



Recursion



Recursion

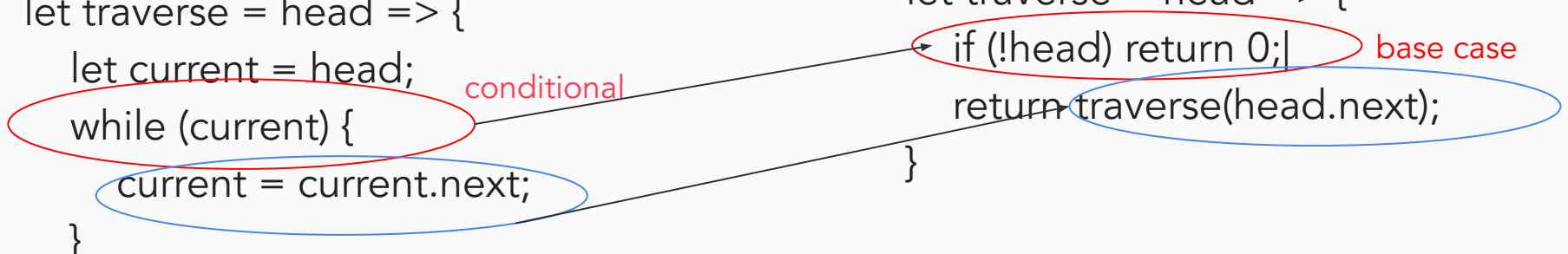
- Recursion is a vital concept that must be understood as a prerequisite for many algorithms that we will see moving forward.
- Linked List problems are a good place to practice recursion due to their simplicity compared to trees or graphs.
- Recursion inherently creates a stack, specifically the call stack, which works very similarly to a stack created by an array.
- All recursion solutions can also be written iteratively, but this can actually overly complicate some problems.



Comparing Recursive and Iterative Code

Iterative:

```
let traverse = head => {  
  let current = head;  
  while (current) {  
    current = current.next;  
  }  
  return 0;  
}
```



Recursive:

```
let traverse = head => {  
  if (!head) return 0; // base case  
  return traverse(head.next);  
}
```

The most important part of a recursive function is the BASE CASE. This is the termination point of the function. It is similar to conditionals put in while loops.



Single Path Demo

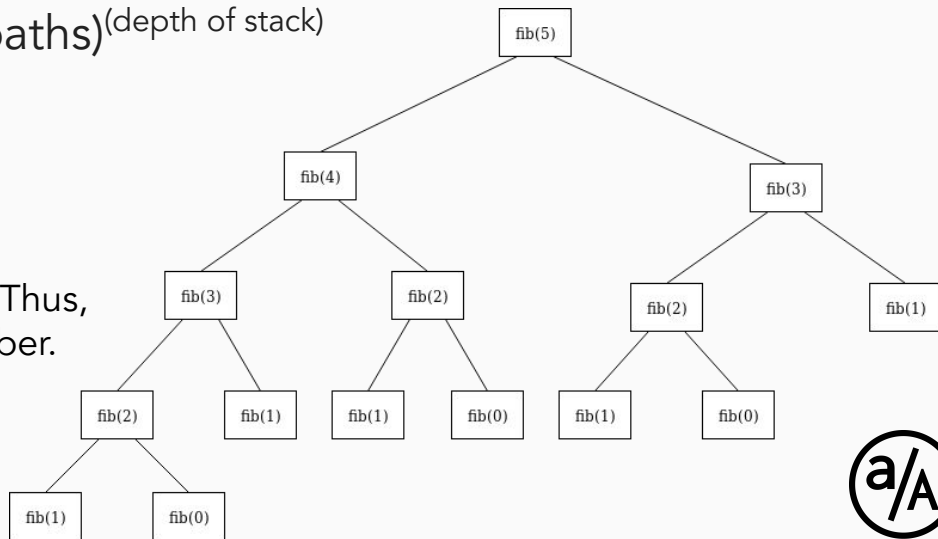
- [Reverse Linked List](#)

Runtime Complexity

- Single-path recursive functions usually run in $O(n)$ where n is the depth of the stack
- Multi-path recursive functions can be analyzed with the following formula:

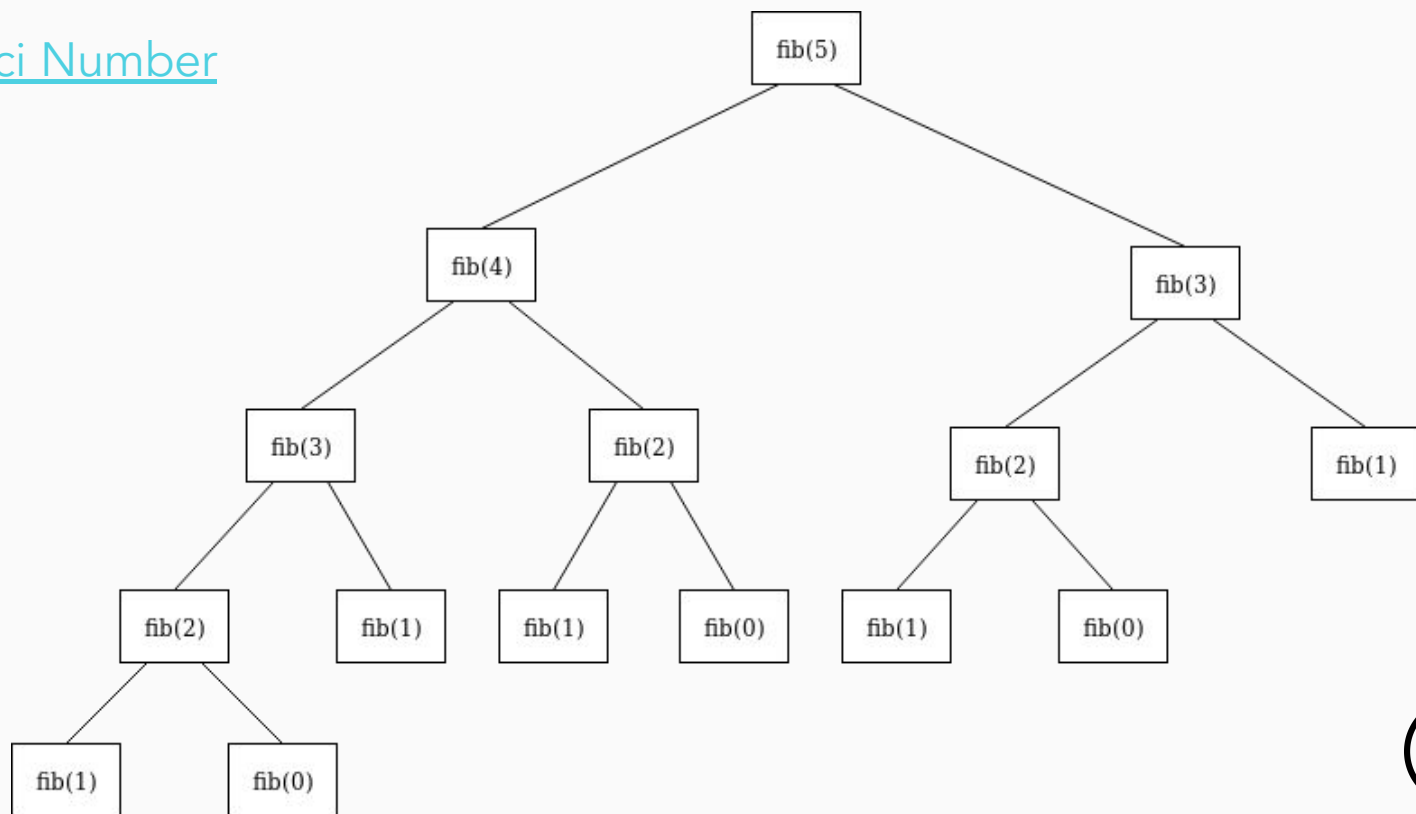
$$(\text{\# of paths})^{(\text{depth of stack})}$$

- The tree on the right represents a recursive Fibonacci function.
- Each node has at most 2 branching paths.
- The depth of the stack is 5, which is the input. Thus, we can call this $O(2^n)$ where n is our input number.



Demo

- Fibonacci Number



Drawbacks of Exponential Runtimes

- Exponential runtimes are extremely inefficient.
- If you run into exponential runtimes, there is likely a way to reduce the time complexity somehow (we will discuss this much later when we get to Dynamic Programming)
- [Unique Paths \(brute force\)](#)



Questions?

