App Academy

DS&A Crash Course Review

Arrays

- Arrays must be contiguous in memory
- Static arrays have a fixed size
- Dynamic arrays solve our space problem if we fill up the array with values and are the default for many languages such as Javascript or Python
- Whenever a dynamic array needs to be resized when adding new elements, it is O(n). However, the amortized time complexity is O(1).

Operations	Big-O Time
Read/ Write ith element	O(1)
Insert / Remove End	O(1)
Insert Middle or Beginning	O(n)
Remove Middle or Beginning	O(n)

Array Algorithms

- Stacks
  - LIFO (last in, first out)
  - useful for when you are working with something in reverse order or need to do some backtracking
- Fixed Sliding Window
  - useful for when you're interested in the contents of a range
- Variable Sliding Window
  - useful when we want to find a min/max range for certain conditions
- Two Pointers
  - useful when we want are interested in two specific indices or values

Operations	Big-O Time
Push	O(1)
Pop	O(1)
Peek	O(1)

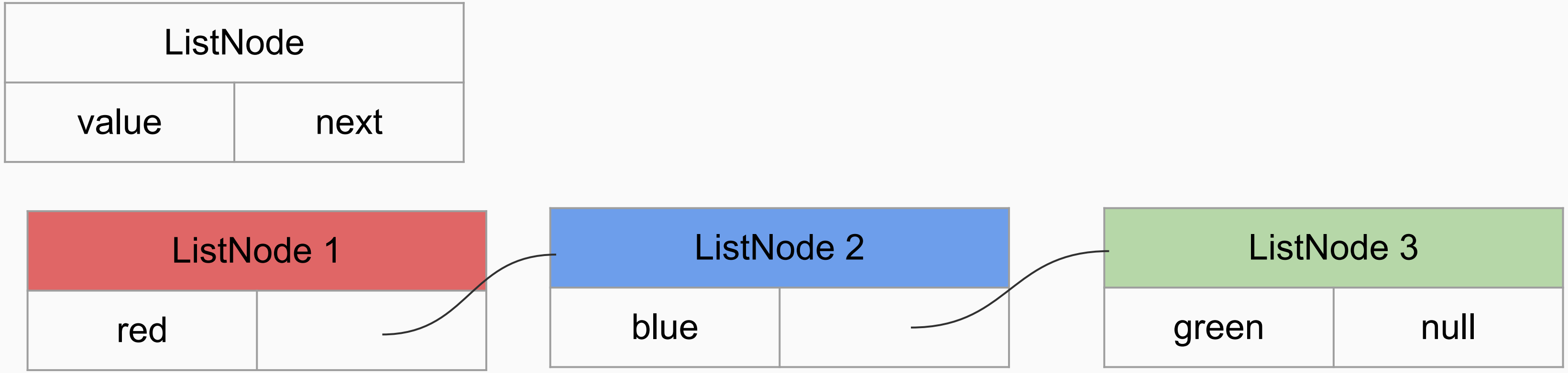
Hash Sets / Maps

- Sets are a hash-like list of values
  - Useful when we need to keep track of unique values that should only appear once and do not need to be assigned to indices
- Maps and Objects store key-value pairs
  - maps are useful over objects when we want to use data types besides strings as keys

Operations	Big-O Time
Insert value	O(1)
Remove value	O(1)
Search value	O(1)

Linked Lists

- Linked lists are made up of “list nodes”
- Each of these nodes encapsulate at minimum 2 components
  - Value
  - Pointer(s) to other nodes
- Linked lists are not contiguous in memory and instead use pointers to different parts of memory



Linked Lists (cont.)

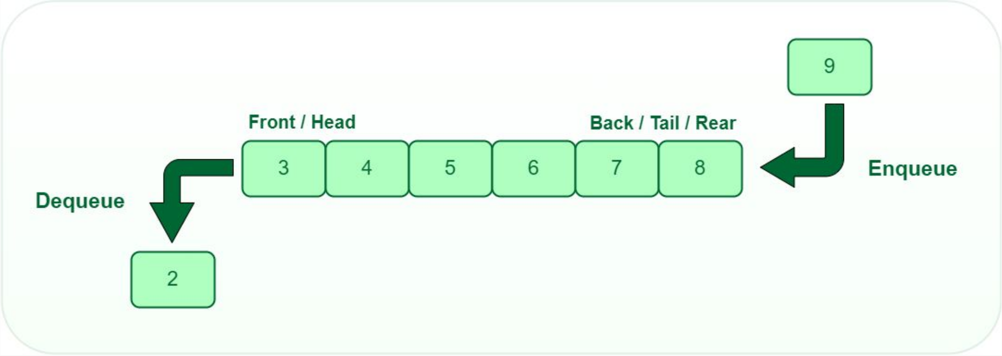
- ***Linked lists have great runtimes, but although reading, writing, and creating new values is technically O(1), we often need to traverse the list to find the node (which is O(n)).

Operations	Big-O Time
Read/ Write ith element***	O(1)
Insert / Remove End	O(1)
Insert Middle or Beginning***	O(1)
Remove Middle or Beginning	O(1)

Queues

- FIFO (first in, first out)
- Because we can add/remove nodes at the beginning of a Linked List in O(1) time, it is actually more efficient to create a queue this way versus an array.

Operations	Big-O Time
Enqueue	O(1)
Dequeue	O(1)

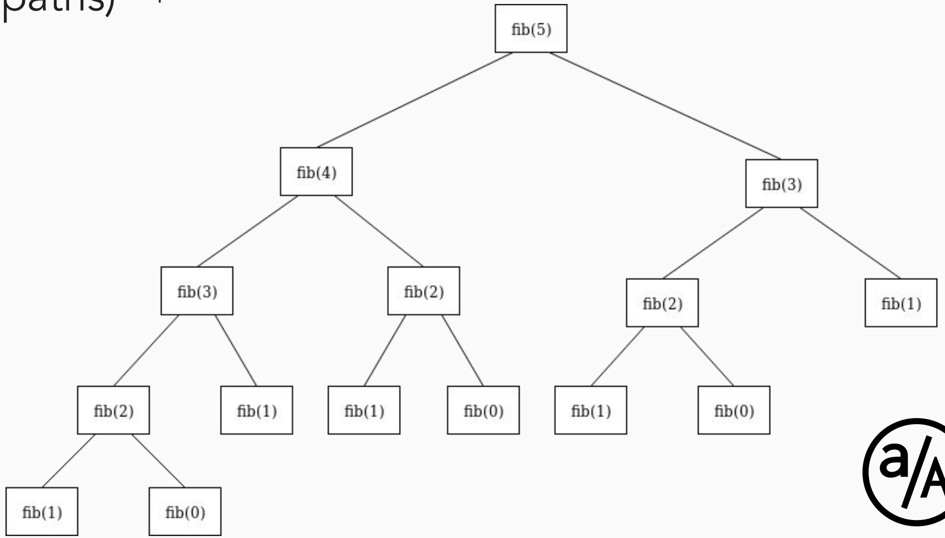


Recursion

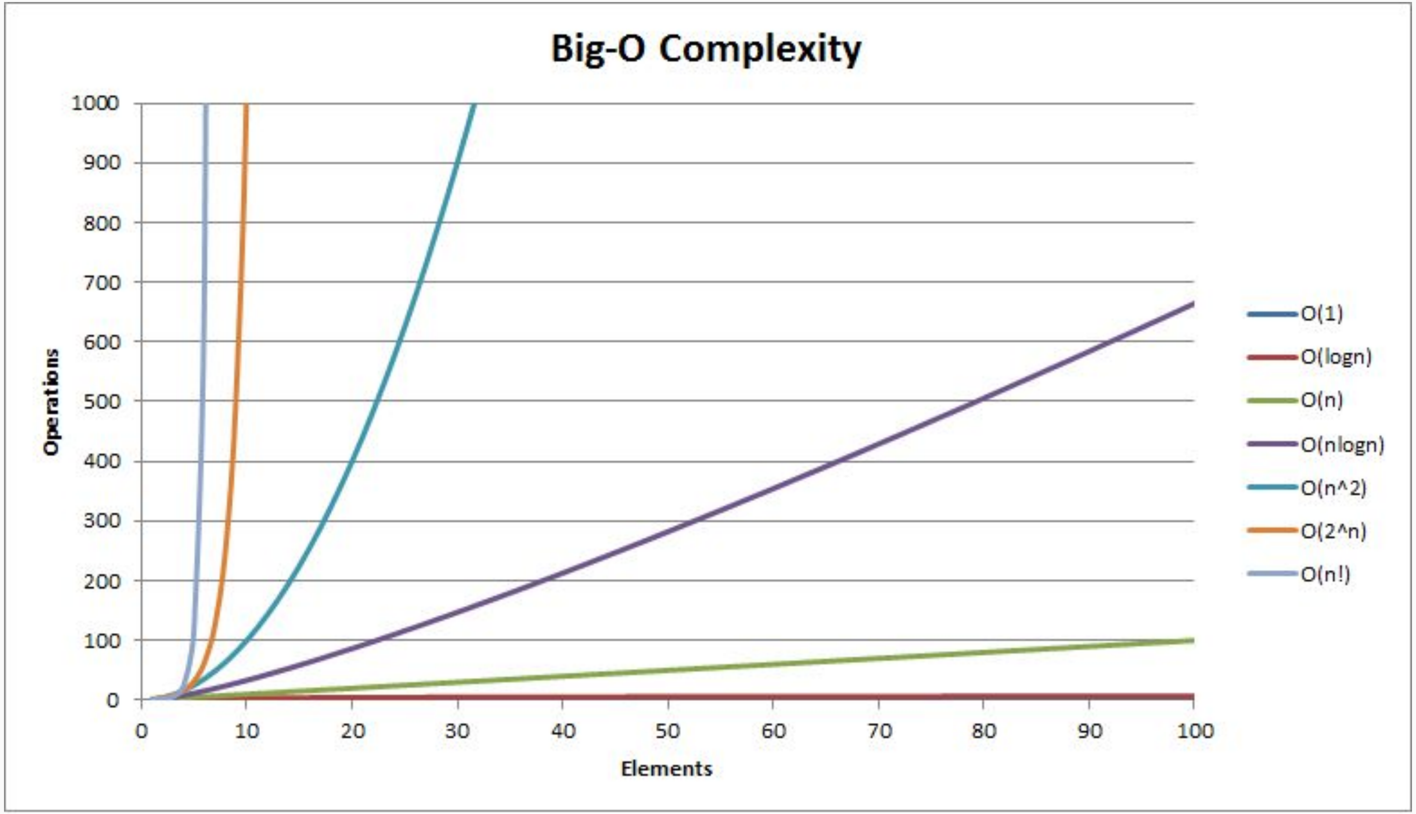
- Single-path recursive functions usually run in O(n) where n is the depth of the stack
- Recursion naturally uses a call stack, so it's perfect for DFS (which uses a stack)
- Multi-path recursive functions can be analyzed with the following formula:

$$(\text{\# of paths})^{(\text{depth of stack})}$$

Operations	Big-O Time
Traverse	$O(P^D)$



Big O Chart



Big-O Complexity
Operations
Elements
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Amortized Complexity

- In most cases, we care about worst case scenarios
- However, amortized complexity is when the worst case scenario happens so infrequently that it’s more useful to consider the average case scenario
    - Examples
        - Dynamic array push: worst case O(n), amortized O(1)
        - Quick sort: worst case O(n), amortized O(log(n))

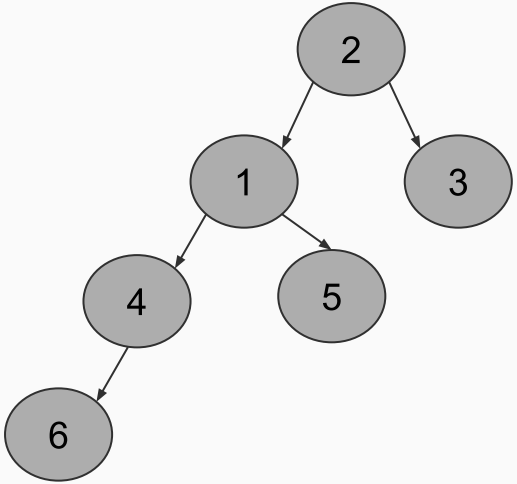
Binary Search

- Recursively cut input size by half to find something quickly
- Very efficient and one of the few algorithms that can run in O(log(n))

Operations	Big-O Time
Search	O(log(n))

Binary Trees

- Made up of nodes with left and right pointers
- Connected nodes have “parent-child” relationship.
- leaf node - A node without a child.
- root - A node without a parent.
- A node is an ancester to descendant nodes.
- height - how far a node is from the leaf nodes.
- depth - how far a node from the root node.
- Binary search trees are sorted trees that allow for efficient searching



Depth First Search

- Searches as deep into one path as possible before searching a different path
- 3 types (process node at diff times):
  - In-order traversal
  - Pre-order traversal
  - Post-order traversal
- Utilizes a stack
  - This makes recursion perfect since recursion inherently uses a stack via the call stack!

Operations	Big-O Time
Search tree	O(n)

```
var inorderTraversal = function(root) {
    if (!root) return;
    inorderTraversal(root.left);
    console.log(root.val)
    inorderTraversal(root.right);
};
```

```
var preorderTraversal = function(root) {
    if (!root) return;
    console.log(root.val)
    inorderTraversal(root.left);
    inorderTraversal(root.right);
};
```

```
var postorderTraversal = function(root) {
    if (!root) return;
    inorderTraversal(root.left);
    inorderTraversal(root.right);
    console.log(root.val)
};
```

Breadth First Search (BFS)

- Searches through nodes level by level
- We process nodes closest to the root first
- This algorithm is very useful when finding shortest paths or when dealing with layers
- Utilizes a queue
    - Because recursion uses a stack, we always want to just run BFS iteratively
- Steps
    - Add the root to the queue
    - Shift out a “current node” from the queue
    - Process current node
    - Push current node’s children into the queue
    - Repeat process until queue is empty

Operations	Big-O Time
Search tree	O(n)

Backtracking

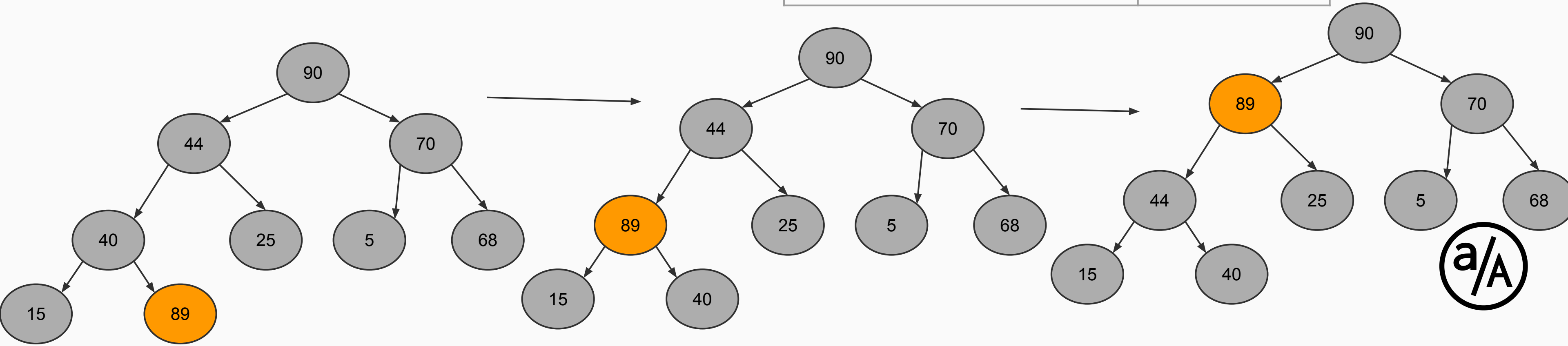
- Backtracking is where we solve problems recursively by “building candidates” and then abandon those candidates (“backtrack”) once we determine that candidate can no longer yield a valid solution.
- Most common type of problem that can be solved with backtracking is "find all possible ways to do something”
- Backtracking problems run in exponential runtime.

```
// let curr represent the thing you are building
// it could be an array or a combination of variables

function backtrack(curr) {
    if (base case) {
        Increment or add to answer
        return
    }

    for (iterate over input) {
        Modify curr
        backtrack(curr)
        Undo whatever modification was done to curr
    }

}
```

Heaps / Priority Queues

- A partially ordered complete binary tree that is useful for efficiently finding minimum or maximum values

Operations	Big-O Time
Push	O(log(n))
Poll	O(log(n))
Heapify	O(n)
Peek	O(1)



Graphs

- A graph is simply a group of connected nodes
  - subsets of graphs include linked lists and trees
- Graphs can be directed or undirected, referring to whether edges point one way or both ways
- Graphs can be disconnected
- $E \leq V^2$, where E = edges and V = vertices
- Ways to represent graphs:
  - Matrix
  - Adjacency Matrix
  - Adjacency List

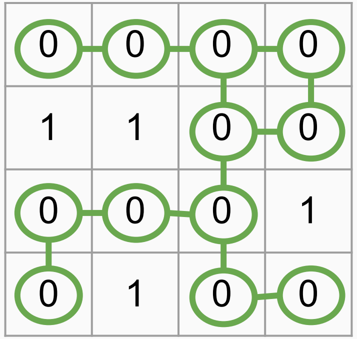
Matrices

- Graphs represented as a 2D array
- Typically an 4-directional undirected graph, but it can also be 8-directional
- Coordinates represented with row and column indices
  - We can find values by using grid[row][col]
  - Some also use x and y, but I recommend against this because I've seen many people get confused during mock interviews.
- Commonly used for path representation

```
grid = [  [0, 0, 0, 0],
          [1, 1, 0, 0],
          [0, 0, 0, 1]
          [0, 1, 0, 0]  ]
```

0 = free

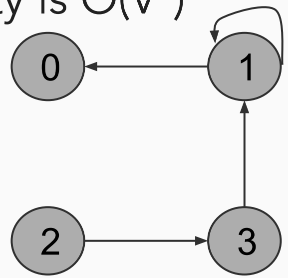
1 = blocked



Adjacency Matrices

- The cells of a adjacency matrix are NOT nodes
- Instead, the indices represent vertices, and the values in the cells represent edges between vertices
- Always a square since both sides represent vertices
- Rare because it is space inefficient. Complexity is $O(V^2)$
- Examples
  - adjMatrix[1][2] === 1
    - An edge exists from vertex 1 to vertex 2
  - adjMatrix[2][1] === 1
    - An edge exists from vertex 2 to vertex 1
    - Order matters!!
  - adjMatrix[0][1] === 0
    - No edge exists from vertex 0 to vertex 1
  - adjMatrix[1][1] === 1
    - There exists a self looping edge at vertex 1

```
adjMatrix=[ [0, 0, 0, 0],
            [1, 1, 0, 0],
            [0, 0, 0, 1],
            [0, 1, 0, 0]  ]
```



	0	1	2	3
0	0	0	0	0
1	1	1	0	0
2	0	0	0	1
3	0	1	0	0

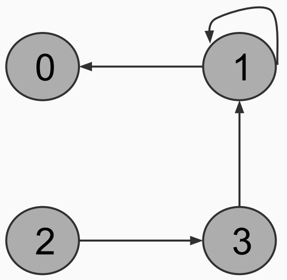
Adjacency Lists

- Very common way to represent a graph
- Uses nodes, similar to linked lists or trees
- Unlike linked lists or trees, there is no predefined number of connected neighbors
- We can represent neighbors in an array or set
- Much more space efficient than adjacency matrix since we only represent nodes that actually exist.
- Sometimes may have to build the adjacency list ourselves

```
class GraphNode {
    constructor(val) {
        this.val = val;
        this.neighbors = [];
    }
}
```

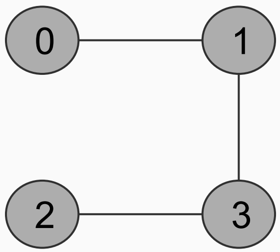
directed graph:

```
graph = {
    0: [],
    1: [0,1],
    2: [3],
    3: [1]
}
```



undirected graph:

```
graph = {
    0: [1],
    1: [0,3],
    2: [3],
    3: [1,2]
}
```



Questions?