CoGrammar

Lecture 12: Searching and Sorting (Data Structure and Algorithms)

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

Lecture Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(FBV: Mutual Respect.)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Open Classes. You can submit these questions here: **Open Class Questions**

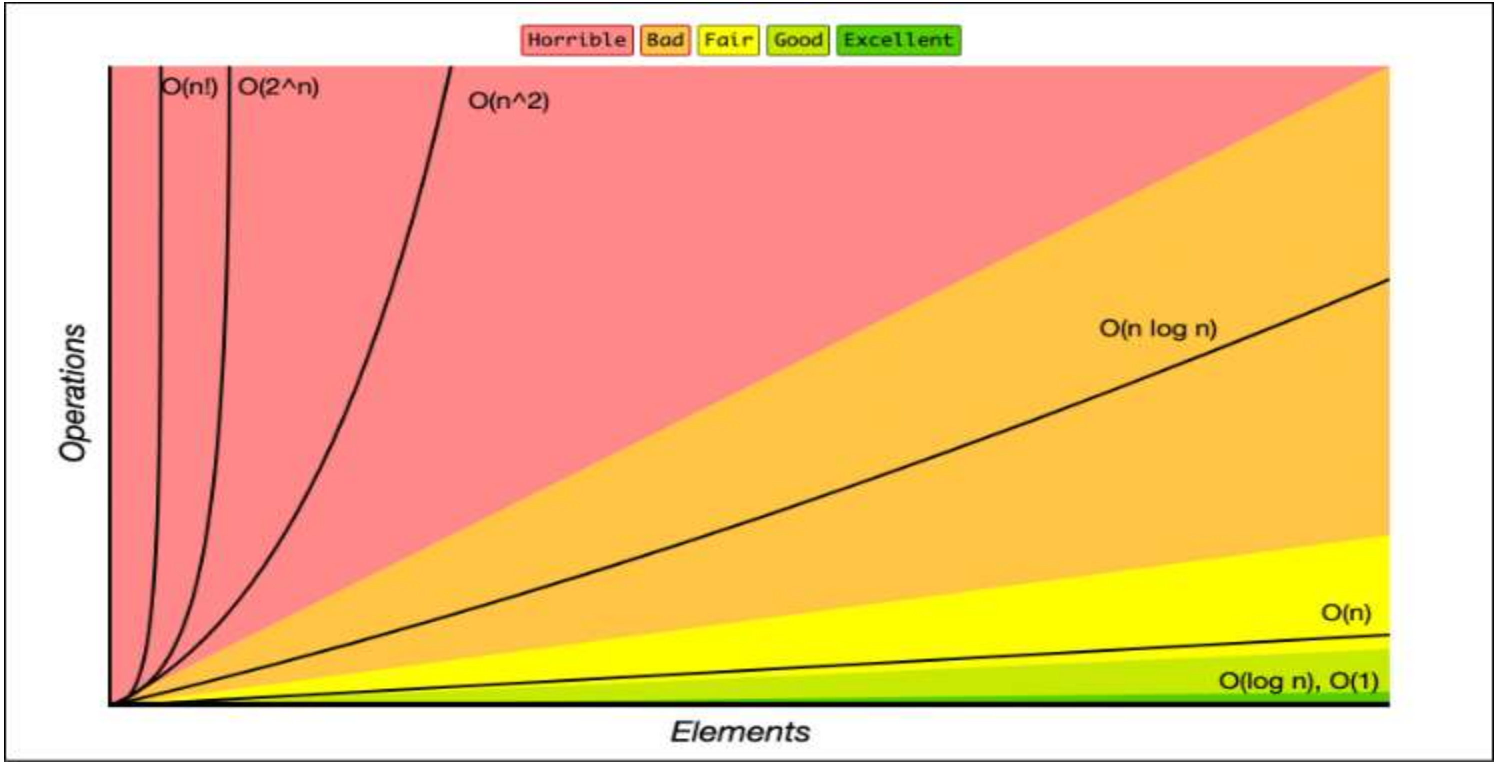
Lecture Housekeeping cont.

- For all **non-academic questions**, please submit a query: www.hyperiondev.com/support
- Report a **safeguarding** incident: www.hyperiondev.com/safeguardreporting
- We would love your **feedback** on lectures: [Feedback on Lectures]()

Lecture Objectives

- Learn about the different data structure and algorithms
- Learn about Big O and how to evaluate the performance of your code
- Making informed decisions as a software engineer

BIG O


Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

BIG O

What is it

- **n** represents the number of inputs that we are passing
  - If we had a list of 10000 records, n would be 10000
- Used to measure the time complexity of an algorithm
- Used to measure the space complexity of an algorithm
- Helps developers make informed decisions on the best algorithm to use
- Usually broken up into, **best, average** and **worst** complexity
  - We usually look at the worst and average cases when choosing an algorithm to implement

BIG O

We Should Aim for

- ★ **O(1)**
  - Mainly when working with data structures, we want to have an O(1) when reading or writing on an **average case** that fits our operations
- ★ **O(log n)**
  - This is the ideal complexity for searching data
- ★ **O(n)**
  - Good when working with unstructured data or performing infrequent operations on a data structure
- ★ **O(n log n)**
  - The worst time complexity that we can realistically accommodate, great for sorting data

BIG O

Terrible Complexities

- **O(n^2)**
  - Okay for small operations, but as we have more data, the operations we perform double
- **O(2^n)**
  - There is no scenario where this is acceptable, for values of n that are less than 100 we we can get into a situation where our code will take years to run (Literally years)
- **O(n!)**
  - Worst case scenario

If we had a list of 1 million records, what time complexity is the most ideal for getting a specific element

A. O(n log n)
B. O(n^2)
C. O(n!)
D. O(log n)

CoGrammar

Q & A SECTION

Please use this time to ask any questions relating to the topic, should you have any.

Data Structures

What Are They

- ★ Used to store our data in memory
- ★ Structure our data in a way that is easy to access

CoGrammar

Basic Data Structures

As software engineers, these are the data structures that you are guaranteed to use at some point in your career

- ★ Arrays
- ★ Hash Table
- ★ Hash Set

Arrays

- ★ Stores a predefined number of elements
- ★ Makes use of indexing to access values
- ★ Only stores values of the same data type
- ★ Basically the same as a list, but with some constraints

0	"H"
1	"E"
2	"L"
3	"L"
4	"O"
5	"."

Arrays vs List

Array

★ Values are stored next to each other in memory (contiguous)

★ Has an O(1) time complexity for inserts and reading

List

★ Dynamically sized, elements can be added and removed easily

★ Have more use-cases than arrays

★ Time complexity of O(n) for key operations.

Hash Table

- ★ Can be called a **Hash Map**
- ★ Uses key value pairs for storing and referencing data
- ★ Highly efficient, O(1) for read and write operations
- ★ Values are not stored in order, knowledge of the values being stored is required in order to access them
- ★ Implemented in Python as **Dictionaries**

CoGrammar

Hash Table

- ★ Keys are used to index the values
- ★ Keys can be of any data type
- ★ Values can be of any data type

"first_name"	"Jack"
"last_name"	"Jackson"
"age"	50
"height"	165

Hash Set

★ Used for storing unique values
★ Does not store values in any order
★ Items are accessed by calling their value

Advanced Data Structures

There are 2 main types of data structures that we can work with in programming, each have their benefits and use-cases

- ★ Linear
- ★ Non-Linear

Linear Data Structures

Linear data structures store data in a directed manner where each item can only have 1 other item pointing to it, and each item can only point to one other item

- ★ Arrays
- ★ Lists
- ★ Stacks
- ★ Queues
- ★ Linked Lists

Stacks

Method of ordering

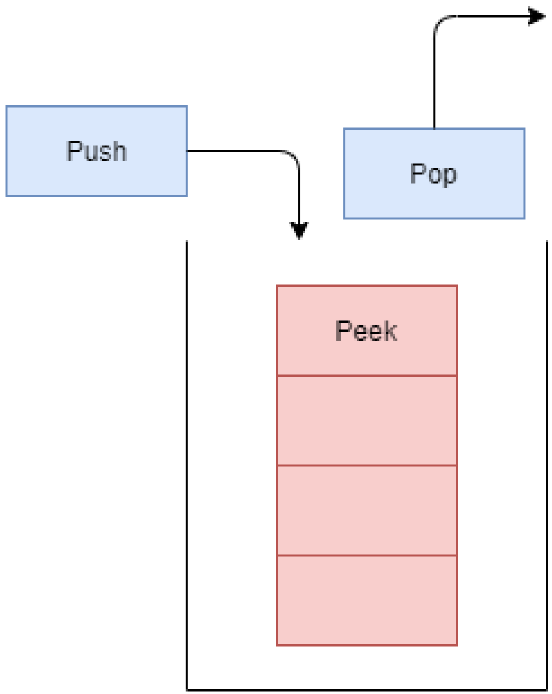
- ★ **LIFO:** Last in, First out
- ★ Elements are added to the top of the stack
- ★ Items are removed from the top of the stack

Operations

- ★ **Push:** Adds an item to the top of the stack
- ★ **Pop:** Removes and returns the top item in the stack
- ★ **Peek:** See what element is coming next in the stack

Note: We can not iterate through a stack, we can only get the top value

Stacks



CoGrammar

Queue

Method of ordering

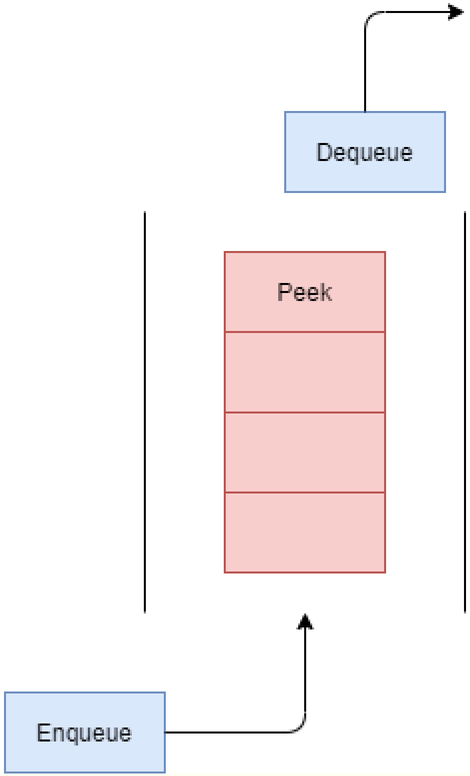
★ **FIFO:** First in, First Out
★ Elements are added to the back of the queue
★ Items are removed from the front of the queue

Operations

★ **Enqueue:** Adds an item to the back of the queue
★ **Dequeue:** Removes and returns the first value in the queue
★ **Peek:** See what element is coming next in the queue

Note: We can not iterate through a queue, we can only get the top value

Queue



Linked List

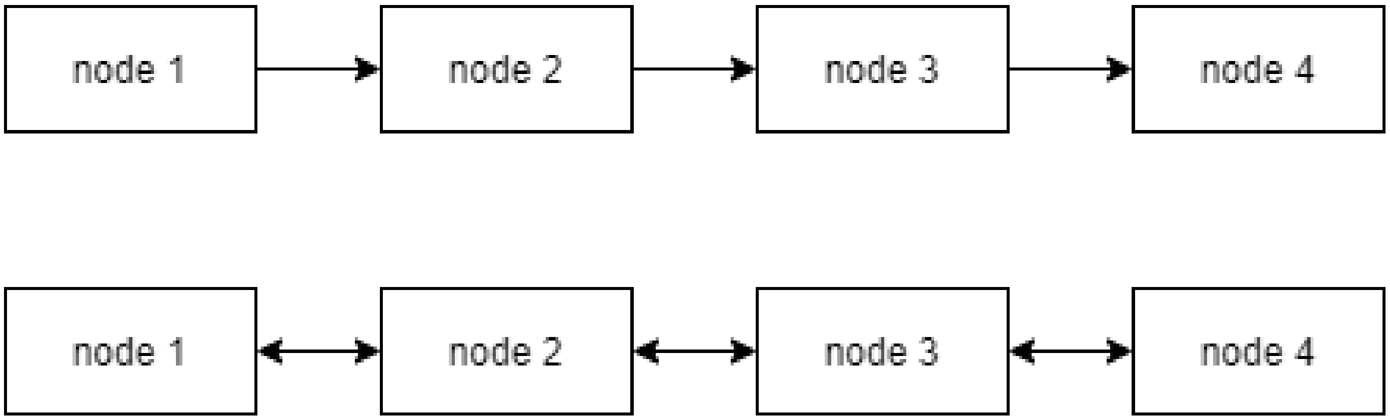
Method of ordering

- **Nodes:** Represent data point
- Nodes point to other nodes to show a relationship

Types of Linked Lists

- **Singly Linked Lists**
  - There is only one direction
  - Each node can only point to one other node
  - The node being pointed to an not point back to the other node
- **Doubly Linked Lists**
  - Bidirectional
  - Node can point back to nodes that point to them

Linked List



Linked List

Performance

- ★ Insertion: **O(n)**
- ★ Searching: **O(n)**
- ★ Deletion: **O(1)**

You have an online store and you need to prepare orders based on the first order made.

Which Data structure would you use?

You are making pancakes, as you make a new pancake, you put it on top of the last pancake that you made, when you eat your pancakes, you start eating the ones at the top first.

Which data structure works in the same way as this?

You need to store 10 items in your application, which data structure would be the best for storing and keeping track of these 10 items and why?

Non-Linear Data Structures

Each data point can have 0 to many connections

- ★ Trees
- ★ Graphs

Trees

★ Represent hierarchy
★ Very good for storing data that needs to be frequently searched

Trees

Structure

- ★ **Node**
  - ○ Represents a data point
- ★ **Edges**
  - ○ Represents the relationship between a child and parent node
- ★ **Parent**
  - ○ The node that connects to 1 or mode children
- ★ **Child**
  - ○ The node that is connected to a parent
- ★ **Leaf**
  - ○ A Node with no children
- ★ **Branch**
  - ○ The path to a given node

Trees

Types of Trees

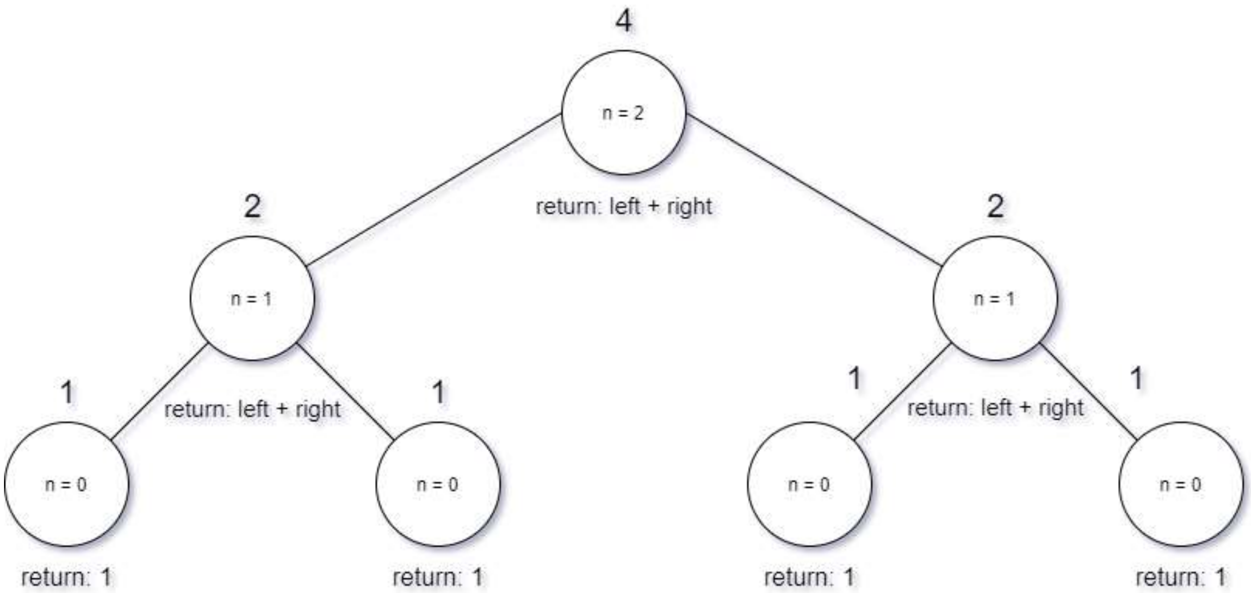
- ★ **Binary**
  - Each node has a maximum of 2 children
- ★ **N-ary Trees**
  - Each node can have more than 2 children

Where are trees used

- ★ **Search trees**
  - Organised data can be efficiently searched in a tree
- ★ **Decision Tree**
  - Each parent can lead to a path based on a decision made
  - Mainly used of machine learning and AI

Trees

total = 2^n where n = 2



Graphs

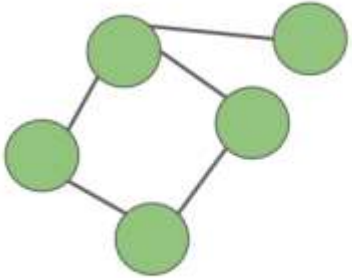
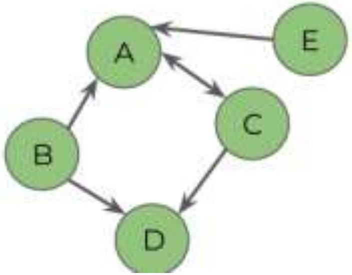
- ★ Represent complex relationships between data points
- ★ Used for certain operations in different applications
    - Machine Learning
    - Game development
    - Operation Optimisation

Graphs

Structure

- ★ **Node**
  - ○ Represents a data point
- ★ **Edges**
  - ○ Represents the relationship between data points
- ★ **Neighbours**
  - ○ Two nodes that are connected at an edge
- ★ **Degree**
  - ○ Number of nodes that are connected

Main Types of Graphs

Undirected Graphs Edges connecting nodes have no direction. For this graph, the order of the pairs of vertices in the edge set does not matter. **Applications:** Social networks, Recommendation Systems	
Directed Graphs Edges connecting nodes have specified directions. Edges may also be bidirectional. This graph cannot contain any loops. **Applications:** Maps, Network Routing, WWW	

Dictionaries and sets are linear data structures

A. True
B. False

Let’s Breathe

Let’s take a small break before moving on to the next topic.

Search and Sorting Algorithms

Things to keep in mind

★ Depending on the industry you land in, you might never use any search and sorting algorithms, with the power of databases, it’s better to perform searching and sorting in the database directly.

CoGrammar

Search and Sorting Algorithms

Why Learn Searching and sorting algorithms

★ Commonly asked in interviews
★ Allow you to think critically about the programming choices that you make
★ If you get into AI, you will need them

Search Algorithms

As you can imagine, when you are working with large amounts of data, you need to be able to get specific values, this is where searching comes into play.

- ★ Linear Search
- ★ Binary Search
- ★ Depth First Search (Covered in Tutorial)
- ★ Breadth First Search (Covered in Tutorial)

Linear Search

If a new programmer was asked to find a value in a list, there is a high probability that they will use a linear search without knowing.

Pros

★ Very simple to understanding and implement

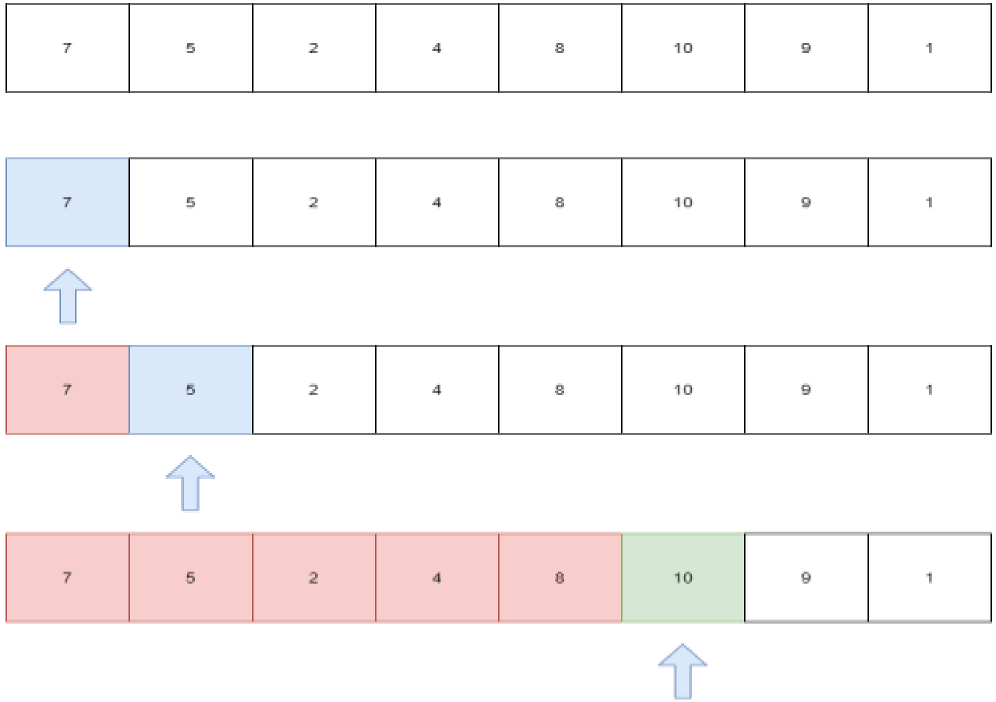
★ The only reliable algorithm for getting values from unsorted lists

Cons

★ **O(n)** Time complexity in the worst case.

Linear Search

Looking for 10



Binary Search

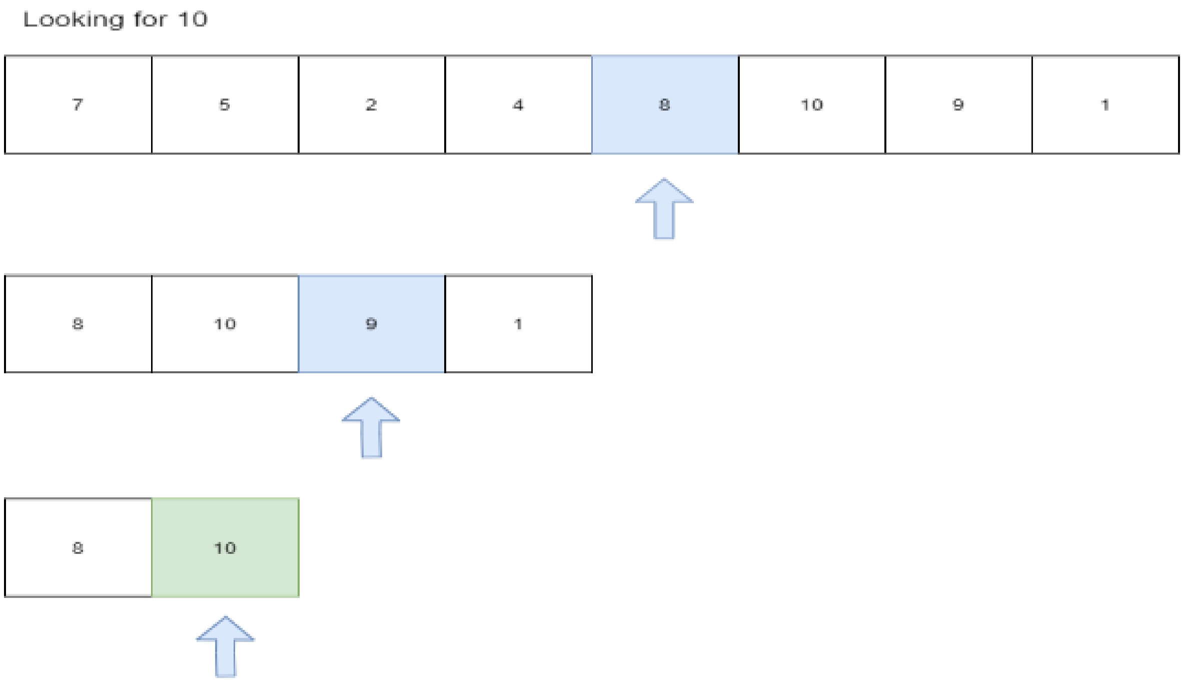
Pros

- Super-efficient with a worst case time complexity of **O(log n)**

Cons

- Only works on sorted datasets

Binary Search



Sorting Algorithms

As seen in the search potion, the way that we store our data is very helpful in allowing us to find values quickly and efficiently. There are a many sorting algorithms to choose from, but here are the most common ones:

★ Insertion Sort
★ Bubble Sort
★ Quick Sort
★ Merge

Insertion Sort

Starts at the beginning of the list and compares the first value to every value and swaps the values where required until we reach the end of the list. The process is repeated for every value in our list

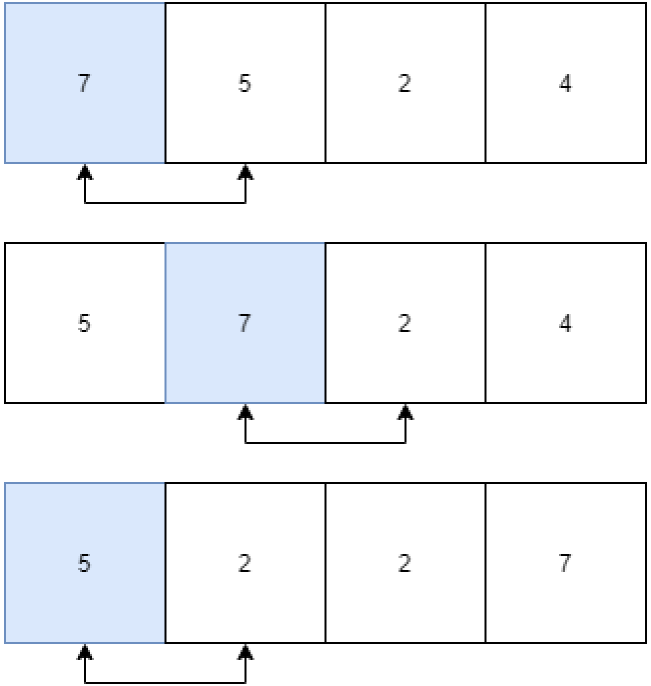
Pros

★ Easy to implement

Cons

★ Really inefficient with a time complexity of **O(n^2)**

Insertion Sort



Bubble Sort

Compares two value in the list at a time and swaps them accordingly allowing the largest number to bubble to the end of the list

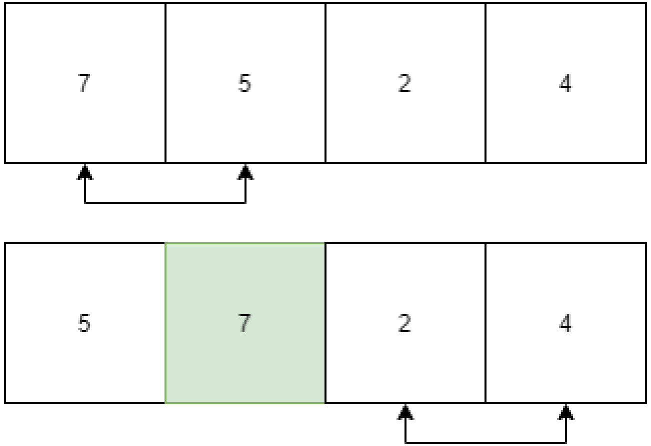
Pros

★ Easy to implement

Cons

★ Really inefficient with a time complexity of **O(n^2)**

Bubble Sort



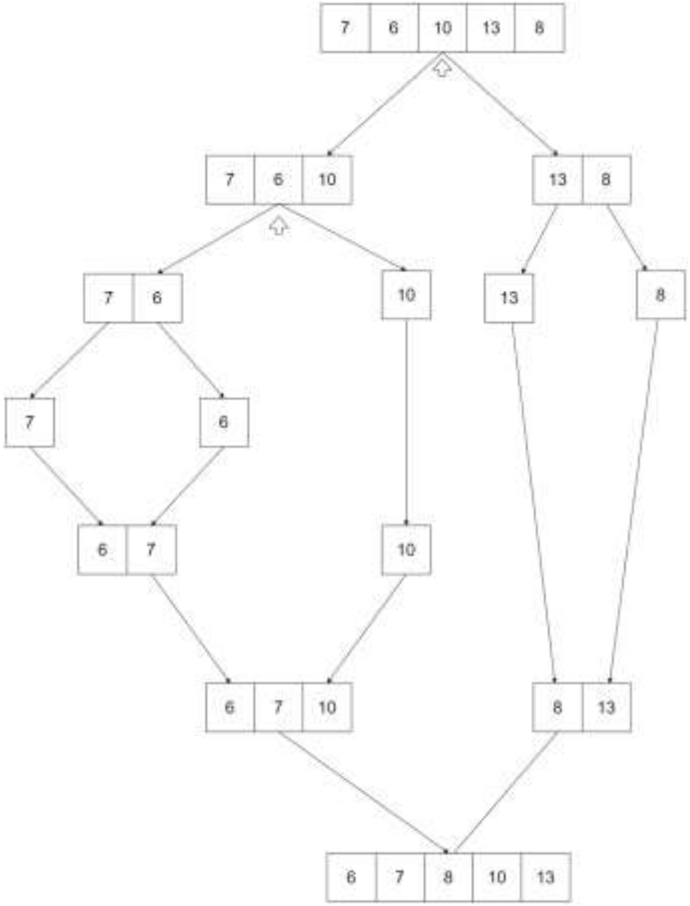
Merge Sort

Divide and conquer algorithm that finds middle point of our data and recursively breaks the values up until we have 1 item left, from there the items are compared to one another to sort them.

Pros

★ Good average time complexity of **O(n log n)**

Merge Sort



Merge Sort

```python
def merge_sort(array):
    if len(array) <= 1:
        return array

    mid = len(array) // 2
    left = merge_sort(array[:mid])
    right = merge_sort(array[mid:])

    return merge(left, right)

def merge(left, right):
    sorted_list = []

    i, j = 0, 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_list.append(left[i])
            i += 1
        else:
            sorted_list.append(right[j])
            j += 1

    sorted_list += left[i:]
    sorted_list += right[j:]

    return sorted_list
```

Quick Sort

Divide and conquer algorithm that finds a pivot value and breaks the array up into sub arrays where the left side contains values lower than the pivot and the right side has values greater than the pivot. Once we reach the base case, the items will be returned in:

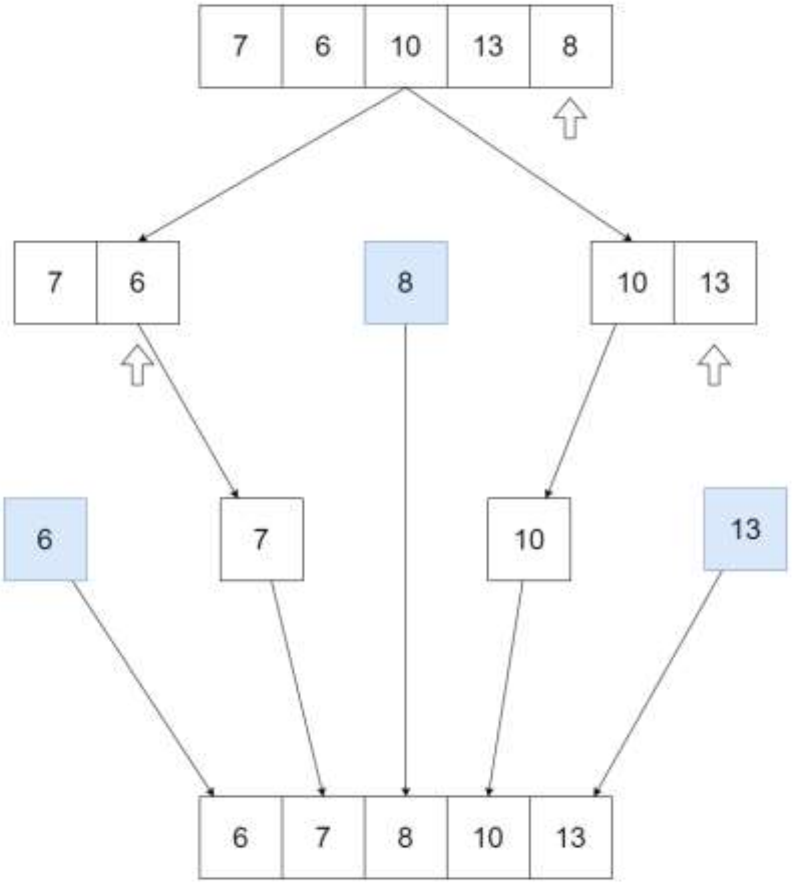
Pros

- ★ Good average time complexity of **O(n log n)**

Cons

- ★ Worst case time complexity of **O(n^2)**

Quick Sort



Quick Sort

```python
def quicksort(arr, low, high):
  if low < high:
    # pi is partitioning index, arr[p] is now at right place
    pi = partition(arr, low, high)

    # Recursively sort elements before and after partition
    quicksort(arr, low, pi-1)
    quicksort(arr, pi + 1, high)

def partition(arr, low, high):
  pivot = arr[high]  # pivot can be the last element or randomly chosen

  i = (low - 1)  # index of smaller element
  for j in range(low, high):

    # Check if current element is smaller than the pivot
    if arr[j] <= pivot:
      i += 1
      arr[i], arr[j] = arr[j], arr[i]

  arr[i + 1], arr[high] = arr[high], arr[i + 1]
  return (i + 1)
```

We’ve got an unordered list of products and their prices, which algorithm would you use to sort these items and why?

You need to find the fastest route to the shops from your house, you have a dataset consisting of the different points that can lead you to your destination.

Which data structure and algorithm would you use to get the optimal path and why?

CoGrammar

Q & A SECTION

Please use this time to ask any questions relating to the topic, should you have any.

CoGrammar

Thank you for joining!