Sorting algorithms in Python: Overview

In this course, you'll learn about...

- What sorting means in a programming context
- Why sorting techniques are an integral part of any programmer's toolbox
- Common sorting algorithms and how they're implemented in Python
- How to evaluate and compare the performance of different algorithms

TABLE OF CONTENTS

What “Sorting” Means in Programming

- **Sorting**: putting a collection of items into a well-defined order (e.g. alphabetical, numerical, etc.)
- This implies that there must be a way to *compare* elements in that collection. Often this can be a simple numerical comparison of some sort, but it can be a lot more complex.
- Imagine you have a database with information about customers at an online store, and you’re trying to determine who to invite to participate in a rewards program. You might want to give first priority to people who have been customers longest, but also might want to weigh how much money each customer has spent and when the last time they made a purchase was.

Sorting Algorithm Keywords

- **Runtime**: How quickly the algorithm runs (O(n), O(n log(n)), O(n^2), etc.)
- **Stable sort**: A sorting algorithm that generates stable results when sorted multiple times (e.g. if you sort a list of store item objects alphabetically, then by price, are the items that are equal in price still sorted alphabetically?)
- **In-place sort**: A sorting algorithm that doesn’t require any additional memory space other than the memory taken up by its input
- **Pathological case:** A case in which the input data is specifically designed to make your algorithm run as slowly as possible

Why Sorting Matters

- **Searching**: Searching for an item on a list works much faster if the list is sorted.
- **Selection**: Selecting items from a list based on their relationship to the rest of the items is easier with sorted data. For example, finding the *kth*-largest or smallest value, or finding the median value of the list, is much easier when the values are in ascending or descending order.
- **Duplicates**: Finding duplicate values on a list can be done very quickly when the list is sorted.
- **Distribution**: Analyzing the frequency distribution of items on a list is very fast if the list is sorted. For example, finding the element that appears most or least often is relatively straightforward with a sorted list.

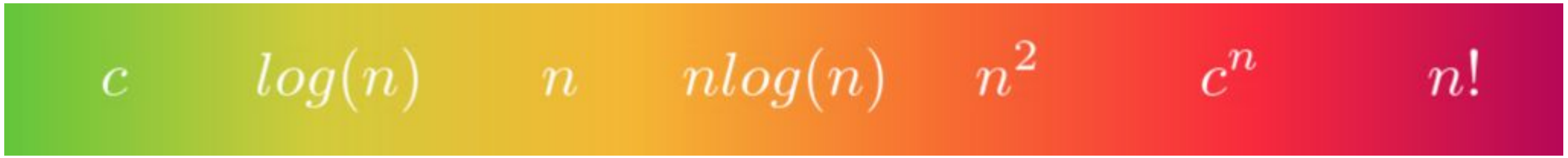
TABLE OF CONTENTS

Time complexity

- Two general ways to measure how long an algorithm takes
    - Directly (in minutes, seconds, milliseconds, etc.)
    - Relatively (in terms of number of operations completed)
- **Big-O Notation**: represents the worst case runtime of an algorithm as a function of number of operations compared to the size of the input, or *n*.
    - If *everything* goes wrong, how many operations would the algorithm do?

Time complexity

Algorithms can be divided into different “classes” based on their worst-case runtime. Of course, certain algorithms in the same classes might be faster or slower than one another in practice, but this is an easy way to classify runtimes generally.



Common runtimes of algorithms

TABLE OF CONTENTS

Bubble Sort

- Basic idea: move the largest item in the list to the end of the list, then the second-largest to the second-to-last position in the list... etc.
- Named because the element in question “bubbles” to the top of the list in a series of swaps with adjacent elements.

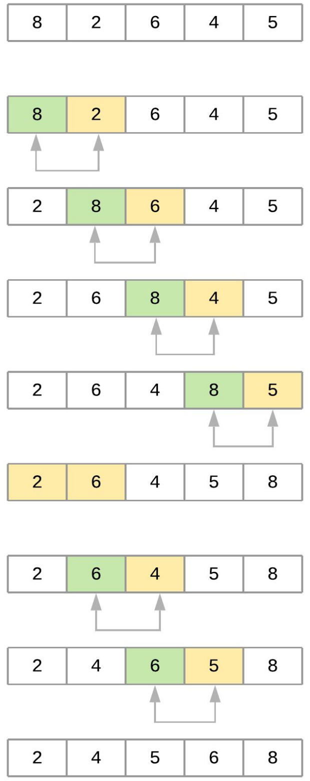


TABLE OF CONTENTS

Insertion Sort

- Basic idea: starting at the second element of the list, move each item so that it's sorted with respect to all items before it.
- "Inserts" each element into its correct place in a sorted sublist that grows to become the full list.

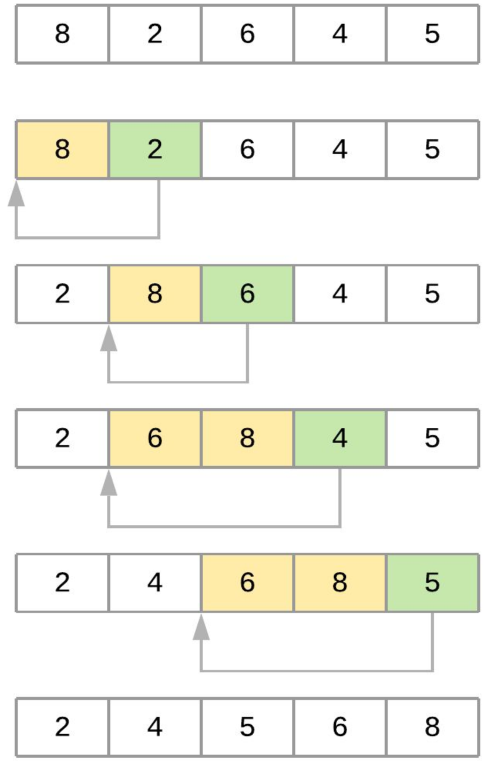


TABLE OF CONTENTS

Merge Sort

- Basic idea: break the list into two halves, sort those halves recursively using merge sort, then “merge” the halves together again.
- Relies on the fact that merging two sorted lists is an O(n) operation.

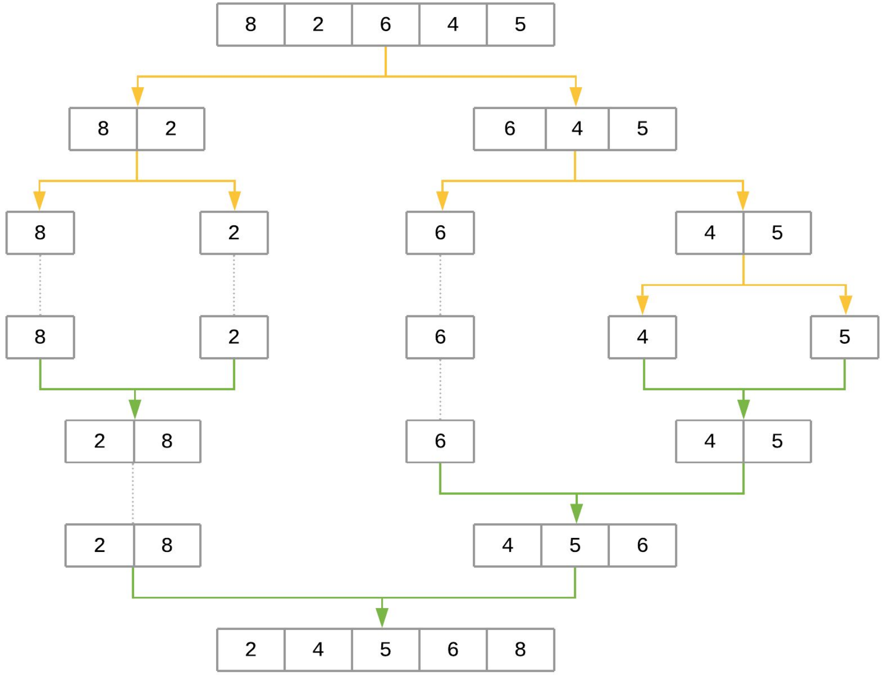


TABLE OF CONTENTS

Quicksort

- Basic idea: Choose a pivot element, then divide the list into three sections → elements that are less than, equal to, and greater than the pivot element.
- Call quicksort recursively on the less-than and greater-than sublists

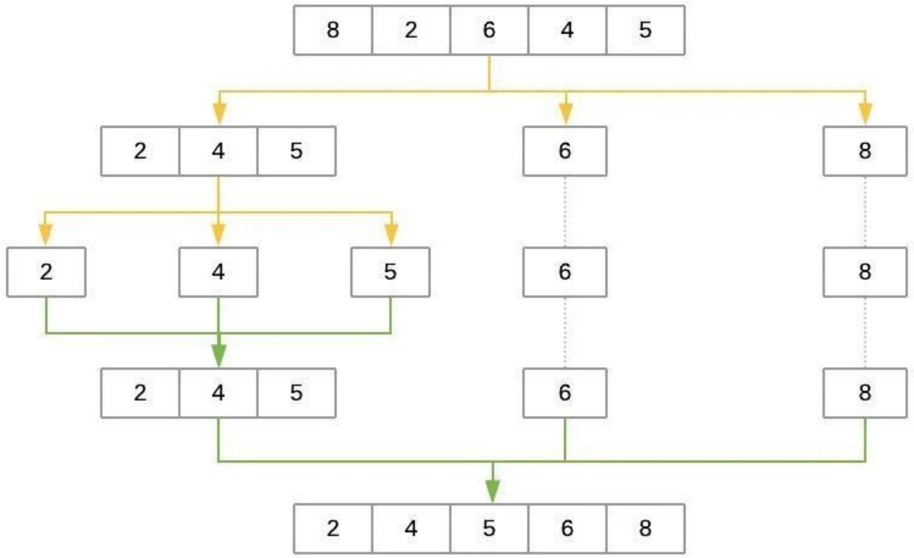


TABLE OF CONTENTS

Timsort

- Basic idea: Divide your list into small sections and use insertion sort to sort all of those sections. Then merge those sections together two at a time until the whole list is sorted.
- Timsort is an example of a “block merge” sorting algorithm
- Leverages the strengths of insertion sort (fast on small lists, stable, O(n) best-case performance) and the strengths of merge sort (improved performance on larger lists, O(n log n) worst-case performance), while mitigating the weaknesses of both algorithms.

Sorting algorithms in Python: Summary

In this course, you've learned...

- What sorting is and why it's such a powerful tool
- Different features of sorting algorithms and the tradeoffs therein
- Common sorting algorithms and how they're implemented in Python

Sorting algorithms in Python: Summary

Algorithm	Worst-case Runtime	Best-case Runtime	Average-case Runtime	Space Complexity	Stable	In-place
Bubble Sort	O(n^2)	O(n)	O(n^2)	O(1)	Yes	Yes
Insertion Sort	O(n^2)	O(n)	O(n^2)	O(1)	Yes	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	No
Quicksort	O(n^2)	O(n log n)	O(n log n)	O(log n)	No	Kind of
Timsort	O(n log n)	O(n)	O(n log n)	O(n)	Yes	No