Sorting

1

Sorting is a basic building block that many other algorithms are built upon.

2

Sorting algorithms are some of the most thoroughly studied algorithms in computer science.

3

You can use sorting to solve a wide range of problems:

Searching: Searching for an item on a list works much faster if the list is sorted.

4

You can use sorting to solve a wide range of problems:

Selection: Selecting items from a list based on their relationship to the rest of the items is easier with sorted data.

5

You can use sorting to solve a wide range of problems:

Duplicates: Finding duplicate values on a list can be done very quickly when the list is sorted.

6

You can use sorting to solve a wide range of problems:

Distribution: Analyzing the frequency distribution of items on a list is very fast if the list is sorted.

7

Python's Built-In Sorting

```
lst = ['text', 'asymmetrical', 'ragamuffins']

sorted(lst) # returns a new, sorted Iterable

lst.sort() # sorts in-place
```

8

Asymptotic Analysis

9

Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical bounding of its run-time performance.

10

Asymptotic Complexities

- Big O notation expresses the upper bound or *worst-case* time complexity as the size of the input grows.
- Big Ω notation expresses the lower bound or *best-case* time complexity as the size of the input grows.
- Big ϴ notation expresses the *average case* time complexity as the size of the input grows.

11

Time Complexity

"In computer science, the time complexity is the computational complexity that describes the amount of computer time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform."

— Wikipedia

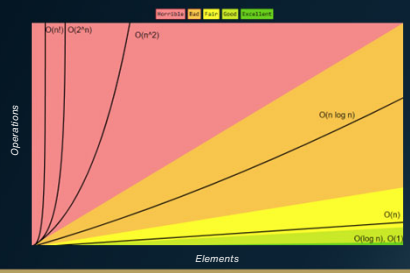
12

Big O Notation

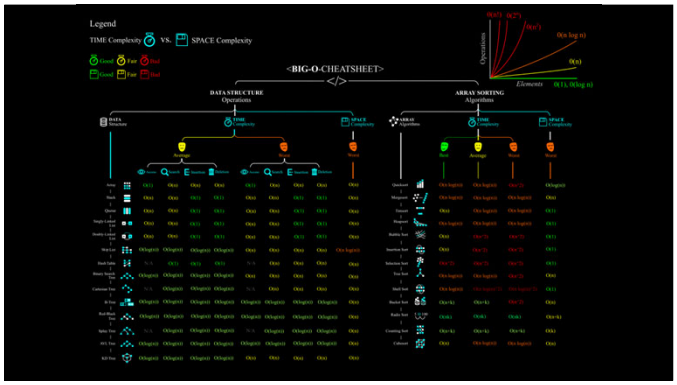
Big O Notation is used to describe the time complexity of algorithms. It calculates the time taken to run an algorithm *as the input grows*. In other words, it calculates the worst-case time complexity of an algorithm.

13

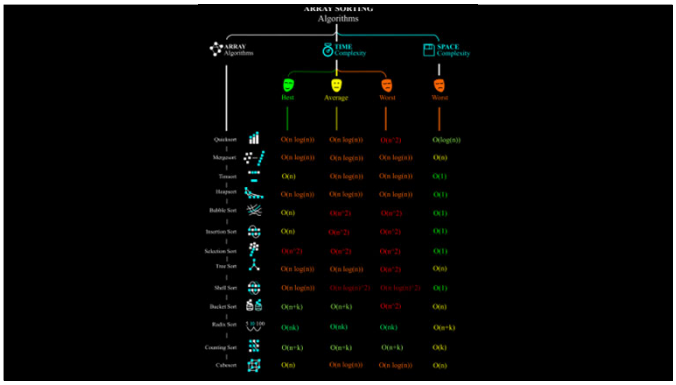
Big-O Complexity Chart



14



15



16

Timing Your Code

17

Python offers the `timeit` module.

`timeit.timeit()` is a little odd to use as it requires the code to be timed to be passed into it as a string.

18

```
timeit(
    stmt="pass",
    setup="pass",
    timer=time.perf_counter,
    number=1_000_000,
    globals=None
)
```

19

Test Harness

20

Measuring Efficiency With Big O Notation

21

The specific time an algorithm takes to run isn't enough information to get the full picture of its time complexity.

22

The time in seconds required to run different algorithms can be influenced by several unrelated factors, including processor speed or available memory.

Big O, on the other hand, provides a way to express runtime complexity in hardware-agnostic terms.

With Big O, you express complexity in terms of how quickly your algorithm's runtime grows relative to the size of the input, especially as the input grows arbitrarily large.

23

Bubble Sort

24

demo

25

Measuring Bubble Sort's Big O Runtime Complexity.

26

Timing Our Bubble Sort Implementation.

27

Analyzing the Strengths and Weaknesses of Bubble Sort

Strengths
- Simplicity

Weaknesses
- Speed

28

Insertion Sort

29

demo

30

Measuring Insertion Sort's Big O Runtime Complexity.

31

Timing Our Insertion Sort Implementation.

32

Analyzing the Strengths and Weaknesses of Insertion Sort

Strengths

- Simplicity
- More efficient than other quadratic sorting algorithms
- Faster than some "better" algorithms on small sets of data

Weaknesses

- Speed
- Still not practical for large data sets

33

Merge Sort

34

Divide and Conquer

1. The original input is broken into several parts, each one representing a subproblem that's similar to the original but simpler.
2. Each subproblem is solved recursively.
3. The solutions to all the subproblems are combined into a single overall solution.

35

demo

36

Measuring Merge Sort's Big O Runtime Complexity.

37

Timing Our Insertion Sort Implementation.

38

Analyzing the Strengths and Weaknesses of Insertion Sort

Strengths

- Very efficient
- Scales well
- Easy to parallelize

Weaknesses

- For small data sets the cost of the recursion can slow performance
- Less memory efficient as copies of its data are made

39

Quicksort

40

demo

41

Selecting the **`pivot`** Element

Quicksort's efficiency often depends on the pivot selection.

If the input array is unsorted, then using the first or last element as the pivot will work the same as a random element.

But if the input array is sorted or almost sorted, using the first or last element as the pivot could lead to a worst-case scenario.

Selecting the pivot at random makes it more likely Quicksort will select a value closer to the median and finish faster.

42

Measuring Insertion Sort's Big O Runtime Complexity.

43

Timing Our Insertion Sort Implementation.

44

Analyzing the Strengths and Weaknesses of Insertion Sort

Strengths

- Very fast!
- Scales well
- Easy to parallelize
- In general, it beats almost all other sorting algorithms

Weaknesses

- Lack of a guarantee that it will achieve its average runtime complexity
- Like merge sort, it trades memory for speed

45