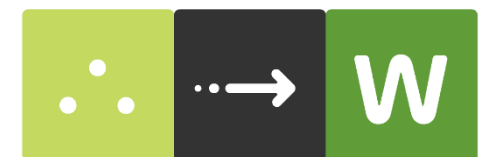




Woodland  
Academy

## 3.1 アルゴリズム例と補足

- 代表的なアルゴリズム
- 補足



*Shape Your Future*

# 目次

- 1 代表的なアルゴリズム
- 2 補足

# 一般的なアルゴリズム

- 実際の開発では、ほとんどのアルゴリズムが既に実装されました。既に実装されている一般的なアルゴリズムを使って慣れ、次に他の既存のアルゴリズムの実装に挑戦し、最後に独自のアルゴリズムを設計という順に追って勉強しましょう。
- ここでは、よくある問題とそれらを解決できるアルゴリズムについて説明します：
  - 配列探索問題：線形探索、二分探索。
  - ソート問題：バブルソート、挿入ソート、クイックソート、マージソート。
  - グラフの検索問題：深さ優先検索、幅優先検索。

# 探索問題

- **探索**[Search]問題とは、データ構造の中から、ある特徴を持ったデータやデータを探し出す問題です。
- また、データは探索された構造にあるかどうかの判断も可能です。

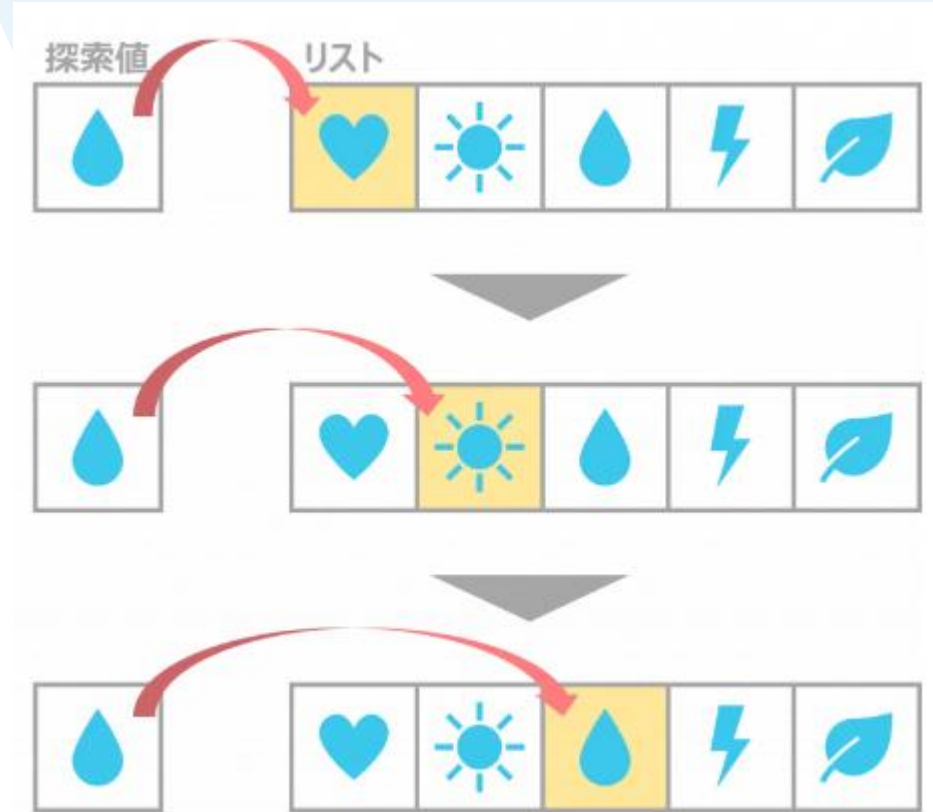
## Example

1. 全生徒のデータから「山田花子」のデータを探します。
  2. 20 世紀の全ての年から最初の閏年を探します。
- まず、単純なデータ構造から考えてみましょう。例えば、配列にあるデータを探索する問題。



# 線形探索

- 最も簡単な方法は、配列全体を**走査**<sup>[Traverse]</sup>し、各要素を順番に取って、条件に一致するかどうかを判断することです。
- 考えましょう：このアルゴリズムの最良の場合とは？ 最悪の場合とは？ それぞれどのくらいの時間がかかるのでしょうか？
- 最良の場合：最初の要素は、望むデータです。時間計算量は  $O(1)$ 。
- 最悪の場合：最後の要素は望むデータか、望むデータが配列にないかのどちらです。時間計算量は  $O(n)$ 。



引用文献：  
<https://udemy.benesse.co.jp/development/python-work/algorithm.html>

# 順序付き配列の探索

- 配列のデータが**順番に並んでいる**場合、この性質を利用して探索を高速化できますか？

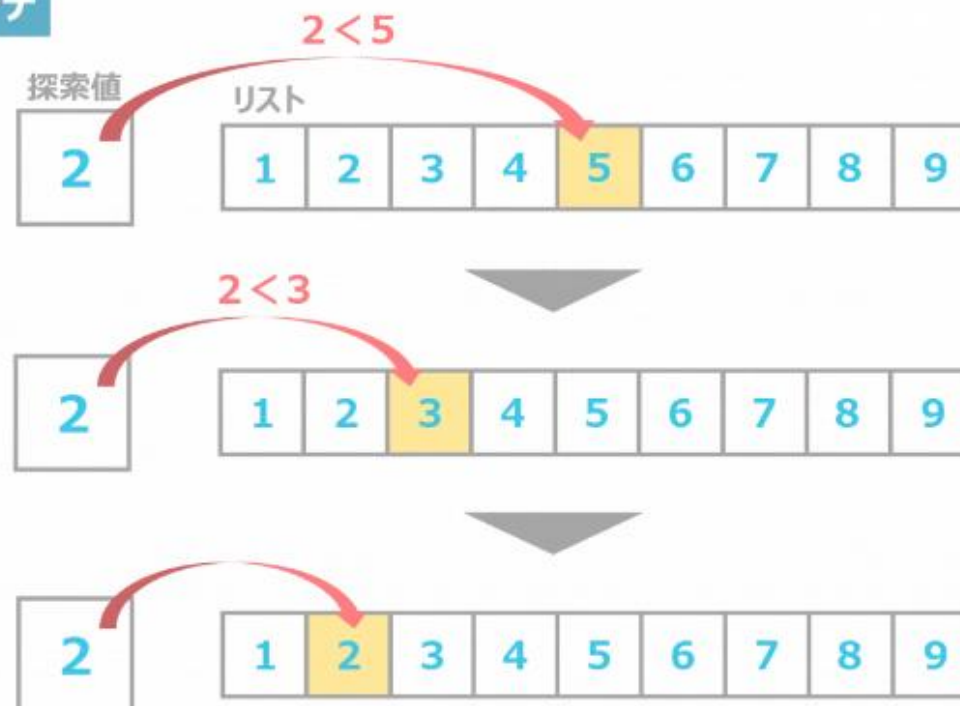
## Example

1. 全生徒を背の低い順に並べました。身長が1.7m 以上の生徒を探します。
2. 電話帳に登録されているすべての電話番号を、小さいものから順に並べ替えました。012-3456-7890 が電話帳に登録されているかどうかを確認します。

# 二分探索

- **二分探索**[Binary Search]は順序配列の高速探索アルゴリズムです。
- 便宜上、以下のデータは小さいものから順に並べていると仮定します。
- 考え方：配列の**真ん中**にあるデータと、探索したいデータを比較します。その二つが同じであれば、結果が見つかりました。探索したいデータの方が小さければ、配列の左半分を検索すればよくて、大きければ、配列の右半分を検索すればよいのです。左半分と右半分の検索では、再び二分探索を使用して検索時間を短縮できます。
- このように、無駄な探索時間が大幅に削減できます。時間計算量は  $O(\log n)$  に最適化されました。

## バイナリサーチ



引用文献：  
<https://udemy.benesse.co.jp/development/python-work/algorithm.html>

# 二分探索の例

## Example ✓

整列された配列 {1, 3, 6, 8, 15, 18, 20}。

6 が配列にあるかどうかを判定。

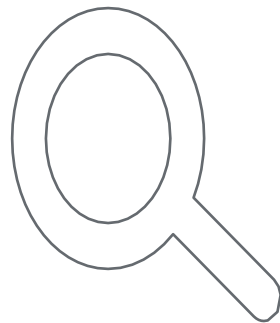
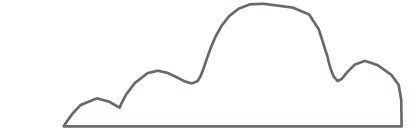
19 が配列にあるかどうかを判定。

Try   
BinarySearch.java





# Q&A



# 整列問題

- **整列**<sup>[Sort]</sup>問題とは、リスト内のデータを一定の順序で並べ替える問題です。
- 数字の場合、単純に小さいものから大きいもの、大きいものから小さいものへと並べることができます。ただし、比較基準を決めておけば、他のデータ型も並べることができます。

## Example ✓

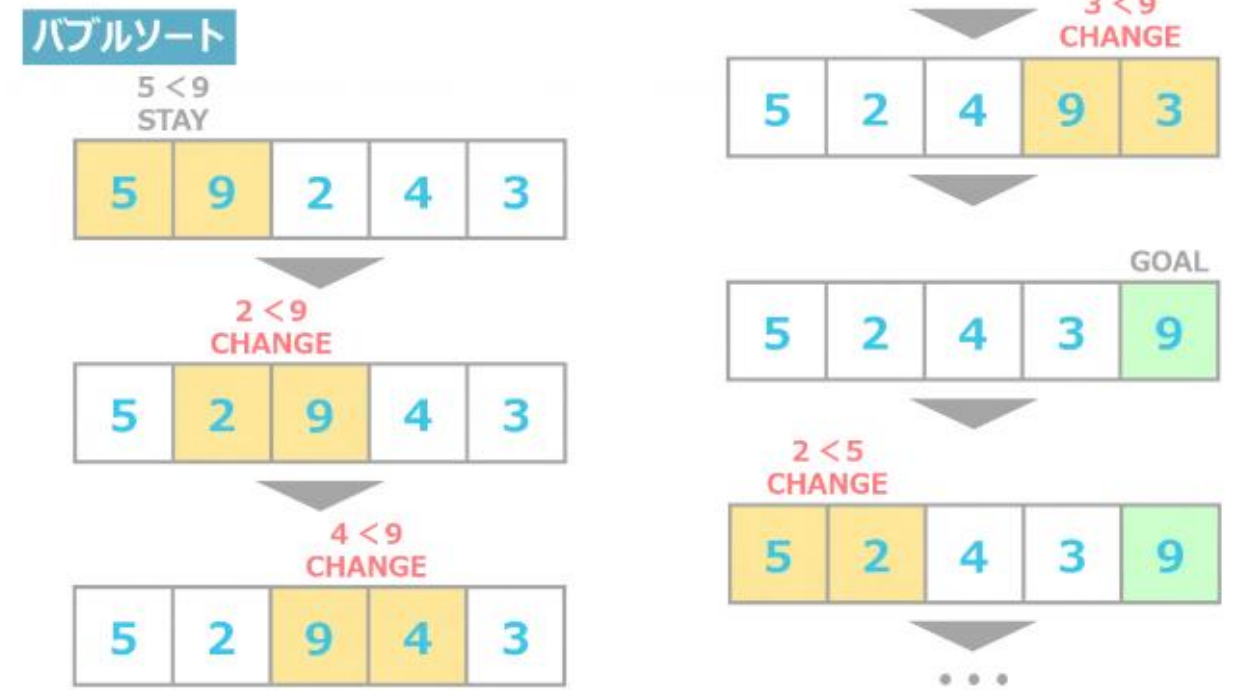
全生徒を背の低い順に整列。  
 全ての企業を平均給与の高いほうから整列。  
 全てのユーザー名を**辞書順**<sup>[Lexicographic Order]</sup>で整列。

- ここは、便宜上、数字の昇順配列のみを考慮します。

# バブルソート

- **バブルソート** [Bubble Sort] は、最も簡単な整列アルゴリズムの一つです。

- 考え方：配列が整列されたら、隣り合う 2 つの数値は、必ず左のほうが右より小さくなるはずです。ですから、ひたすら隣り合う数字のペアを見て、左が右より大きければそれらを入れ替えます。この操作を、配列が整列されるまで繰り返します。



引用文献：

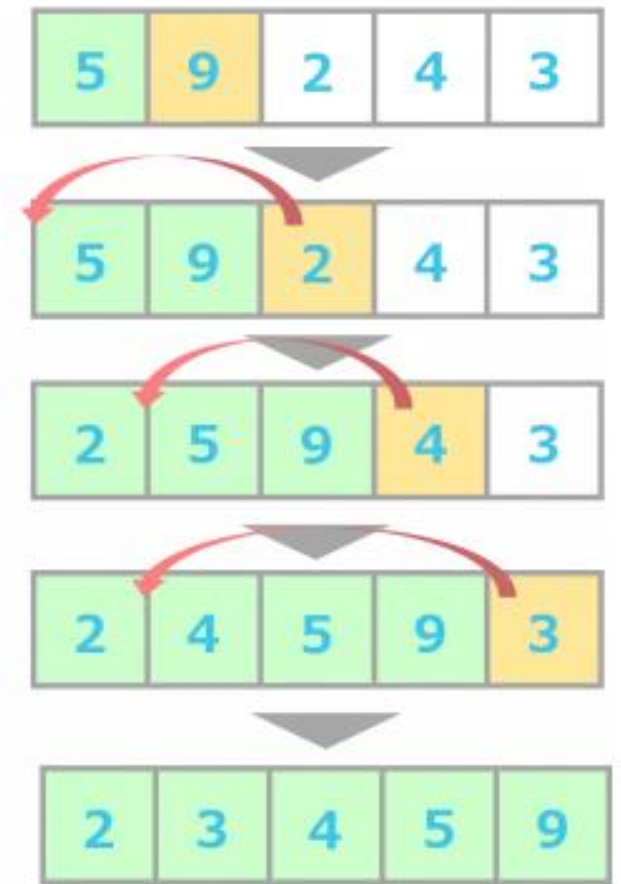
<https://udemy.benesse.co.jp/development/python-work/algorithm.html>

- アルゴリズムの動画はここでご覧いただけます：  
<https://visualgo.net/en/sorting>

# 挿入ソート

- **挿入ソート** [Insertion Sort] も、簡単なソートアルゴリズムの一つです。


- 考え方：ポーカーでカードを管理するのと似ています。まず、全てのカードを右手に適当に並べます。右手から無作為にカードを一枚取りだして、左手にそれを置きます。そして、右手から一枚ずつカードを取って、左手の正しい位置にカードを入れて、常に左手のカードを順番に並べていきます。これを、全てのカードは順番になるまで繰り返します。



- アルゴリズムの動画を見ましょう。

引用文献：  
<https://udemy.benesse.co.jp/development/python-work/algorithm.html>

# クイックソート

- **クイックソート**<sup>[Quicksort]</sup>は、実装が複雑ですが、効率のいい整列アルゴリズムです。
- クイックソートの考え方自体は簡単です：
  1. 適当に数  $k$  を取り、全ての数を、「 $k$  より大きい数」と「 $k$  より小さい数」に分けます。
  2.  $k$  より小さい数を  $k$  の左側に置き、それらをクイックソートで並べ替えます。
  3.  $k$  より大きい数を  $k$  の右側に置き、それらをクイックソートで並べ替えます。
- 気づきましたか？これは、前に（ § 1.3.3）紹介した再帰の考え方です。再帰の終了条件は何でしょうか？
- アルゴリズムの動画を見ましょう。

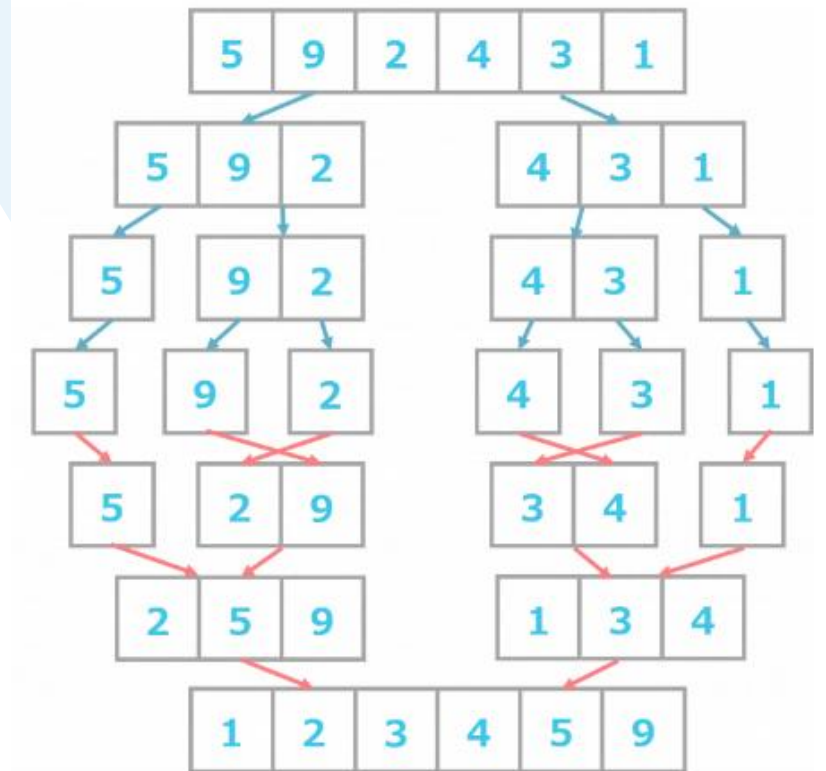


引用文献：  
<https://udemy.benesse.co.jp/development/python-work/algorithm.html>



# マージソート

- **マージソート** [Merge Sort] も、複雑な整列アルゴリズムの一つです。
- 考え方もまた再帰を利用：
  1. 配列を真ん中から 2 つに分けます。
  2. 左半分をマージソートで整列します。
  3. 右半分をマージソートで整列します。
  4. この 2 つの整列された配列を、1 つの整列された配列に結合（マージ [Merge]）します。
- マージソートが効率高いのは、ステップ 4 のマージ操作の時間計算量は  $O(n)$  だけからです。
- アルゴリズムの動画を見ましょう。



引用文献：

<https://udemy.benesse.co.jp/development/python-work/algorithm.html>

# 整列アルゴリズムの計算量の比較

- 各整列アルゴリズムの時間計算量は何でしょうか。
- これらのアルゴリズムの時間および空間計算量はこの通り：

アルゴリズム	最悪時間計算量	平均時間計算量	空間計算量
バブルソート	$O(n^2)$	$O(n^2)$	$O(1)$
挿入ソート	$O(n^2)$	$O(n^2)$	$O(1)$
クイックソート	$O(n^2)$	$O(n \log n)$	$O(\log n)$
マージソート	$O(n \log n)$	$O(n \log n)$	$O(n)$

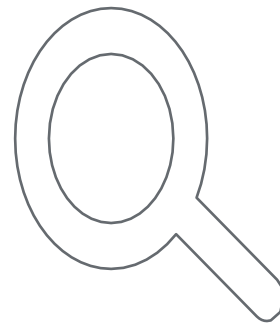
- 実際、データの比較に基づいた整列アルゴリズムでは、 $O(n \log n)$  より早いもの存在しません。

# 整列アルゴリズムの使用

- 複雑すぎて覚えられないとお思いですか？実際には、効率的なクイックソートとマージソートのアルゴリズムだけを使用すればよいので、心配は要りません。
- そして、それらを実装する必要もありません：現代のプログラミング言語では、基本的に**標準的な整列メソッド**が提供されているので、**それを直接利用**すればよいです（これについては次章で説明します）。
- 例えば、Java のリストでは標準な整列アルゴリズムが提供されて、マージソートの変種で実現されました。



Q&A



# グラフの探索問題

- 偶に、複雑なデータ構造の中にあるデータを探索する必要があります。例えば、**グラフ**です。

## Example

地下鉄で東京駅から渋谷駅を探します。

ユーザーのパソコンからインターネット上にあるサーバのパソコンを検索します。

ルートディレクトリから、指定されたファイル名のファイルを見つけます。

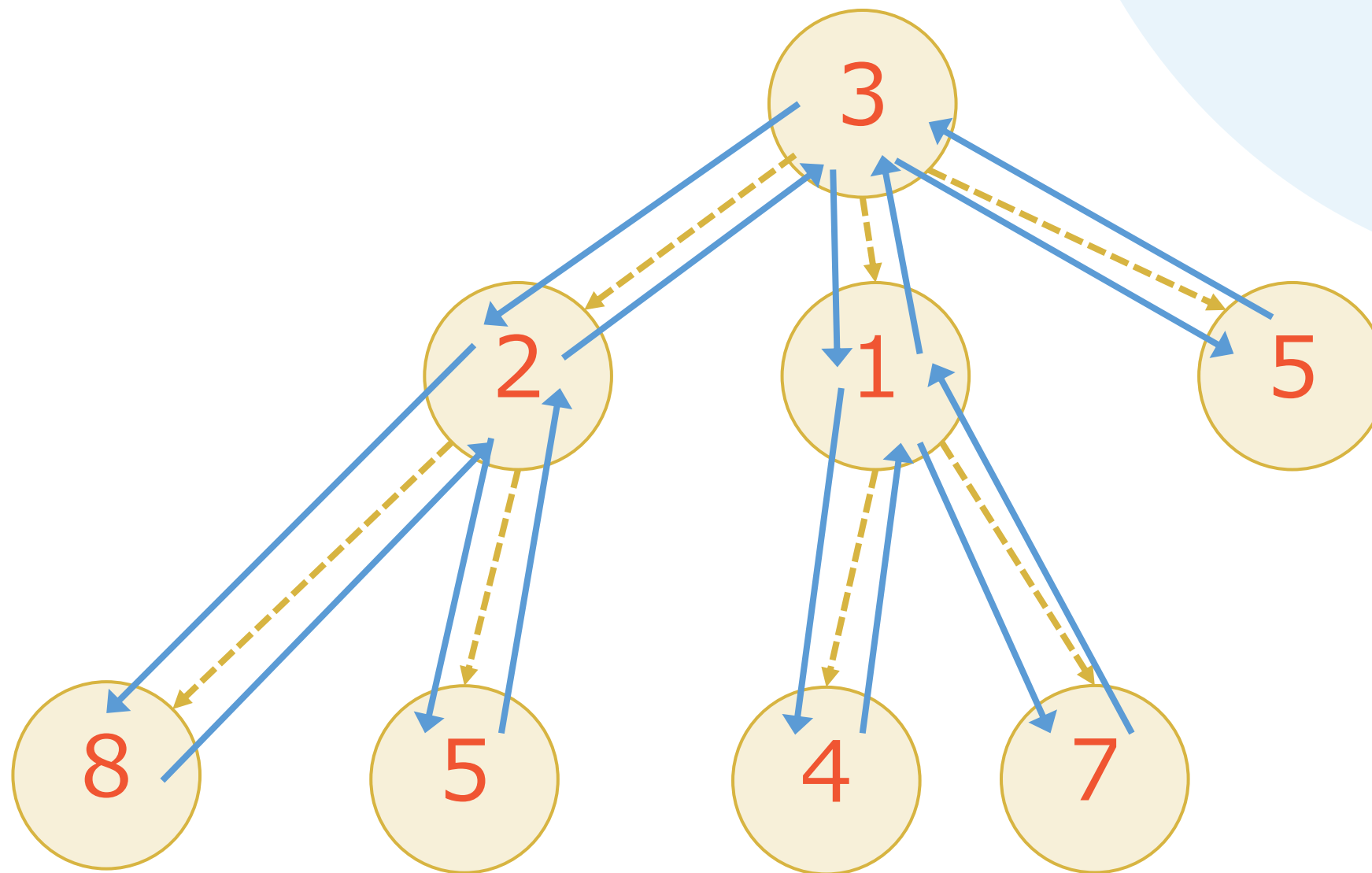


# グラフの探索アルゴリズム

- 最も基本的で一般的に使用されているグラフの探索アルゴリズムは 2 つあります：
  - **深さ優先探索**[Depth First Search, DFS]。
  - **幅優先探索**[Breadth First Search, BFS]。
- どちらの探索方法も、グラフ内のすべてのノードを重複や省略なしに探索しようとしています。
- ここで、簡単なグラフである**木**を例として、それらのアルゴリズムがどのように実装されるか見ていきましょう。より複雑なグラフでも考え方は基本同じです。

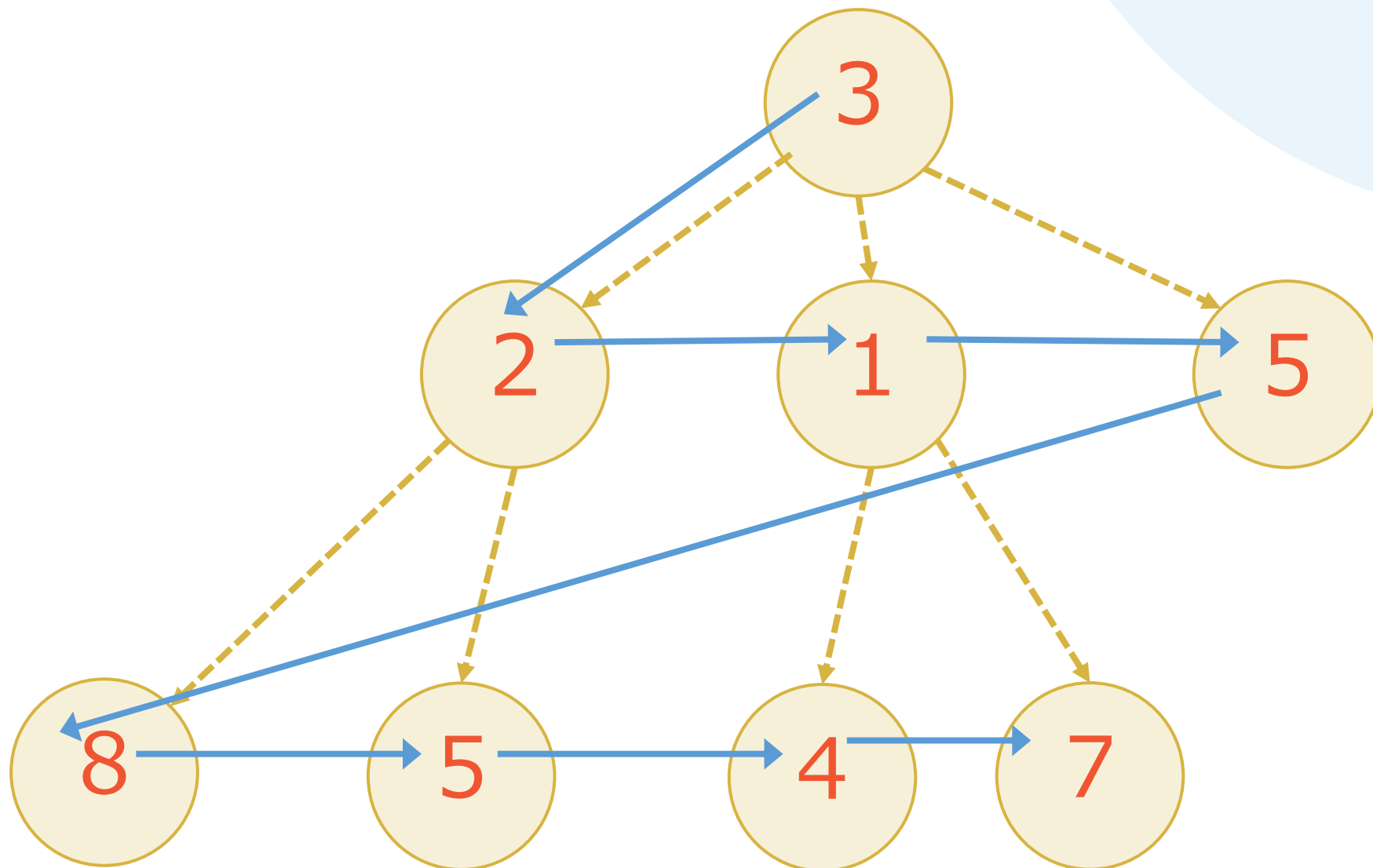
# 深さ優先探索

- **深さ優先探索**は、ルートから始まり、できるだけ**子**を探索し、全ての子を探索し終わったら親ノードに戻ります：



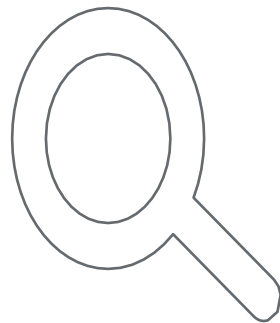
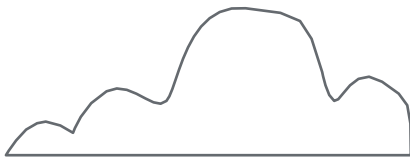
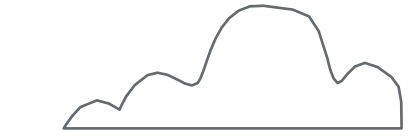
# 幅優先探索

- **幅優先探索**もルートから始まり、その子をできるだけ探索してから、子の子をできるだけ探索し、子の子の子を探索……の順に探索を行います。





# Q&A



# 目次

1 代表的なアルゴリズム

2 補足



# その他のデータ構造とアルゴリズム

- この章では、より高度的に実用的データ構造とアルゴリズムを紹介します。
- 今は、これらのデータ構造とアルゴリズムが何の機能を持つかの印象を持つだけでよく、詳細な実装方法と計算量などは、実際に使いたいときに調べましょう。

# 二分探索木

- **二分木** [Binary Tree] は、特殊な木構造で、各ノードは2つ以上の子を持ちません。
- **二分探索木** [Binary Search Tree, BST] は、二分探索の考え方に基づく特殊な二分木です。データを順番に保存することができ、データの検索、データの削除、新しいデータの挿入の平均時間計算量は  $O(\log n)$  です。
- **平衡二分探索木** [Balanced BST] は二分探索木の最適化です。通常の BST では、最悪なケースの計算量が  $O(n)$  に対し、平衡化された二分木は最悪の場合も  $O(\log n)$  を保証します。AVL 木、赤黒木、スプレー木など様々な実装方法があります。
- 二分探索木は、主に頻繁に変更する必要がある順序付けられたデータを格納するために使用されます。

# ヒープと優先度付きキュー

- **ヒープ** [Heap] は特別な木です：各ノードは、そのどの子ノードよりも大きい（最大ヒープ）・小さい（最小ヒープ）。一般に使用されるヒープは二分ヒープです。
- **優先度付きキュー** [Priority Queue] は抽象データ型で、通常はヒープで実装されます。優先度付きキューに格納されたデータは順番に保存され、以下の操作ができます：
  - データを挿入。ヒープで実装した場合の計算量は  $O(\log n)$ 。
  - 最大データを取得（そして削除）。ヒープで計算量は  $O(\log n)$ 。
  - 最大データを取得（削除せず）。ヒープで計算量は  $O(1)$ 。
- 優先度付きキューは、優先順位によって使用する必要があるデータを格納するために使います。

# 特別な集合

- **ビット集合**<sup>[Bitset]</sup> は、特別な集合で、整数しか格納できませんが、使われるメモリの容量を大幅に節約できます。
- **素集合**<sup>[Disjoint Set]</sup> は、特殊な集合で、共通のない複数の集合を扱います。集合の基本操作に加え、主に以下の集合間操作が可能：
  - ある要素が複数の集合のどれに属するかを調べます。
  - 二つの集合を一つにまとめます。
- 素集合の実装にはいくつか種類がありますが、最適化されたものは上記の操作の時間計算量が全て  $O(1)$  くらいになります。
- 素集合は、主にいくつかのグラフのアルゴリズムを補助するために使用されます。

# 他の整列アルゴリズム

- 先ほど説明した整列アルゴリズム以外にも、様々なアルゴリズムがあります：
  - **選択ソート** [Selection Sort]。もう一つの単純なソートアルゴリズム。時間計算量は  $O(n^2)$ 。
  - **ヒープソート** [Heap Sort]。ヒープに基づく整列アルゴリズムです。時間計算量は  $O(n \log n)$ ，マージやクイックソートの代わりに使われることもあります。
  - **シェルソート** [Shell Sort]、**カウントソート** [Counting Sort]、**バケットソート** [Bucket Sort] など。これらのアルゴリズムは、データの比較に基づいていないために、時間計算量は  $O(n)$  にあります。ただし、限定的な状況のみ使えます。例えば、データがある範囲内の正の整数のみである場合。



# グラフのアルゴリズム

- **ダイクストラ法**[\[Dijkstra's Algorithm\]](#) : あるノードから他の全てのノードまでの最短距離を計算する。
- **ベルマン - フォード法**[\[Bellman-Ford Algorithm\]](#) : ノードからの最短距離を計算。ただし、エッジの値が負数でも可能。
- **ワーシャル - フロイド法**[\[Floyd-Warshall Algorithm\]](#) : 全ノード間の最短距離を計算。
- **A\* アルゴリズム** : 2 つのノード間の最短なルートを探索でき、道案内やナビゲーションに広く使用されている。
- **フォード - ファルカーソン法**[\[Ford-Fulkerson Algorithm\]](#) : ある場所から別の場所への最大トラフィック量を計算。


# 文字列のアルゴリズム

- **KMP アルゴリズム**[\[Knuth–Morris–Pratt Algorithm\]](#)：長い文字列の中の特定の単語の位置を求める場合で広く使用されます。
- **ボイヤー - ムーア法**[\[Boyer-Moore Algorithm\]](#)：特定の単語の出現位置を求めるのにも使われるが、従来のアルゴリズムより効率的です。このアルゴリズムは、検索エンジンで広く使用されます。
- **トライ木**[\[Trie\]](#)は、多数の文字列の集合（辞書）を保存するためのデータ構造で、入力補完やスペルチェックなどで広く使用されます。

# アルゴリズム設計パターン

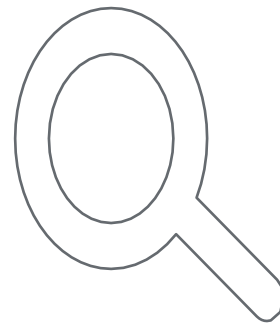
- これまでの紹介は、既に存在するアルゴリズムについてのものでしたが、自分たちでアルゴリズムを設計するとしたらどうでしょうか。ここでは、より一般的なアルゴリズム設計のアイデアを紹介します。自分で勉強したい方は、まずここから始めてみましょう：
  - **しらみつぶし**[\[Brute-force\]](#)
  - **貪欲法**[\[Greedy\]](#)
  - **分割統治法**[\[Divide-and-conquer\]](#)
  - **バックトラック法**[\[Backtracking\]](#)
  - **動的計画法**[\[Dynamic Programming, DP\]](#)

# さらなる勉強

- 様々なデータ構造やアルゴリズムに慣れ、それらを実装し、さらには設計する能力を向上させるには、実際にプログラミングして、問題を解決するほどいい方法はありません。
- 実際に解くべき問題がない場合は、 LeetCode などのアルゴリズム問題練習サイトを通じてオンラインで練習するのもいいでしょう：  
 <https://leetcode.com/>



# Q&A



# まとめ

Sum Up



1.基本的なアルゴリズム：探索、ソート問題：

① 二分探索の考え方。

2.さらなる勉強方法：実践。

# Thank you!

From Seeds to Woodland — Shape Your Future.



*Shape Your Future*