Space Complexity

- the amount of working space required by an algorithm in addition to the input
- how does the amount of memory needed change as the input grows?

```
SUMARRAY(A)
1  total = 0
2  for i = 0 to n − 1
3      total = total + A[i]
4  return total
```

O(1) space

O(1) space

Time Complexity: $\Theta(n)$

Space Complexity: $O(1)$

```
RECURSIVESUMARRAY(A, start)
1  if start = A.length
2        return 0
3  return A[start] + RECURSIVESUMARRAY(A, start + 1)
```

Assume no compiler optimization

Time Complexity:

$T(n) = T(n-1) + 1 \quad \Theta(n)$

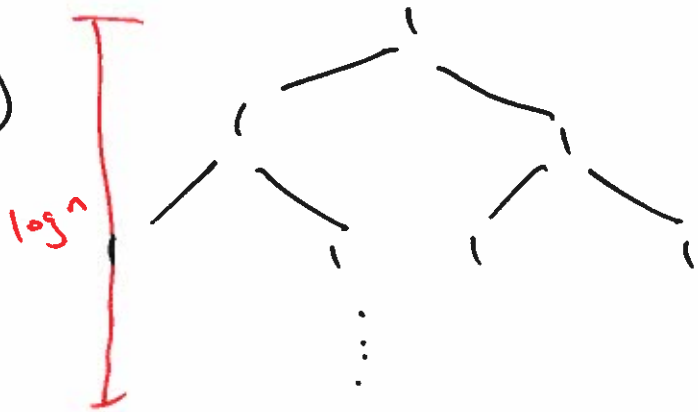
Space Complexity:

$\Theta(n)$ ← because the max depth of the recursion stack is directly related to the size of A

```
DCSumArray(A, start, end)
1  if start = end
2        return A[start]
3  mid = ⌊(start + end)/2⌋
4  return DCSumArray(A, start, mid − 1) + DCSumArray(A, mid, end
```

Time Complexity

$$T(n) = 2T\left(\frac{n}{2}\right) + 1 \in \Theta(n)$$



Space Complexity

- Max depth of the recursion stack

$$\Theta(\log n)$$

```
NaiveFibonacci(n)
1  if n ≤ 1
2        return n
3  return NaiveFibonacci(n − 1) + NaiveFibonacci(n − 2)
```

basic op

Time Complexity

Input size: value of n

$$T(n) = T(n-1) + T(n-2) + 1$$

$$\leq 2T(n-1) + 1 \quad \in O(2^n)$$

$$\geq 2T(n-2) + 1 \quad \in \Omega(2^{\frac{n}{2}})$$

Actual Complexity

$\Theta(\phi^n)$

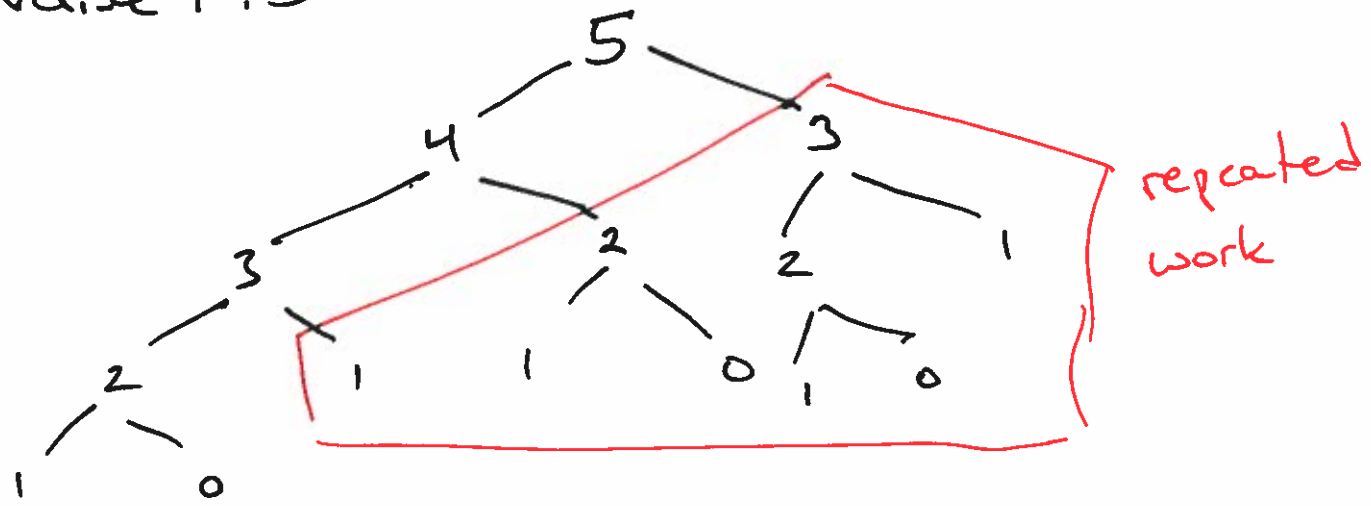
$\Theta\left(\left(\frac{\sqrt{5}+1}{2}\right)^n\right)$

Dynamic Programming

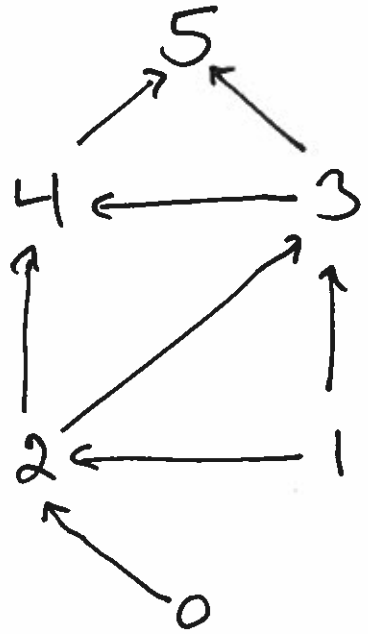
- general idea: trade ~~time~~ space for ~~space~~ time

- A method for solving a problem by breaking it into a series of sub-problems and storing the solution to each subproblem

- Naive Fib



In dynamic programming, instead of a tree we want to think of the subproblems as a DAG



```
MemoizedFib(n)
    // Assume fibs to be a globally available array initialized to nil
1   if fibs[n] = nil
2       if n ≤ 1
3           fibs[n] = n
4       else
5           fibs[n] = MemoizedFib(n − 1) + MemoizedFib(n − 2)
6   return fibs[n]
```

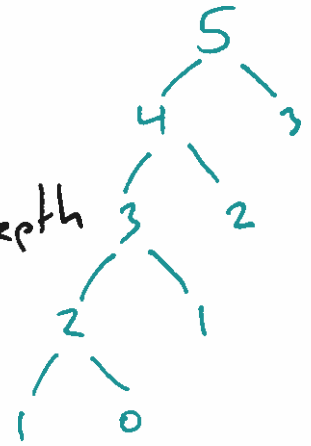
Space Complexity $\Theta(n)$

- size of fibs

$n+1$

- max recursion depth

n



Time Complexity

$\Theta(n)$

$T(n) = T(n-1) + 1$

TABULATIONFIB(n)

```
1  fibs = array of n + 1 integers
2  fibs[0] = 0
3  fibs[1] = 1
4  for i = 2 to n
5        fibs[i] = fibs[i − 1] + fibs[i − 2]
6  return fibs[n]
```

Time Complexity:

$\Theta(n)$

Space Complexity:

$\Theta(n)$

Two main approaches

Memoization

- top down approach (recursive)
- prior to solving a subproblem first check if we have already solved it
- requires recursion overhead
- easy to implement if we already have the recursive solution
- if not all subproblems are required this is faster than tabulation

Tabulation

- bottom up solution (iterative)
- starting with the smallest subproblems
  compute each before it is required
- faster in general (if all or most
  subproblems are required)
- easier to analyze
- more difficult to craft the algorithm

Fibonacci Revisited Give an algorithm that finds the value of the nth number in the Fibonacci sequence in linear time and constant space.