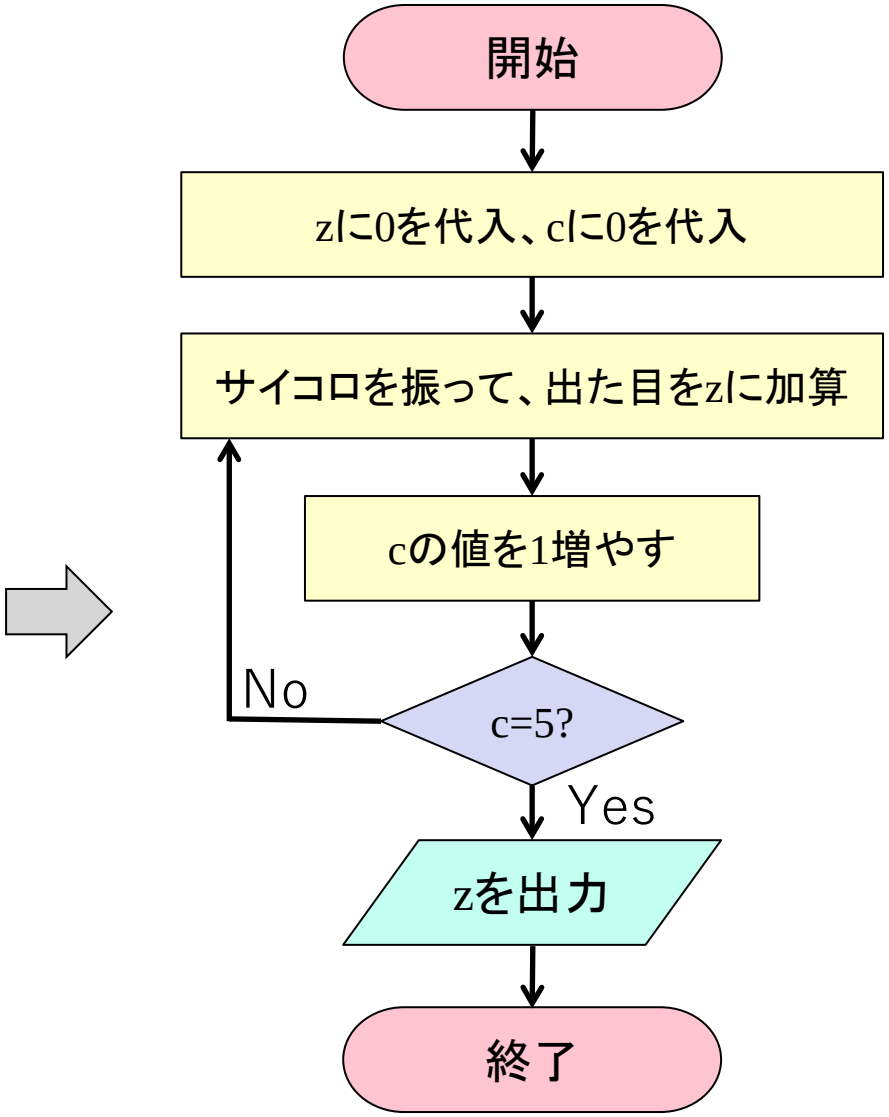
アルゴリズム3改：

手順1：z=0とし、c=0とする

手順2：サイコロを振って、出た目をzに加える

手順3：cの値を1増やす

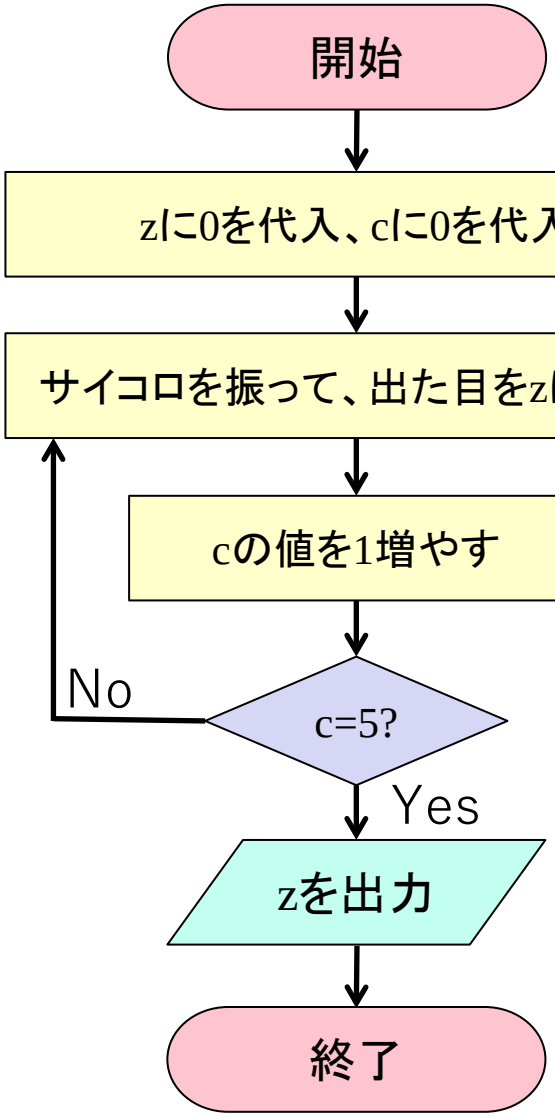
手順4：cが5ならzを出力してアルゴリズムを終了する。そうでなければ、手順2に戻る



東京大学 数理・情報教育研究センター 小林浩二 2021 CC BY-NC-SA

フローチャートとは

- 新しい記号が出てきたので説明します。
- 判断：手続きの分岐を行うための端子です。条件を記述し、その条件が満たされる場合と満たされない場合とで分岐します。右の例の判断では、cが5であれば、zを出力する手続きへ進み、そうでなければ、サイコロを振る手続きに戻ります



記号の名前

端子
処理
処理
処理
判断
データ
端子

東京大学 数理・情報教育研究センター 小林浩二 2021 CC BY-NC-SA

フローチャートの例

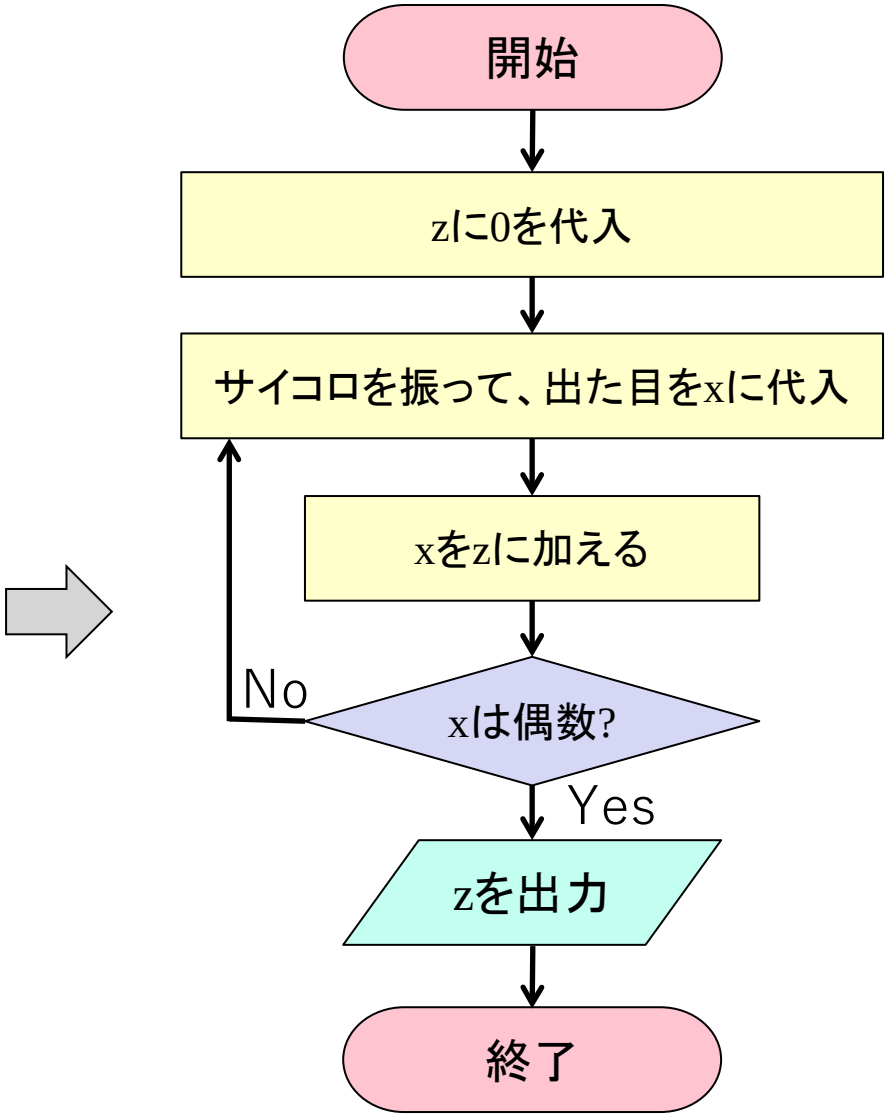
- 別のフローチャートの例です。

アルゴリズム4：

手順1：z=0とする

手順2：サイコロを振って、出た目をxに代入し、xをzに加える

手順3：xが偶数ならzを出力してアルゴリズムを終了する。そうでなければ、手順2に戻る



フローチャートの例

- 別のフローチャートの例です。

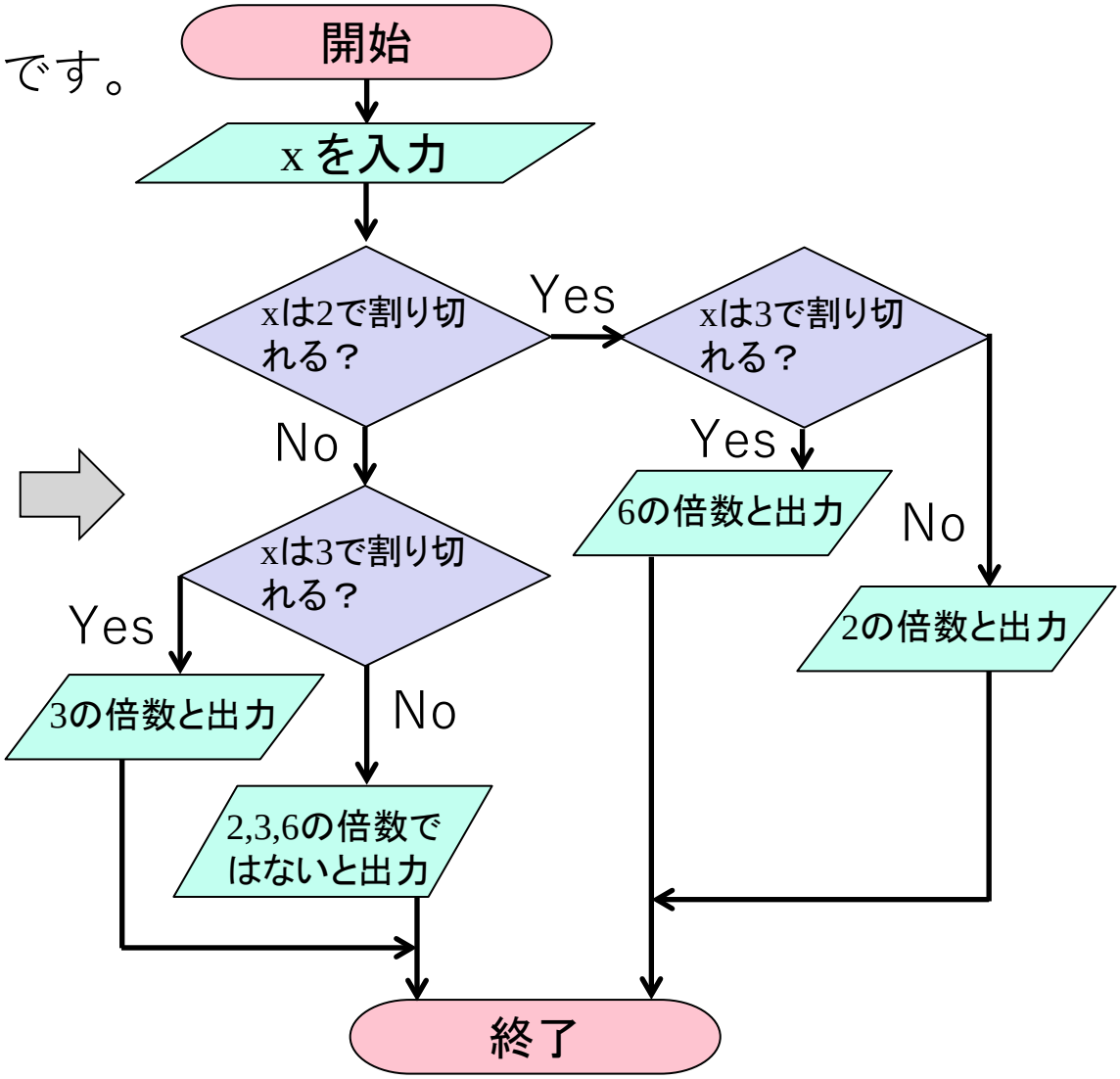
アルゴリズム5：

手順1：整数を入力して、xに代入する

手順2：xが2で割り切れれば、手順3へ。そうでなければ手順4へ

手順3：xが3で割り切れれば、「xは6の倍数」と出力して終了。そうでなければ「xは2の倍数」と出力して終了

手順4：xが3で割り切れれば、「xは3の倍数」と出力して終了。そうでなければ「xは2,3,6の倍数ではない」と出力して終了



代表的なアルゴリズム

- アルゴリズムについて説明した頁において、紹介した以下の2つの問題は代表的な問題であり、この問題を解くための様々なアルゴリズが考案されています。
  - 探索（サーチ）：データの集まりの中から、あるデータを探し出す
  - 並べ替え（ソート）：数を小さい順（昇順）に並べる
- これからそれぞれの問題に対する代表的なアルゴリズムを紹介します。

東京大学 数理・情報教育研究センター 小林浩二 2021 CC BY-NC-SA

探索のアルゴリズム

- 探索（サーチ）は、データの集まりの中からあるデータを探し出す問題です。
  - 例えば、数の集まりの中から特定の値（例えば、最小値、最大値、ある値以上の値など）を見つけたり、文字列の中から特定の文字列を見つけたりすることは探索の問題とみなせます。
  - あとで説明する並べ替えのアルゴリズムでも探索が利用されています。（選択ソートで最小値を探したり、挿入ソートで数の適切な挿入位置を探したりします）
- 本教材では、以下の代表的な探索アルゴリズムを説明します。
  - リスト探索
  - 木探索

 CC BY-NC-SA

リスト探索

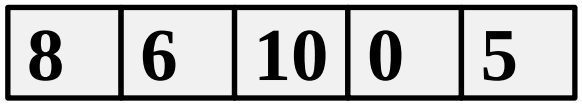
- リスト探索は、順序付けられたデータ（値）の中から目的の値を探す（探索を行う）問題です。
- 「順序付けられたデータ」はやや複雑な表現ですが、要するに「○番目のデータ」と呼ぶことができる形式で保管されているデータを考えるということです。この教材では「配列」と呼ばれるデータ構造に格納されていると考えます。
  - この教材では下の図の様な、長方形を繋げた図で配列を表します。

- 本教材では、以下の代表的なリスト探索のアルゴリズムを説明します。
  - 線形探索
  - 二分探索

 CC BY-NC-SA

配列について

- 探索について説明する前に配列の基本について説明します。
- 配列は値を順序付けて格納します。例の図では一番左を先頭（1番目）の値、一番右を最後（例では、5番目）の値として扱います。それぞれ「配列のi番目の数」という呼び方をすることにします。
  - 以下の例では、配列の1番目の値は8、3番目の値は10です。
- 配列に格納している値の数を、「配列の要素数」と呼ぶことにします。
  - 以下の例では、配列の要素数は5になります。



線形探索

- 線形探索は、配列の先頭から順に目的の値を探すアルゴリズムです。
- 単純なアルゴリズムですが、配列に格納されている値が規則的に並んでいない場合は線形探索が最も有効なアルゴリズムとなります。
  - 配列が短い場合、人間の目では目的の値が格納されている位置はすぐに分かりますが、アルゴリズムを考える（問題を解く）には配列の長さがどうであろうと明確な手順に従って目的の値を探す必要があります。
  - 逆に、規則的に並んでいる場合とは、例えば、値が昇順に並んでいる場合が該当します。

値が規則的に並んでいない

8	6	10	0	5

値が規則的に並んでいる

0	5	6	8	10

 CC BY-NC-SA

線形探索

- 以下の例で、目的の値を7とし、配列の中から7を探す場合を考えます。
- まず1番目の値が7かどうか調べます。1番目の値は10ですので、7ではありません。

- 2番目の値が7かどうか調べます。2番目の値は5ですので、7ではありません。

- 3番目の値が7かどうか調べます。3番目の値は13ですので、7ではありません。

- 4番目の値が7かどうか調べます。4番目の値は7ですので、7を発見したと報告してアルゴリズムを終了します。

 CC BY-NC-SA

線形探索

- 配列に含まれていない値を目的の値として探索した場合は、先頭から最後まで目的の値を探索した後に、その値が見つからなかったと報告してアルゴリズムを終了します。
  - 例えば、以下の配列の例では、8を探索した場合は、その様にして探索を終了します。

 CC BY-NC-SA

線形探索

- 線形探索のアルゴリズムは次の様になります。（配列の要素数をnとします）

  手順1：x を入力　（探索する値を入力）

  手順2：i=1　（配列のi番目の値が目的の値xか調べたい）

  手順3：配列のi番目の数とxを比較し、一致するならば、xを発見したと報告（出力）して終了する。そうでなければ手順4へ

  手順4：iを1増加する

  手順5：iがn+1より小さい場合、手順3へ戻る。そうでなければ手順6へ

  手順6：xを発見できなかったと報告（出力）して終了する

- 以降の頁では分岐における「そうでなければ手順○○へ」の一文は省略します。例えば、手順3において分岐があり、条件が成立しない場合は手順4を実行するものとします。

 CC BY-NC-SA

線形探索のフローチャート

- 線形探索のフローチャートです。

手順1：x を入力

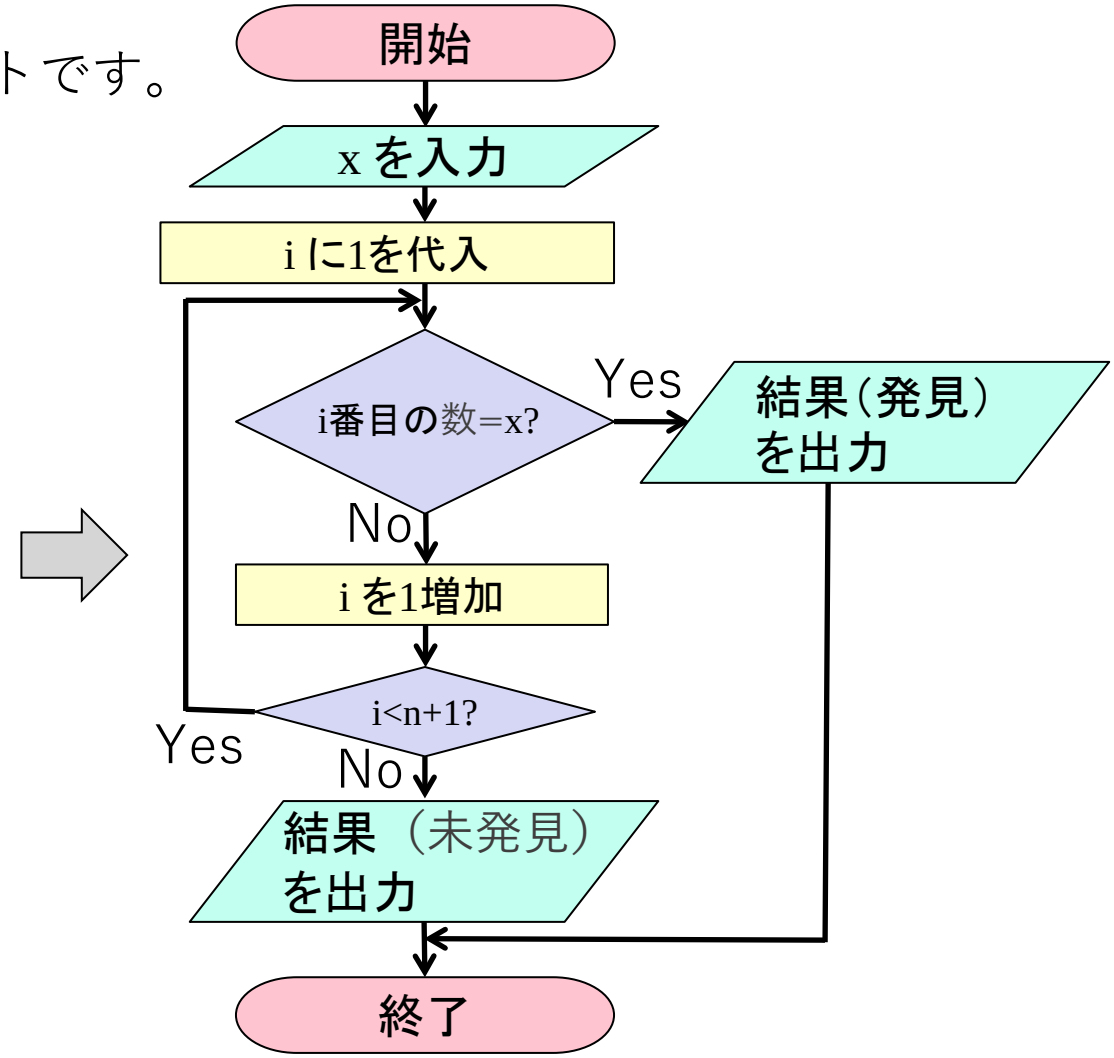
手順2：i=1

手順3：配列のi番目の数とxを比較し、一致するならば、xを発見したと報告して終了する。そうでなければ手順4へ

手順4：iを1増加する

手順5：iがn+1より小さい場合、手順3へ戻る。そうでなければ手順6へ

手順6：xを発見できなかったと報告して終了する



計算量

- アルゴリズムが動作を終了するのにかかる時間を、そのアルゴリズムの計算量と呼びます。
- そして、アルゴリズムの性能の良し悪しはこの計算量によって評価します。
  - つまり、一般に計算量が小さければ小さい程、良いアルゴリズムと言って構いません。
- 計算量はアルゴリズムを動かすのに使用しているコンピュータの性能に依存して決まるのは好ましくない為、実際にコンピュータで動かしてみたときにかかる時間ではなく、「アルゴリズムが終了するまでにどれくらいの数の手続きを実行したか」で評価します。
  - 例えば、線形探索の場合は「手順1から6を何回実行したのか」で評価します。
- そして、計算量は厳密に評価するのではなく、どのくらいの値に比例する時間がかかるのか、で評価を行います。

 CC BY-NC-SA

線形探索の計算量

- では、線形探索の計算量を評価してみましょう。配列の大きさをnとした場合、各手順の実行回数は以下の通りです。
  - 手順1, 2, 6：1回
  - 手順3, 4, 5：最大でn回
- つまり、合計すると最大で（つまり、一番時間がかかる最悪の場合）3n+3回手順を実行します。
- よって、線形探索の計算量は最悪の場合、配列の大きさである n に比例することが分かります。
  - 「x に比例する大きさ」は O(x) （Oは大文字のオー）という記号で表し、「x のオーダー」、「オーダー x 」などと読みます。
  - つまり、線形探索の最悪の計算量はO(n)である、ということもできます。

 CC BY-NC-SA

二分探索

- 二分探索は、線形探索と同様に配列から目的の値を探すアルゴリズムですが、探索の対象となる配列は値が昇順（もしくは降順）、すなわち小さい順に並べてあります。
- 二分探索では探索する値が見つかるまで、探索対象とする配列の範囲を毎回半分になる様に探索を行います。そうすることで線形探索よりも効率的に（つまり、より小さい計算量で）探索を行います。

 CC BY-NC-SA

二分探索

- 以下の例で、目的の値を7とし、配列の中から7を探す場合を考えます。先の述べた様に二分探索を行う場合は配列の中身は昇順に並べておく必要があります。

- まず真ん中（4番目）の値が7かどうか調べます。4番目の値は9ですので7ではありません。

- このとき、配列の値は昇順に並んでいますので、真ん中より後（5番目から7番目）の値は4番目の値 (9) より大きくなります。その為、7が配列に含まれているならば、真ん中よりも手前（1番目から3番目）ということになります。そこで以降は4番目から7番目は探索しなくても良く、1番目から3番目に7が含まれているかどうか調べれば良くなります。

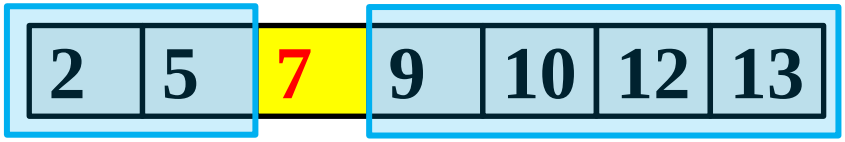
 CC BY-NC-SA

二分探索

- そこで、2番目に探索を行う場合、先程と同様に探索対象となる配列の真ん中の値が7かどうか調べます。ただし、1回目の探索によって今回は1番目から3番目のみが探索の対象とすれば良いのでした。そこで、今回は1番目から3番目までの真ん中（2番目）の値が7かどうか調べることになります。2番目の値は5ですので7ではありません。

4番目から7番目は無視する

- 先程と同様に、配列の値は昇順に並んでいますので真ん中より手前（1番目）の値は2番目の値より必ず小さくなります。その為、7が配列に含まれているならば、2番目よりも後（3番目）ということになります。



1,2番目、4番目から7番目は無視する

- 次に、3番目の値が7かどうか調べます。3番目の値は7ですので、 7を発見したと報告してアルゴリズムを終了します。

 CC BY-NC-SA

二分探索

- 先の例では、最後に目的の値が見つかりましたが、勿論途中で見つかる場合もありますし、最後まで見つからないで探索を終了する場合もあり得ます。
  - 例えば、以下の配列の例では、8を目的の値として探索した場合は、8を最後まで発見できずに探索を終了しますし、12を探索した場合には2回目の探索で12を発見することになります。

 CC BY-NC-SA

二分探索

- 二分探索のアルゴリズムは次の様になります。（配列の要素数をnとします）

手順1：x を入力　（探索する値を入力）

手順2：left=1, right=n　（配列のleft番目からright番目が探索範囲）

手順3：left>rightならば、xを発見できなかったと報告（出力）して終了する

手順4：mid=[(left+right)/2] （ただし、[x]はx以下の最大の整数。例えば、[5/2]なら2を表す）

手順5：配列のmid番目の値をMで表す。Mとxを比較し、一致するならば、xを発見したと報告（出力）して終了する

手順6：M<xならば、left=mid+1とする。M>xならば、right=mid-1とする（探索範囲を半分にする）

手順7：手順3に戻る

 CC BY-NC-SA

二分探索のフローチャート

- 二分探索のフローチャートです。

手順1：x を入力

手順2：left=1, right=n

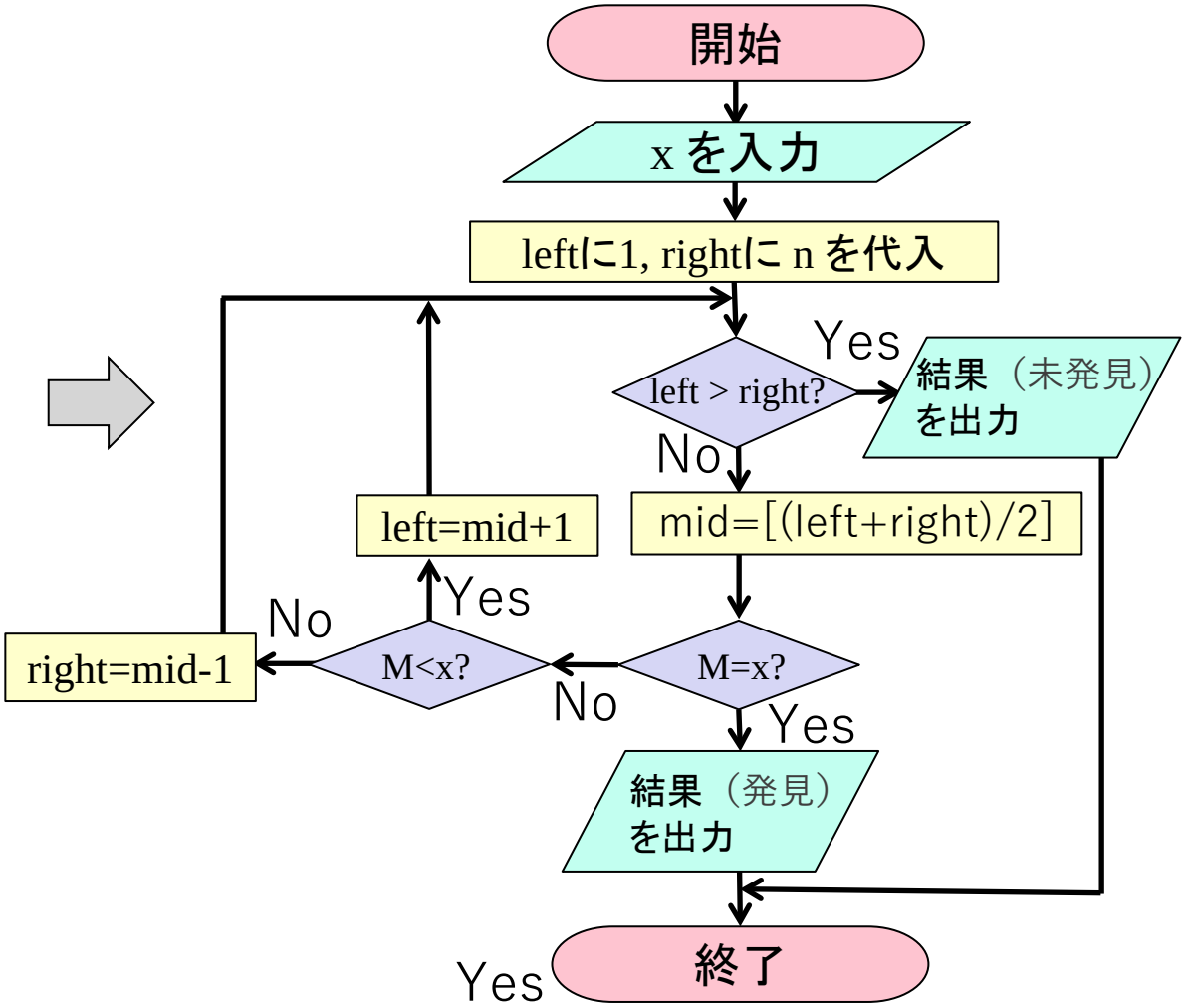
手順3：left>rightならば、xを発見できなかったと報告（出力）して終了する

手順4：mid=[(left+right)/2]

手順5：配列のmid番目の値をMで表す。Mとxを比較し、一致するならば、xを発見したと報告（出力）して終了する。

手順6：M<xならば、left=mid+1とする。M>xならば、right=mid-1とする

手順7：手順3に戻る



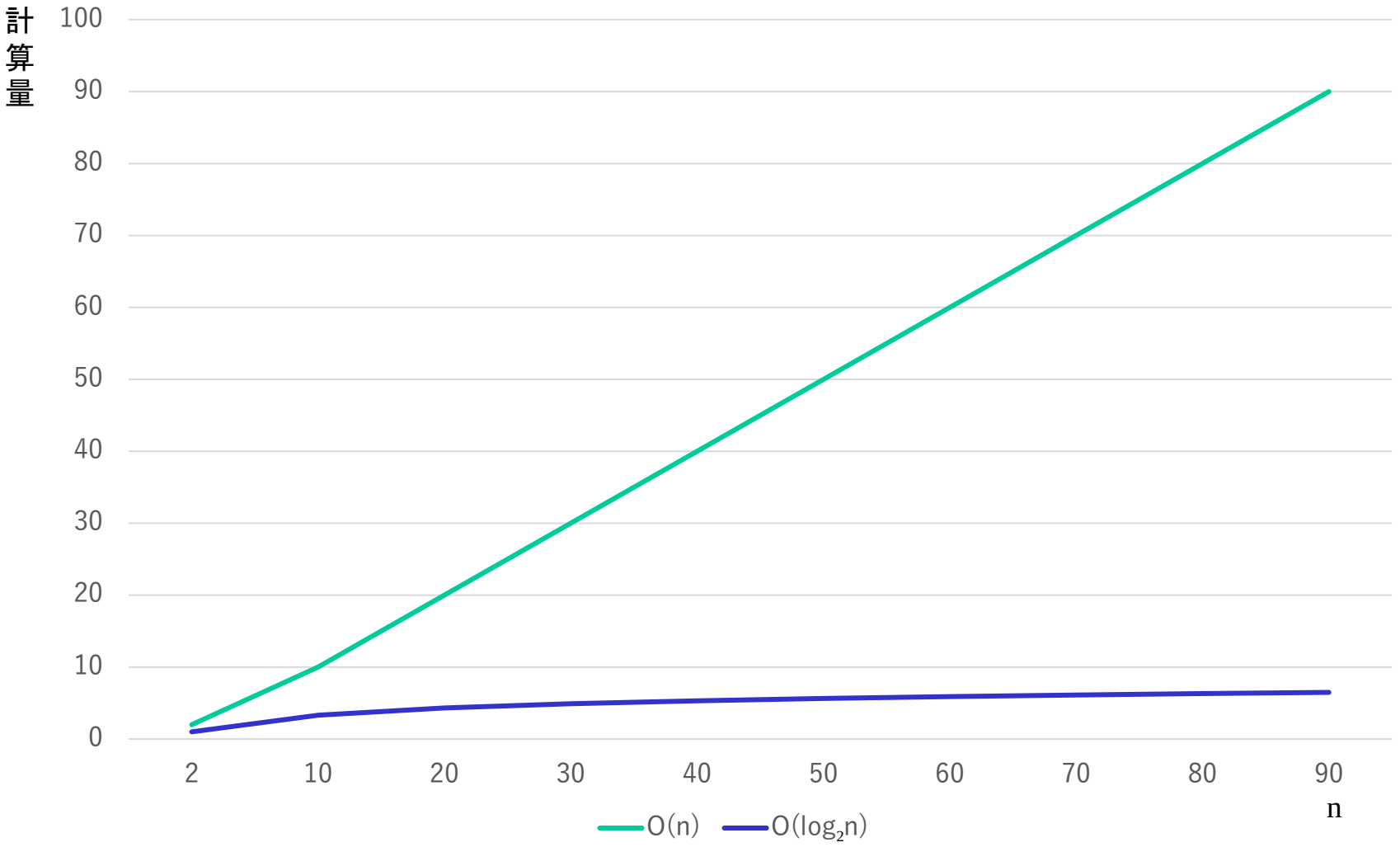
二分探索の計算量

- 二分探索の計算量を評価してみましょう。配列の大きさをnとした場合を考えましょう。一番時間がかかる場合、探索対象となる配列の範囲は n → n/2 → n/4 → n/8 → … → 1 となりますので、約 $\log_2 n$ 回のmidの計算と比較（手順3から7）が行われます。
  - n → n/ 2 → $n/2^2$ → $n/2^3$ → … → $n/2^k$ なので、k+1回手順3から7を行います。このとき、 $n/2^k$ =1とすると$k = \log_2 n$が成立します
- つまり、各手順の実行回数は以下の通りです。
  - 手順1,2：1回
  - 手順3,4,5,6,7：最大で約$\log_2 n$回
- つまり、合計すると最大で約 $5 \log_2 n + 2$回手順を実行します。
- よって、二分探索の計算量は $\log_2 n$ に比例する（すなわち、$O(\log_2 n)$ である）ことが分かり、線形探索の計算量である n よりも小さくなることが分かります。

 CC BY-NC-SA

計算量の比較

- 線形探索 (O(n)) と二分探索 (O($\log_2 n$)) の計算量を比較してみます。n が大きくなるほど二分探索が高速であることが良く分かります。



東京大学 数理・情報教育研究センター 小林浩二 2021 CC BY-NC-SA

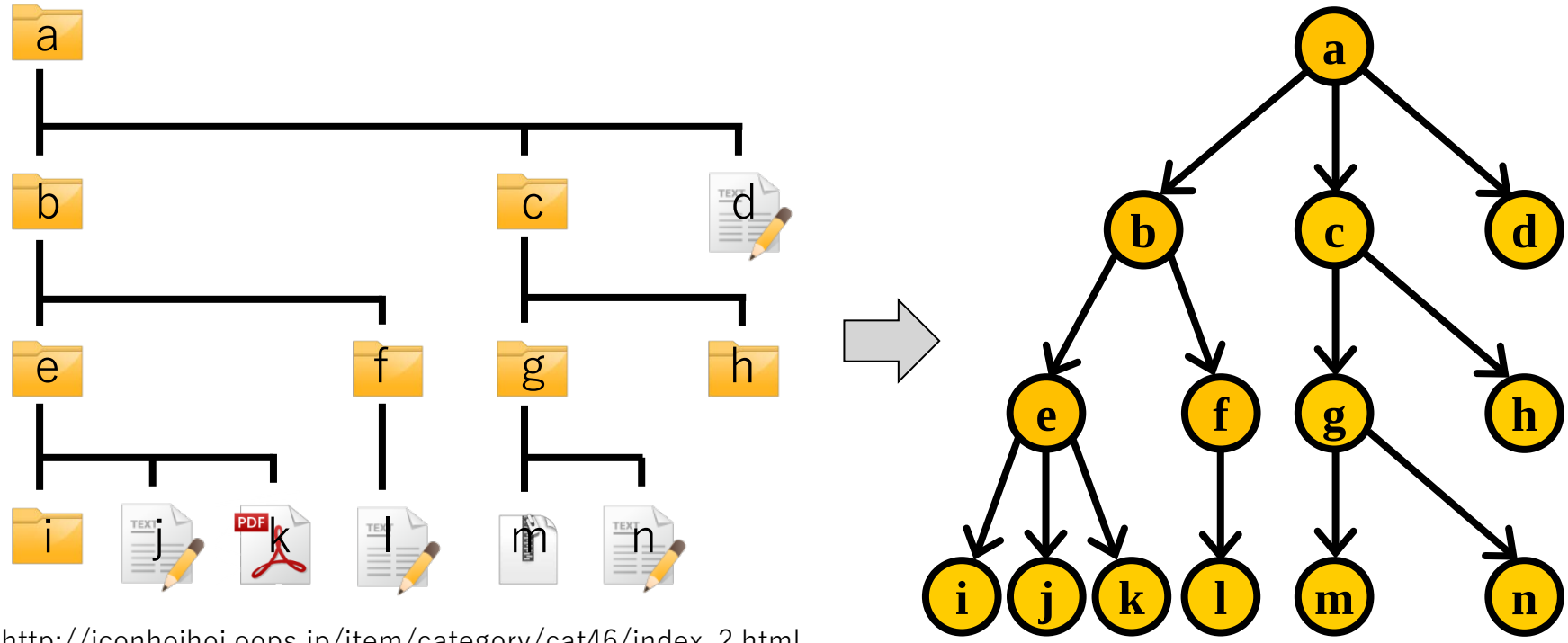
木探索

- 木探索は、樹形図状のデータ構造である「木構造」に格納された値の中から目的の値を探す（探索を行う）問題です。
- 木構造は樹形図状のデータ構造です。組織図や家系図、コンピュータ上のファイル・フォルダの階層構造などは樹形図状に表現できますが、こういった様々な分野で用いるデータ構造です。
- 本教材では、以下の代表的な木探索のアルゴリズムを説明します。
  - 幅優先探索
  - 深さ優先探索

 CC BY-NC-SA

木構造について

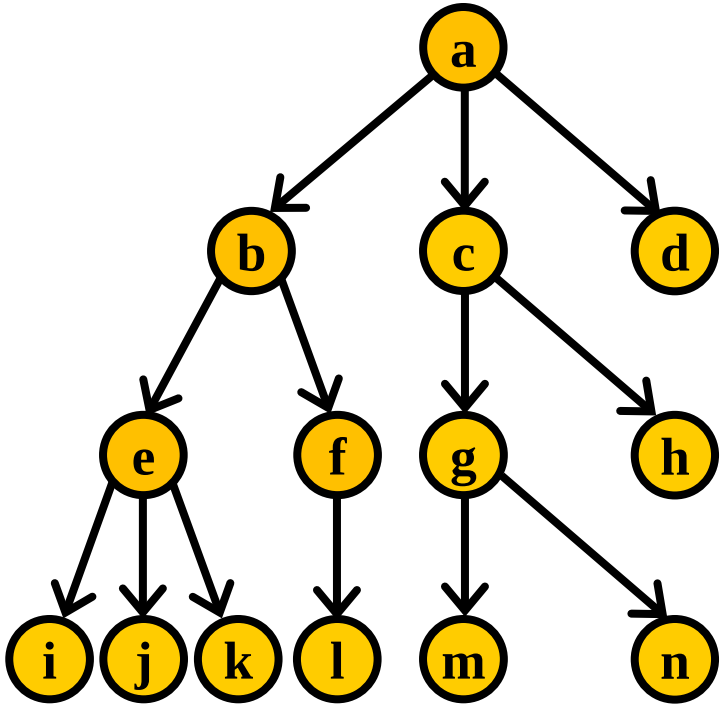
- 例えば、ファイル・フォルダの階層構造を表す木探索では目的のファイルを探すという操作を考えられます。
- 以下はファルダ類の階層構造の例ですが、この教材では右の図の様な丸と矢印からなる図で木構造で表すことにします。



http://iconhoihoi.oops.jp/item/category/cat46/index_2.html

木構造について

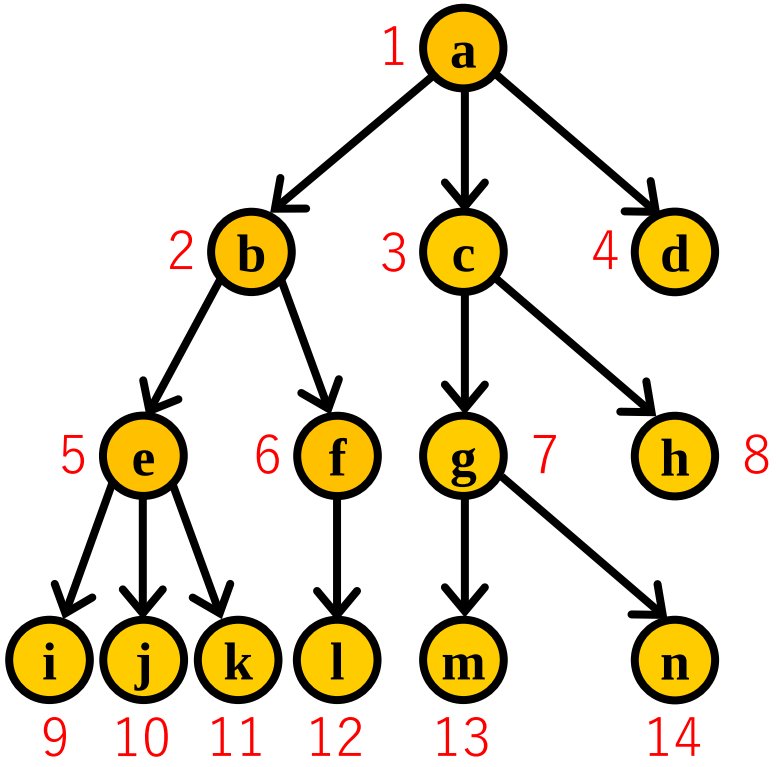
- 木構造において用いる簡単な用語を説明します。
- 図の丸 ● で表す部分を点と呼び、矢印を枝と呼びます。矢印が1本も入って来ない点を根と呼びます。下の例では点aが根です。ある点から出た矢印の先の点をその点の子と呼びます。下の例では点aの子は点b,c,dの3つです。



東京大学 数理・情報教育研究センター 小林浩二 2021 CC BY-NC-SA

幅優先探索

- 木構造に対する探索は一般に、根から探索を開始し、根以外の点をどの様な順番で調べる（探索する）か、を定義します。幅優先探索は根から探索を開始し、根に近い点から順番に探索を行います。
  - 同じ近さの点が2個以上存在する場合は、その中から自由に選んで構いません。
- 配列の探索では配列に格納されている値を探索しましたが、木探索では簡単のため、点の名前を探索するものとします。
- この例では、a,b,c,d,e,f,g,h,i,j,k,l,m,nの順で探索を行います（右図の赤字）。



幅優先探索

- 幅優先探索のアルゴリズムは次の様になります。

手順1：x を入力　（探索する値＝点の名前を入力）

手順2：v を探索する木構造の根とする

手順3：v = x ならば、xを発見したと報告（出力）して終了する

手順4：今までに探索した点（vに代入されたことがある点）の子であり、かつまだ探索していない点が存在するなら、その中で一番根に近い点をvとして手順3に戻る　（次に探索する点を決める）

手順5： xを発見できなかったと報告（出力）して終了する

 CC BY-NC-SA

幅優先探索のフローチャート

- 幅優先探索のフローチャートです。

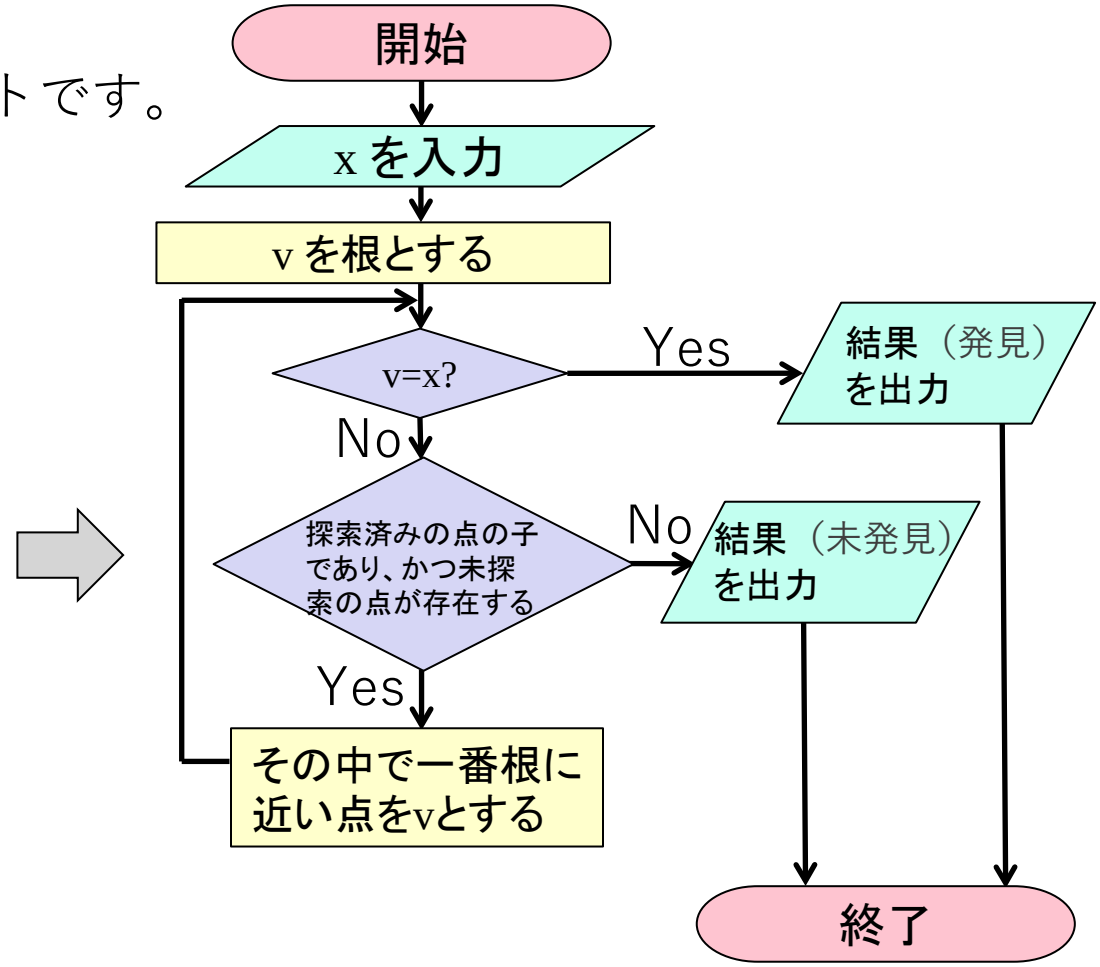
手順1：x を入力

手順2：v を探索する木構造の根とする

手順3：v = x ならば、xを発見したと報告（出力）して終了する

手順4：今までに探索した点（vに代入されたことがある点）の子であり、かつまだ探索していない点が存在するなら、その中で一番根に近い点をvとして手順3に戻る

手順5： xを発見できなかったと報告（出力）して終了する



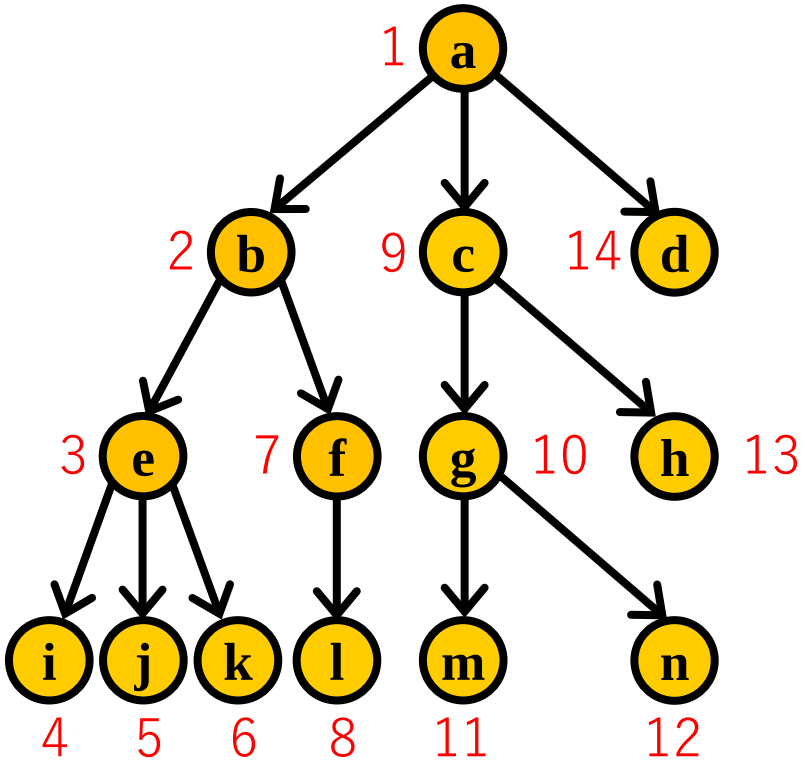
幅優先探索の計算量

- 幅優先探索の計算量を評価してみましょう。探索する木構造の点の数をnとした場合、各手順の実行回数は以下の通りです。
  - 手順1,2,5：1回
  - 手順3, 4：最大でn回
- つまり、合計すると最大で2n+3回手順を実行します。
- よって、幅優先探索の計算量は最悪の場合、木構造の点の数である n に比例することが分かります。

 CC BY-NC-SA

深さ優先探索

- 深さ優先探索は根から探索を開始し、できるだけ根から遠い点から順番に探索を行います。つまり、幅優先探索とはある意味逆の順番で探索を行います。
- 以下の例では、a,b,e,i,j,k,f,l,c,j,k,l,m,nの順で探索を行います（下図の赤字）。



深さ優先探索

- 深さ優先探索のアルゴリズムは次の様になります。

手順1：x を入力　（探索する値＝点の名前を入力）

手順2：v を探索する木構造の根とする

手順3：v = x ならば、xを発見したと報告（出力）して終了する

手順4：今までに探索した点（vに代入されたことがある点）の子であり、かつまだ探索していない点が存在するなら、その中で一番根から遠い点をvとして手順3に戻る　（次に探索する点を決める）

手順5： xを発見できなかったと報告（出力）して終了する

 CC BY-NC-SA

深さ優先探索のフローチャート

- 深さ先探索のフローチャートです。

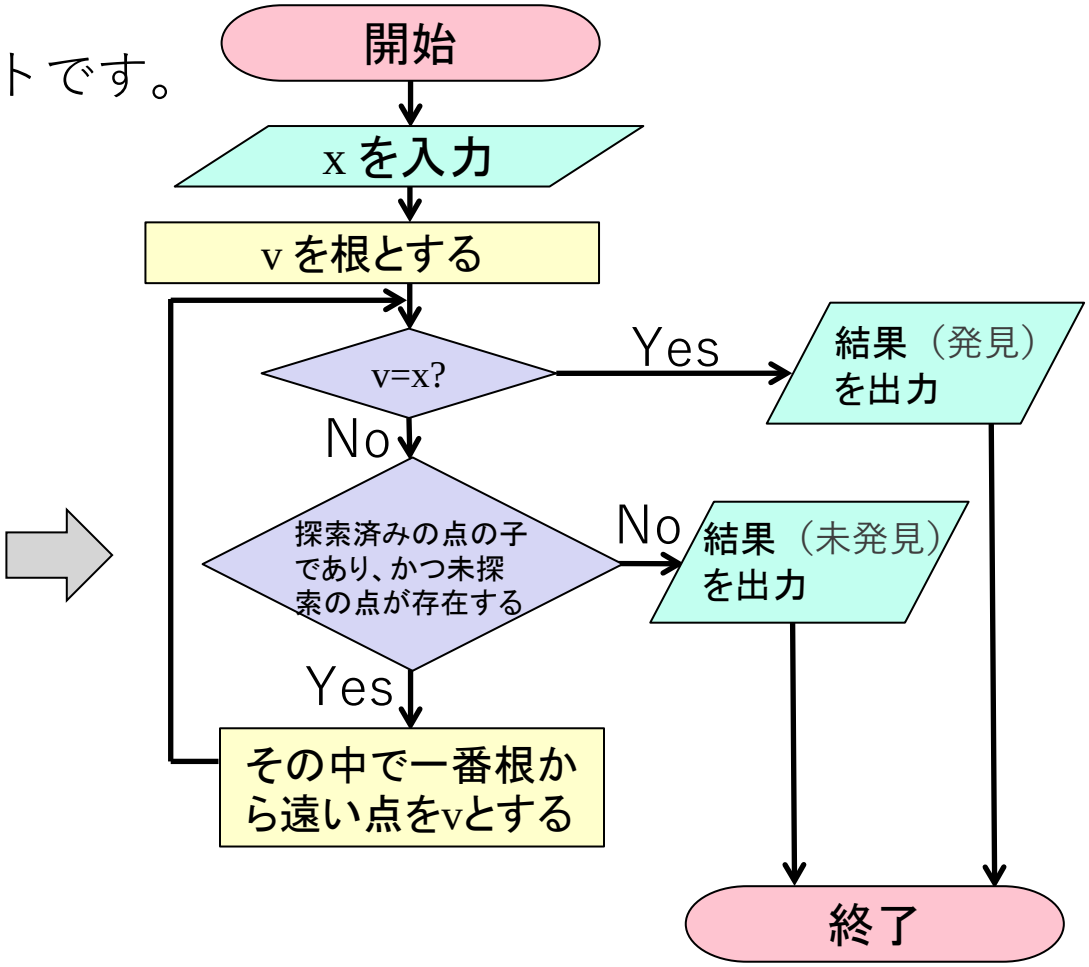
手順1：x を入力

手順2：v を探索する木構造の根とする

手順3：v = x ならば、xを発見したと報告（出力）して終了する

手順4：今までに探索した点（vに代入されたことがある点）の子であり、かつまだ探索していない点が存在するなら、その中で一番根に近い点をvとして手順3に戻る

手順5： xを発見できなかったと報告（出力）して終了する



 CC BY-NC-SA

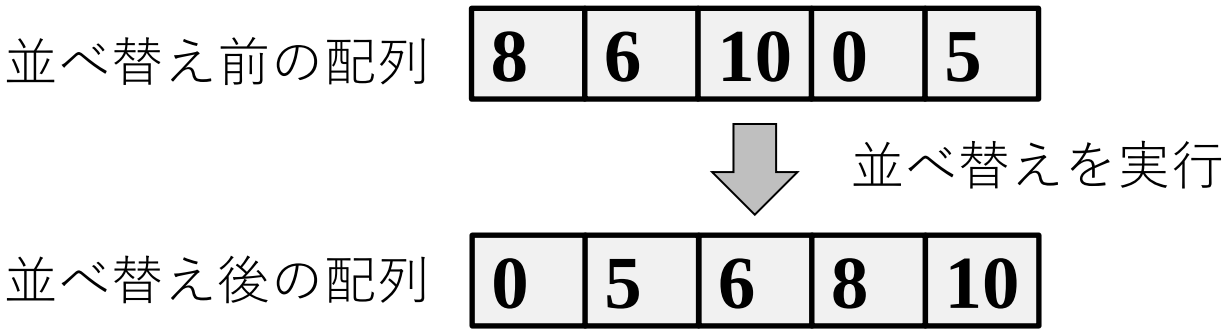
深さ優先探索の計算量

- 深さ優先探索の計算量を評価してみましょう。定義やフローチャートなどからも分かりますが、探索の順番は全く違いますが、深さ優先探索と幅優先探索は非常によく似ており、計算量も同様です。
- 探索する木構造の点の数をnとした場合、各手順の実行回数は以下の通りです。
  - 手順1,2,5：1回
  - 手順3, 4：最大でn回
- つまり、合計すると最大で2n+3回手順を実行します。
- よって、深さ優先探索の計算量も最悪の場合には木構造の点の数である n に比例することが分かります。

 CC BY-NC-SA

並べ替えのアルゴリズム

- 並べ替え（ソート, sort）は、数を小さい順（昇順）に並べる問題です。
  - 大きい順（降順）に並べたり、それ以外の順序で並べたりする場合もあります。数の代わりに文字列を並べる場合もあります。
- 並べ替えの対象となる数は配列に格納されているものとします。



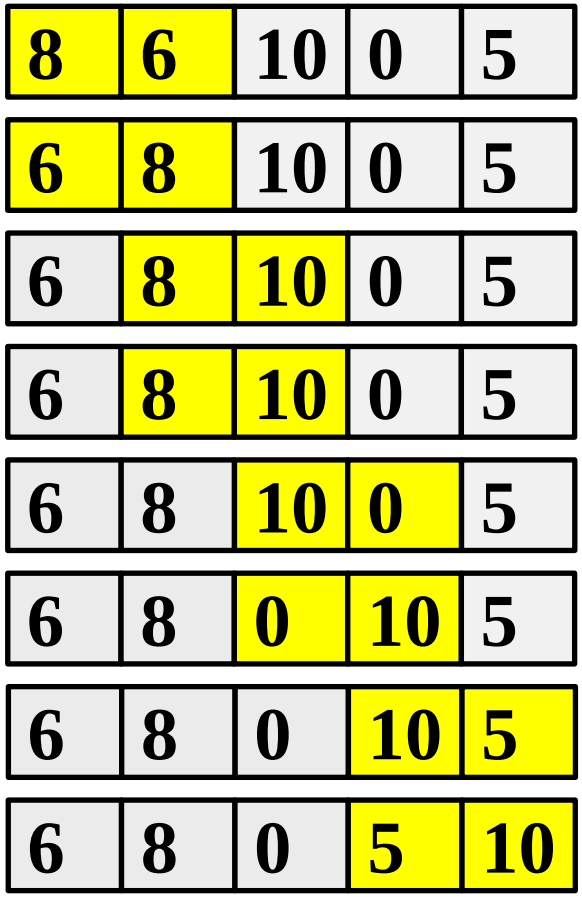
並べ替えのアルゴリズム

- 並べ替えを解くアルゴリズムには様々なアルゴリズムがあります。並べ替えを行うことが問題の目的ですので、得られる結果は同じですが、手順や効率（計算量の大きさ）が異なります。
  - 並べ替えのアルゴリズムは、二分探索の実行前に配列の値を並べ替えたり、コンピュータでCGを描画する際に描くべきものの順に並べたりと様々な場面で使われています。
- 本教材では、以下の代表的な並べ替えアルゴリズム（ソートアルゴリズム）を説明します。
  - バブルソート（交換法）
  - 選択ソート
  - 挿入ソート

 CC BY-NC-SA

バブルソート

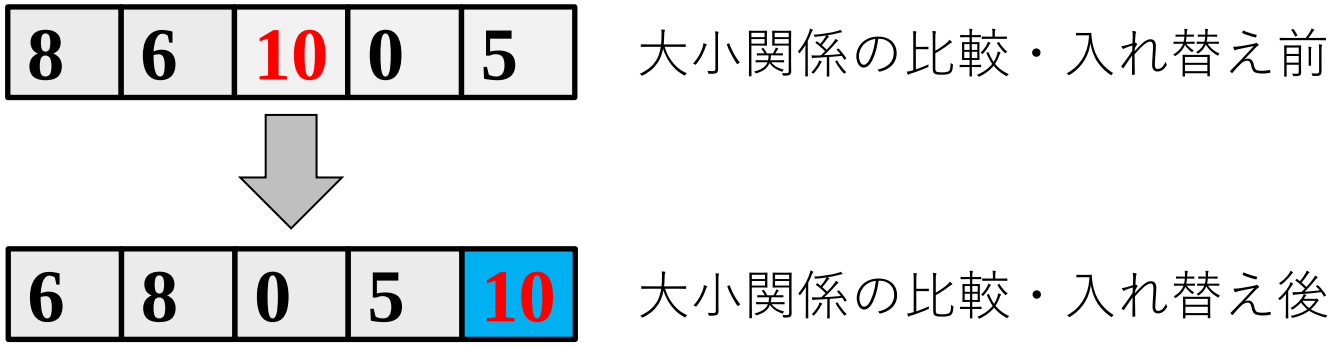
- バブルソート（交換法）では、配列の先頭から順に、隣り合う2つの数の大小を比較し、その大小が目的の順序（昇順）になっていない場合、その2つの数の順序を入れ替える、という手続きをまず行います。例えば、以下の図の通りです。



東京大学 数理・情報教育研究センター 小林浩二 2021 CC BY-NC-SA

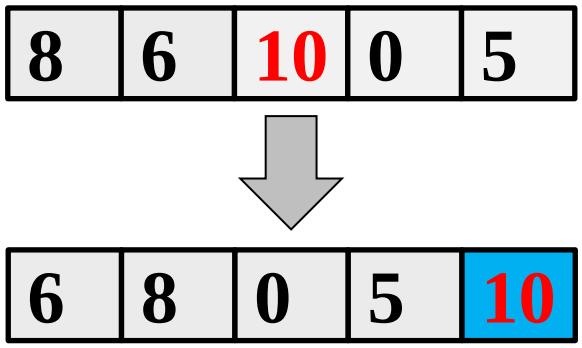
バブルソート

- 前頁の手順を実行完了すると、並べ替えの対象となっている数の中で最大の数が配列の最後に格納されていることに注意して下さい。
  - 例では、最大の数は10です。10は元は配列の3番目に格納されていましたが、先の手順完了後は配列の最後（5番目）に格納されています。
  - 最大の値が泡（バブル, bubble）の様に浮かび上がって最後尾に移動するのでバブルソートという名前がついています。



バブルソート

- これは部分的に並べ替えが完了したとみなすことができます。
- どういうことかと言うと、並べ替えは数を小さい順に並べる必要があります。つまり、「一番大きい数は配列の一番最後に格納したい」ということです。そして、先の比較・入れ替えの手順によって、配列の最後に最大の数が格納されています。
- つまり、元は配列の先頭から最後まで全ての数を並べ替える必要がありましたが、今は先頭から最後から2番目までの数だけを並べ替えれば良くなっていると見なせるということです。
  - 例では、元は配列の1番目から5番目まで並べ替えの必要があり、比較・入れ替えの実行後は、配列の1番目から4番目まで並べ替えれば良くなっています。



1番目から5番目を並べ替えたい

1番目から4番目を並べ替えたい

バブルソート

- そこで、再度、配列の先頭から順に、隣り合う2つの数の大小を比較・入れ替えを行っていきます。ただし、今度は、最後までではなく、最後から2番目まで行う様にします。
- 例えば、以下の図の通りです。

1番目(6)と2番目(8)を比較

1番目(6)の方が小さいのでそのまま

2番目(8)と3番目(0)を比較

3番目(0)の方が小さいので入れ替え

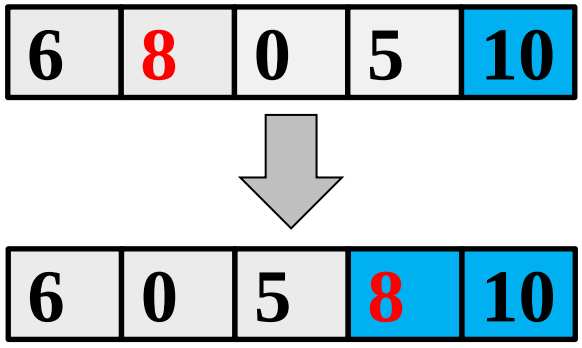
3番目(8)と4番目(5)を比較

4番目(5)の方が小さいので入れ替え

 CC BY-NC-SA

バブルソート

- 再度、比較・入れ替えの一連の手順を実行完了すると、並べ替えの対象となっている数（先頭から最後から2番目までの数）の中で最大の数が配列の最後から2番目に格納されることになります。
  - 最後から2番目に格納された数は配列の中で2番目に大きい数です。
  - 例では、先頭から最後から2番目までの数の中で最大の数（配列全体では2番目に大きい数）は8です。先の手順によって最後から2番目（先頭から4番目）に格納されています。



大小関係の比較・入れ替え前

大小関係の比較・入れ替え後

バブルソート

- 以降、比較・入れ替えの範囲を徐々に狭めながら、一覧の比較・入れ替えを繰り返していくと並べ替えが完了します。

1番目(6)と2番目(0)を比較

2番目(0)の方が小さいので入れ替え

2番目(6)と3番目(5)を比較

3番目(5)の方が小さいので入れ替え

 CC BY-NC-SA

バブルソート

- 前のページの続きです。

3番目に大きい数(6)が最後から3番目に格納されている

1番目(0)と2番目(5)を比較

1番目(0)の方が小さいのでそのまま

4番目に大きい数(5)が最後から4番目、5番目に大きい数(0)が最後から5番目（先頭）に格納されている

 CC BY-NC-SA

バブルソート

- バブルソートのアルゴリズムは次の様になります。（配列の要素数をnとします）

手順1：j=1　（j番目の比較・入れ替え＝j番目に大きい数を求める）

手順2：i=1

手順3：i番目とi+1番目の数を比較し、i+1番目の方が小さい場合、i番目とi+1番目の数を入れ替える　（i番目とi+1番目の比較・入れ替え）

手順4：iを1増加する

手順5：iがn-jより小さいなら手順3に戻る。そうでなければ、jを1増加する

手順6：jがnより小さいなら、手順2に戻る。そうでなければ、結果(配列）を出力してアルゴリズムを終了する

 CC BY-NC-SA

バブルソートのフローチャート

- バブルソートのフローチャートです。

手順1：j=1

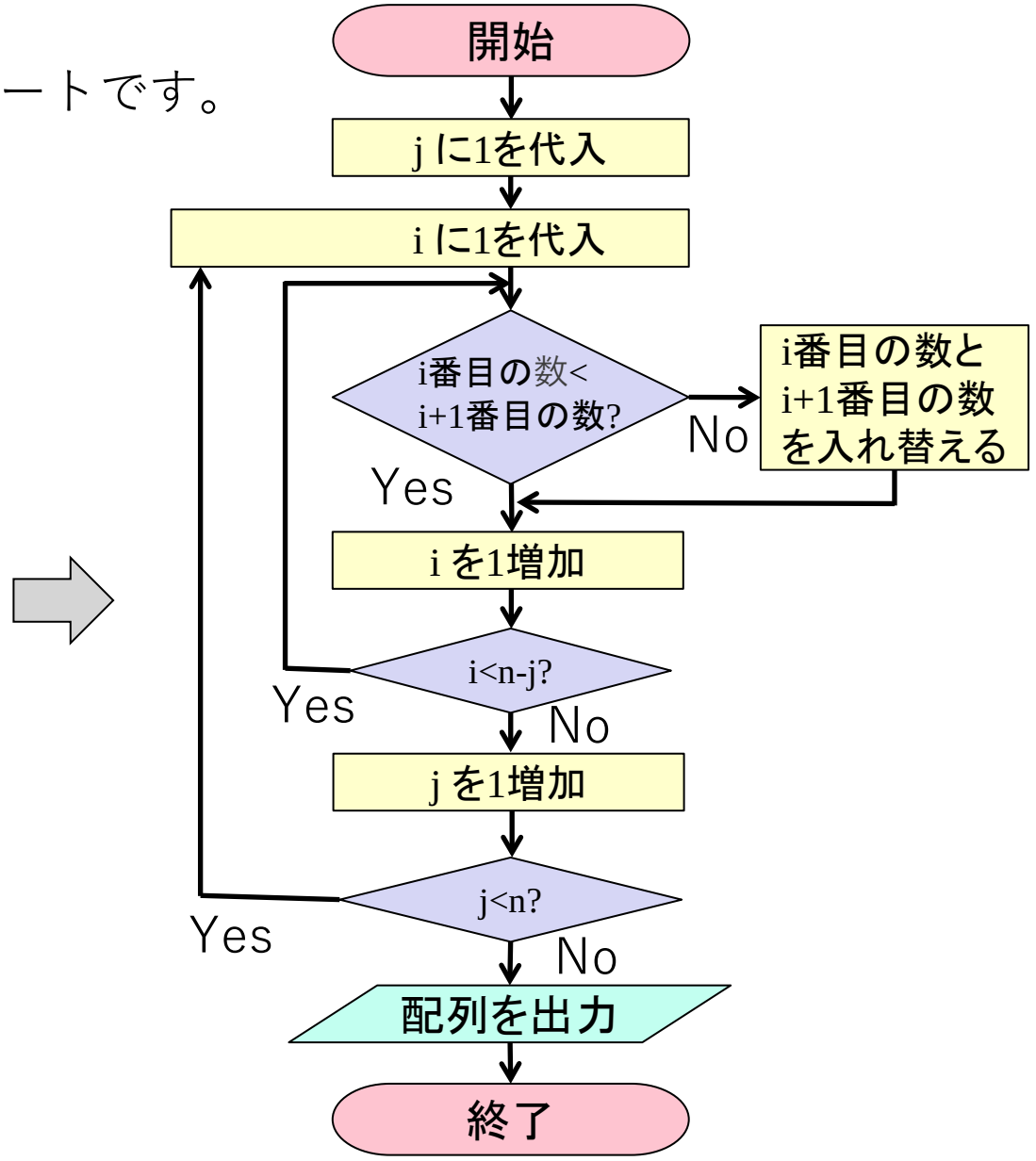
手順2：i=1

手順3：i番目とi+1番目の数を比較し、i+1番目の方が小さい場合、i番目とi+1番目の数を入れ替える

手順4：iを1増加する

手順5：iがn-jより小さいなら手順3に戻る。そうでなければ、jを1増加する

手順6：jがnより小さいなら、手順2に戻る。そうでなければ、結果(配列）を出力してアルゴリズムを終了する



 CC BY-NC-SA

バブルソートの計算量

- バブルソートの計算量を評価してみましょう。並べ替える数の個数をnとした場合、各手順の実行回数は以下の通りです。
  - 手順1：1回
  - 手順2：n-1回
  - 手順3,4,5：変数jの値がaのときn-a回であり、jは1からn-1までの場合が起こり得る。よって、合計で(n-1)+…+1=n(n-1)/2回
  - 手順6:n-1回
- つまり、合計すると約n^2回手順を実行します。
- よって、バブルソートの計算量は n^2 に比例することが分かります。

 CC BY-NC-SA

選択ソート

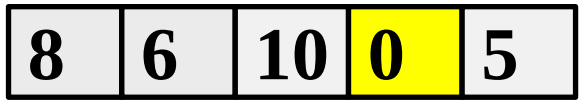
- 選択ソートは、配列の中から最小の数を見つける→2番目に小さい数を見つける→3番目に小さい数を見つける→…を繰り返すことで昇順に数を並べ替えるアルゴリズムです。
- ただし、見つけたi番目に小さい数は配列のi番目に移動させます。そして、i+1番目に小さい数を見つけるときには、配列のi+1番目から最後までを探す様にします。そうすることで、1番目からi番目に小さい数が格納されている場所を探さない様にして、目的の数を見つけるのに費やす時間を短くしています。
- 例えば、以下の例ではi=3、すなわち、1から3番目に小さい数を配列の1から3番目にそれぞれ格納した状況です。

- また、i+1番目に小さい数を見つける手順は、配列のi+1番目から最後までの値の中で最小の数を見つける手順に等しいことに注意して下さい。

 CC BY-NC-SA

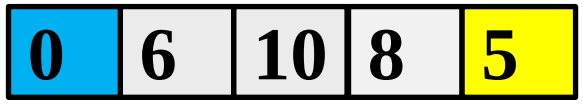
選択ソート

- 選択ソートの実行例を示します。

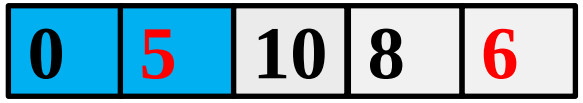


最小の数 (0) を見つける

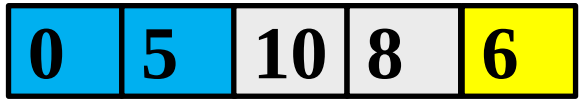
配列の1番目の数 (8) と0を入れ替え



配列の2番目から最後までの中で最小の数 (5) を見つける



配列の2番目の数 (6) と5を入れ替え



配列の3番目から最後までの中で最小の数 (6) を見つける

 CC BY-NC-SA

選択ソート

- 前の頁の続きです。

配列の3番目の数 (10) と6を入れ替え

配列の4番目から最後までの
中で最小の数 (8) を見つける

配列の4番目の数 (8) と8を入れ替え（8は
元から4番目なので何もしない）

最大の数 (10) は配列の最後にあるので
何もしない

 CC BY-NC-SA

選択ソート

- 選択ソートのアルゴリズムは次の様になります。（配列の要素数をnとします）

手順1：j=1　（j番目に小さい数を求める）

手順2：i=j+1, x=j　（手順3から4において、配列のj番目から最後までの中から最小の数を探す。暫定の最小の数は配列のx番目に格納されている）

手順3：配列のi番目の数をx番目の数と比較し、i番目の数の方が小さい場合、xにiを代入する　（暫定の最小の数の格納場所xを更新する）

手順4：iを1増加する

手順5：iがn+1より小さい場合、手順3に戻る。そうでなければ、配列のx番目の数とj番目の数を入れ替えて、jを1増加する　（配列のj番目にj番目に小さい数を移動させる。その数は配列のx番目に格納されている）

手順6：jがnより小さい場合、手順2に戻る。そうでなければ結果(配列)を出力してアルゴリズムを終了する

 CC BY-NC-SA

選択ソートのフローチャート

- 選択ソートのフローチャートです。

手順1：j=1

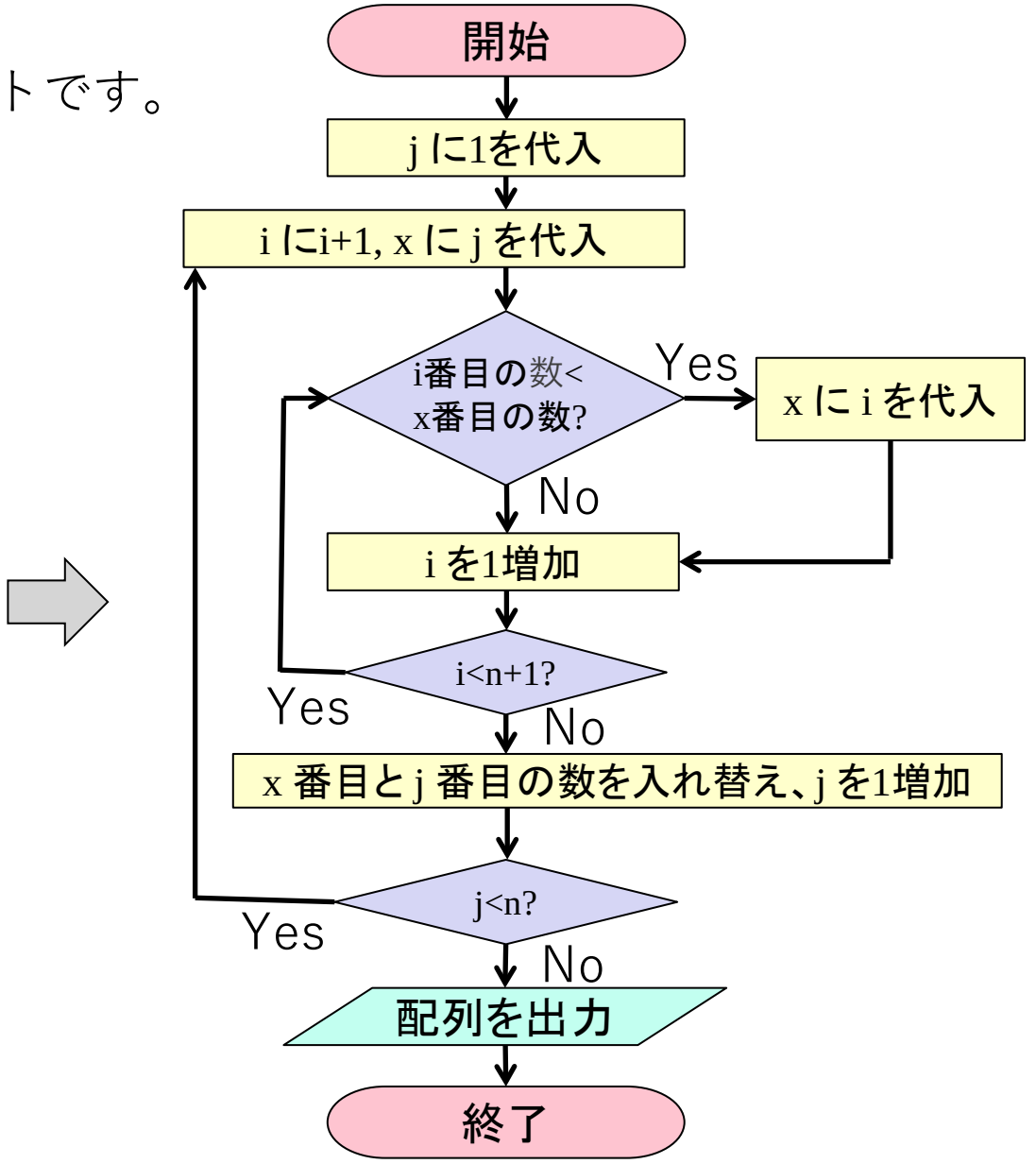
手順2：i=j+1, x=j

手順3：配列のi番目の数をx番目の数と比較し、i番目の数の方が小さい場合、xにiを代入する

手順4：iを1増加する

手順5：iがn+1より小さい場合、手順3に戻る。そうでなければ、配列のx番目の数とj番目の数を入れ替えて、jを1増加する

手順6：jがnより小さい場合、手順2に戻る。そうでなければ結果(配列）を出力してアルゴリズムを終了する



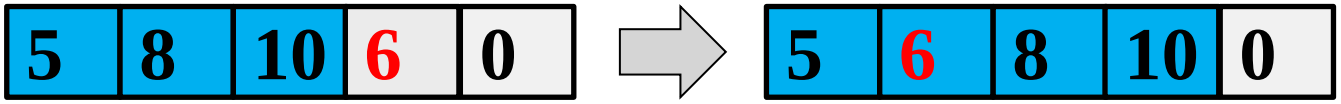
選択ソートの計算量

- 選択ソートの計算量を評価してみましょう。並べ替える数の個数をnとした場合、各手順の実行回数は以下の通りです。
  - 手順1：1回
  - 手順2,6：n-1回
  - 手順3,4,5：変数jの値がaのときn-a回であり、jは1からn-1まで増加する。よって、合計で(n-1)+…+1=n(n-1)/2回
- つまり、合計すると約n^2回手順を実行します。
- よって、選択ソートの計算量は n^2 に比例することが分かります。

 CC BY-NC-SA

挿入ソート

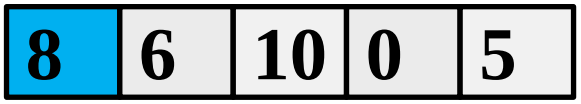
- 挿入ソートは、並べ替え済みの配列の適切な位置に、新しい数を挿入することで数を並べ替えるアルゴリズムです。
- 具体的には、配列の1番目からj番目の数が並べ替え済みである状態で、配列のj+1番目の数を配列の1番目からj+1番目の適切な位置（x番目としましょう）に移動させます。このとき、x番目からj番目の数はそれぞれx+1番目からj+1番目に移動させます。
- 例えば、以下の例はj=3、すなわち、配列の1,2,3番目は並べ替え済みです。そして、4番目の値 (6) を適切な位置に移動させます。この場合は、1番目の値 (5) と2番目の間 (8) に移動するのが適切です。よって、x=2、すなわち、2番目と3番目の値を3番目と4番目にそれぞれスライドさせます。



- もう少し具体例を見てみましょう。

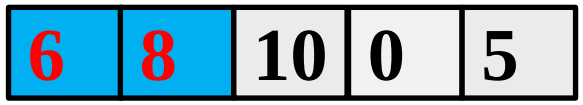
挿入ソート

- 挿入ソートの実行例を示します。



配列の1番目の数 (8) は並べ替え済みとみなす

配列の2番目の数 (6) を並べ替え済みの配列（青い部分）に挿入したい



6は配列の1番目の数 (8) より小さいので8の前に挿入（8を1つ後ろにずらす）

配列の3番目の数 (10) を挿入したい

10は配列の2番目の数 (8) より大きいので8の後ろに挿入

 CC BY-NC-SA

挿入ソート

- 前の頁の続きです。

配列の4番目の数 (0) を挿入したい

0は配列の1番目の数 (5) より小さいので6の前に挿入（6,8,10を1つ後ろにずらす）

配列の5番目の数 (5) を挿入したい

5は配列の1番目の数 (0) より大きく、2番目の数 (6) より小さい6の前に挿入（6,8,10を1つ後ろにずらす）

 CC BY-NC-SA

挿入ソート

- 挿入ソートのアルゴリズムは次の様になります。（配列の要素数をnとします）

手順1：j=2　（配列の1番目からj-1番目が並べ替え済み、j番目の値を挿入したい）

手順2：i=j-1, y=j番目の値　（j-1番目から1番目の中で適切な挿入位置を探す）

手順3：配列のi番目の数とyを比較し、yの方が大きい場合、手順6へ。そうでなければ、i番目の数をi+1番目に代入する　（適切な挿入位置が見つかったら手順6を実行する。そうでなければ、i番目の数を後ろにずらす）

手順4：iを1減少する

手順5：iが0より大きい場合、手順3へ戻る（iが0だと、yを1番目に挿入）

手順6：配列のi+1番目にyを代入する。jを1増加する（適切な位置に挿入）

手順7：jがn+1より小さい場合、手順2に戻る。そうでなければ結果(配列）を出力してアルゴリズムを終了する

 CC BY-NC-SA

挿入ソートのフローチャート

- 挿入ソートのフローチャートです。

手順1：j=2

手順2：i=j-1, y=j番目の値

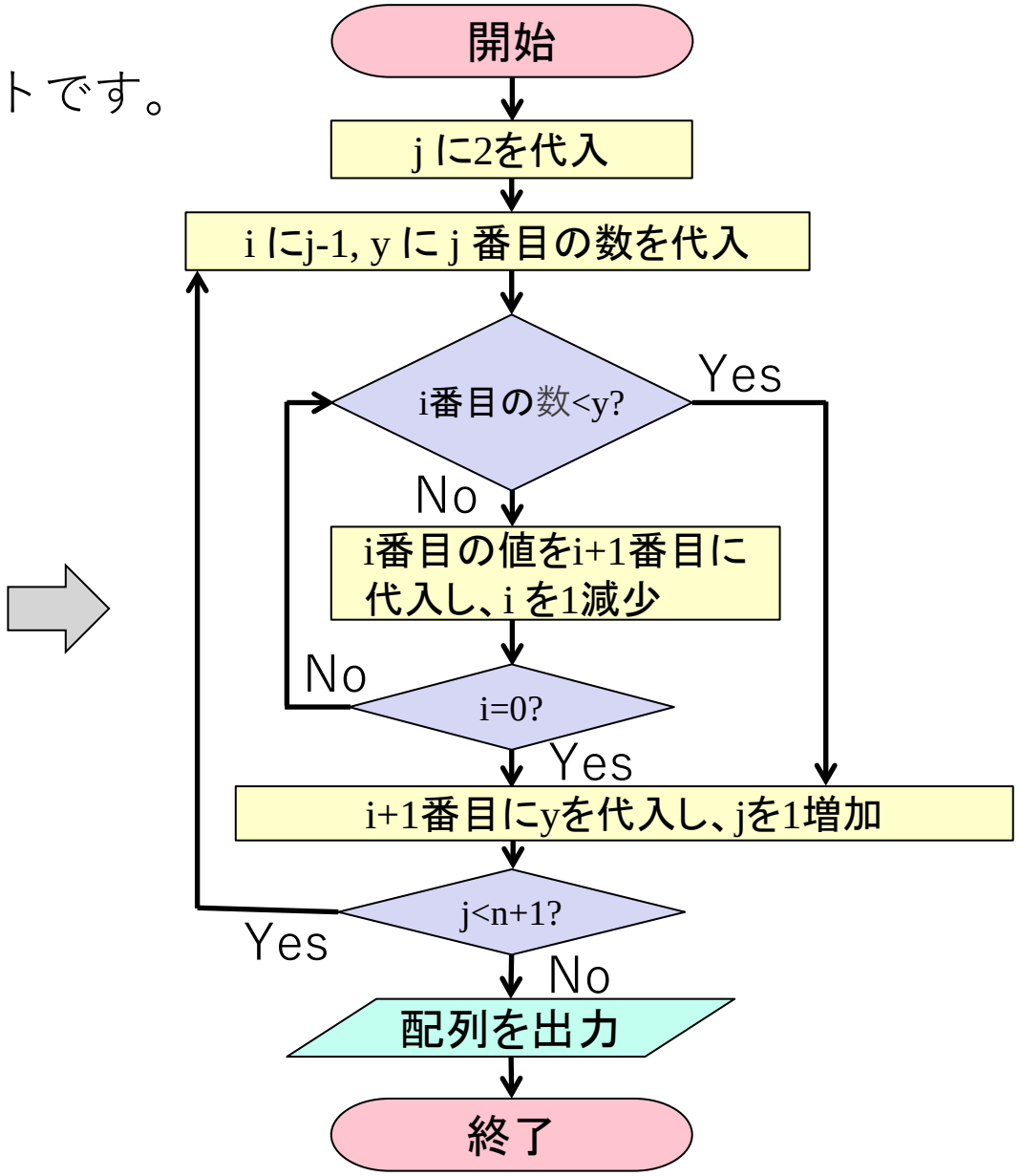
手順3：配列のi番目の数とyを比較し、yの方が大きい場合、手順6へ。そうでなければ、i番目の数をi+1番目に代入する

手順4：iを1減少する

手順5：iが0より大きい場合、手順3へ戻る

手順6：配列のi+1番目にyを代入する。jを1増加する

手順7：jがn+1より小さい場合、手順2に戻る。そうでなければ結果(配列）を出力してアルゴリズムを終了する



東京大学 数理・情報教育研究センター 小林浩二 2021 CC BY-NC-SA

挿入ソートの計算量

- 挿入ソートの計算量を評価してみましょう。並べ替える数の個数をnとした場合、各手順の実行回数は以下の通りです。
  - 手順1：1回
  - 手順2,6,7：n-1回
  - 手順3,4,5：変数jの値がaのとき最大でa-1回（最小で1回）であり、jは2からnまで増加する。よって、合計で 最大で1+…+(n-1)=n(n-1)/2回
- つまり、合計すると約n^2回手順を実行します。
- よって、挿入ソートの計算量は最大で n^2 に比例することが分かります。
  - ただし、手順3,4,5は各 j に対して最小のとき（例えば、最初から配列が並べ替え完了している様な場合）は 1 回ずつしか実行されませんので、この場合は計算量は n に比例します。

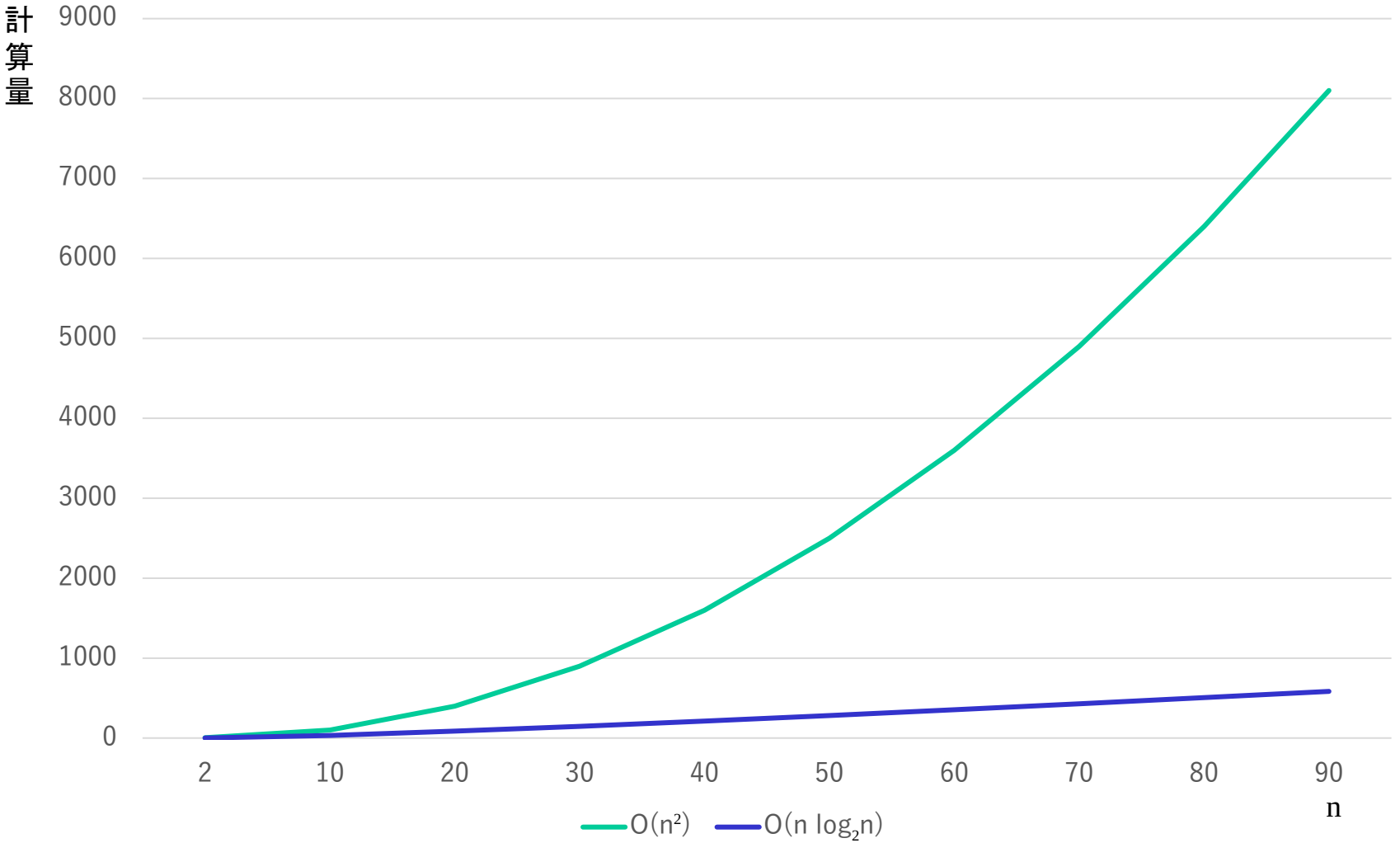
より高速な並べ替えのアルゴリズム

- 今まで見てきたバブルソート・選択ソート・挿入ソートの計算量は（最大で） n^2 に比例していましたが、これらより高速な（計算量の少ない）アルゴリズムが開発されています。
- 例えば、クイックソート、マージソート、ヒープソートという様なアルゴリズムの計算量は概ね $n \log_2 n$ に比例することが知られています。

 CC BY-NC-SA

より高速な並べ替えのアルゴリズム

- 最後に、バブルソートなど($O(n^2)$) とクイックソートなど($O(n \log_2 n)$) の計算量を比較してみましょう。クイックソートの方が高速であることが分かりますね。



東京大学 数理・情報教育研究センター 小林浩二 2021 CC BY-NC-SA