The AlgoDaily Book: Core Essentials

The Visual Guide to Technical Interviews

By **Jacob Zhang and the AlgoDaily.com team**

Content

Algorithm Complexity and Big O Notation

Objective: In this lesson, we'll cover the topics of Algorithm Complexity and Big O Notation. By the end, you should:

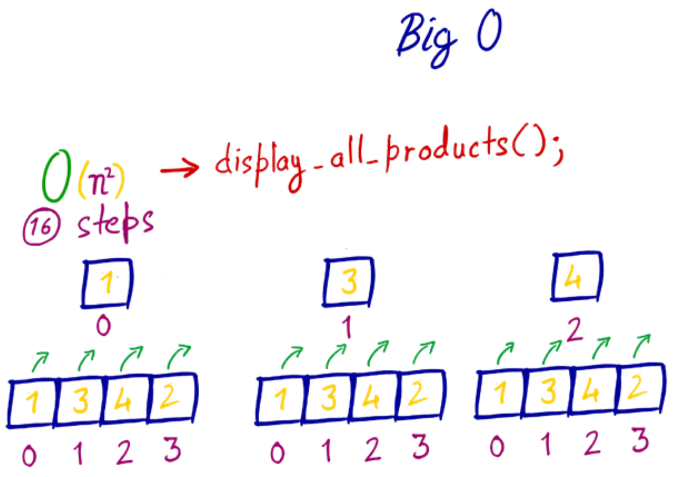
- Be familiar with these terms and what they mean.
- See their use in practice.
- Use these tools to measure how "good" an algorithm is.
- In software engineering, developers can write a program in several ways.

For instance, there are many ways to search an item within a data structure. You can use **linear search**, **binary search**, **jump search**, **interpolation search**, among many other options.

Our focus in this lesson is on mastering Algorithm Complexity and Big O Notation. But what we want to do with this knowledge is to improve the performance of a software application. This is why it's important to understand which algorithm to use, depending upon the problem at hand.

Let's start with basics. A computer algorithm is a series of steps the machine takes in order to compute an output. There are several ways to measure its performance. One of the metrics used to compare algorithms is this notion of algorithm complexity.

Sounds challenging, doesn't it? **Don't worry, we'll break it down**.



Algorithm complexity can be further divided into two types: time complexity and space complexity. Let's briefly touch on these two:

- The time complexity, as the name suggests, refers to the time taken by the algorithm to complete its execution.
- The space complexity refers to the memory occupied by the algorithm.

In this lesson, we will study time and space complexity with the help of many examples. Before we jump in, let’s see an example of why it is important to measure algorithm complexity.

Importance of Algorithm Complexity

To study the importance of algorithm complexity, let’s write two simple programs. Both the programs raise a number x to the power y.

Here's the first program: see the sample code for **Program 1.**

PYTHON
```python
def custom_power(x, y):
    result = 1
    for i in range(y):
        result = result * x
    return result

print(custom_power(3, 4))
```

Let’s now see how much time the previous function takes to execute. In Jupyter Notebook or any `Python` interpreter, you can use the following script to find the time taken by the algorithm.

PYTHON
```python
from timeit import timeit

func = '''
def custom_power(x, y):
    result = 1
    for i in range(y):
        result = result * x
    return result
'''

t = timeit("custom_power(3,4)", setup=func)
print(t)
```

You'll get the time of execution for the program.

I got the following results from running the code.

SNIPPET
```
600 ns ± 230 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Let's say that the `custom_power` function took around `600` nanoseconds to execute. We can now compare with **program 2**.

Program 2:

Let’s now use the Python’s built-in `pow` function to raise `3` to the power `4` and see how much time it ultimately takes.

PYTHON
```python
from timeit import timeit

t = timeit("pow(3, 4)")
print(t)
```

Again, you'll see the time taken to execute. The results are as follows.

SNIPPET
```
308 ns ± 5.03 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Please note that the built-in function took around `308` nanoseconds.

What does this mean?

You can clearly see that `Program 2` is twice as fast as `Program 1`. This is just a simple example. However if you are building a real-world application, the wrong choice of algorithms can result in sluggish and inefficient user experiences. Thus, it is very important to determine algorithm complexity and optimize for it.

Big(O) Notation

Let's now talk about `Big(O) Notation`. Given what we've seen, you may ask-- why do we need `Big O` if we can just measure execution speed?

In the last section, we recorded the clock time that our computer took to execute a function. We then used this clock time to compare the two programs. But clock time is hardware dependent. **An efficient program may take more time to execute on a slower computer than an inefficient program on a fast computer**.

So clock time is not a good metric to find time complexity. Then how do we compare the algorithm complexity of two programs in a standardized manner? The answer is `Big(O) notation`.

Big(O) notation is an algorithm complexity metric. It defines the relationship **between the number of inputs and the step taken by the algorithm to process those inputs**. Read the last sentence very carefully-- it takes a while to understand. Big(O) is NOT about measuring speed, it is about measuring the amount of work a program has to do **as an input scales**.

We can use `Big(O)` to define both time and space complexity. The below are some of the examples of Big(O) notation, starting from "fastest"/"best" to "slowest"/"worst".

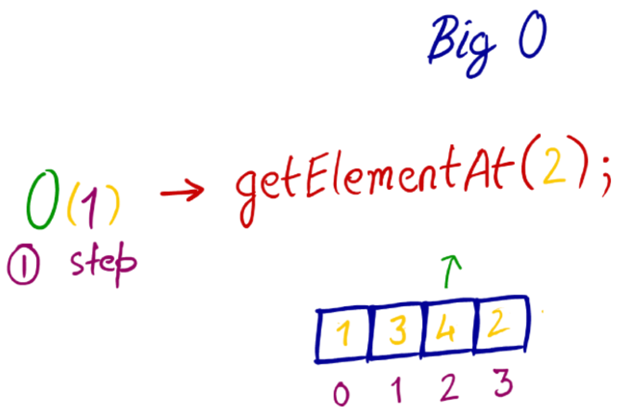
Function	Big(O) Notation
Constant	O(c)
Logarithmic	O(log(n))
Linear	O(n)
Quadratic	O(n^2)
Cubic	O(n^3)
Exponential	O(2^n)
Factorial	O(n!)

Let's see how to calculate time complexity for some of the aforementioned functions using Big(O) notation.

Big(O) Notation for Time Complexity

In this section we will see examples of finding the Big(O) notation for certain constant, linear and quadratic functions.

Constant Complexity



In the constant complexity, the steps taken to complete the execution of a program remains the same irrespective of the input size. Look at the following example:

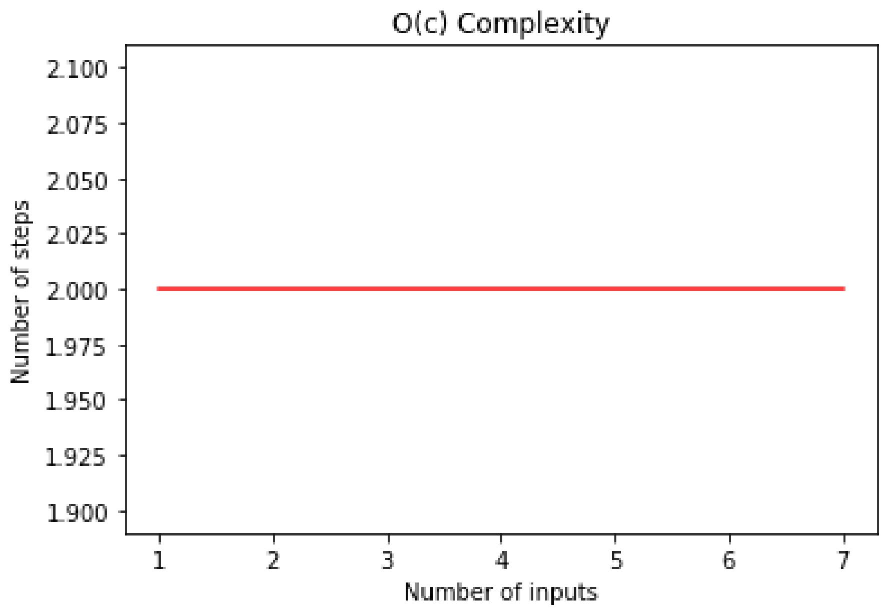
PYTHON
```python
import numpy as np

def display_first_cube(items):
    result = pow(items[0],3)
    print (result)

inputs = np.array([2,3,4,5,6,7])
display_first_cube(inputs)
```

In the above example, the display_first_cube element calculates the cube of a number. That number happens to be the first item of the list that passed to it as a parameter. No matter how many elements there are in the list, the display_first_cube function always performs two steps. First, calculate the cube of the first element. Second, print the result on the console. Hence the algorithm complexity remains constant (it does not scale with input).

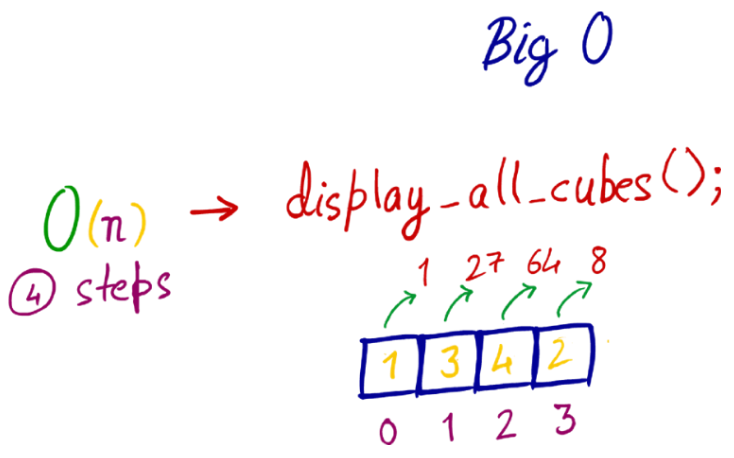
Let’s plot the constant algorithm complexity.

PYTHON
```python
import numpy as np
import matplotlib.pyplot as plt

num_of_inputs = np.array([1,2,3,4,5,6,7])
steps = [2 for n in num_of_inputs]

plt.plot(num_of_inputs, steps, 'r')
plt.xlabel('Number of inputs')
plt.ylabel('Number of steps')
plt.title('O(c) Complexity')

plt.show()

# No matplotlib support in this terminal.
# Go to the next screen for the results!
```

In the above code we have a list `num_of_inputs` that contains a different number of inputs. The `steps` list will always contain 2 for each item in the `num_of_inputs` list. If you plot the `num_of_inputs` list on x-axis and the `steps` list on y-axis you will see a straight line as shown in the output:



Linear Complexity*

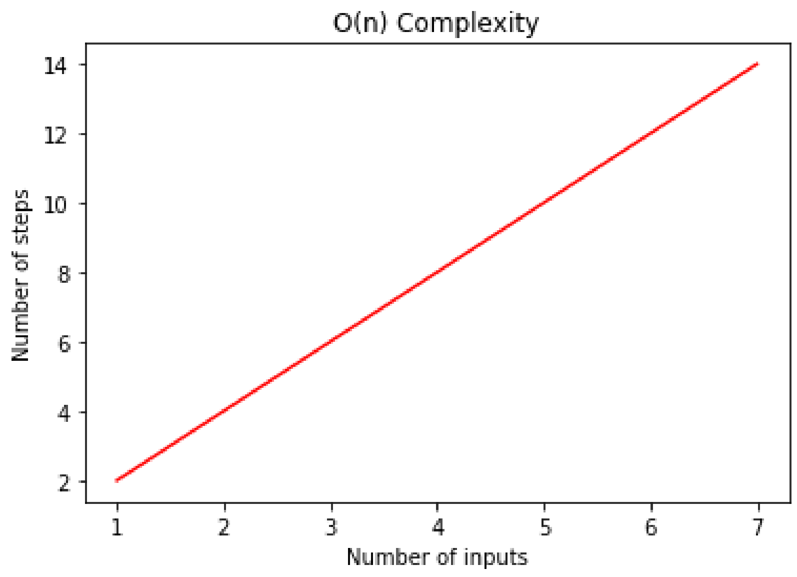


In functions or algorithms with linear complexity, a single unit increase in the input causes a unit increase in the steps required to complete the program execution.

A function that calculates the cubes of all elements in a list has a linear complexity. This is because as the input (the list) grows, it will need to do one `unit` more work per item. Look at the following script.

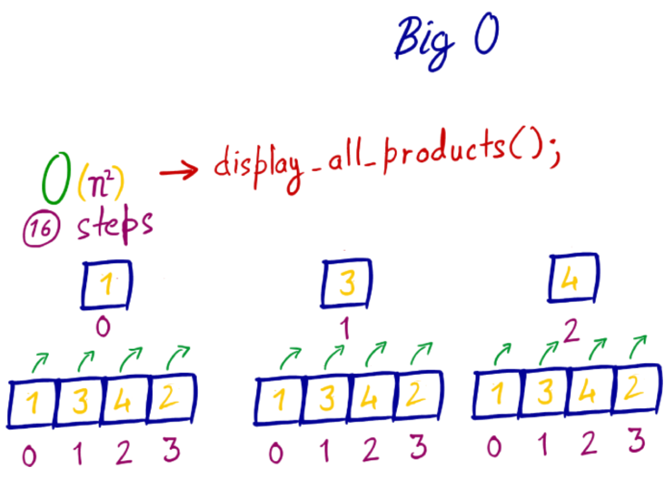
PYTHON
```python
import numpy as np

def display_all_cubes(items):
    for item in items:
        result = pow(item,3)
        print (result)

inputs = np.array([2,3,4,5,6,7])
display_all_cubes(inputs)
```

For each item in the `items` list passed as a parameter to `display_all_cubes` function, the function finds the cube of the item and then displays it on the screen. If you double the elements in the input list, the steps needed to execute the `display_all_cubes` function will also be doubled. For the functions, with linear complexity, you should see a straight line increasing in positive direction as shown below:



PYTHON
```
num_of_inputs = np.array([1,2,3,4,5,6,7])
steps = [2*n for n in no_inputs]

plt.plot(no_inputs, steps, 'r')
plt.xlabel('Number of inputs')
plt.ylabel('Number of steps')
plt.title('O(n) Complexity')
plt.show()

# No matplotlib support in this terminal.
# Go to the next screen for the results!
```

Quadratic Complexity



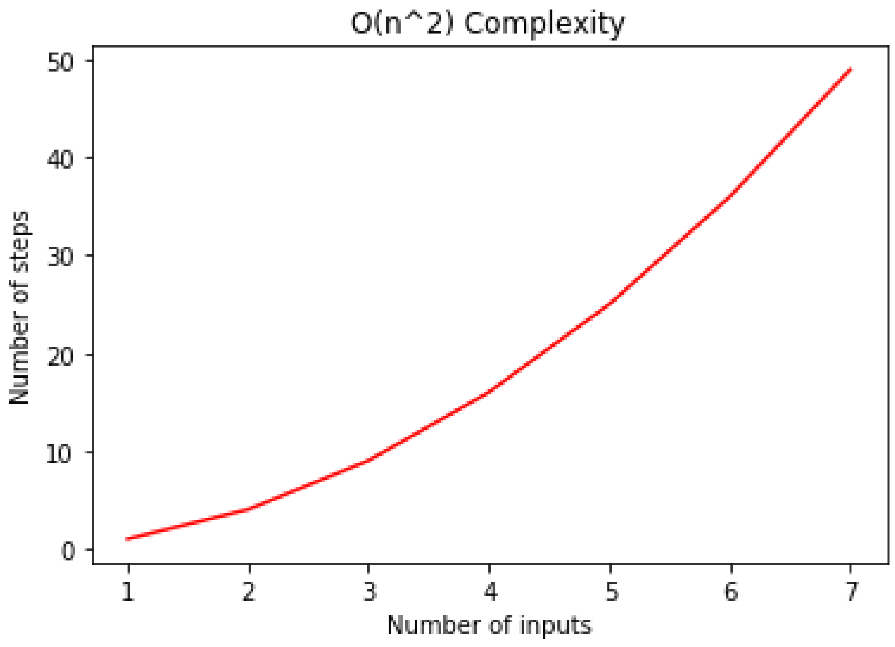
As you might guess by now-- in a function with quadratic complexity, the output steps increase quadratically with the increase in the inputs. Have a look at the following example:

PYTHON
```python
import numpy as np

def display_all_products(items):
    for item in items:
        for inner_item in items:
            print(item * inner_item)

inputs = np.array([2,3,4,5,6])
display_all_products(inputs)
```

In the above code, the `display_all_products` function multiplies each item in the “items” list with all the other elements.

The outer loop iterates through each item, and then for each item in the outer loop, the inner loop iterates over each item. This makes total number of steps `n x n` where `n` is the number of items in the “items” list.

If you plot the inputs and the output steps, you should see the **graph** for a quadratic equation as shown below. Here is the output:



PYTHON
```
num_of_inputs = np.array([1,2,3,4,5,6,7])
steps = [pow(n,2) for n in no_inputs]

plt.plot(no_inputs, steps, 'r')
plt.xlabel('Number of inputs')
plt.ylabel('Number of steps')
plt.title('O(n^2) Complexity')
plt.show()

# No matplotlib support in this terminal.
# Go to the next screen for the results!
```

Here no matter the size of the input, the `display_first_cube` has to allocate memory only once for the `result` variable. Hence the `Big(O)` notation for the space complexity of the "display_first_cube" will be `O(1)` which is basically `O(c)` i.e. constant.

Similarly, the space complexity of the `display_all_cubes` function will be `O(n)` since for each item in the input, space has to be allocated in memory.

Big(O) Notation for Space Complexity

To find space complexity, we simply calculate the space (working storage, or memory) that the algorithm will need to allocate against the items in the inputs. Let's again take a look at the `display first cube` function.

PYTHON
```
import numpy as np

def display_first_cube(items):
    result = pow(items[0],3)
    print (result)

inputs = np.array([2,3,4,5,6,7])
display_first_cube(inputs)
```

Best vs Worst Case Complexity

You may hear the term **worst case** when discussing complexities.

An algorithm can have two types of complexities. They are best case scenarios and worst case scenarios.

The best case complexity refers to the complexity of an algorithm in the ideal situation. For instance, if you want to search an item X in the list of N items. The best case scenario is that we find the item at the **first** index in which case the algorithm complexity will be `O(1)`.

The worst case is that we find the item at the `nth` (or last) index, in which case the algorithm complexity will be O(N) where `n` is the total number of items in the list. When we use the term algorithmic complexity, we generally refer to `worst case complexity`. This is to ensure that we are optimizing for the least ideal situation.

Conclusion

Algorithm complexity is used to measure the performance of an algorithm in terms of time taken and the space consumed.

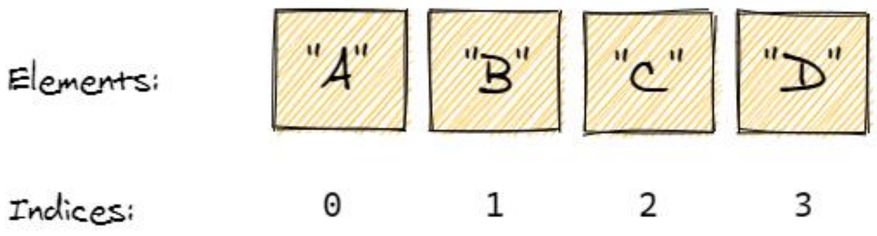
`Big(O) notation` is one of the most commonly used metrics for measuring algorithm complexity. In this article you saw how to find different types of time and space complexities of algorithms using Big(O) notation. You'll now be better equipped to make trade-off decisions based on complexity in the future.

An Executable
Data Structures Cheat Sheet
For Interviews

This cheat sheet uses the Big O notation to express time complexity.

- For a reminder on Big O, see **Understanding Big O Notation and Algorithmic Complexity**.
- For a quick summary of complexity for common data structure operations, see the **Big-O Algorithm Complexity Cheat Sheet**.

Array



- **Quick summary:** a collection that stores elements in order and looks them up by index.
- **Also known as:** fixed array, static array.
- **Important facts:**
    - Stores elements sequentially, one after another.
    - Each array element has an index. Zero-based indexing is used most often: the first index is 0, the second is 1, and so on.
    - Is created with a fixed size. Increasing or decreasing the size of an array is impossible.
    - Can be one-dimensional (linear) or multi-dimensional.
    - Allocates contiguous memory space for all its elements.
- **Pros:**
    - Ensures constant time access by index.
    - Constant time append (insertion at the end of an array).

- **Cons:**
  - Fixed size that can't be changed.
  - Search, insertion and deletion are O(n). After insertion or deletion, all subsequent elements are moved one index further.
  - Can be memory intensive when capacity is underused.
- **Notable uses:**
  - The String data type that represents text is implemented in programming languages as an array that consists of a sequence of characters plus a terminating character.
- **Time complexity (worst case):**
  - Access: O(1)
  - Search: O(n)
  - Insertion: O(n) (append: O(1))
  - Deletion: O(n)
- **See also:**
  - **A Gentle Refresher Into Arrays and Strings**
  - **Interview Problems: Easy Strings**
  - **Interview Problems: Basic Arrays**
  - **Interview Problems: Medium Arrays**

JAVASCRIPT

```
// instantiation
let empty = new Array();
let teams = new Array('Knicks', 'Mets', 'Giants');

// literal notation
let otherTeams = ['Nets', 'Patriots', 'Jets'];

// size
console.log('Size:', otherTeams.length);

// access
console.log('Access:', teams[0]);

// sort
const sorted = teams.sort();
console.log('Sorted:', sorted);

// search
const filtered = teams.filter((team) => team === 'Knicks');
console.log('Searched:', filtered);
```

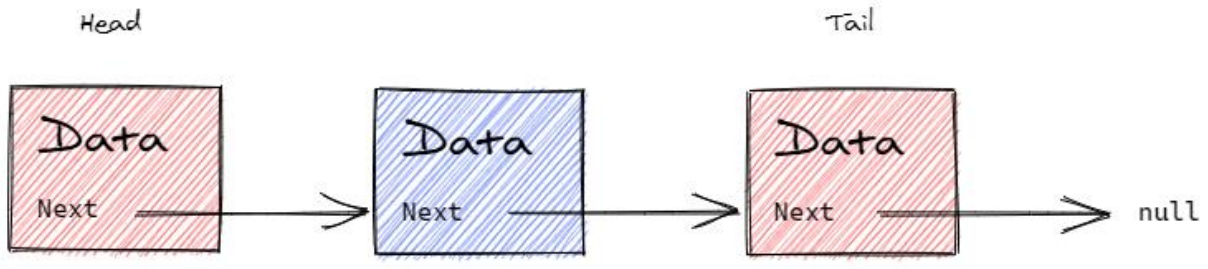
Dynamic array

- **Quick summary:** an array that can resize itself.
- **Also known as:** array list, list, growable array, resizable array, mutable array, dynamic table.
- **Important facts:**
  - Same as array in most regards: stores elements sequentially, uses numeric indexing, allocates contiguous memory space.
  - Expands as you add more elements. Doesn't require setting initial capacity.
  - When it exhausts capacity, a dynamic array allocates a new contiguous memory space that is double the previous capacity, and copies all values to the new location.
  - Time complexity is the same as for a fixed array except for worst-case appends: when capacity needs to be doubled, append is O(n). However, the average append is still `O(1)`.
- **Pros:**
  - Variable size. A dynamic array expands as needed.
  - Constant time access.
- **Cons:**
  - Slow worst-case appends: O(n). Average appends: O(1).
  - Insertion and deletion are still slow because subsequent elements must be moved a single index further. Worst-case (insertion into/deletion from the first index, a.k.a. prepending) for both is O(n).
- **Time complexity (worst case):**
  - Access: O(1)
  - Search: O(n)
  - Insertion: O(n) (append: O(n))
  - Deletion: O(n)
- **See also:** same as arrays (see above).

JAVASCRIPT

```
/* Arrays are dynamic in Javascript :-) */

// instantiation
let empty = new Array();
let teams = new Array('Knicks', 'Mets', 'Giants');

// literal notation
let otherTeams = ['Nets', 'Patriots', 'Jets'];

// size
console.log('Size:', otherTeams.length);

// access
console.log('Access:', teams[0]);

// sort
const sorted = teams.sort();
console.log('Sorted:', sorted);

// search
const filtered = teams.filter((team) => team === 'Knicks');
console.log('Searched:', filtered);
```

Linked List



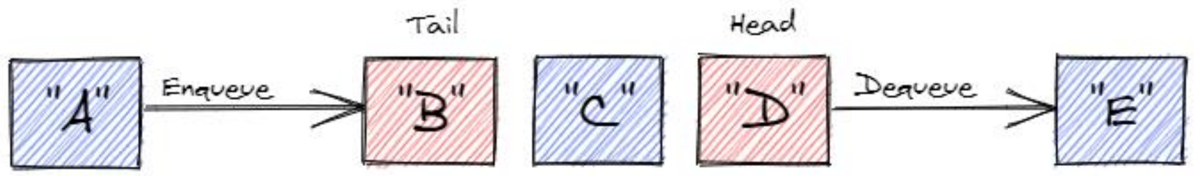
- **Quick summary**: a linear collection of elements ordered by links instead of physical placement in memory.
- **Important facts**:
  - Each element is called a *node*.
    - The first node is called the *head*.
    - The last node is called the *tail*.
  - Nodes are sequential. Each node stores a reference (pointer) to one or more adjacent nodes:
    - In a **singly linked list**, each node stores a reference to the next node.
    - In a **doubly linked list**, each node stores references to both the next and the previous nodes. This enables traversing a list backwards.
    - In a **circularly linked list**, the tail stores a reference to the head.
  - Stacks and queues are usually implemented using linked lists, and less often using arrays.
- **Pros**:
  - Optimized for fast operations on both ends, which ensures constant time insertion and deletion.
  - Flexible capacity. Doesn't require setting initial capacity, can be expanded indefinitely.
- **Cons**:
  - Costly access and search.
  - Linked list nodes don't occupy continuous memory locations, which makes iterating a linked list somewhat slower than iterating an array.
- **Notable uses**:
  - Implementation of stacks, queues, and graphs.
- **Time complexity** (worst case):
  - Access: O(n)
  - Search: O(n)

    - Insertion: O(1)
    - Deletion: O(1)
- **See also:**
    - What Is the Linked List Data Structure?
    - Implement a Linked List
    - Interview Problems: Linked Lists

JAVASCRIPT

```
function LinkedListNode(val) {
  this.val = val;
  this.next = null;
}

class MyLinkedList {
  constructor() {
    this.head = null;
    this.tail = null;
  }

  prepend(newVal) {
    const currentHead = this.head;
    const newNode = new LinkedListNode(newVal);
    newNode.next = currentHead;
    this.head = newNode;

    if (!this.tail) {
      this.tail = newNode;
    }
  }

  append(newVal) {
    const newNode = new LinkedListNode(newVal);
    if (!this.head) {
      this.head = newNode;
      this.tail = newNode;
    } else {
      this.tail.next = newNode;
      this.tail = newNode;
    }
  }
}

var linkedList1 = new MyLinkedList();
linkedList1.prepend(25);
linkedList1.prepend(15);
linkedList1.prepend(5);
linkedList1.prepend(9);
console.log(linkedList1);
```

Queue

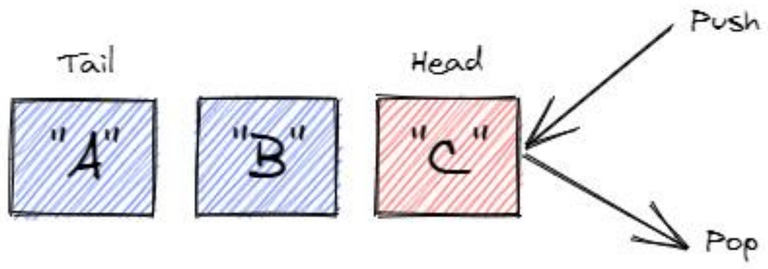


- **Quick summary**: a sequential collection where elements are added at one end and removed from the other end.
- **Important facts**:
    - Modeled after a real-life queue: first come, first served.
    - First in, first out (FIFO) data structure.
    - Similar to a linked list, the first (last added) node is called the *tail*, and the last (next to be removed) node is called the *head*.
    - Two fundamental operations are enqueuing and dequeuing:
        - To *enqueue*, insert at the tail of the linked list.
        - To *dequeue*, remove at the head of the linked list.
    - Usually implemented on top of linked lists because they're optimized for insertion and deletion, which are used to enqueue and dequeue elements.
- **Pros**:
    - Constant-time insertion and deletion.
- **Cons**:
    - Access and search are O(n).
- **Notable uses**:
    - CPU and disk scheduling, interrupt handling and buffering.
- **Time complexity** (worst case):
    - Access: O(n)
    - Search: O(n)
    - Insertion (enqueuing): O(1)
    - Deletion (dequeuing): O(1)
- **See also**:
    - Understanding the Queue Data Structure
    - Interview Problems: Queues

JAVASCRIPT

```javascript
class Queue {
  constructor() {
    this.queue = [];
  }

  enqueue(item) {
    return this.queue.unshift(item);
  }

  dequeue() {
    return this.queue.pop();
  }

  peek() {
    return this.queue[this.length - 1];
  }

  get length() {
    return this.queue.length;
  }

  isEmpty() {
    return this.queue.length === 0;
  }
}

const queue = new Queue();
queue.enqueue(1);
queue.enqueue(2);
console.log(queue);

queue.dequeue();
console.log(queue);
```

Stack



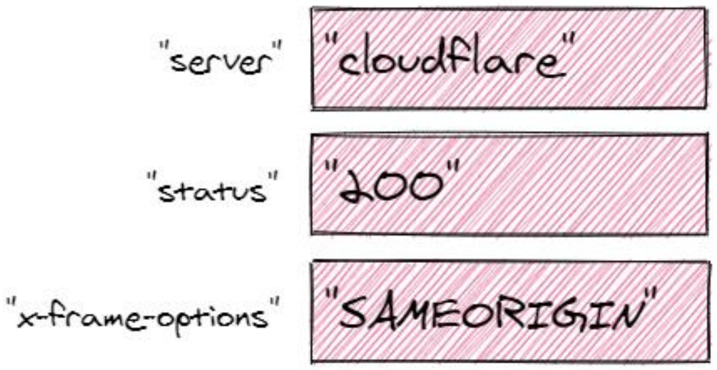
- **Quick summary**: a sequential collection where elements are added to and removed from the same end.

- **Important facts**:
    - First-in, last-out (FILO) data structure.
    - Equivalent of a real-life pile of papers on desk.
    - In stack terms, to insert is to *push*, and to remove is to *pop*.
    - Often implemented on top of a linked list where the head is used for both insertion and removal. Can also be implemented using dynamic arrays.
- **Pros**:
    - Fast insertions and deletions: O(1).
- **Cons**:
    - Access and search are O(n).
- **Notable uses**:
    - Maintaining undo history.
    - Tracking execution of program functions via a call stack.
    - Reversing order of items.
- **Time complexity** (worst case):
    - Access: O(n)
    - Search: O(n)
    - Insertion (pushing): O(1)
    - Deletion (popping): O(1)
- **See also**:
    - **The Gentle Guide to Stacks**
    - **Interview Problems: Stacks**

JAVASCRIPT

```javascript
class Stack {
  constructor() {
    this.stack = [];
  }

  push(item) {
    return this.stack.push(item);
  }

  pop() {
    return this.stack.pop();
  }

  peek() {
    return this.stack[this.length - 1];
  }

  get length() {
    return this.stack.length;
  }

  isEmpty() {
    return this.length === 0;
  }
}

const newStack = new Stack();
newStack.push(1);
newStack.push(2);
console.log(newStack);

newStack.pop();
console.log(newStack);
```

Hash Table



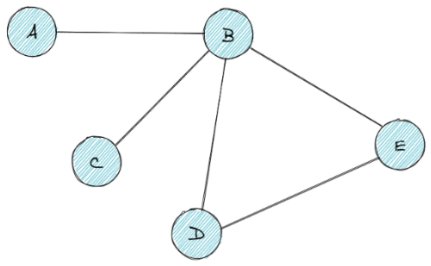
- **Quick summary**: unordered collection that maps keys to values using hashing.
- **Also known as**: hash, hash map, map, unordered map, dictionary, associative array.
- **Important facts**:
  - Stores data as key-value pairs.
  - Can be seen as an evolution of arrays that removes the limitation of sequential numerical indices and allows using flexible keys instead.
  - For any given non-numeric key, *hashing* helps get a numeric index to look up in the underlying array.
  - If hashing two or more keys returns the same value, this is called a *hash collision*. To mitigate this, instead of storing actual values, the underlying array may hold references to linked lists that, in turn, contain all values with the same hash.
  - A *set* is a variation of a hash table that stores keys but not values.
- **Pros**:
  - Keys can be of many data types. The only requirement is that these data types are hashable.
  - Average search, insertion and deletion are O(1).
- **Cons**:
  - Worst-case lookups are O(n).
  - No ordering means looking up minimum and maximum values is expensive.
  - Looking up the value for a given key can be done in constant time, but looking up the keys for a given value is O(n).
- **Notable uses**:
  - Caching.
  - Database partitioning.

- **Time complexity** (worst case):
    - Access: O(n)
    - Search: O(n)
    - Insertion: O(n)
    - Deletion: O(n)
- **See also**:
    - **Interview Problems: Hash Maps**

JAVASCRIPT

```
class Hashmap {
  constructor() {
    this._storage = [];
  }

  hashStr(str) {
    let finalHash = 0;
    for (let i = 0; i < str.length; i++) {
      const charCode = str.charCodeAt(i);
      finalHash += charCode;
    }
    return finalHash;
  }

  set(key, val) {
    let idx = this.hashStr(key);

    if (!this._storage[idx]) {
      this._storage[idx] = [];
    }

    this._storage[idx].push([key, val]);
  }

  get(key) {
    let idx = this.hashStr(key);

    if (!this._storage[idx]) {
      return undefined;
    }

    for (let keyVal of this._storage[idx]) {
      if (keyVal[0] === key) {
        return keyVal[1];
      }
    }
  }
}

var dict = new Hashmap();
dict.set("james", "123-456-7890");
dict.set("jess", "213-559-6840");
console.log(dict.get("james"));
```

Graph



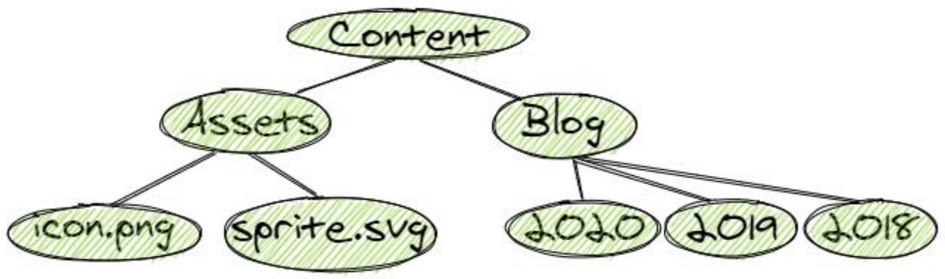
- **Quick summary**: a data structure that stores items in a connected, non-hierarchical network.
- **Important facts**:
    - Each graph element is called a *node*, or *vertex*.
    - Graph nodes are connected by *edges*.
    - Graphs can be *directed* or *undirected*.
    - Graphs can be *cyclic* or *acyclic*. A cyclic graph contains a path from at least one node back to itself.
    - Graphs can be *weighted* or *unweighted*. In a weighted graph, each edge has a certain numerical weight.
    - Graphs can be *traversed*. See important facts under *Tree* for an overview of traversal algorithms.
    - Data structures used to represent graphs:
        - *Edge list*: a list of all graph edges represented by pairs of nodes that these edges connect.
        - *Adjacency list*: a list or hash table where a key represents a node and its value represents the list of this node's neighbors.
        - *Adjacency matrix*: a matrix of binary values indicating whether any two nodes are connected.
- **Pros**:
    - Ideal for representing entities interconnected with links.
- **Cons**:
    - Low performance makes scaling hard.
- **Notable uses**:
    - Social media networks.
    - Recommendations in ecommerce websites.
    - Mapping services.
- **Time complexity** (worst case): varies depending on the choice of algorithm. O(n*lg(n)) or slower for most graph algorithms.

- **See also**:
    - [The Simple Reference to Graphs]()

JAVASCRIPT

```javascript
class Graph {
  constructor() {
    this.adjacencyList = {};
  }

  addVertex(nodeVal) {
    this.adjacencyList[nodeVal] = [];
  }

  addEdge(src, dest) {
    this.adjacencyList[src].push(dest);
    this.adjacencyList[dest].push(src);
  }

  removeVertex(val) {
    if (this.adjacencyList[val]) {
      this.adjacencyList.delete(val);
    }

    this.adjacencyList.forEach((vertex) => {
      const neighborIdx = vertex.indexOf(val);
      if (neighborIdx >= 0) {
        vertex.splice(neighborIdx, 1);
      }
    });
  }

  removeEdge(src, dest) {
    const srcDestIdx = this.adjacencyList[src].indexOf(dest);
    this.adjacencyList[src].splice(srcDestIdx, 1);

    const destSrcIdx = this.adjacencyList[dest].indexOf(src);
    this.adjacencyList[dest].splice(destSrcIdx, 1);
  }
}

var graph = new Graph(7);
var vertices = ["A", "B", "C", "D", "E", "F", "G"];
for (var i = 0; i < vertices.length; i++) {
  graph.addVertex(vertices[i]);
}
graph.addEdge("A", "G");
graph.addEdge("A", "E");
graph.addEdge("A", "C");
graph.addEdge("B", "C");
graph.addEdge("C", "D");
graph.addEdge("D", "E");
graph.addEdge("E", "F");
graph.addEdge("E", "C");
graph.addEdge("G", "D");
console.log(graph.adjacencyList);
```

Tree



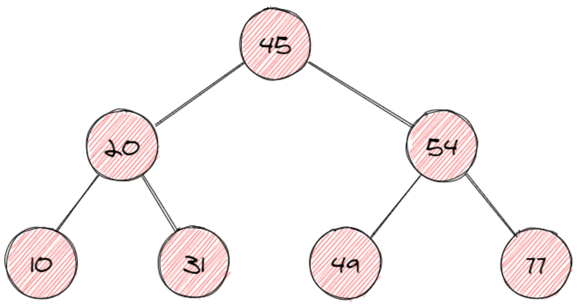
- **Quick summary**: a data structure that stores a hierarchy of values.
- **Important facts**:
    - Organizes values hierarchically.
    - A tree item is called a *node*, and every node is connected to 0 or more child nodes using *links*.
    - A tree is a kind of graph where between any two nodes, there is only one possible path.
    - The top node in a tree that has no parent nodes is called the *root*.
    - Nodes that have no children are called *leaves*.
    - The number of links from the root to a node is called that node's *depth*.
    - The height of a tree is the number of links from its root to the furthest leaf.
    - In a *binary tree*, nodes cannot have more than two children.
        - Any node can have one left and one right child node.
        - Used to make *binary search trees*.
        - In an unbalanced binary tree, there is a significant difference in height between subtrees.
        - An completely one-sided tree is called a *degenerate tree* and becomes equivalent to a linked list.
    - Trees (and graphs in general) can be *traversed* in several ways:
        - *Breadth first search* (BFS): nodes one link away from the root are visited first, then nodes two links away, etc. BFS finds the shortest path between the starting node and any other reachable node.
        - *Depth first search* (DFS): nodes are visited as deep as possible down the leftmost path, then by the next path to the right, etc. This method is less memory intensive than BFS. It comes in several flavors, including:

        - *Pre order traversal* (similar to DFS): after the current node, the left subtree is visited, then the right subtree.
        - *In order traversal*: the left subtree is visited first, then the current node, then the right subtree.
        - *Post order traversal*. the left subtree is visited first, then the right subtree, and finally the current node.
- **Pros**:
    - For most operations, the average time complexity is O(log(n)), which enables solid scalability. Worst time complexity varies between O(log(n)) and O(n).
- **Cons**:
    - Performance degrades as trees lose balance, and re-balancing requires effort.
    - No constant time operations: trees are *moderately* fast at everything they do.
- **Notable uses**:
    - File systems.
    - Database indexing.
    - Syntax trees.
    - Comment threads.
- **Time complexity**: varies for different kinds of trees.
- **See also**:
    - **Interview Problems: Trees**

JAVASCRIPT

```
function TreeNode(value) {
  this.value = value;
  this.children = [];
  this.parent = null;

  this.setParentNode = function (node) {
    this.parent = node;
  };

  this.getParentNode = function () {
    return this.parent;
  };

  this.addNode = function (node) {
    node.setParentNode(this);
    this.children[this.children.length] = node;
  };

  this.getChildren = function () {
    return this.children;
  };

  this.removeChildren = function () {
    this.children = [];
  };
}

const root = new TreeNode(1);
root.addNode(new TreeNode(2));
root.addNode(new TreeNode(3));

const children = root.getChildren();
for (let i = 0; i < children.length; i++) {
  for (let j = 0; j < 5; j++) {
    children[i].addNode(new TreeNode("second level child " + j));
  }
}

console.log(root);
children[0].removeChildren();
console.log(root);
console.log(root.getParentNode());
console.log(children[1].getParentNode());
```

Binary Search Tree



- **Quick summary**: a kind of binary tree where nodes to the left are smaller, and nodes to the right are larger than the current node.
- **Important facts**:
    - Nodes of a binary search tree (BST) are ordered in a specific way:
        - All nodes to the left of the current node are smaller (or sometimes smaller or equal) than the current node.
        - All nodes to the right of the current node are larger than the current node.
    - Duplicate nodes are usually not allowed.
- **Pros**:
    - Balanced BSTs are moderately performant for all operations.
    - Since BST is sorted, reading its nodes in sorted order can be done in O(n), and search for a node closest to a value can be done in O(log(n))
- **Cons**:
    - Same as trees in general: no constant time operations, performance degradation in unbalanced trees.
- **Time complexity** (worst case):
    - Access: O(n)
    - Search: O(n)
    - Insertion: O(n)
    - Deletion: O(n)
- **Time complexity** (average case):
    - Access: O(log(n))
    - Search: O(log(n))
    - Insertion: O(log(n))
    - Deletion: O(log(n))
- **See also**:
    - An Intro to Binary Trees and Binary Search Trees

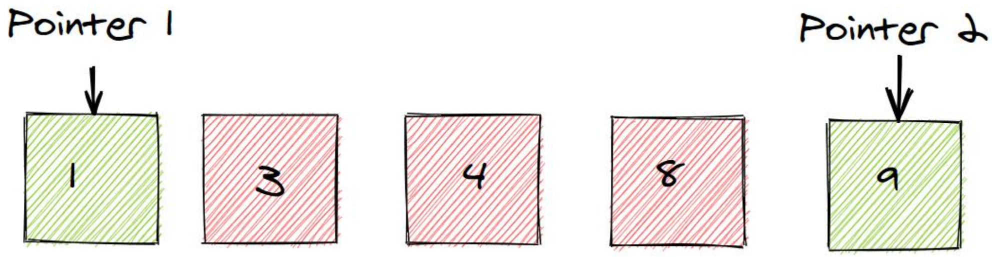
- [Interview Problems: Binary Search Trees]

JAVASCRIPT

```javascript
function Node(val) {
  this.val = val;
  this.left = null;
  this.right = null;
}

class BST {
  constructor(val) {
    this.root = new Node(val);
  }

  add(val) {
    let newNode = new Node(val);

    function findPosAndInsert(currNode, newNode) {
      if (newNode.val < currNode.val) {
        if (!currNode.left) {
          currNode.left = newNode;
        } else {
          findPosAndInsert(currNode.left, newNode);
        }
      } else {
        if (!currNode.right) {
          currNode.right = newNode;
        } else {
          findPosAndInsert(currNode.right, newNode);
        }
      }
    }

    if (!this.root) {
      this.root = newNode;
    } else {
      findPosAndInsert(this.root, newNode);
    }
  }

  remove(val) {
    let self = this;
    let removeNode = function (node,  val) {
      if (!node) {
        return null;
      }
      if (val === node.val) {
        if (!node.left && !node.right) {
          return null;
        }
        if (!node.left) {
          return node.right;
        }
        if (!node.right) {
          return node.left;
        }
        let temp = self.getMinimum(node.right);
        node.val = temp;
        node.right = removeNode(node.right, temp);
        return node;
      } else if (val < node.val) {
        node.left = removeNode(node.left, val);
        return node;
      } else {
        node.right = removeNode(node.right, val);
        return node;
      }
    };
    this.root = removeNode(this.root, val);
  }

  getMinimum(node) {
    if (!node) {
      node = this.root;
    }
    while (node.left) {
      node = node.left;
    }
    return node.val;
  }

  // helper method
  contains(value) {
    let doesContain = false;

    function traverse(bst) {
      if (this.root.value === value) {
        doesContain = true;
      } else if (this.root.left !== undefined && value < this.root.value) {
        traverse(this.root.left);
      } else if (this.root.right !== undefined && value > this.root.value) {
        traverse(this.root.right);
      }
    }

    traverse(this);
    return doesContain;
  }
}

const bst = new BST(4);
bst.add(3);
bst.add(5);
bst.remove(3);
console.log(bst);
```

The Two Pointer
Technique

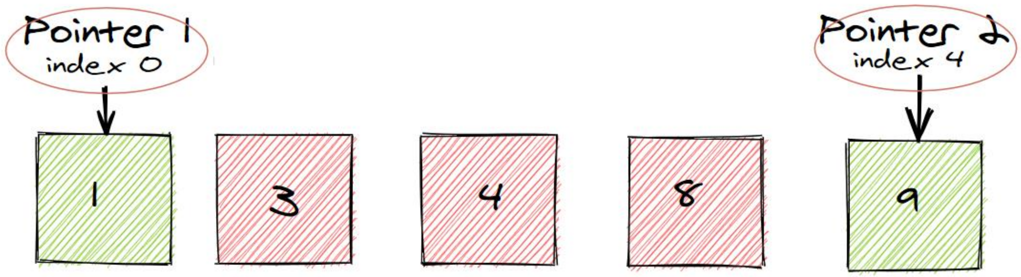
The **two pointer technique** is a near necessity in any software developer's toolkit, especially when it comes to technical interviews. In this guide, we'll cover the basics so that you know when and how to use this technique.



What is the pattern?

The name `two pointers` does justice in this case, as it is exactly as it sounds. It's the use of two different pointers (usually to keep track of array or string indices) to solve a problem involving said indices with the benefit of saving time and space. See the below for the two pointers highlighted in yellow.

But what are `pointers`? In computer science, a `pointer` is a reference to an object. In many programming languages, that object stores a memory address of another value located in computer memory, or in some cases, that of memory-mapped computer hardware.



When do we use it?

In many problems involving collections such as arrays or lists, we have to analyze each element of the collection compared to its other elements.

There are many approaches to solving problems like these. For example we usually start from the first index and iterate through the `data structure` one or more times depending on how we implement our code.

Sometimes we may even have to create an additional `data structure` depending on the problem's requirements. This approach might give us the correct result, but it likely won't give us the most space and time efficient result.

This is why the `two-pointer technique` is efficient. We are able to process two elements per loop instead of just one. Common patterns in the two-pointer approach entail:

1. Two pointers, each starting from the beginning and the end until they both meet.
2. One pointer moving at a slow pace, while the other pointer moves at twice the speed.

These patterns can be used for string or array questions. They can also be streamlined and made more efficient by iterating through two parts of an object simultaneously. You can see this in the **Two Sum problem** or **Reverse a String** problems.

Running through an example

One usage is while searching for pairs in an array. Let us consider **a practical example**: assume that you have a sorted array `arr`.

You're tasked with figuring out the pair of elements where `arr[p]` + `arr[q]` add up to a certain number. (To try this problem out, check the **Two Sum** and **Sorted Two Sum** problems here.)

The brute force solution is to compare each element with every other number, but that's a time complexity of `O(n^2)`. We can do better!

So let's optimize. You need to identify the indices `pointer_one` and `pointer_two` whose values sum to the integer `target`.

Let's initialize two variables, `pointer_one` and `pointer_two`, and consider them as our two pointers.

PYTHON
```
pointer_one = 0
pointer_two = len(arr)-1
```

Note that `len(arr)-1` helps to get the last index possible in an array.

Also observe that when we start, `pointer_one` points to the first element of the array, and `pointer_two` points to the last element.

This won't always be the case with this technique (we'll explore the **sliding window concept** later, which uses two pointers but have them move in a different direction). For our current purposes, it is more efficient to start wide, and iteratively narrow in (particularly if the array is sorted).

PYTHON
```
def two_sum(arr, target):
    pointer_one = 0
    pointer_two = input.length - 1

    while pointer_one < pointer_two:
```

Since the array is already sorted, and we're looking to process an index at each iteration, we can use two pointers to process them faster. One pointer *starts from the beginning of the array*, and the other pointer *begins from the end of the array*, and then we add the values at these pointers.

Once we're set up, what we want to do is check if the current pointers already sum up to our target. This might happen if the correct ones are on the exact opposite ends of the array.

Here's what the check might look like:

PYTHON
```
if sum == targetValue:
  return true
```

However, it likely will not be the target immediately. Thus, we apply this logic: if the sum of the values is less than the target value, we increment the left pointer (move your left pointer `pointer_one` one index rightwards).

And if the sum is higher than the target value, we decrement the right pointer (correct the position of your pointer `pointer_two` if necessary).

PYTHON
```
elif sum < targetValue:
  pointer_one += 1
else:
  pointer_two -= 1
```

In other words, understand that if `arr[pointer_one]` < `target-arr[pointer_two]`, it means we should move forward on `pointer_one` to get closer to where we want to be in magnitude.

This is what it looks like all put together:

PYTHON
```
def two_sum(arr, target):
    pointer_one = 0
    pointer_two = input.length - 1

    while pointer_one < pointer_two:
        sum = input[pointer_one] + input[pointer_two]

        if sum == targetValue:
            return true
        elif sum < targetValue:
            pointer_one += 1
        else:
            pointer_two -= 1

    return false
```

It's crucial to see that how both indices were moving in conjunction, and how they depend on each other.

We kept moving the pointers until we got the sum that matches the target value-- or until we reached the middle of the array, and no combinations were found.

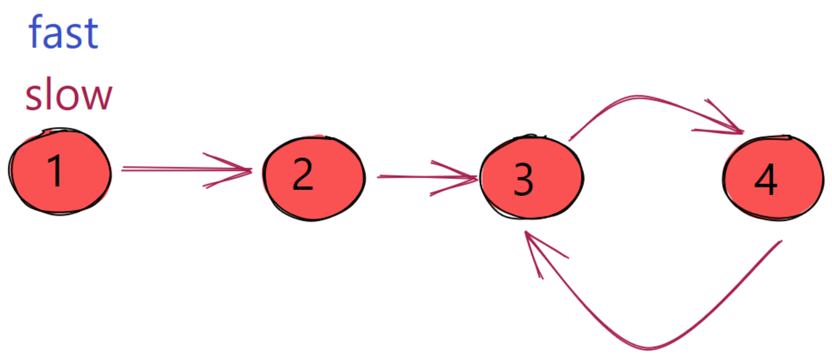
The time complexity of this solution is `O(n)` and space complexity is `O(1)`, a significant improvement over our first implementation!

Another Example

In addition, to the previous example, the two pointer technique can also involve the pattern of using a fast pointer and a slow pointer.

PYTHON
```
Node fast = head, slow = head
```

One usage is through detecting cycles in a `linked list` data structure. For example, a cycle (when a node points back to a previous node) begins at the last node of the `linked list` in the example below.

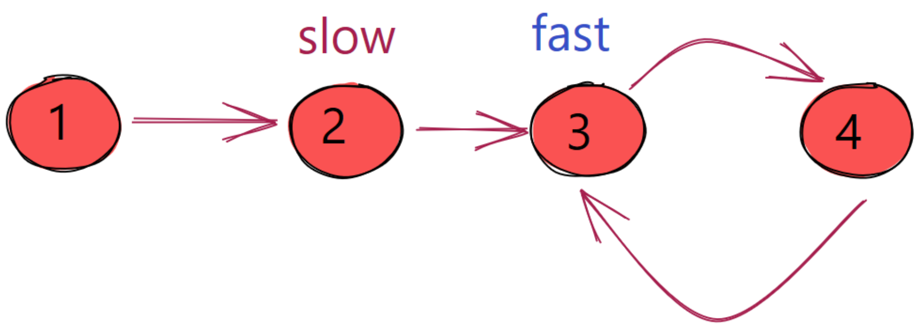


STEP 1

SNIPPET

```
1 -- > 2 --> 3 --> 4

             ^     |

             |     |

             <- - -
```

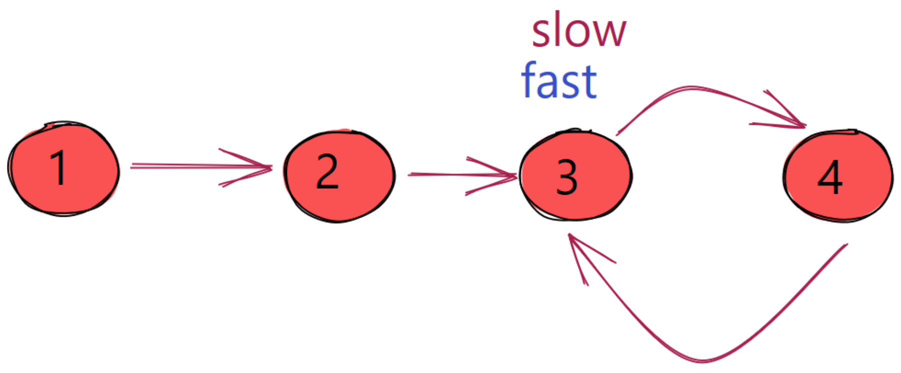
The idea is to move the fast pointer twice as quickly as the slow pointer so the distance between them increases by `1` at each step.



STEP 2

TEXT/X-JAVA
```
while(fast!=null && fast.next!=null){
  slow = slow.next;

  fast = fast.next.next;
```

However, if at some point both pointers meet, then we have found a cycle in the linked list. Otherwise we'll have have reached the end of the list and no cycle is present.



STEP 3

TEXT/X-JAVA
```
if (slow==fast) return true;
```

The attached code is what the entire method would look like all together.

The time complexity would be `O(N)` or linear time.

TEXT/X-JAVA
```
public static boolean detectCycle(Node head){
    Node fast = head, slow = head;

    while(fast!=null && fast.next!=null){
        slow = slow.next;

        fast = fast.next.next;

        if(slow==fast) return true;

    }
    return false;
}
```

Reverse a String

Question

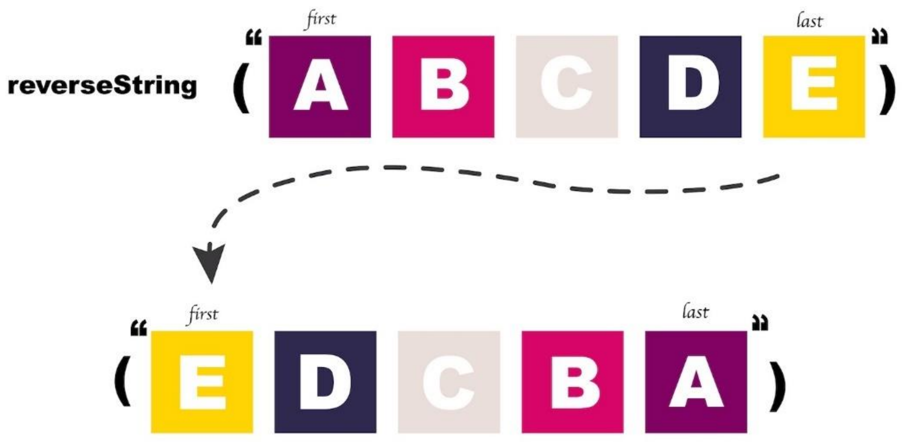
Hey there, welcome to the **challenges** portion of the **AlgoDaily** technical interview course! Over the next few days, you'll get some hands-on experience with the essential data structures and algorithms that will help you nail your interview, and land your dream software engineering job.

The only way to get better at solving these problems is to power through a few.

We covered the best ways to prepare **in this lesson**, in **this one**, and **here**. Be sure to visit those if you need some more guidance. Otherwise, let's jump into it.

Reverse a String

Let's **reverse a string**!



We'll usually start with a simple prompt, as you would in a regular technical interview. Like most, this one will be intentionally open-ended.

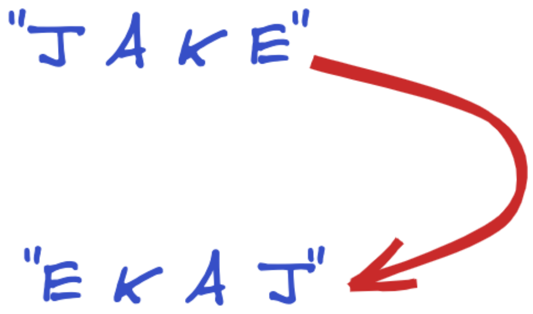
Prompt

Can you write a function that reverses an inputted string without using the built-in `Array#reverse` method?

Let's look at some examples. So, calling:

`reverseString("jake")` should return `"ekaj"`.

`reverseString("reverseastring")` should return `"gnirtsaesrever"`.



True or False?

In Java, C#, JavaScript, Python and Go, strings are `immutable`. This means the string object's state can't be changed after creation.

Solution: True

On Interviewer Mindset

Today on AlgoDaily, we're going to reverse a string. Reversing a string is one of the most common technical interview questions that candidates get. Interviewers love it because it's deceptively simple. After all, as a software engineer, you'd probably call the `#reverse` method on your favorite `String` class and call it a day!

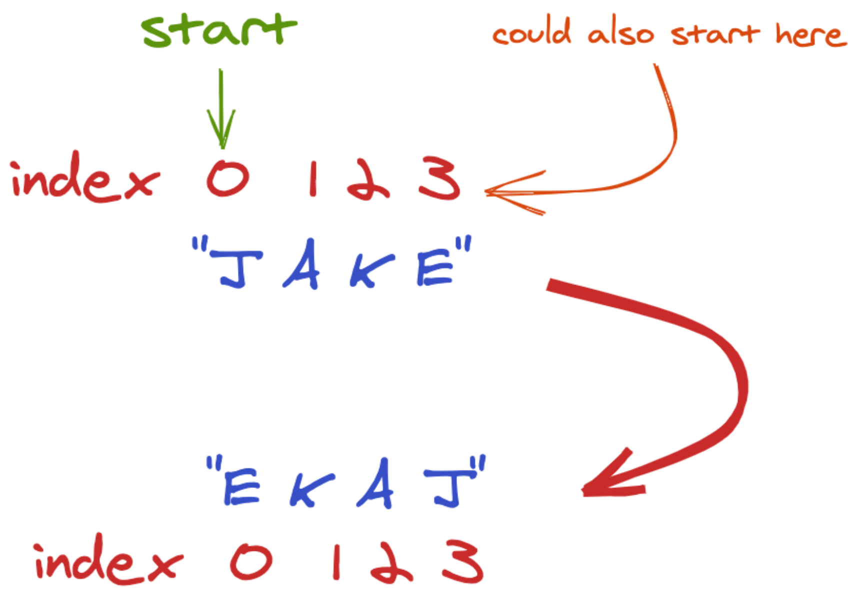
So don't overlook this one-- it appears a surprising amount as a warm-up or build-up question. Many interviewers will take the approach of using an easy question like this one, and actually judge much more harshly. You'll want to make you sure really nail this.

How We'll Begin Solving

We want the **string reversed**, which means that we end up with all our letters positioned backwards. *If you need a quick review of `string`s, check out [our lesson on arrays and strings](#).*

We know that `string`s can be thought of as character arrays-- that is, each element in the array is a single character. And if we can assume that, then we know the location (array position) of each character, as well as the index when the `array` ends.

We can apply that paradigm to operating on this string. Thus we can step through each of its indices. Stepping through the beginning of the string, we'll make these observations at each point:



```
JAVASCRIPT
const str = "JAKE";
// position 0 - "J"
// position 1 - "A"
// ...
```

Since a reversed string is just itself backwards, **a brute force solution** could be to use the indices, and iterate from *the back to the front*.

See the code attached and try to run it using `Run Sample Code`. You'll see that we log out each character from the back of the string!

JAVASCRIPT

```
function reverseString(str) {
  let newString = '';

    // start from end
  for (let i = str.length-1; i >= 0; i--) {
    console.log('Processing ', newString, str[i]);
        // append it to the string builder
    newString = newString + str[i];
  }

    // return the string
  return newString;
}

console.log(reverseString('test'));
```

Fill In

We want to `console.log` out:

JAVASCRIPT

```
5
4
3
2
1
```

What's the missing line here?

JAVASCRIPT

```
var arr =  [1, 2, 3, 4, 5];

for (var i = ___________; i >= 0; i--) {
    console.log(arr[i]);
}
```

Solution: Arr.length - 1

Can We Do Better Than Brute Force?

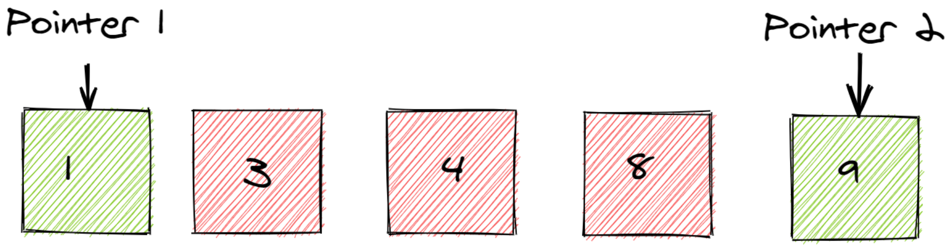
However, it wouldn't really be an interesting algorithms question if there wasn't a better way. Let's see how we can optimize this, or make it run faster. When trying to make something more efficient, it helps to think of things to *cut or reduce*.

One thing to note is that we're going through the *entire* string-- do we truly need to iterate through every single letter?

Let's examine a worst case scenario. What if the string is a million characters long? That would be a million operations to work through! Can we improve it?

Yes, With More Pointers!

Well, we're only working with a single pointer right now. The iterator from our loop starts from the back, and appends each character to a new string, one by one. Having gone through The Two Pointer Technique, we may recognize that some dramatic improvements can be had by increasing the number of pointers we use.



By this I mean, we can **cut the number of operations in half**. How? What if we did some **swapping** instead? By using a `while` loop and two pointers-- one on the left and one on the right.

With this in mind-- the big reveal is that, at each iteration, we can swap the letters at the pointer indices. After swapping, we would increment the `left` pointer while decrementing the `right` one. That could be hard to visualize, so let's see a basic example listed out.

SNIPPET
```
jake    // starting string

eakj    // first pass
^  ^


ekaj    // second pass
 ^^
```

Multiple Choice

What's a good use case for the two pointers technique?

- Shifting indices to be greater at each iteration
- Reducing a solution with a nested for-loop and O(n^2) complexity to O(n)
- Finding pairs and duplicates in a for-loop
- None of these

Solution: Reducing a solution with a nested for-loop and O(n^2) complexity to O(n)

With two pointers, we've cut the number of operations in half. It's much faster now! However, similar to the brute force, the time complexity is still `O(n)`.

Why Is This?

Well, if `n` is the length of the string, we'll end up making `n/2` swaps. But remember, `Big O Notation` isn't about the raw number of operations required for an algorithm-- it's about *how the number scales with the input*.

So despite requiring half the number operations-- a `4`-character string would require `2` swaps with the two-pointer method. But an `8`-character string would require `4` swaps. The input doubled, and so did the number of operations.
If you haven't by now, try to do the problem in `MY CODE` before moving on.

Final Solution

JAVASCRIPT
```javascript
function reverseString(str) {
  let strArr = str.split("");
  let start = 0;
  let end = str.length - 1;

  while (start <= end) {
    const temp = strArr[start];
    strArr[start] = strArr[end];
    strArr[end] = temp;
    start++;
    end--;
  }

  return strArr.join("");
}
```

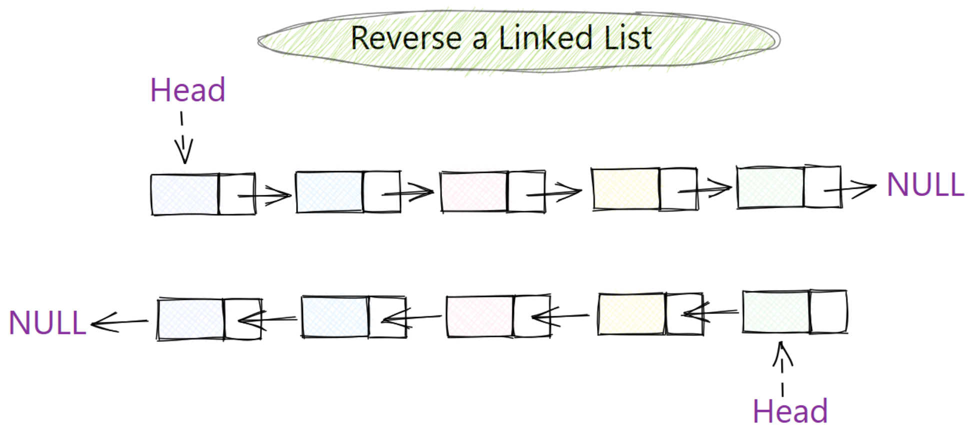
Reverse a Linked List

Question

You're sent a `linked list` of numbers, but it's been received in the opposite order to what you need. This has happened multiple times now, so you decide to write an algorithm to reverse the lists as they come in. The list you've received is as follows:

```
JAVASCRIPT
// 17 -> 2 -> 21 -> 6 -> 42 -> 10
```

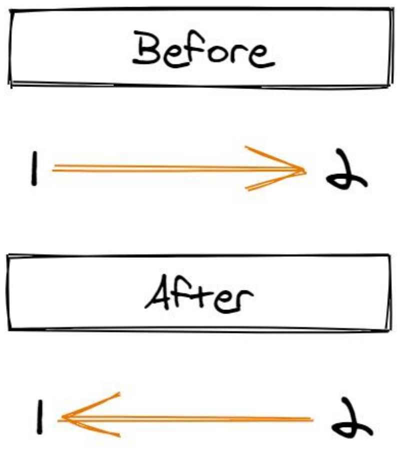
Write an algorithm for a method `reverseList` that takes in a `head` node as a parameter, and reverses the linked list. It should be capable of reversing a list of any length.



You may use the example `linked list` for testing purposes. Your method will be called as such:

```
JAVASCRIPT
class LinkedListNode {
  constructor(val, next = null) {
    this.val = val;
    this.next = next;
  }
}

l1 = new LinkedListNode(1);
l1.next = new LinkedListNode(2);
reverseList(l1);
```

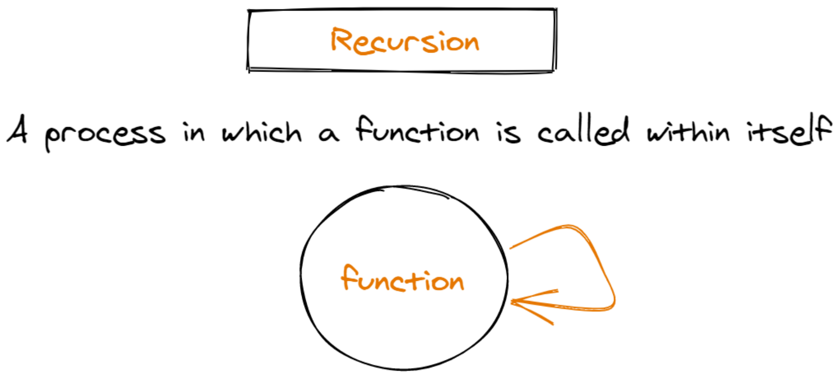
Seems pretty easy, right? To reverse an entire linked list, simply reverse every pointer. If `1` is pointing at `2`, flip it so `2` should point to `1`.



JAVASCRIPT
```javascript
// 17 -> 2 -> 21 -> 6 -> 42 -> 10
// becomes
// 17 <- 2 <- 21 <- 6 <- 42 <- 10
```

The actual reversal method is actually pretty straightforward, but be aware that it takes some time to reason out. It's easy to get lost, so make sure you draw lots of diagrams.

As this is a problem (reversing an entire linked list) that can be broken up into sub-problems (reverse the pointer between two nodes), it seems like a good opportunity to use recursion.



There are many ways to do the actual reversal, and we'll cover both an **iterative** and **recursive** approach, but the general methodology is as follows:

Begin by creating 3 pointers: `newHead`, `head` and `nextNode`.

`newHead` and `nextNode` are initialized to `null`.

`head` starts off pointing to the head of the linked list.

Iiterate (or recursively do) through the following process until `head` is `null`. This means that the end of the list has been reached:

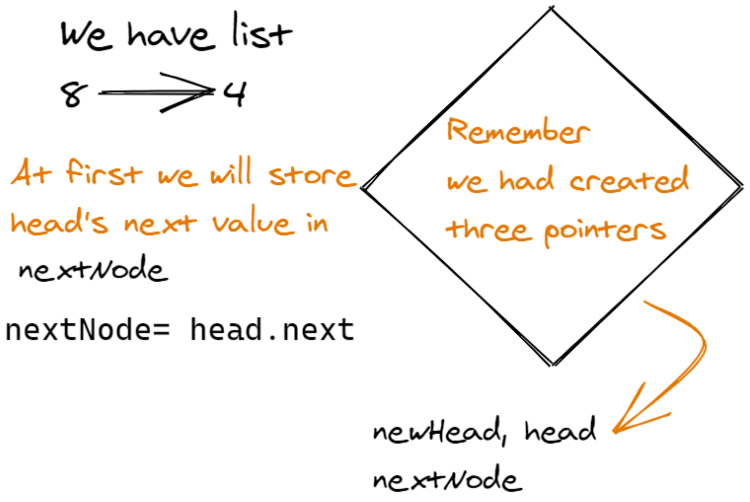
JAVASCRIPT

```
class LinkedListNode {
  constructor(val, next = null) {
    this.val = val;
    this.next = next;
  }
}

l1 = new LinkedListNode(1);
l2 = new LinkedListNode(2);
l1.next = l2;

// we start at head
let head = l1;
let newHead = null;
while (head != null) {
  // store the node to the right to reuse later
  let nextNode = head.next;
  // set the current node's next to point backwards
  head.next = newHead;
  // store the current node, to be used as the new next later
  newHead = head;
  // the previously right-side node is now processed
  head = nextNode;
}

console.log(l2);
```

It's difficult to visualize this chain of events, so let's use comments to visualize it. During the interview, *try not to keep it in your head*.

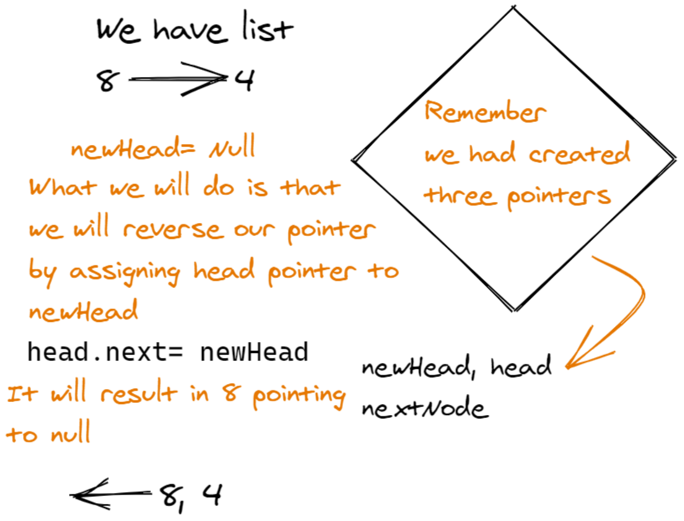
It'll be especially difficult while balancing your nerves and talking to the interviewer. Take advantage of the whiteboard not just to record things, but also to think through potential steps.

Let's walk through it step by step and then look at working code. Let's reverse an extremely basic list, like `8 -> 4`. The first line is `let nextNode = head.next;`, which will store the node to the right.


We have list
8 → 4
At first we will store head's next value in nextNode
nextNode= head.next
Remember we had created three pointers
newHead, head
nextNode

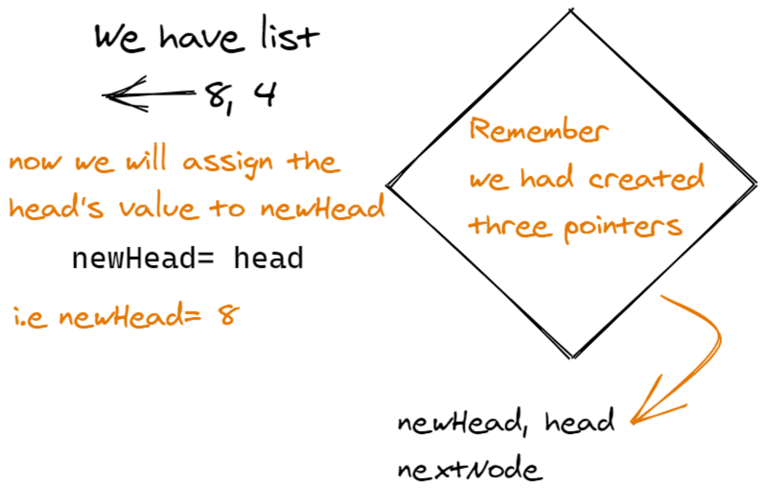
JAVASCRIPT
```
nextNode = 4
// 8 -> 4
```

Then we'll do `head.next = newHead;`, which will set the current node's `next` to point backwards.


We have list
8 → 4
newHead= Null
What we will do is that we will reverse our pointer by assigning head pointer to newHead
head.next= newHead
It will result in 8 pointing to null
← 8, 4
Remember we had created three pointers
newHead, head
nextNode

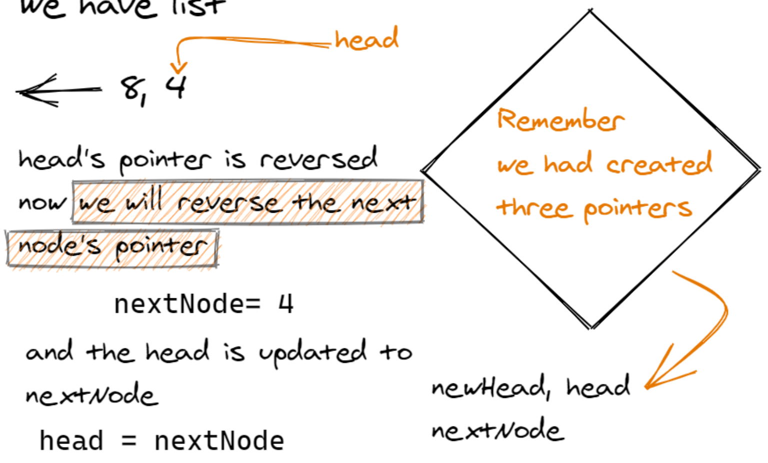
JAVASCRIPT
```
nextNode = 4
// <- 8, 4
```

Now `newHead = head;` will store the current node, to be used as the new next later.


We have list
← 8, 4
now we will assign the head's value to newHead
newHead= head
i.e newHead= 8
Remember we had created three pointers
newHead, head
nextNode

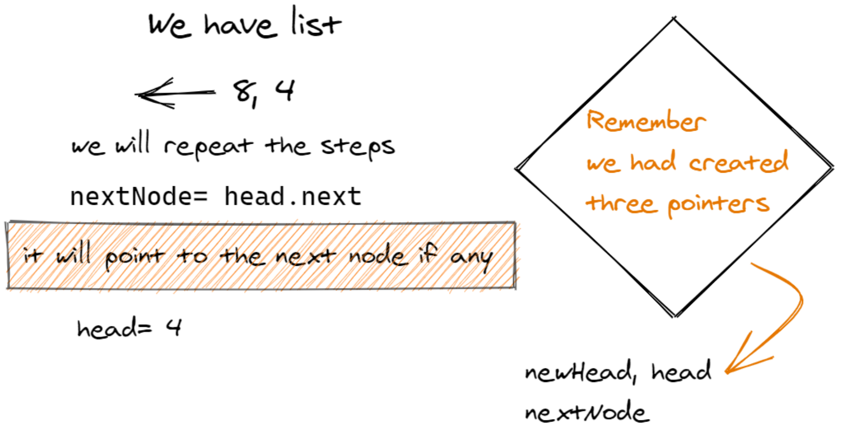
```
JAVASCRIPT
newHead = 8
nextNode = 4
// <- 8, 4
```

Finally, the previously right-side node is now processed:


We have list
head
← 8, 4
head's pointer is reversed
now we will reverse the next node's pointer
nextNode= 4
and the head is updated to nextNode
head = nextNode
Remember we had created three pointers
newHead, head
nextNode

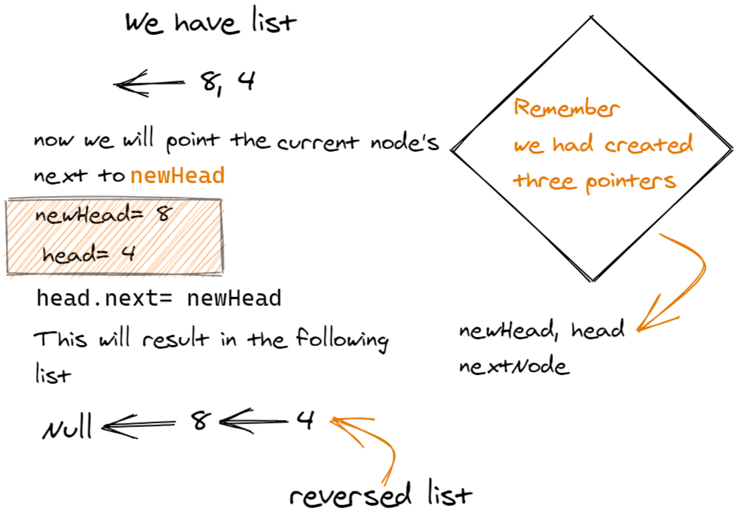
```
JAVASCRIPT
newHead = 8
nextNode = 4
// <- 8, 4
         ^
   current node
```

Now we process the next one with the same steps. `nextNode = head.next;` will store the node to the right.



```
JAVASCRIPT
newHead = 8
nextNode = null
// <- 8, 4
         ^
   current node
```

Again, set the current node's `next` to point backwards with `head.next = newHead;`. Recall that `newHead` is `8`! This is where we make the switch:



```
JAVASCRIPT
newHead = 8
nextNode = null
// <- 8 <- 4
           ^
     current node
```

Now let's see this all put together in code, with lots of comments for edification!

JAVASCRIPT
```javascript
class LinkedListNode {
  constructor(val, next = null) {
    this.val = val;
    this.next = next;
  }
}

l1 = new LinkedListNode(8);
l2 = new LinkedListNode(4);
l1.next = l2;

// start at head, 8
let head = l1;
// example: 8 -> 4
let newHead = null;
while (head) {
  /* FIRST PASS */
  // store the node to the right
  let nextNode = head.next;
  // nextNode = 4, still 8 -> 4
  // set the current node's next to point backwards
  head.next = newHead;
  // 8 -> null
  // store the current node, to be used as the new next later
  newHead = head;
  // newHead = 8
  // the previously right-side node is now processed
  head = nextNode;
  // head = 4

  /* SECOND PASS */
  // store the node to the right
  nextNode = head.next;
  // nextNode = null
  // set the current node's next to point backwards
  head.next = newHead;
  // 4 -> 8
  // store the current node as the previous one
  newHead = head;
  // the previously right-side node is now processed
  head = nextNode;
}

console.log(l2);
```

Does that all make sense? Be sure to go through the iterative approach a few times.

Here's the recursive way to do it. This can also be tricky, especially on first glance, but realize most of the magic happens when it gets to the end.

JAVASCRIPT
```
function reverseList(head) {
  if (!head || !head.next) {
    return head;
  }

  let rest = reverseList(head.next);

  head.next.next = head;
  delete head.next;
  return rest;
}
```

Let's take an easy example of `8 -> 4` again `let rest = reverseList(head.next);` takes `4` and calls `reverseList` on it.

Calling `reverseList` on `4` will have us reach the termination clause because there is no `.next`:

JAVASCRIPT
```
if (!head || !head.next) {
  return head;
}
```

We go up the stack back to when `8` was being processed. `rest` now simply points to `4`. Now notice what happens:

JAVASCRIPT
```
// remember, head is 8 - it is being processed
// head.next is 4
head.next.next = head;
// head.next.next was null since 4 wasn't pointing to anything
// but now head.next (4) points to 8
```

And we return `4` - which is pointing to `8`. And we can simply extrapolate that to longer linked lists! Note that the recursive approach requires more space because we need to maintain our call stack.

Final Solution

JAVASCRIPT

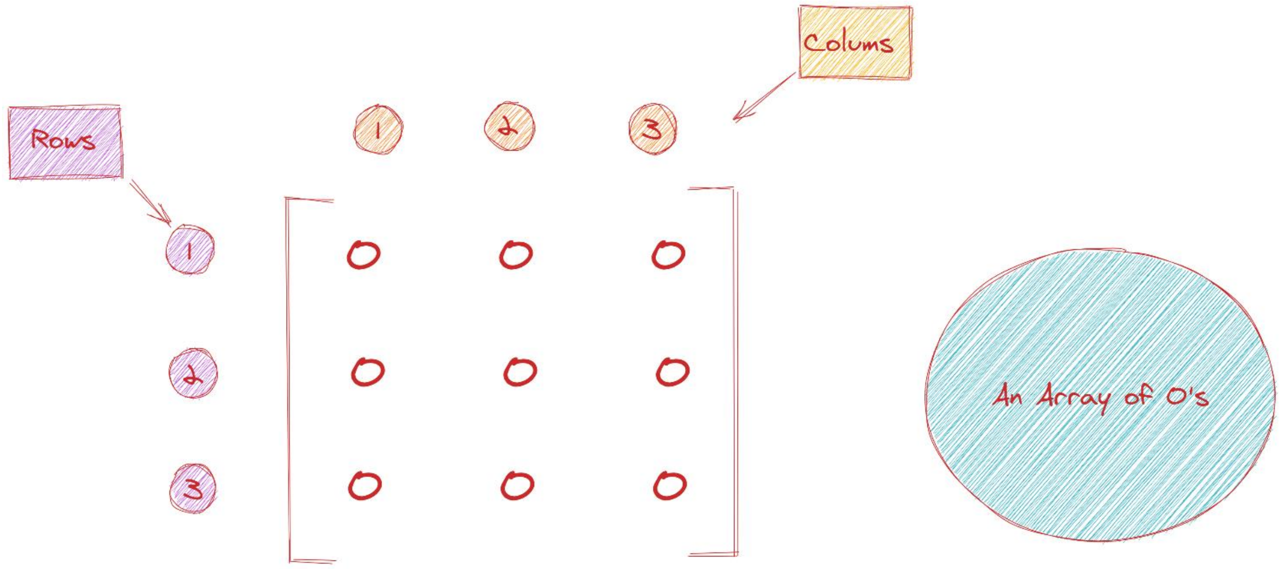
```javascript
// iterative
function reverseList(head) {
  if (head.length < 2) {
    return;
  }
  let newHead = null;
  while (head != null) {
    let nextNode = head.next;
    head.next = newHead;
    newHead = head;
    head = nextNode;
  }
  return newHead;
}

// recursive
function reverseList(head) {
  if (!head || !head.next) {
    return head;
  }

  let rest = reverseList(head.next);

  head.next.next = head;
  delete head.next;
  return rest;
}
```

Matrix Operations

Question

Here's a fun one: let's say we have a 2D array matrix that is a size of `m` rows and `n` columns. It is initially prefilled with `0`s and looks like the following:

JAVASCRIPT
```
[[0, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]]
```

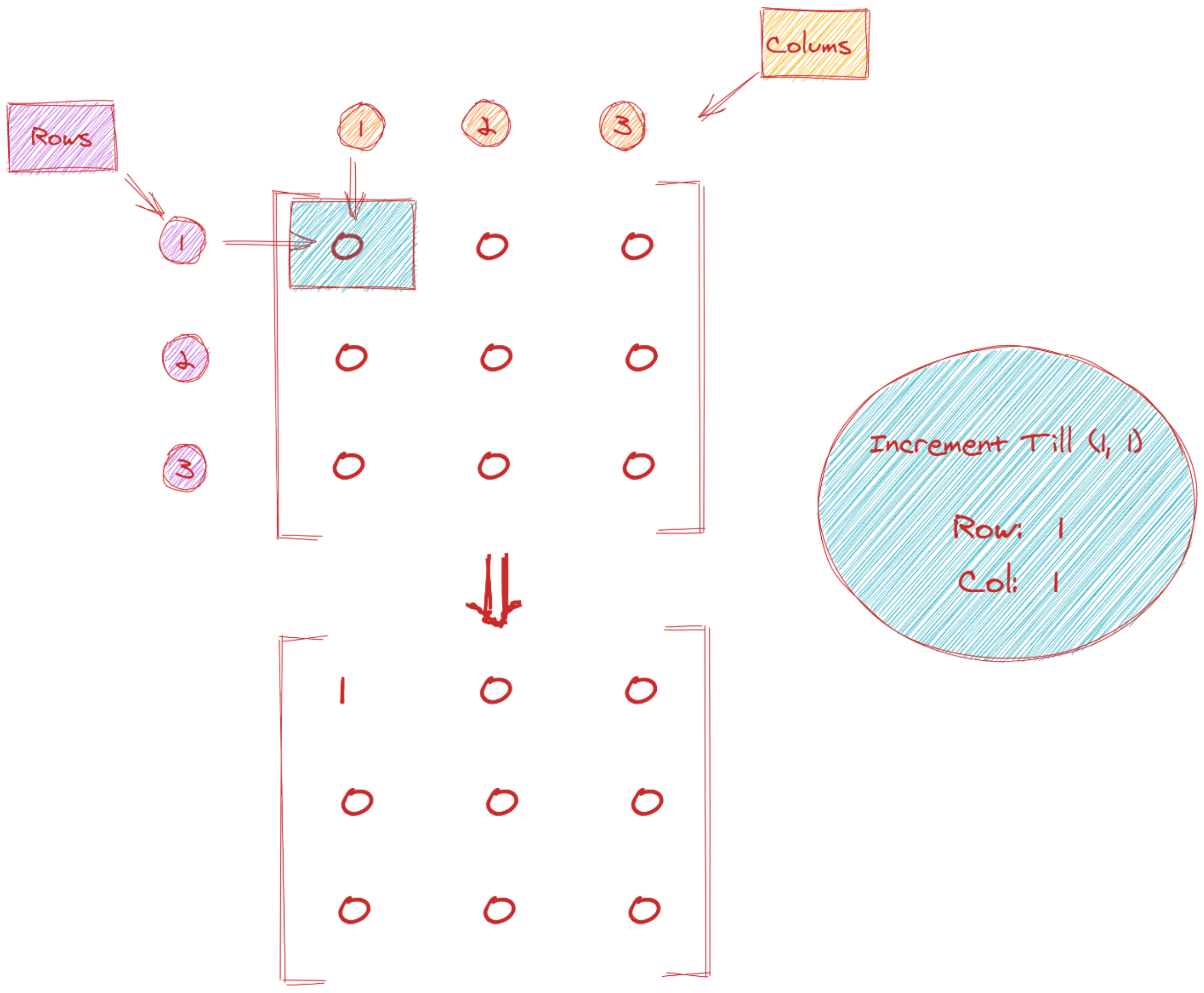


We are then given a list of increment operations to be performed on the nested matrix array. An increment operation is exactly as it sounds-- it will dictate when to add `1` to a number in the matrix.

Each increment operation will consist of two numbers, a `row` and `column` value, that dictate to what extent the array's cells under that range should increment by 1.

For example, given the above matrix, if we get `[1, 1]` as an operation, it results in:

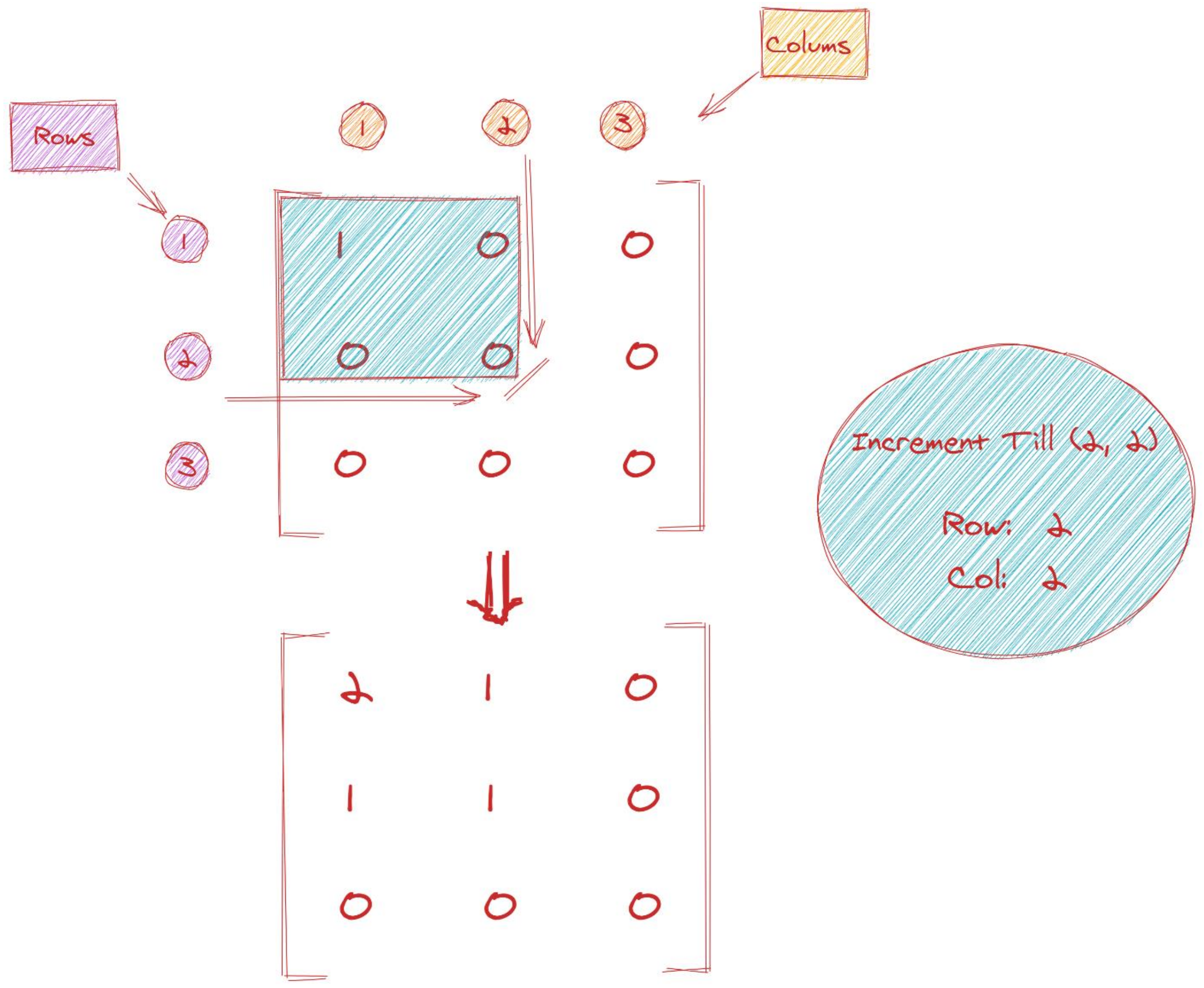
JAVASCRIPT
```
[[1, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]]
```



This is because we look at `matrix[1][1]`, and look northwest to see what the "range" is. So what we've done is incremented all cells from `matrix[0][0]` to `matrix[row][col]` where `0 <= row < m`, and `0 <= column < n`.

If we then got another operation in the form of `[2, 2]`, we'd get:

JAVASCRIPT
```javascript
[[2, 1, 0, 0],
 [1, 1, 0, 0],
 [0, 0, 0, 0]]
```



Can you write a method that returns the frequency of the max integer in the matrix? In the above example, it would be `1` (since `2` shows up just once).

So let's analyze the example given. Starting from a blank state of all `0`s, if we are given the operations `[1, 1]` and then `[2, 2]`, we obtain:

JAVASCRIPT
```javascript
[[2, 1, 0, 0],
 [1, 1,  0, 0],
 [0, 0, 0, 0]]
```

The largest number is `2`, and it's in the upper left. It got there because `matrix[0][0]` was incremented by `1` in both the operations.

Let's take a step back: so how do numbers in the matrix increment? It's via the overlap of the operations! What do we mean by this? The first operation gave us:

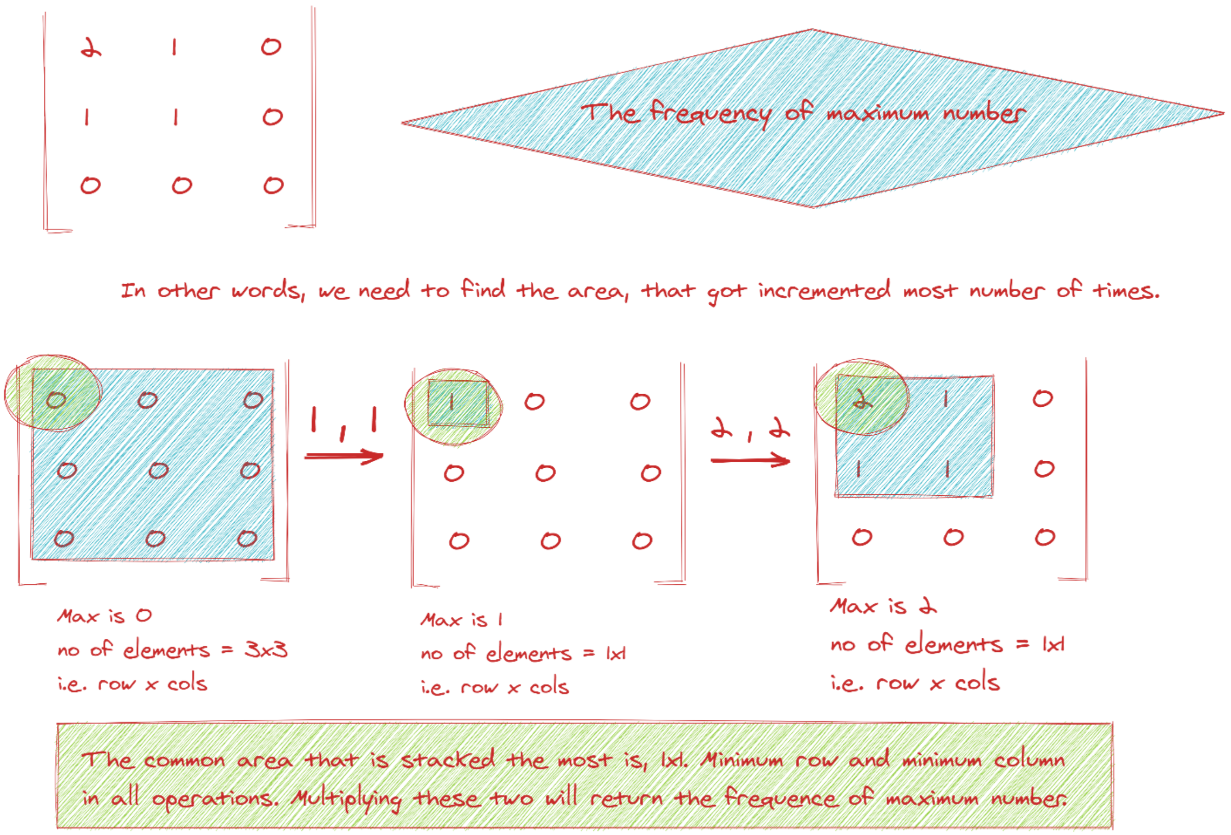
JAVASCRIPT
```
[[1, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]]
```

`matrix[0][0]` is now a `1`. It gets bumped up again since it falls under the `[2, 2]` range. We can imagine the operations as layering over each other.

There is an associated insight from this one: the cells that are layered over the most are the ones with the greatest numbers.

So really the cells we're looking for are the ones that got layered over the most.

The easiest way to do is by looking at the minimum rows and columns of each operation, since those are the edges of coverage. The minimum guarantees that the greatest increments will at at least reach those boundaries.


2 1 0
1 1 0
0 0 0
The frequency of maximum number
In other words, we need to find the area, that got incremented most number of times.
0 0 0
0 0 0
0 0 0
1, 1
1 0 0
0 0 0
0 0 0
2, 2
2 1 0
1 1 0
0 0 0
Max is 0
no of elements = 3x3
i.e. row x cols
Max is 1
no of elements = 1x1
i.e. row x cols
Max is 2
no of elements = 1x1
i.e. row x cols
The common area that is stacked the most is, 1x1. Minimum row and minimum column in all operations. Multiplying these two will return the frequence of maximum number.

Time and space complexity is `O(n)`.

Final Solution

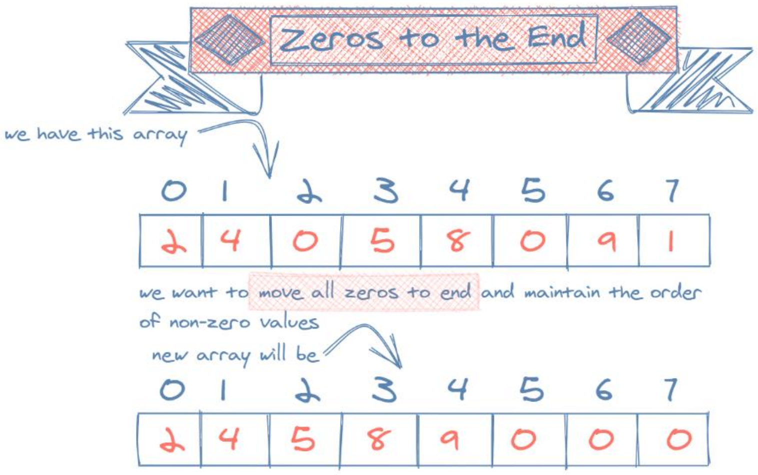
JAVASCRIPT
```
function maxFromOps(m, n, operations) {
  var minCol = m;
  var minRow = n;

  for (let op of operations) {
    minCol = Math.min(minCol, op[0]);
    minRow = Math.min(minRow, op[1]);
  }
  return minCol * minRow;
}
```

Zeros to the End

Question

Write a method that moves all zeros in an array to its end. You should maintain the order of all other elements. Here's an example:

JAVASCRIPT
```javascript
zerosToEnd([1, 0, 2, 0, 4, 0])
// [1, 2, 4, 0, 0, 0]
```



Here's another one:

JAVASCRIPT
```javascript
zerosToEnd([1, 0, 2, 0, 4, 0])
// [1, 2, 4, 0, 0, 0]
```

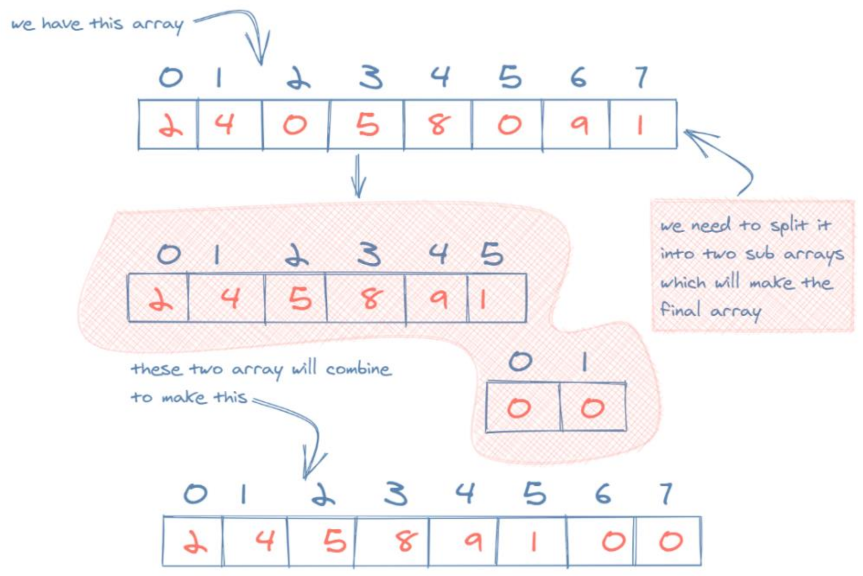
Fill in the following function signature:

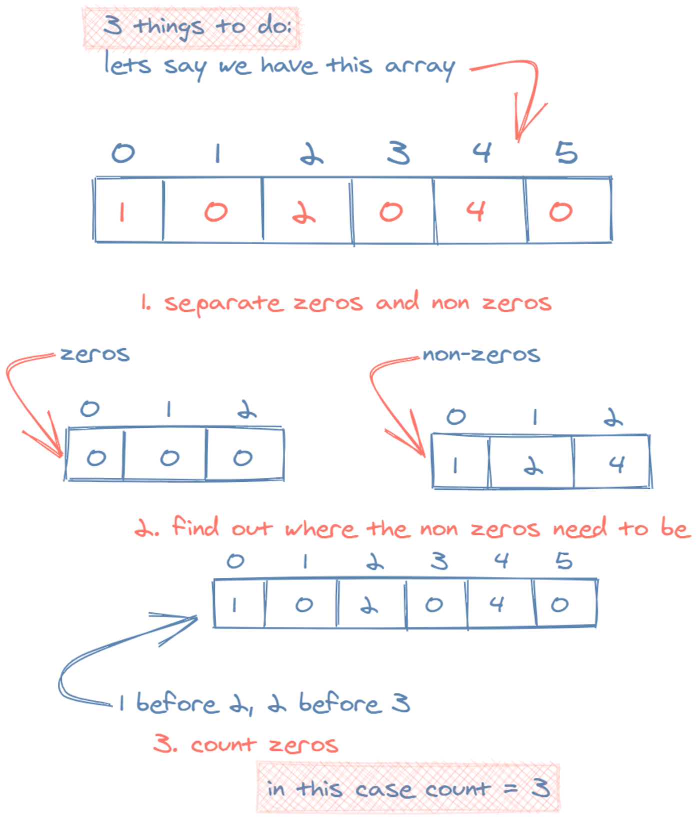
JAVASCRIPT
```javascript
function zerosToEnd(nums) {
  return;
};
```

Can you do this without instantiating a new array?

Always start by running through some examples to get a feel for the problem. With `[1, 0, 2, 0, 4, 0]`, let's walk through the steps-- in this case, it may be best to work backwards.

We want to end up with `[1, 2, 4, 0, 0, 0]`. To do that, it seems like we need to separate out `[1, 2, 4]` and `[0, 0, 0]`, so there's 3 things to consider.


we have this array
0 1 2 3 4 5 6 7
2 4 0 5 8 0 9 1
we need to split it into two sub arrays which will make the final array
0 1 2 3 4 5
2 4 5 8 9 1
0 1
0 0
these two array will combine to make this
0 1 2 3 4 5 6 7
2 4 5 8 9 1 0 0

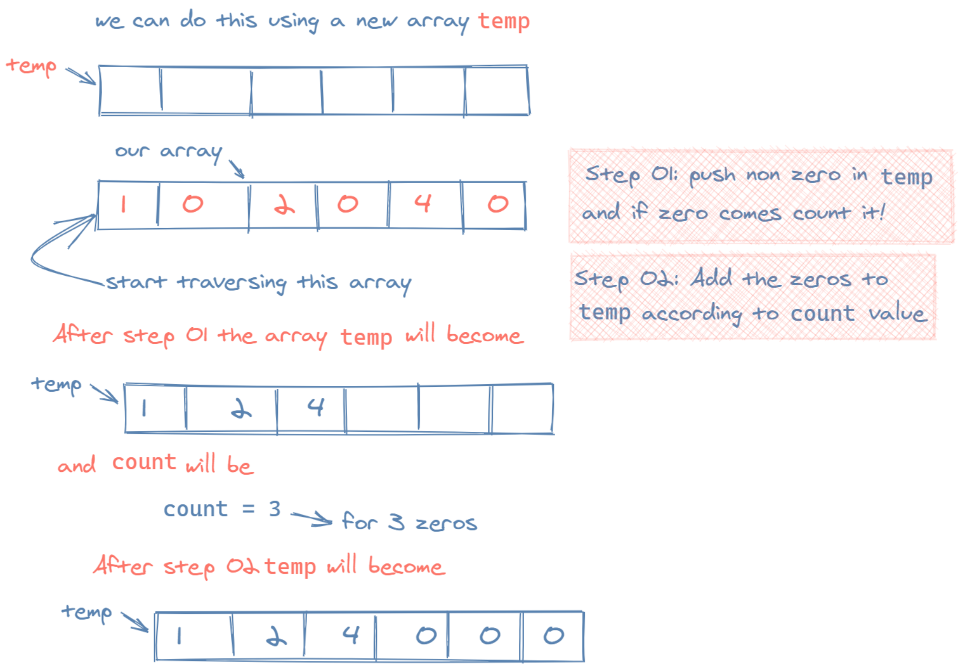


We need to do these 3 things:

Separate the `0`s and non-`0`s.

Find out where the non-`0`s need to be index-wise

Keeping track of the number of `0`s



we can do this using a new array temp
temp
our array
1 0 2 0 4 0
start traversing this array
Step 01: push non zero in temp and if zero comes count it!
Step 02: Add the zeros to temp according to count value
After step 01 the array temp will become
temp
1 2 4
and count will be
count = 3 for 3 zeros
After step 02 temp will become
temp
1 2 4 0 0 0

This could be our pseudocode.

SNIPPET

```
temp = []
zero_count = 0
iterate through array:
  if nonzero, push to new temp
  if zero, increment count
for zero_count times:
  push to temp
return temp
```

The above snippet would get the job done, but requires extra space with the new `temp` array.

Without a `temp` array, we'll need to change the position of elements in the array in-place. This requires us to keep track of indexes.

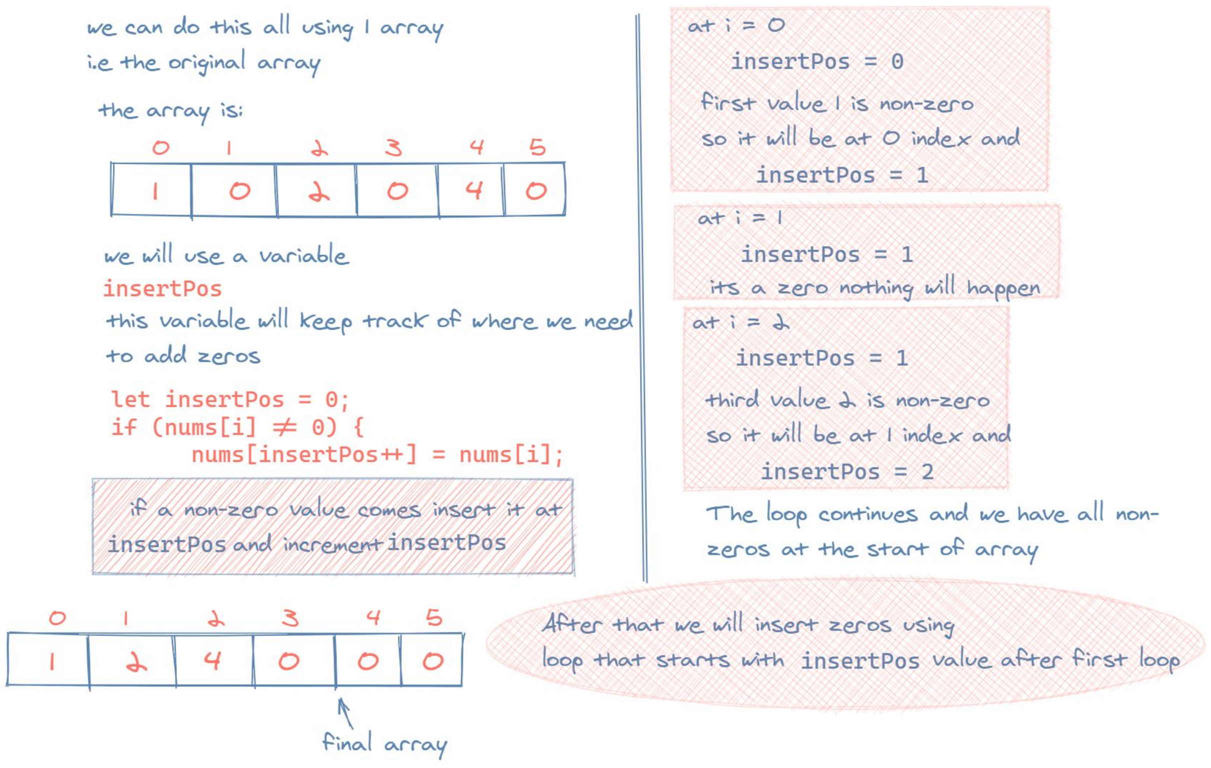
JAVASCRIPT

```javascript
const arr = [1, 0, 2, 0, 4, 0];
const nonZeros = [1, 2, 4]
```

Perhaps we can try to simplify-- what if we just kept track of one index?

Because we don't need or care about what is in the array after non-zero elements, we can simply keep a separate pointer of where to start the zeros.

Despite having two loops, the time complexity simplifies to `O(n)`. However, the space complexity is constant since we're using the same array.



we can do this all using 1 array
i.e the original array
the array is:
0 1 2 3 4 5
1 0 2 0 4 0
we will use a variable
insertPos
this variable will keep track of where we need to add zeros
let insertPos = 0;
if (nums[i] ≠ 0) {
nums[insertPos++] = nums[i];
if a non-zero value comes insert it at insertPos and increment insertPos
at i = 0
insertPos = 0
first value 1 is non-zero
so it will be at 0 index and
insertPos = 1
at i = 1
insertPos = 1
its a zero nothing will happen
at i = 2
insertPos = 1
third value 2 is non-zero
so it will be at 1 index and
insertPos = 2
The loop continues and we have all non-zeros at the start of array
0 1 2 3 4 5
1 2 4 0 0 0
final array
After that we will insert zeros using loop that starts with insertPos value after first loop

Final Solution

JAVASCRIPT
```javascript
function zerosToEnd(nums) {
  let insertPos = 0;
  for (let i = 0; i < nums.length; i++) {
    if (nums[i] != 0) {
      nums[insertPos++] = nums[i];
    }
  }

  for (let j = insertPos; j < nums.length; j++) {
    nums[j] = 0;
  }

  return nums;
}

console.log(zerosToEnd([1, 0, 2, 0, 4, 0]));
```

Sum of Perfect Squares

Question

A perfect square is a number made by squaring a whole number.

Some examples include `1`, `4`, `9`, or `16`, and so on -- because they are the squared results of `1`, `2`, `3`, `4`, etc. For example:

SNIPPET
```
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
...
```

However, `15` is not a perfect square, because the square root of `15` is not a whole or natural number.

Perfect Square	Factors
1	1 * 1
4	2 * 2
9	3 * 3
16	4 * 4
25	5 * 5
36	6 * 6
49	7 * 7
64	8 * 8
81	9 * 9
100	10 * 10

Given some positive integer *n*, write a method to return the fewest number of perfect square numbers which sum to *n*.

The follow examples should clarify further:

JAVASCRIPT
```
var n = 28
howManySquares(n);
// 4
// 16 + 4 + 4 + 4 = 28, so 4 numbers are required
```

On the other hand:

JAVASCRIPT
```
var n = 16
howManySquares(n);
// 1
// 16 itself is a perfect square
// so only 1 perfect square is required
```

True or False?

The perfect squares are the squares of the whole numbers: `1, 4, 9, 16, 25, 36, 49, 64, 81, 100`

Solution: True

We need to find the least number of perfect squares that sum up to a given number. With problems like this, the brute-force method seems to lean towards iterating through all the possible numbers and performing some operation at each step.

It seems from the get-go that we'll need to find all the perfect squares available to us before we reach the specified number in an iteration.

This means if the number is `30`, we'll start at `1`, then `2`, and conduct a logical step at each number. But what logic do we need? Well, if we're given a number-- say, `100`, how would we manually solve this problem and find the perfect squares that sum up to `100`?

Like we said, we'd probably just start with `1` and get its perfect square. Then we'll move on to `2`, and get the perfect square of `4`. But in this case, we'll keep summing as we calculate each square. Mathematically speaking, it would look like:

SNIPPET
```
// while (1*1) + (2*2) + (3*3) + (4*4) + ... <= 100
```

Fill In

What line below would give us the proper count?

TEXT/X-JAVA

```
class Main {
    public static int howManySquares(int num) {
        int count = 0;

        for (int i = 1; i <= num; i++)

            // Is current number 'i' a perfect square?
            for (int j = 1; j * j <= i; j++)
                if (j * j == i)
                    _______________________

        return count;
    }

    public static void main(String args[]) {
        this.howManySquares(4);
    }
}
```

Solution: Count++

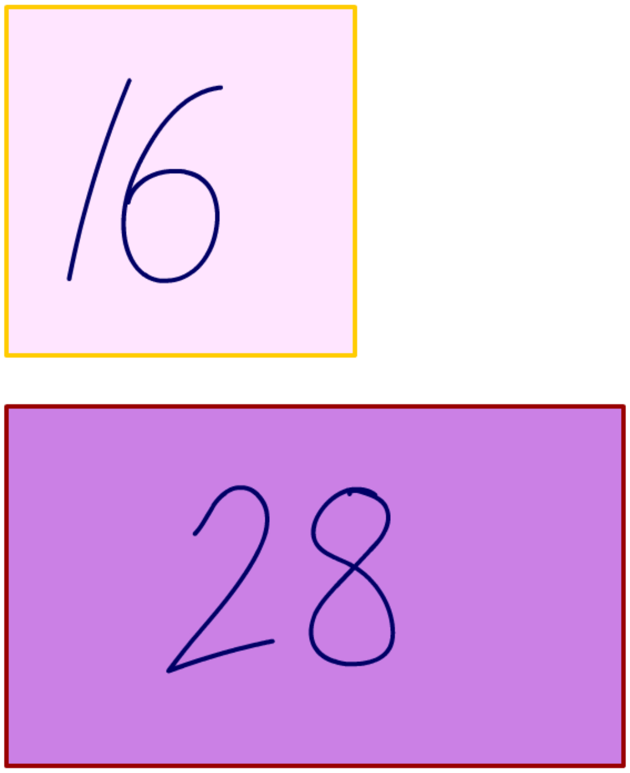
Our brute force solution isn't very optimal, so let's try to make it more efficient. At the very least, notice how we can limit our number of perfect squares to just the ones that are less than our number, since larger ones (in this case, greater than `100`) wouldn't make sense. That reduces the time necessary to a degree.

So if we were looking for all the perfect squares that sum up to `12`, we wouldn't want to consider `16` or `25`-- they're too large for our needs.

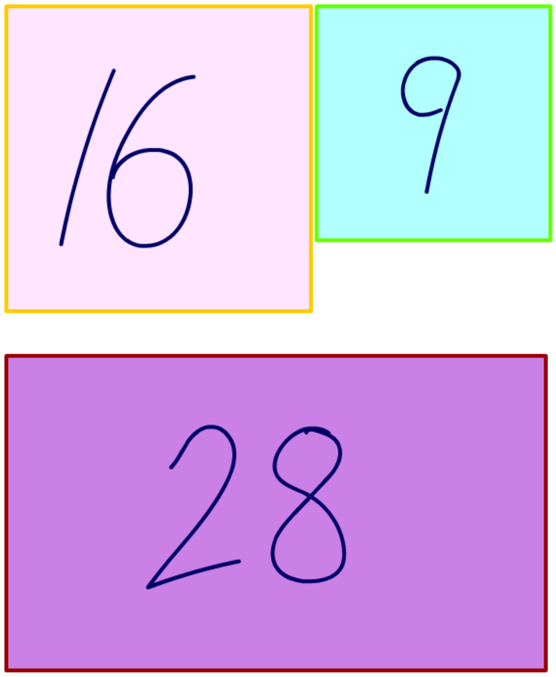
To translate this into code, given our previous example, we'd essentially iterate through all numbers smaller than or equal to `100`. At each iteration, we'd see if we can find a perfect square that is larger to "fit in" to the "leftover space".

Picture these steps. Start with `28`:

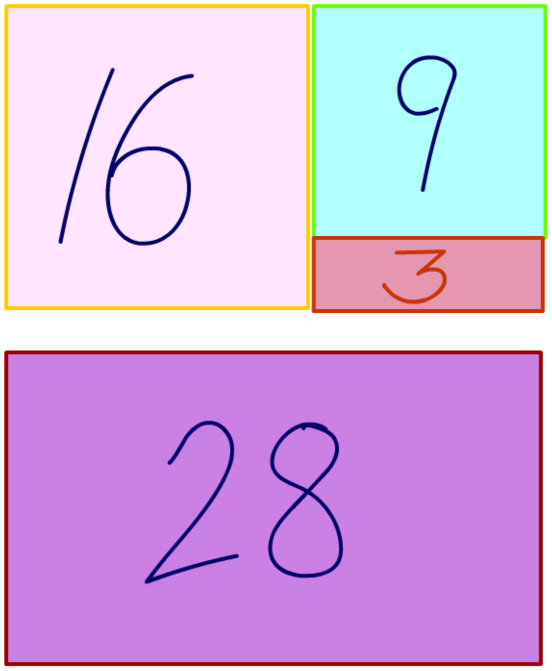
Try 16:



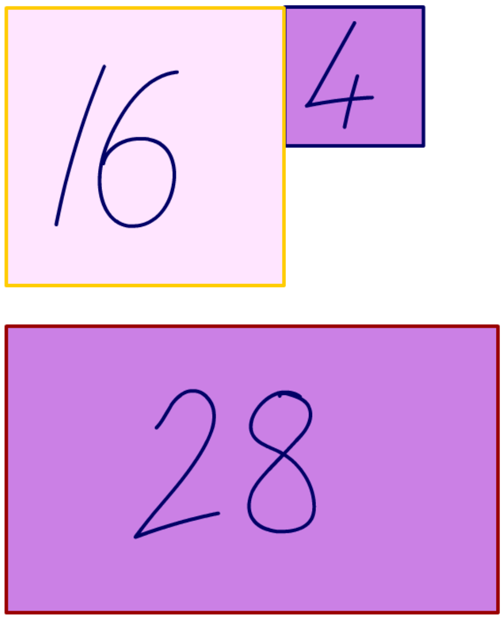
Cool, it fits-- let's keep 16. Try 9:



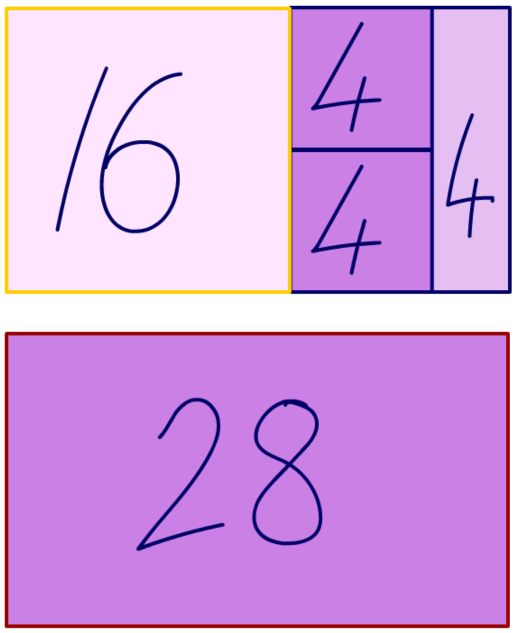
That also fits. What works with 16 and 9?



That won't work-- we can only fit `3` but it's not a perfect square. What else fits with `16`?



OK, `4` works, and is a perfect square. Notice that it not only matches both conditions (fits and is a perfect square), but we can fit 3 of them in!



Given that approach, here's the logic translated to code. Read the comments carefully for clarification:

JAVASCRIPT

```javascript
function howManySquares(num) {
    if (num <= 3) {
        return num;
    }

    result = num;

    for (let i = 1; i < num + 1; i++) {
    // get squared number
        temp = i * i;
        if (temp > num) {
            break;
        } else {
      // get whatever is lower:
      // 1 - the current minimum, or
      // 2 - the minimum after considering the next perfect square
            result = Math.min(result, 1 + howManySquares(num - temp));
        }
    }

    return result;
}

console.log(howManySquares(16));
```

The above, however, has an exponential time complexity-- that is, `O(2^n)`. Additionally, we don't want to find all the possible perfect squares each time, that wouldn't be very time efficient.

How can we turn this into a shortest path problem? Notice the branching, too-- how might we construct a `graph` to help us work backwards from a sum to perfect squares? Additionally, there's the opportunity for reuse of calculations.

Well, if we used a `data structure` for storing the calculations-- let's say a `hash`, we can memo-ize the calculations. Let's imagine each key in the `hash` representing a number, and its corresponding value as being the minimum number of squares to arrive at it.

Alternatively, we can use an `array`, and simply store the sequence of perfect squares for easy lookup.

Final Solution

JAVASCRIPT

```javascript
function howManySquares(n) {
  let perfectSqNumsLength = 1;
  while (perfectSqNumsLength * perfectSqNumsLength < n) {
    perfectSqNumsLength++;
  }

  if (perfectSqNumsLength * perfectSqNumsLength > n) {
    perfectSqNumsLength--;
  }

  const perfectSqNums = [];

  // Fill the array backwards so we get the numbers to work with
  for (let i = perfectSqNumsLength - 1; i >= 0; i--) {
    perfectSqNums[perfectSqNumsLength - i - 1] = (i + 1) * (i + 1);
  }

  // instantiate a hashmap of possible paths
  const paths = {};
  paths[1] = 1; // 1 = 1
  paths[0] = 0; // 0 means you need 0 numbers to get 0

  return numSquares(paths, perfectSqNums, n);
}

function numSquares(paths, perfectSqNums, n) {
  if (paths.hasOwnProperty(n)) {
    // we already knew the paths to add up to n.
    return paths[n];
  }

  let min = Number.MAX_SAFE_INTEGER;
  let thisPath = 0;

  for (let i = 0; i < perfectSqNums.length; i++) {
    if (n - perfectSqNums[i] >= 0) {
      const difference = n - perfectSqNums[i];
      // this is key - recursively solve for the next perfect square
      // that could sum to n by traversing a graph of possible perfect square
sums
      thisPath = numSquares(paths, perfectSqNums, difference);

      // compare the number of nodes required in this path
      // to the current minimum
      min = Math.min(min, thisPath);
    }
  }

  min++; // increment the number of nodes seen
  paths[n] = min; // set the difference for this number to be the min so far

  return min;
}
```

Longest Palindromic Substring

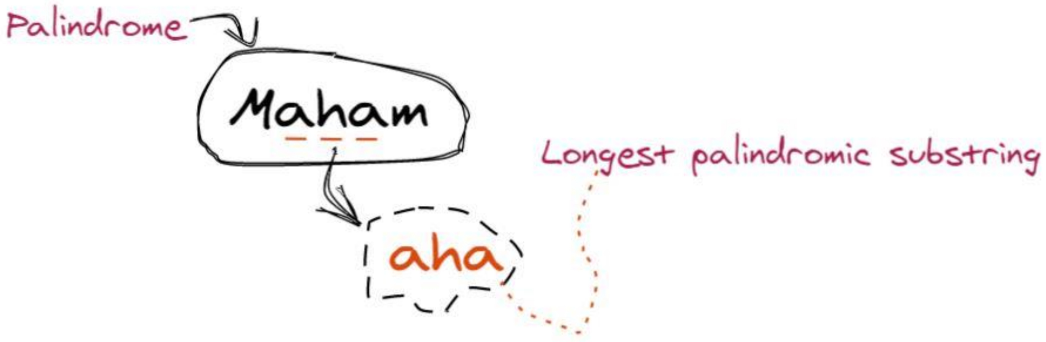
Question

We're given a string that's a mixture of several alphabetical characters.

JAVASCRIPT
```javascript
const str = "algototheehtotdaily";
```

Could you write a method to find the longest substring that is considered a palindrome? In the above example, it would be `"totheehtot"`.

Where there are multiple longest palindromic substrings of the same length, for example in `"abracadabra"` (`"aca"` and `"ada"`), return the first to appear.



Longest Palindromic Substring

I have a sister named `"Maham"`. For my cousins and I, her name was fascinating, as it is the same spelling if we read it backwards and forwards.

Later on, as I grew up, I came to know that this interest is a problem in the field of Computer Science. Yes, you guessed it right, I'm talking about the study of `palindromes`.

In Computer Science, the palindrome problem has always been tricky. We have to find a solution for a word that reads the same backward as read forward keeping the solution optimized. Don’t worry ,once you’ll get the concept, it will become easy for you to deal with any problem related to palindromes.

We will take you on a journey where you will learn about the `longest palindromic substring`. A palindromic substring is a substring which is a palindrome. Let’s say that we have a string “"Maham"”-- the longest palindromic substring would be “"aha"”. For the function signature, we will pass a simple string as a parameter. Our program should give us an output that will display the longest palindromic substring of the input string.

Let’s dig deeper into it.

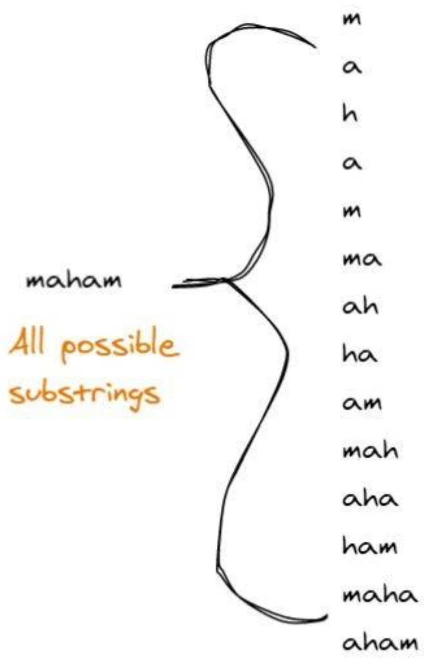
True or False?

The second half of a palindromic word is the same as the first, but backwards.

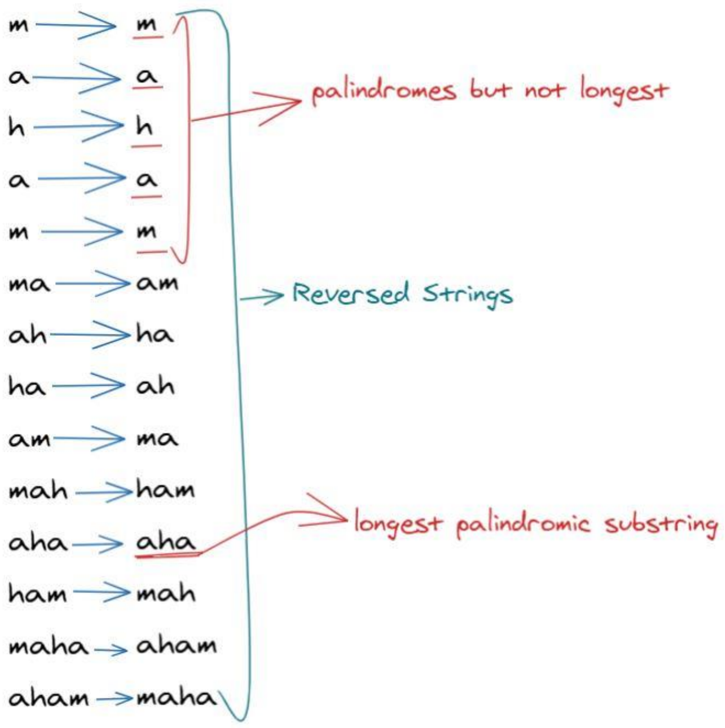
Solution: True

Coming to the main problem, let's find a brute force method to find the longest palindromic substring.

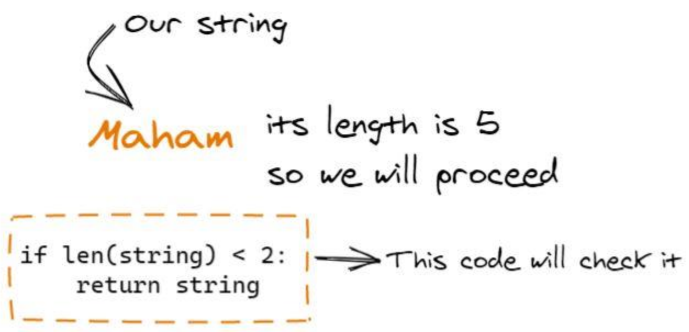
For the brute force method, the logic is that you'll find all substrings. Let's say we have the string `"Maham"` again. You'd try to generate all substrings in it.



After obtaining all the substrings, what we'll do is reverse each substring, and check whether it's palindromic or not. This way we'll naturally keep the longest palindromic one.

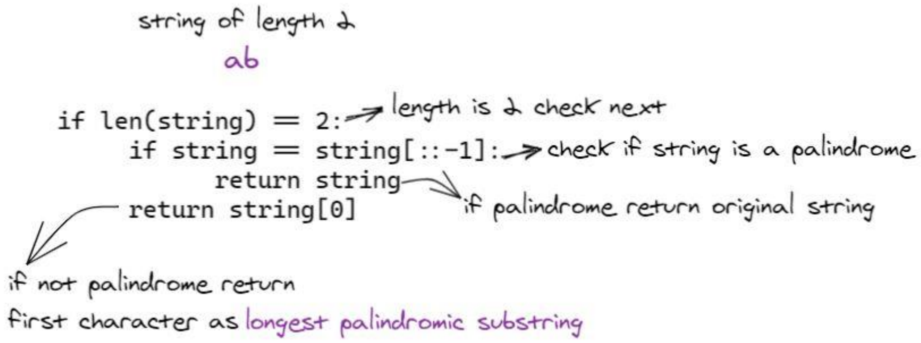


Let's see how we will implement it through code. At first, we will check if our string is of length `1` or not. If it has a length of `1`, then we don't have to proceed, as you know that a single character is a palindrome by itself.



```python
if len(string) < 2:
    return string
```

Then we can check if our string is of length `2` or not. We are checking these specific lengths to avoid extra work, as our function doesn't have to execute all the awy. Adding these checks will improve the method's performance.


string of length 2
ab
length is 2 check next
check if string is a palindrome
if palindrome return original string
if not palindrome return
first character as longest palindromic substring

```python
if len(string) == 2:
    if string == string[::-1]:
        return string
    return string[0]
```

What will happen if we pass the string `"Maham"`?

Multiple Choice

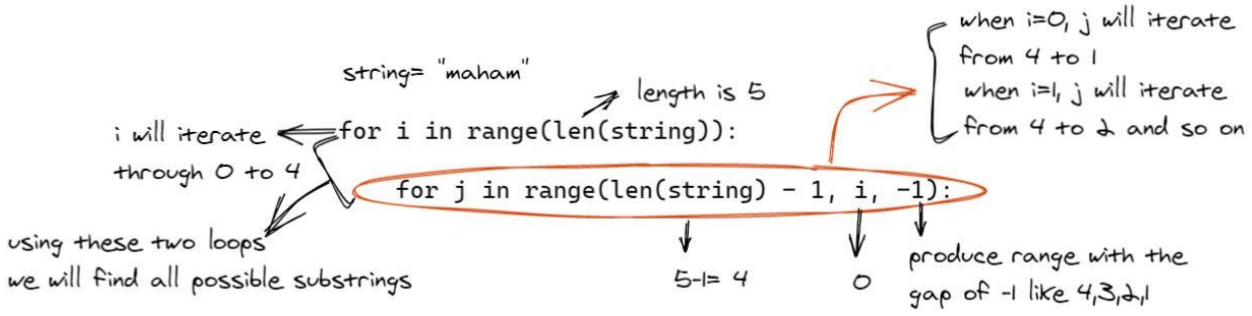
Why will the interpreter skip the previous code snippet for the input string `"Maham"`?

- The length of the input string is less than 2
- Our code is not correct
- The length of the input string is not equal to 2

Solution: The length of the input string is not equal to 2

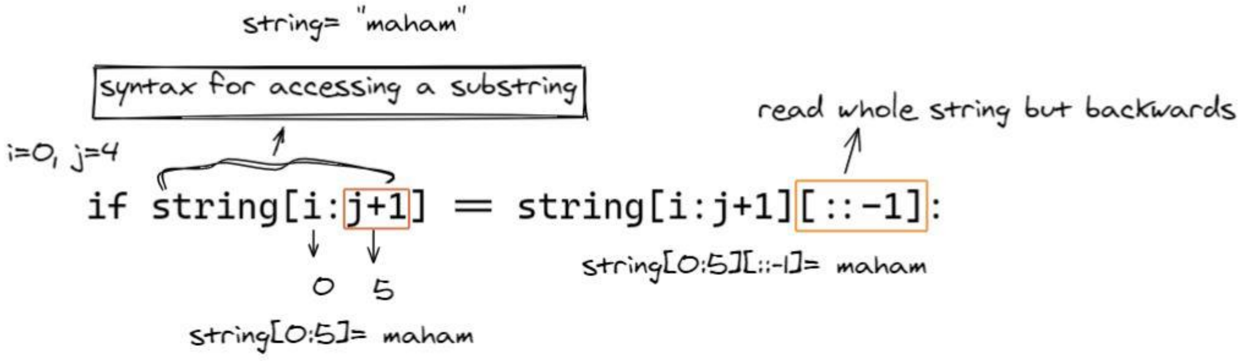
Let's move on as we haven't completed our solution yet. Now the problem is that the input string is neither less than `2` nor equal to `2`. The only possibility left is that it will be greater than `2`. So the question arises, how are we going to deal with such strings?

In this case, we will find all possible substrings with the help of loops. Here `i` is the starting point of a substring and `j` is the ending point.

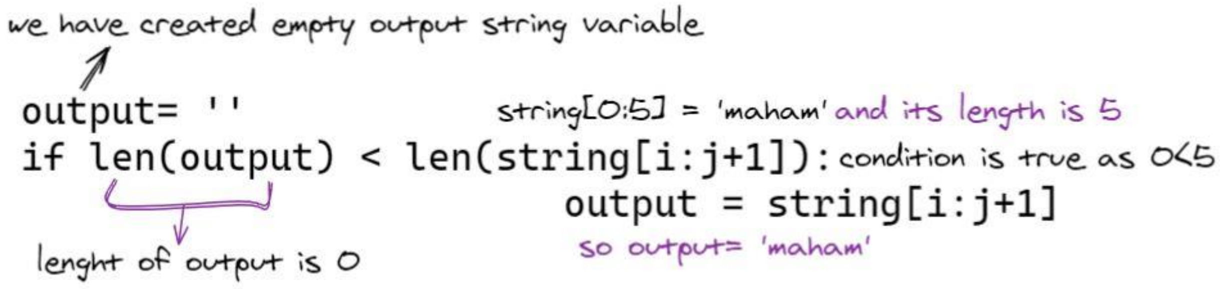

string= "maham"
length is 5
i will iterate through 0 to 4
for i in range(len(string)):
for j in range(len(string) - 1, i, -1):
when i=0, j will iterate from 4 to 1
when i=1, j will iterate from 4 to 2 and so on
using these two loops we will find all possible substrings
5-1= 4
0
produce range with the gap of -1 like 4,3,2,1

The loops will form the substrings with indices `(0, 4)`, `(0, 3)`, `(0, 2)`, and so on.

The next step is to check if our substring is a palindrome or not. We will check it using the following line of code.

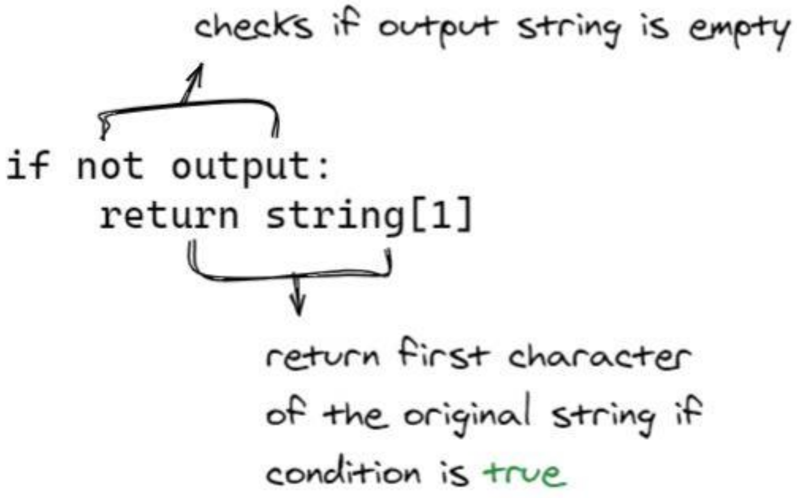

string= "maham"
syntax for accessing a substring
read whole string but backwards
i=0, j=4
if string[i:j+1] == string[i:j+1][::-1]:
0
5
string[0:5]= maham
string[0:5][::-1]= maham

In the following step, if the substring is a palindrome, we will check if it is the longest palindromic substring. For that purpose, we can create an `output` variable that will store the longest palindromic substring found. Let's then compare our substring with the output string. If it is longer than the output string, we'll update it.


we have created empty output string variable
output= ''
if len(output) < len(string[i:j+1]):
output = string[i:j+1]
string[0:5] = 'maham' and its length is 5
condition is true as 0<5
lenght of output is 0
so output= 'maham'

In this way, our loop will iterate through the length of the string and we will find the longest palindromic substring.

If no palindromic substring is found, then `output` will remain empty and the function will return the first character of the string.


checks if output string is empty
if not output:
return string[1]
return first character of the original string if condition is true

The time complexity for this function will be `O(n*3)`, as the complexity to find all possible strings is `O(n*2)` and the one to check if it's a palindrome is `O(n)`. This results in `O(n2*n)` = `O(n3)`.

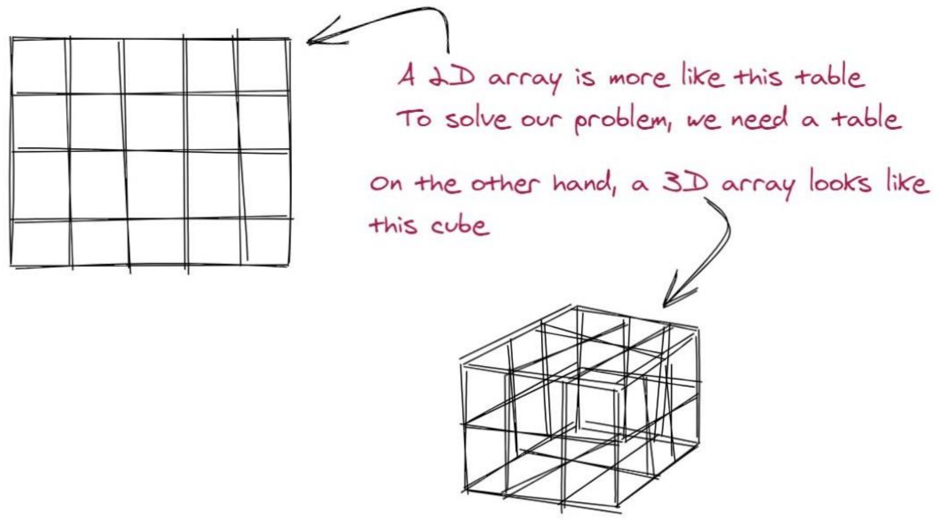
Is there a better, more efficient method to solve this problem?

Of course! There are many solutions to a problem if we'll look hard enough for them. We will share another solution with you that will use the dynamic programming technique to find the longest palindromic substring.

PYTHON
```python
def longestPalindromicString(string: str) -> str:
    if len(string) < 2:
        return string

    if len(string) == 2:
        if string == string[::-1]:
            return string
        return string[0]

    output = ""
    for i in range(len(string)):
        for j in range(len(string) - 1, i, -1):
            if string[i : j + 1] == string[i : j + 1][::-1]:
                if len(output) < len(string[i : j + 1]):
                    output = string[i : j + 1]

    if not output:
        return string[1]

    return output
```

In this method, we will create an indexed table (2D array) having `n` rows and `m` columns where `n` and `m` are equal to the length of the string. You may wonder why we're using such a table.

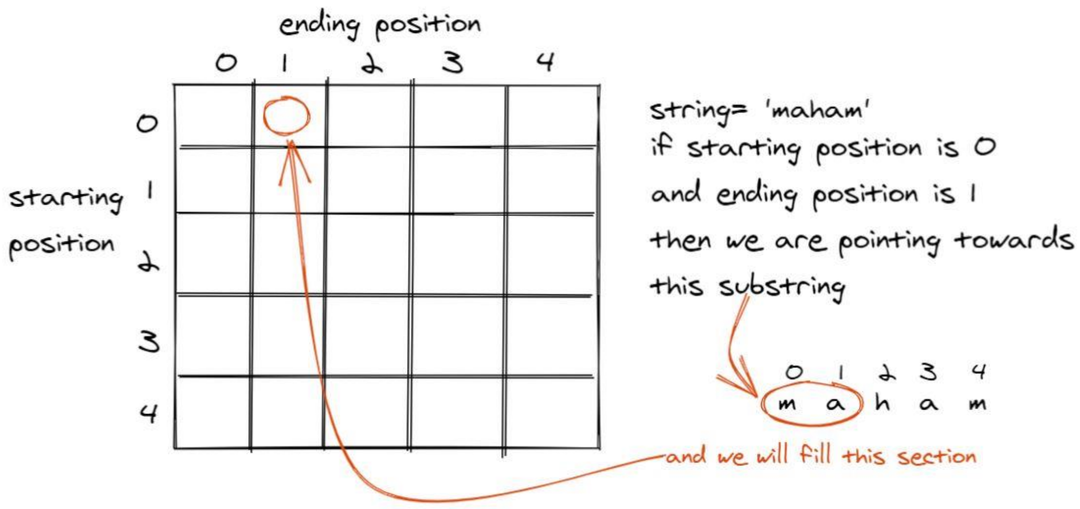
Well, **dynamic programming** is about using solving sub-problems of a larger problems, and then using those results to avoid any repeat work. The shape of the two-dimensional array allows us to more intuitively hold solutions of subproblems. This will make more sense as we continue-- the table helps us easily model our findings thus far.


A 2D array is more like this table
To solve our problem, we need a table
On the other hand, a 3D array looks like this cube

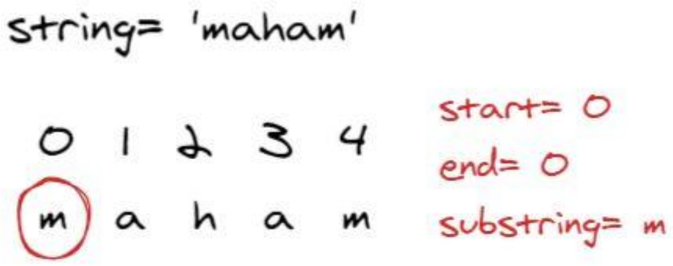
A sample 2-dimensional array syntax is as follows:

PYTHON
```
array = [[...], [...], ..., [...]]
```

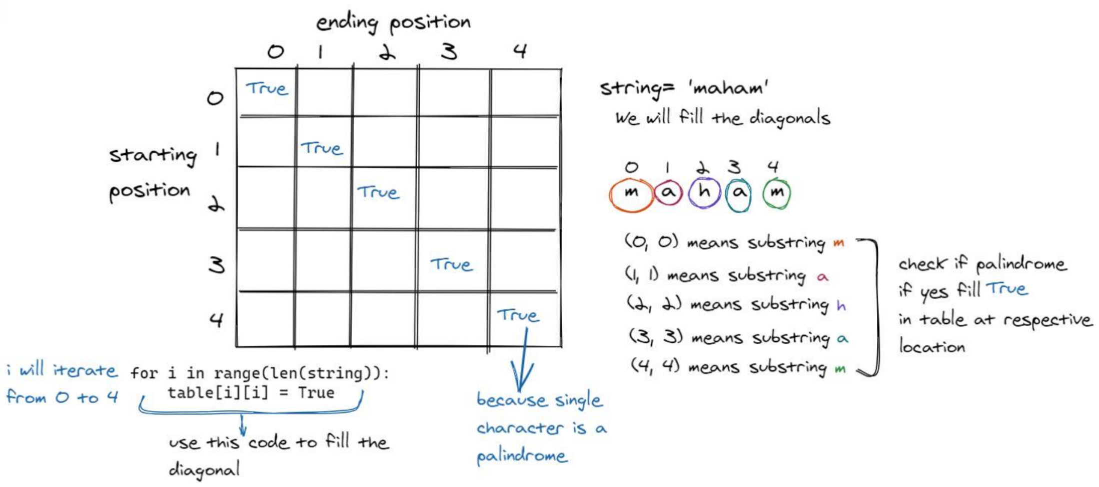
Let us revisit the string `"maham"`. We'll create a table having rows and columns equal to the length of the string. This is `5` in our case.



Now we'll show you how to fill this table so that we can find the longest palindromic substring. We will fill the diagonal first because the diagonal indicates a single element (for example, if the starting point is `0` and the ending point is also `0`, then what we're operating on is just the substring `m`). See the diagram below.



The logic is similar to our brute force solution-- we want to find the substring and check if it is a palindrome or not. If it is a palindrome, then fill `True` in the table at that position. With that said, let's fill up the diagonal.

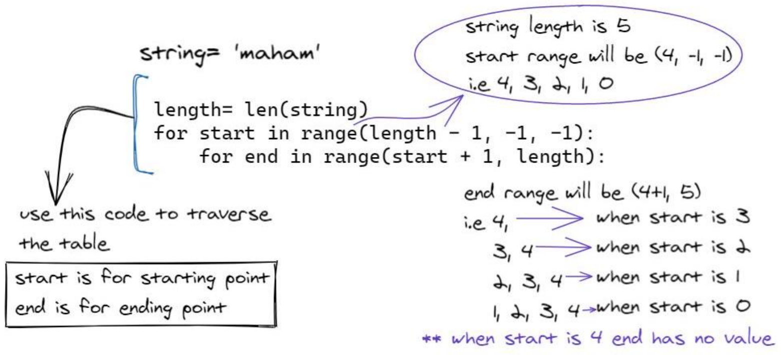

ending position
0 1 2 3 4
starting position
0 1 2 3 4
True
string= 'maham'
We will fill the diagonals
m a h a m
(0, 0) means substring m
(1, 1) means substring a
(2, 2) means substring h
(3, 3) means substring a
(4, 4) means substring m
check if palindrome if yes fill True in table at respective location
i will iterate from 0 to 4
for i in range(len(string)):
    table[i][i] = True
use this code to fill the diagonal
because single character is a palindrome

Now let's operate on all remaining substrings. How will we check if the substring is a palindrome? We will compare the element in the starting position and element in the ending position, and move "in-wards", saving a lot of effort by eliminating unnecessary work.

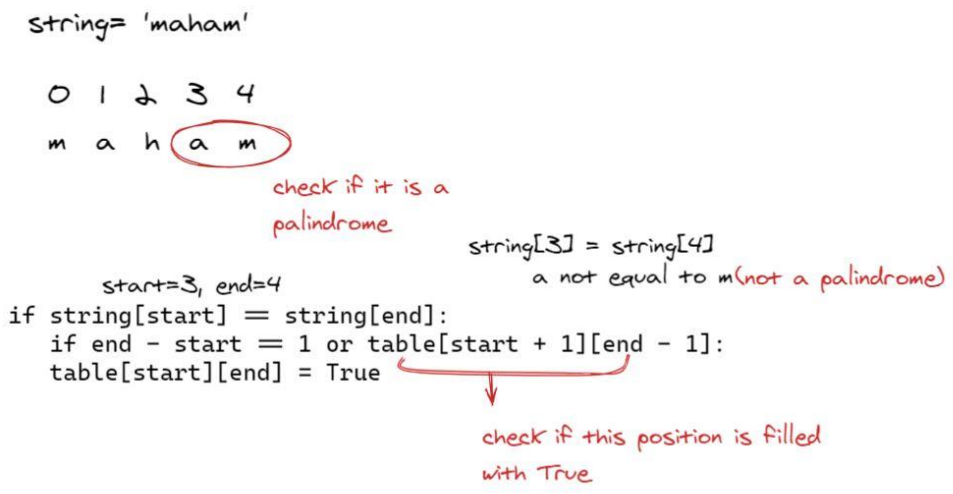
TEXT
```
for start = end, "a" is palindromic,
for start + 1 = end, "aa" is palindromic (if string[start] = string[end])
for start + 2 = end, "aba" is palindromic (if string[start] = string[end] and
"b" is palindromic)
for start + 3 = end, "abba" is palindromic (if string[start] = string[end] and
"bb" is palindromic)
```

The big idea is that we won't have to repeatedly check substrings that can't possibly be a palindrome. If both elements are the same, our substring is a palindrome, and we continue the check by moving towards the middle with two pointers. Otherwise, if it's not a palindrome, our "starting point" already verifies that.

But we'll need to build up this table to know. For traversing the table above the diagonal, we can use the following code:


```
length= len(string)
for start in range(length - 1, -1, -1):
    for end in range(start + 1, length):
```
string= 'maham'
string length is 5
start range will be (4, -1, -1)
i.e 4, 3, 2, 1, 0
use this code to traverse the table
start is for starting point
end is for ending point
end range will be (4+1, 5)
i.e 4, → when start is 3
3, 4 → when start is 2
2, 3, 4 → when start is 1
1, 2, 3, 4 → when start is 0
** when start is 4 end has no value

So, the above loops will traverse the area above the diagonal in a bottom-up manner (for example, `(3, 4)`, `(2, 3)`, `(2, 4)`.. and so on). Then we'll see if the substring is a palindrome or not. At the first iteration, the loop will check the `(3, 4)` substring (i.e start is 3 and the end is 4).


string= 'maham'
0 1 2 3 4
m a h a m
check if it is a palindrome
string[3] = string[4]
a not equal to m(not a palindrome)
start=3, end=4
check if this position is filled with True

```
if string[start] == string[end]:
   if end - start == 1 or table[start + 1][end - 1]:
   table[start][end] = True
```

In this case, there will be no value at `(3, 4)` because the substring is not a palindrome. We will keep checking the substrings and get the longest palindromic substring at the end.

The key to understanding the rest of the implementation is this pseudocode:

TEXT
```
for start + distance = end, str[start, end] will be palindromic
if str[start] == str[end]
and
str[start + 1, end - 1] (the "inner" part of the string) is palindromic
```

What we're doing is moving "inwards" at each step, so we do `start + 1` and `end - 1`. It also means we can use this state transition equation:

TEXT
```
state(start, end) is true if:
for start = end,
for start + 1 = end,  if str[start] == str[end]
for start + 2 <= end, if str[start] == str[end] && state(start + 1, end - 1) is
true
```

The time complexity of this program is `O(n*2)` as the time complexity for creating the table is `O(n*2)`. For traversing and filling the table it is `O(n*2)` as well, and thus reduces to an `O(n)` solution.

The `maxLen` variable keeps track of the longest palindromic substring and `output` has stored the longest palindromic string found so far. In the end, the function returns the `output` i.e the longest palindromic substring.

PYTHON

```
def longestPalindromicString(string: str) -> str:
    length = len(string)
    table = [[False] * length for _ in range(length)]
    output = ""

    for i in range(length):
        table[i][i] = True
        output = string[i]

    maxLen = 1
    for start in range(length - 1, -1, -1):
        for end in range(start + 1, length):
            if string[start] == string[end]:
                if end - start == 1 or table[start + 1][end - 1]:
                    table[start]  [end] = True
                    if maxLen < end - start + 1:
                        maxLen = end - start + 1
                        output = string[start : end + 1]
    return output

print(longestPalindromicString('algoog'))
```

Final Solution

JAVASCRIPT
```javascript
const longestPalindrome = (str) => {
  if (!str || str.length <= 1) {
    return str;
  }

  let longest = str.substring(0, 1);
  for (let i = 0; i < str.length; i++) {
    let temp = expand(str, i, i);
    if (temp.length > longest.length) {
      longest = temp;
    }
    temp = expand(str, i, i + 1);
    if (temp.length > longest.length) {
      longest = temp;
    }
  }
  return longest;
};

const expand = (str, begin, end) => {
  while (begin >= 0 && end <= str.length - 1 && str[begin] === str[end]) {
    begin--;
    end++;
  }
  return str.substring(begin + 1, end);
};
```