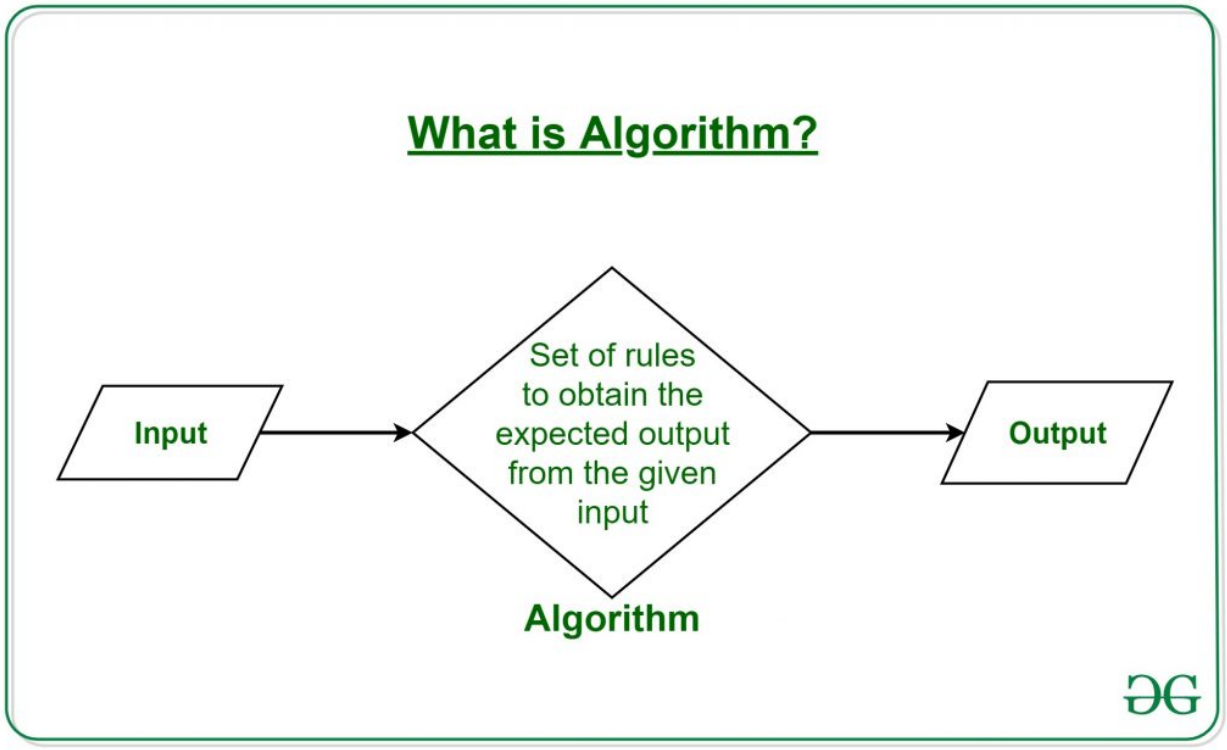
Module 13 Day 3

Lesson 1

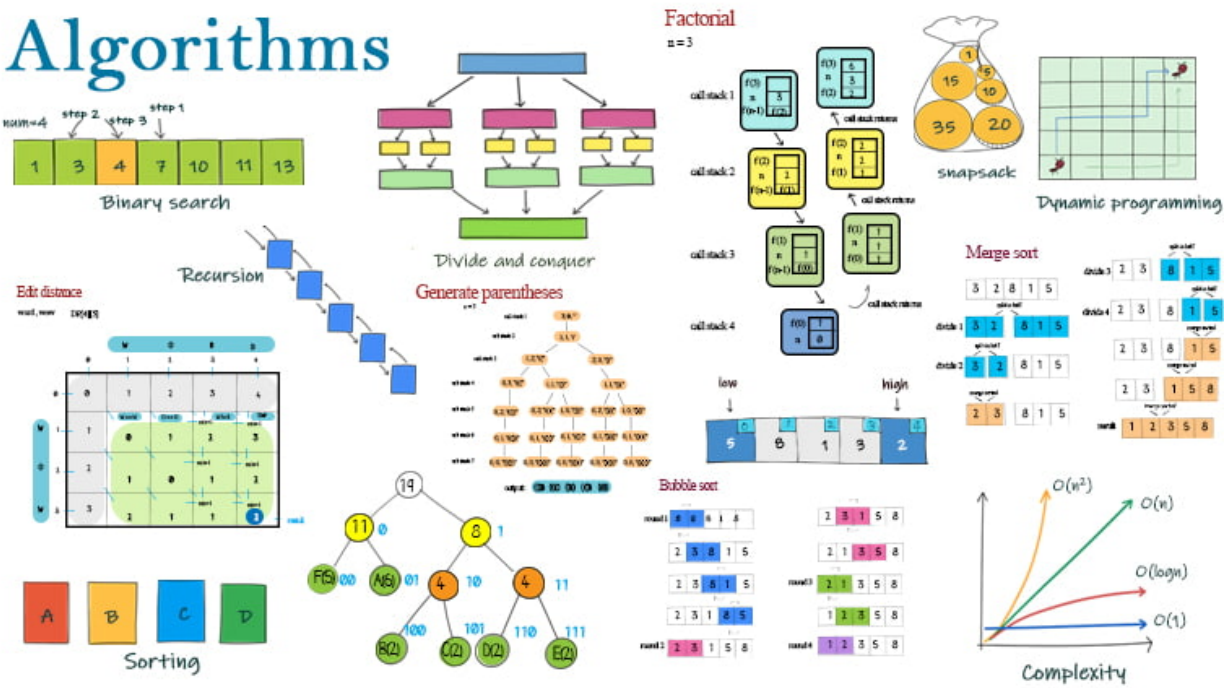
Algorithms



Source: [What is Algorithm | Introduction to Algorithms - GeeksforGeeks]

Definition: An algorithm is a set of instructions designed to perform a specific task or solve a particular problem. It is a step-by-step procedure that takes input, processes it, and produces the desired output. In computer programming, algorithms are written in the form of functions or methods, and they are used to implement various functionalities in software applications.

https://www.khanacademy.org/computing/computer-science/algorithms



Source: [Algorithm types and algorithm examples – illustrated]

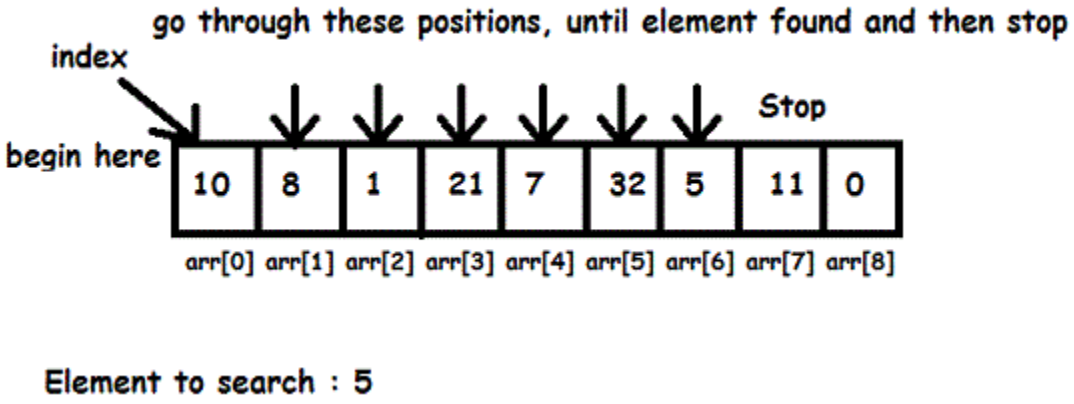
Use Cases

1. Sorting: Sorting algorithms are used to arrange data in a specific order, such as alphabetical, numerical, or chronological order. They are used in databases, spreadsheets, and other applications that handle large amounts of data.
2. Search: Search algorithms are used to find specific items in a list or database. They are used in search engines, recommendation systems, and e-commerce sites.
3. Machine learning: Machine learning algorithms are used to identify patterns in data and create predictive models. They are used in many fields, including finance, healthcare, and marketing.
4. Computer vision: Computer vision algorithms are used to analyze images and videos to extract information, such as object recognition and face detection. They are used in security systems, autonomous vehicles, and medical imaging.
5. Cryptography: Cryptographic algorithms are used to encrypt and decrypt data to ensure secure communication and transactions. They are used in online banking, e-commerce, and other applications that require secure data transmission.
6. Optimization: Optimization algorithms are used to find the best solution to a problem within a set of constraints. They are used in logistics, resource allocation, and scheduling.

Careers

1. Software Engineer: Software engineers are responsible for designing and developing software applications and systems. They often need to create efficient algorithms to solve complex problems and improve the performance of their software.
2. Data Scientist: Data scientists work with large datasets to extract insights and knowledge. They often use algorithms to analyze data and create predictive models.
3. Computational Scientist: Computational scientists use computer simulations to model complex systems and processes, such as climate change or protein folding. They often need to create specialized algorithms to solve these complex problems.
4. Game Developer: Game developers use algorithms to create realistic and engaging game environments, characters, and mechanics.
5. Robotics Engineer: Robotics engineers design and develop robots for a variety of applications, such as manufacturing, healthcare, and space exploration. They often need to create algorithms to control the movements and actions of robots.

Linear Search



Source:
https://medium.com/karuna-sehgal/an-simplified-explanation-of-linear-search-5056942ba965

https://codepen.io/shafferma08/pen/oNaeZbq

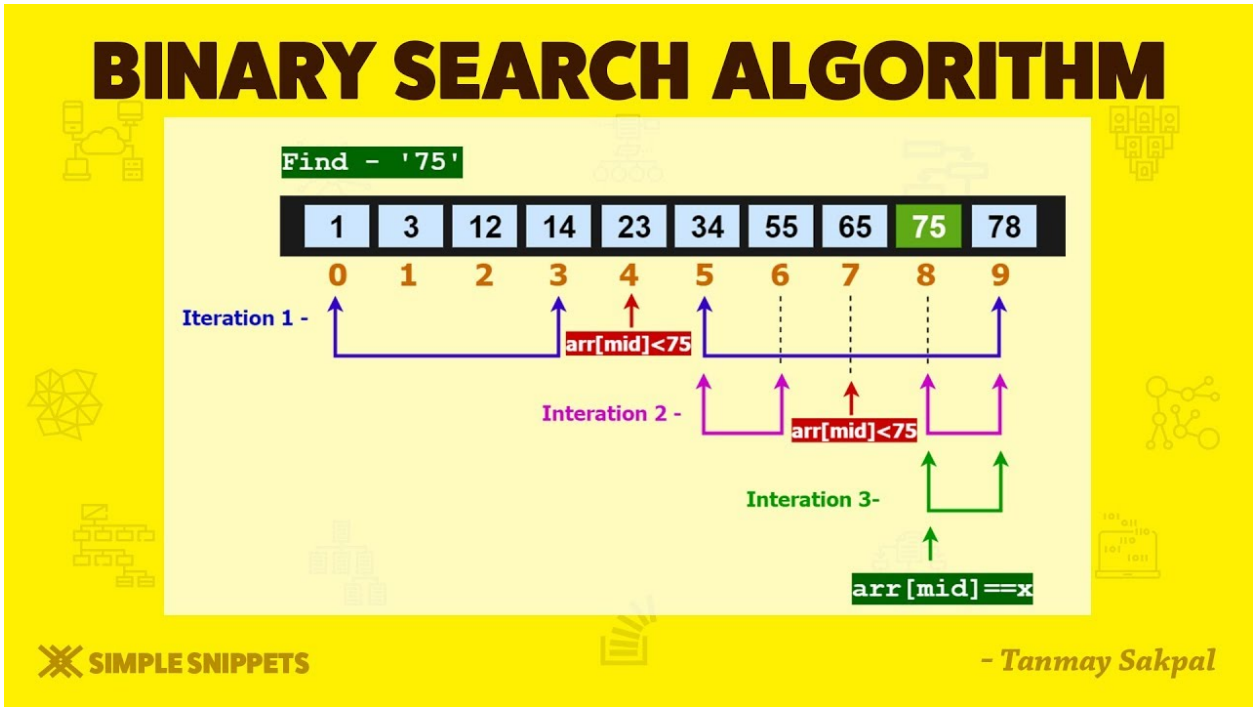
Linear search is a simple searching algorithm that searches for an element in an array or a list by sequentially checking each element until a match is found or the entire list is searched. It starts at the beginning of the list and compares each element with the search element until the desired element is found. If the element is not present in the list, the algorithm returns a message indicating that the element is not found.

The time complexity of the linear search algorithm is O(n), where n is the number of elements in the list. This means that the time taken by the algorithm to search for an element increases linearly with the number of elements in the list.

Use Case Examples:

- Checking if a particular name exists in a phonebook.
- Finding the first occurrence of a character in a string.
- Searching for a specific element in an unsorted list.

Binary Search



Source: https://youtu.be/HlEz93t628E?feature=shared

Video: https://www.youtube.com/watch?v=MFhxShGxHWc

https://codepen.io/shafferma08/pen/yLRoMOO

Binary search is a more efficient searching algorithm that works by dividing the list into two halves and then recursively searching the desired element in either the left half or the right half, depending on whether the element is smaller or larger than the middle element of the list. This process is repeated until the element is found or the entire list is searched.

The time complexity of the binary search algorithm is O(log n), where n is the number of elements in the list. This means that the time taken by the algorithm to search for an element

increases logarithmically with the number of elements in the list, making it much faster than linear search for large lists.

Use Case Examples:

- Searching for a word in a dictionary.
- Finding a specific item in a sorted list of items.
- Searching for a specific record in a database sorted by a particular field, such as last name or date.

Algorithm	Use Cases	Pros	Cons	Time Complexity
Linear Search	Searching for an element in an unsorted list.	Easy to implement.	Time taken to search increases linearly with the size of the list. Not suitable for large datasets.	O(n)
Binary Search	Searching for an element in a sorted list.	Fast search time with a logarithmic increase in time taken as the size of the list increases. Suitable for large datasets.	Requires the list to be sorted beforehand. More complex to implement than linear search.	O(log n)

Regarding Code Examples in Codepen:

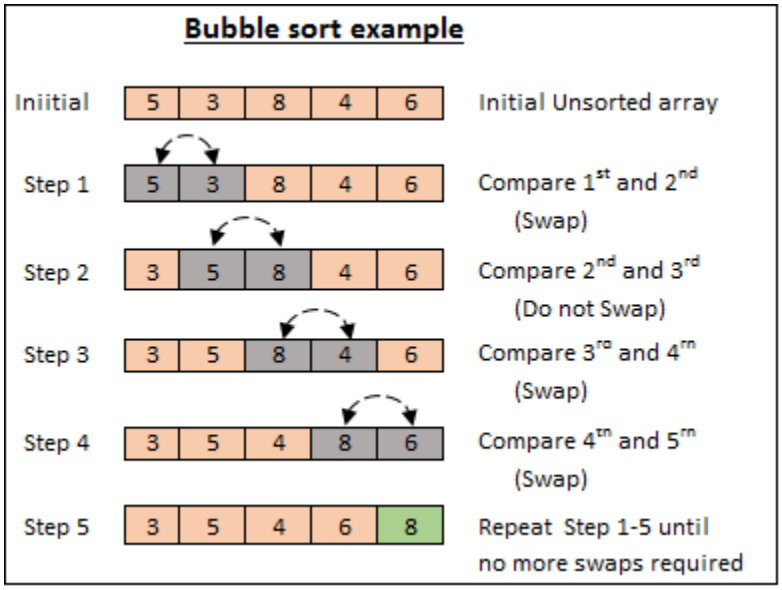
1. The input list needs to be sorted for binary search: In the linear search example, the input list can be unsorted, and the algorithm will still work correctly. However, in the binary search example, the input list must be sorted beforehand for the algorithm to work correctly.

2. Different search algorithms: The two examples use different search algorithms to find the desired element. The linear search algorithm checks each element of the list in sequence until a match is found, while the binary search algorithm divides the list in half and checks the middle element. This difference in algorithms leads to significant differences in performance for large lists.
3. Different time complexities: The time complexity of the linear search algorithm is O(n), where n is the number of elements in the list. This means that the time taken by the algorithm to search for an element increases linearly with the number of elements in the list. The time complexity of the binary search algorithm is O(log n), which means that the time taken by the algorithm to search for an element increases logarithmically with the number of elements in the list. This makes binary search much faster for large lists.
4. Different implementation details: The implementation details of the two algorithms are different, with different variables and loops used to perform the search. For example, the binary search algorithm uses variables to keep track of the start and end indices of the search space, while the linear search algorithm uses a loop to iterate through each element of the list.

Lesson 2

Sorting Algorithms

Bubble Sort:

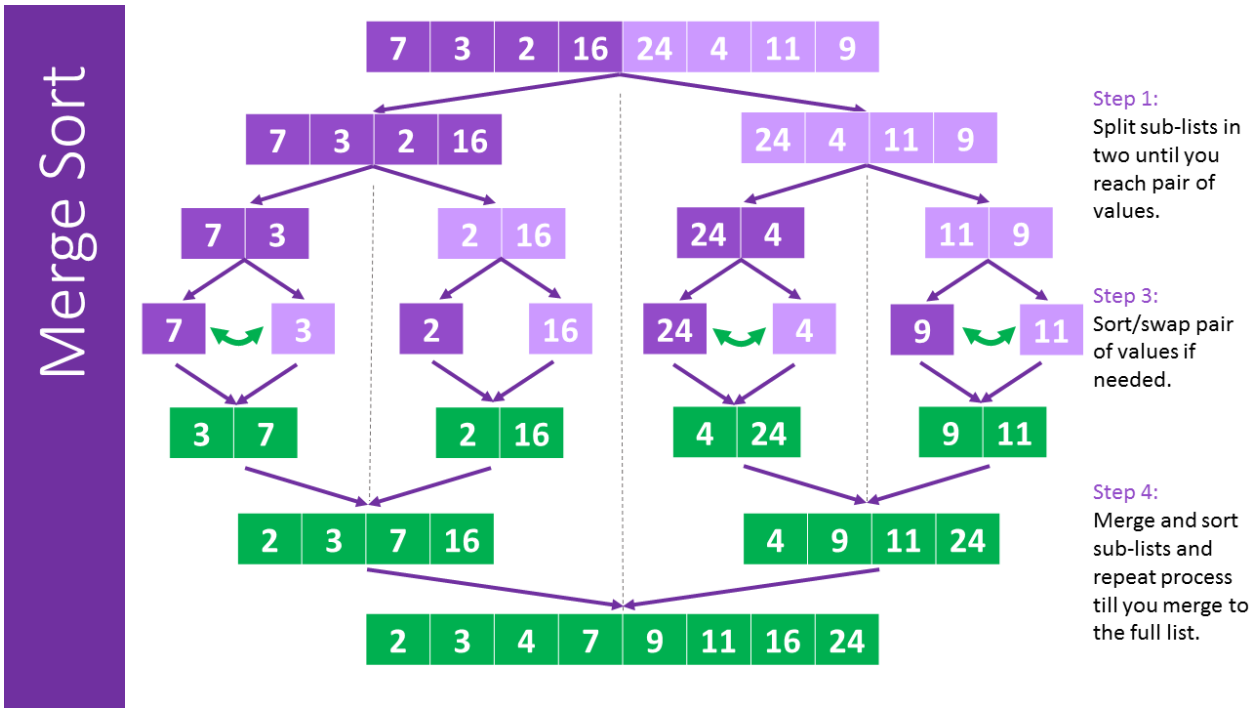


Source: [An Introduction to Bubble Sort. This blog post is a continuation of a… | by Karuna Sehgal]

Bubble sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. It starts by comparing the first two elements, then the second and third, and so on until the end of the list is reached. If a swap is made, the algorithm continues from the beginning of the list. The process is repeated until no more swaps are needed.

https://codepen.io/shafferma08/pen/OJBjpmK
https://codepen.io/shafferma08/pen/WNaEpOQ

Merge Sort:



Source: Merge Sort Algorithm - 101 Computing

Merge sort is a divide-and-conquer sorting algorithm that works by repeatedly dividing the list into smaller sub-lists until each sub-list contains only one element. Then, the sub-lists are merged back together in order. The merging process involves comparing the first element of each sub-list and adding the smaller of the two to the final list. This process is repeated until all elements have been added to the final list.

https://codepen.io/shafferma08/pen/abRyJXj

https://www.geeksforgeeks.org/merge-sort/

Sorting Algorithm	Use Cases	Differences	Pros	Cons	Average Time Complexity	Worst-Case Time Complexity
Bubble Sort	Small datasets, educational purposes	Simple implementation	Easy to understand and implement	Inefficient for large datasets	O(n^2)	O(n^2)
Selection Sort	Small datasets, educational purposes	Simple implementation	In-place sorting	Inefficient for large datasets	O(n^2)	O(n^2)
Insertion Sort	Small datasets, partially sorted datasets	Simple implementation, efficient for small datasets	In-place sorting, efficient for partially sorted datasets	Inefficient for large datasets, slower than other sorting algorithms	O(n^2)	O(n^2)
Merge Sort	Large datasets, stable sorting needed	Divide-and-conquer algorithm	Efficient for large datasets, stable sorting	Uses extra memory	O(n log n)	O(n log n)

Quick Sort	Large datasets, efficient sorting needed	Divide-and-conquer algorithm	Efficient for large datasets, in-place sorting	Worst case performance can be O(n^2)	O(n log n)	O(n^2)
Radix Sort	Sorting strings, integers with a fixed number of digits	Non-comparative sorting algorithm	Efficient for sorting integers with a fixed number of digits, works well with strings	Uses extra memory, not suitable for floating point numbers	O(d * (n + k))	O(d * (n + k))

More on Merge Sort

Optimization	**Description**	**Time Complexity**	**Space Complexity**	**Advantages**	**Use Cases**
Insertion Sort	Use insertion sort on small subarrays	O(n^2)	O(1)	Fast for small input sizes	Small input sizes

In-Place Merge Sort	Perform the merge operation in-place	O(nlogn)	O(1)	Space-efficient	Memory-constrained environments
Tail Recursion	Optimize recursive function calls	O(nlogn)	O(logn)	Efficient use of function call stack	Recursive algorithms with large input sizes

Lesson 3

Tree Traversal

https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

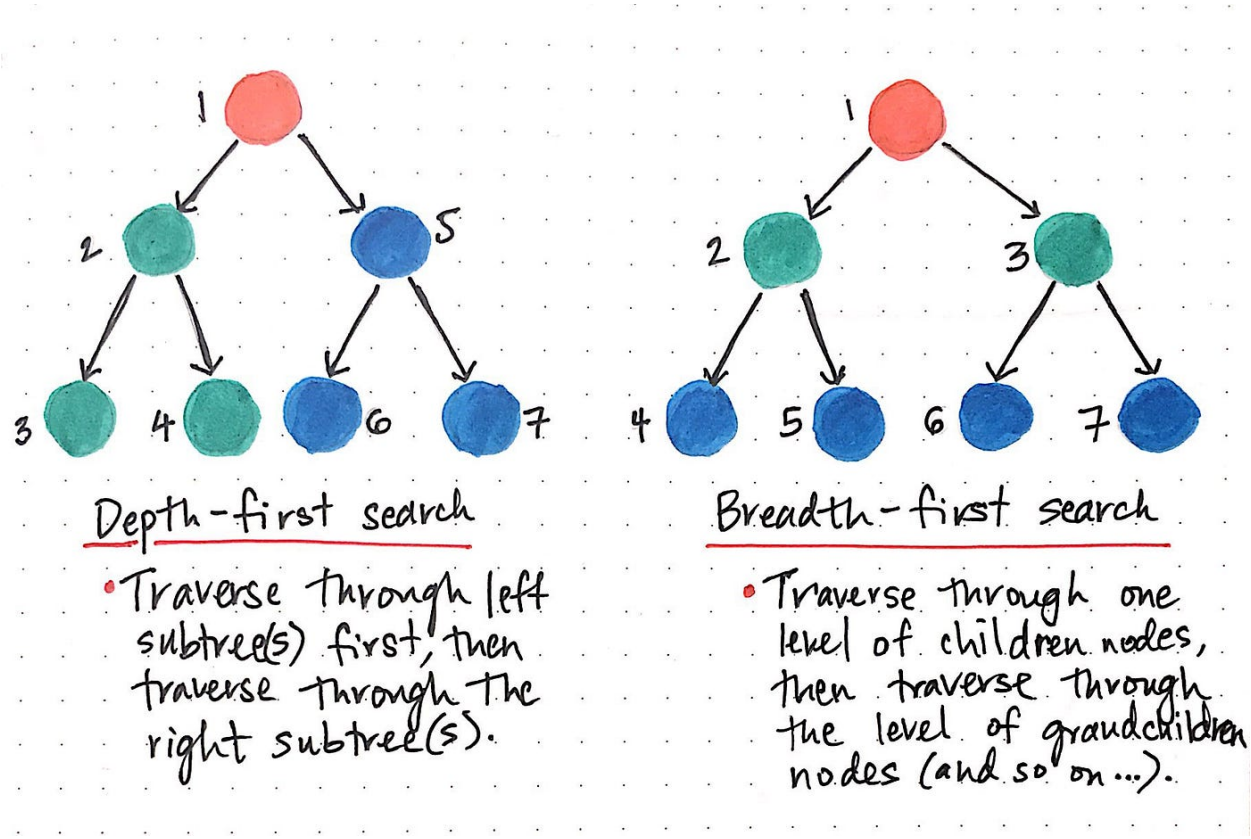
Tree traversal refers to the process of visiting all nodes of a tree data structure in a particular order. There are two main ways to traverse a tree: breadth-first search (BFS) and depth-first search (DFS).

Algorithm	Use Case	Advantages	Disadvantages
Breadth-First Search (BFS)	Finding the shortest path between two nodes	Guaranteed to find the shortest path in an unweighted graph	Can require a lot of memory to store all nodes at a given depth

			level
Depth-First Search (DFS)	Traversing a tree or graph and searching for a specific node or value	Uses less memory than BFS and can be implemented with a simple recursive function	Not guaranteed to find the shortest path and can get stuck in an infinite loop if there is a cycle in the graph
In-order Traversal	Traversing a binary search tree and returning values in ascending order	Useful for implementing binary search on a binary search tree	Not applicable to general trees or graphs
Pre-order Traversal	Generating a prefix expression for a binary expression tree	Useful for parsing and evaluating expressions	Not applicable to general trees or graphs
Post-order Traversal	Generating a postfix expression for a binary expression tree	Useful for parsing and evaluating expressions	Not applicable to general trees or graphs

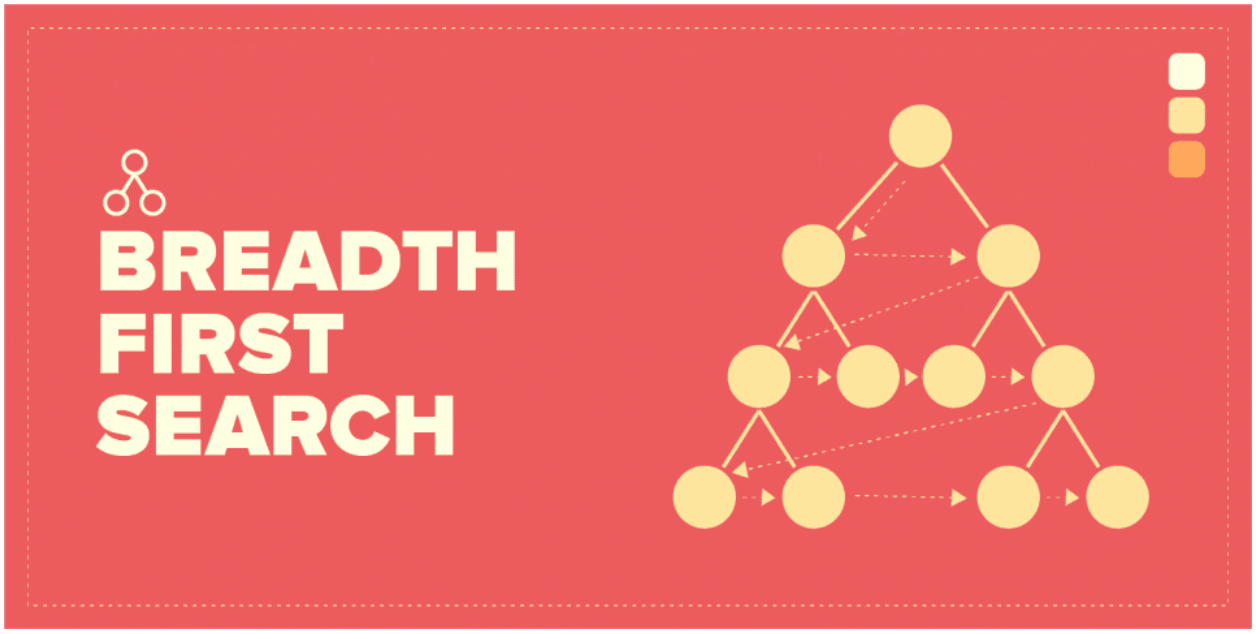
Why?

Tree traversal algorithms are essential in many applications involving trees, such as data analysis, graph algorithms, and computer science applications like compilers and interpreters.

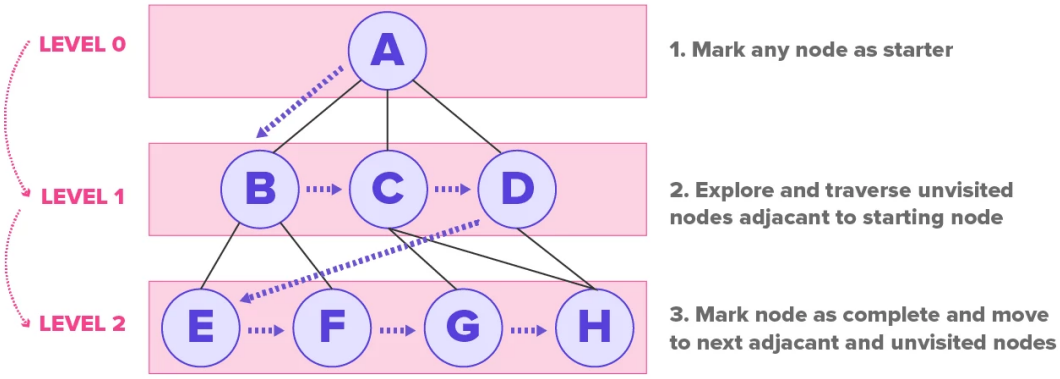


Source: https://medium.com/basecs/breaking-down-breadth-first-search-cebe696709d9

Breadth First Search



ARCHITECTURE OF BFS

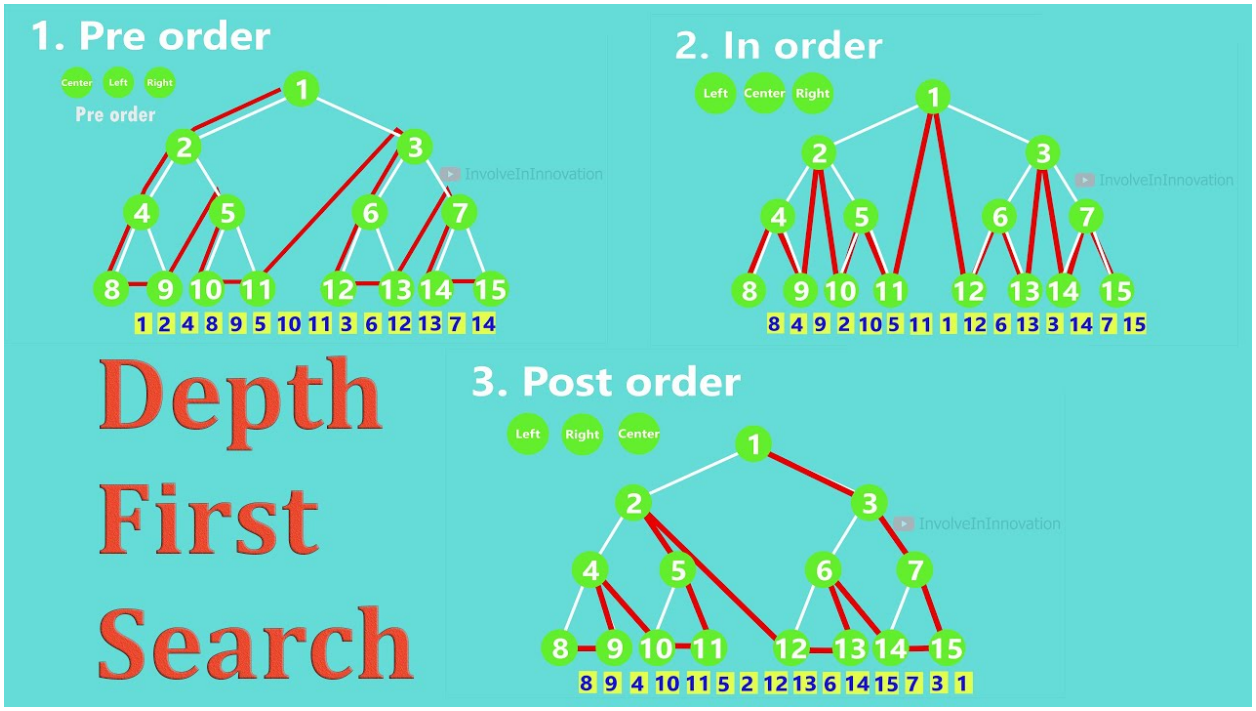


Source: https://hackr.io/blog/breadth-first-search-algorithm

BFS is a traversal algorithm that starts at the root node and visits all nodes at the current depth level before moving on to the next level. This means that BFS visits all nodes at the shallowest depth before visiting deeper nodes. BFS is commonly implemented using a queue data structure to keep track of nodes to visit.

Codepen: https://codepen.io/shafferma08/pen/BaMaLEO

Depth First Search



Source: https://youtu.be/5a9NUeRsBPo?feature=shared

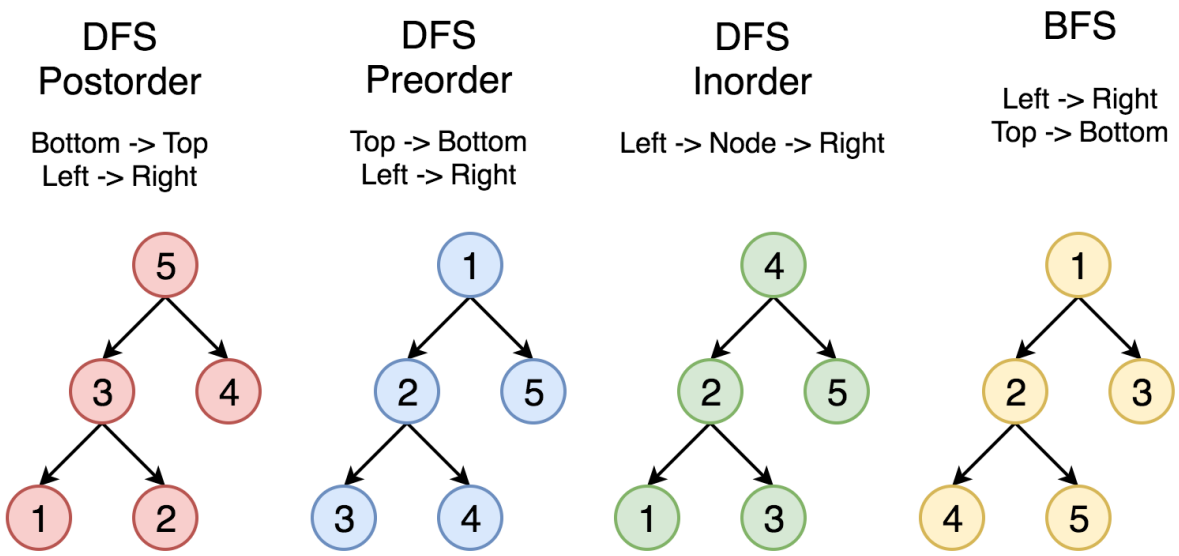
DFS is a traversal algorithm that explores as far as possible along each branch before backtracking. This means that DFS visits all nodes along one path from the root node to a leaf node before moving on to the next path. DFS can be implemented using either a stack or recursion.

In-order traversal

In-order traversal visits the left subtree, then the root node, and then the right subtree. This traversal order is often used to visit the nodes of a binary search tree in ascending order.

Post-order traversal

Post-order traversal visits the left subtree, then the right subtree, and then the root node. This traversal order is often used to perform a depth-first search of a binary tree where the nodes are visited in a "bottom-up" order.



Source: [Tree Traversal (In-order / Level-order) – Jun Zhang]

Codepen: https://codepen.io/shafferma08/pen/BaMaQQb

Lesson 4

Common Patterns

Pattern Name	Description	Use Case	Advantage	Disadvantage	Why It's Important
Brute Force	Tries all solutions until correct one is found.	Simple problems where solution space is manageable.	Straightforward, guaranteed to find a solution if one exists.	Inefficient for large problems, can be time-consuming.	Foundation for understanding problem-solving; sometimes the only

					clear method.
Greedy Algorithm	Makes locally optimal choices.	Coin change problems, certain optimization problems.	Often fast, provides “good enough” solutions.	May not always find globally optimal solution.	Useful for real-time decisions and when exact solution isn’t mandatory.
Divide and Conquer	Divides problem into subproblems and solves recursively.	Merge sort, binary search.	Can drastically reduce computation time.	Overhead of recursion, sometimes hard to define subproblems.	Splits complex problems into manageable pieces, often improving efficiency.
Dynamic Programming	Breaks problem into subproblems, stores solutions.	Knapsack problem, shortest path in directed acyclic graph.	Avoids redundant calculations, often highly efficient.	Requires understanding of problem’s substructure, uses more memory.	Turns exponential problems into polynomial ones, optimizing solutions.
Backtracking	Tries solutions, backs up upon mistakes.	Puzzle solving (like Sudoku), permutatio	Systematic trial-and-error, finds all solutions.	Can be slow; may explore many wrong paths.	Helps in situations where solution

		n problems.			structure is unknown.
Sliding Window	Uses a fixed-size window to slide over data.	Finding subarrays with given sums, longest substring without repeating characters.	Often linear time complexity, reduces nested loops.	Requires understanding of window conditions.	Efficiently processes arrays/strings without reprocessing elements.
Two Pointers	Uses two pointers in tandem, one moving faster.	Sum problems, finding pairs in sorted arrays.	Reduces need for nested loops, often linear time.	Array usually needs to be sorted, may miss solutions if not applied correctly.	Increases efficiency especially in sorted arrays or linked lists.
Memoization	Stores results of costly function calls.	Recursive calculations, Fibonacci sequence.	Reduces repeated calculations, often transforming exponential problems to linear/ polynomial.	Uses extra memory for storage.	Speeds up programs by reusing previously computed results.

FizzBuzz

1. Understand the problem: Before you start coding, make sure you understand what the FizzBuzz problem is asking for. You need to write a program that prints the numbers from 1 to N, with some variations depending on whether the number is divisible by 3, 5, or both.
2. Plan out the solution: Take a few minutes to plan out your solution before you start coding. Break the problem down into smaller steps and figure out how to tackle each step one by one.
3. Use a loop: Since you need to print a sequence of numbers, it's a good idea to use a loop to iterate through the numbers from 1 to N.
4. Use conditionals: Inside the loop, use conditional statements (if-else) to check whether the current number is divisible by 3, 5, or both. If it is, print the appropriate string ("Fizz", "Buzz", or "FizzBuzz").
5. Test the solution: Once you have written your code, test it with different input values to make sure it works correctly.
6. Refactor the code: If you have time, try to make your code more efficient or readable. For example, you could use a switch statement instead of if-else statements, or you could use a ternary operator to simplify the conditional checks.

Starter: https://codepen.io/shafferma08/pen/LYgjyEa