# Dynamic programming

## fibonacci sequence: memoization/top-down

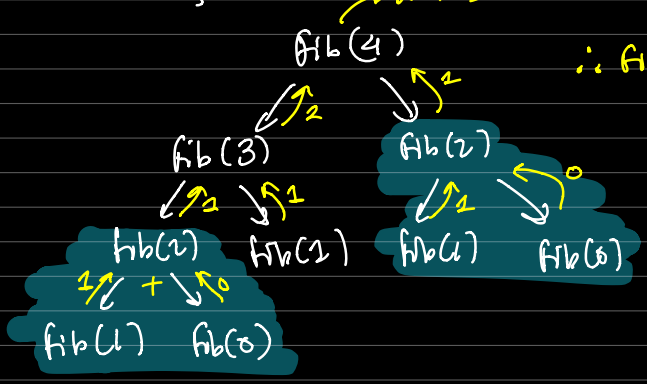
↓ Subproblems

fib(0) = 0
fib(1) = 1

fibonacci sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, 34...

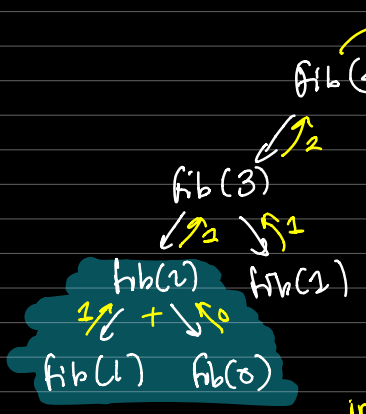
```
int fib(int n) {
  // base case
  if(n <= 1) return n;
  return (fib(n-1) + fib(n-2));
}
```

Runtime: $O(2^n)$
Space: O(n)
↓
call stack




∴ fib(4) = 3

← Repeated work




Runtime: O(n)

Space: O(n)
↓
Call stack
+ memo table

```
int fib(int n, int * memo) {
  if(memo == NULL) {
      memo = int[n+1]{0};
  }
  if(n <= 1) return n;  // base case
  if(!memo[n]) {
      memo[n] = fib[n-1] + fib[n-2];
  }
  return memo[n];
}
```

# Bottom-up approach / ground up

first compute fib(1) & fib(0)
then build memo-table.

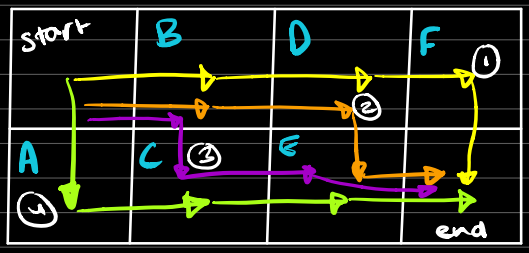
```
int fib(int n) {
    if (n<=1) return n;
    int memo = new int[n];
    memo[0] = 0
    memo[1] = 1
    for (int i=2; i<=n; i++){
        memo[i] = memo[i-1] + memo[i-2];
    }
    return memo[n];
}
```

iterative

n=4

0
1
1
2
3

# Count Paths




Paths = 4

Paths(start, end)

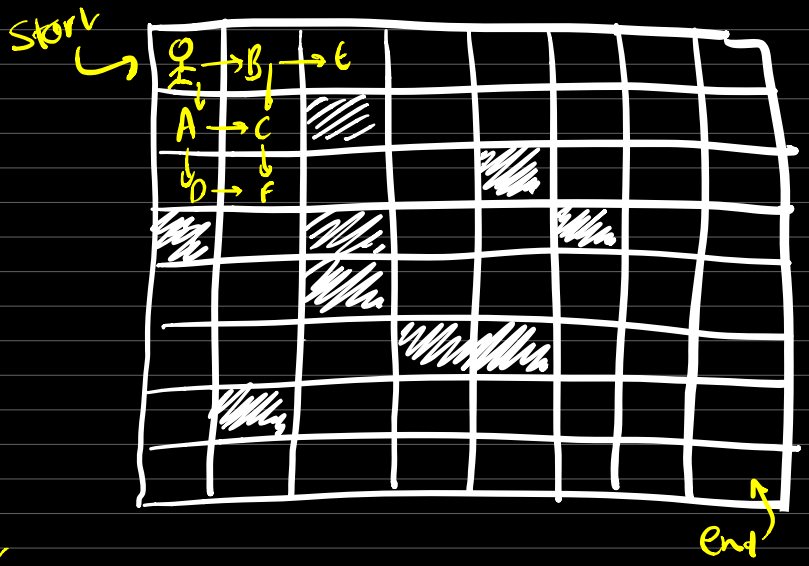
Paths(A, end) + Paths(B, end)

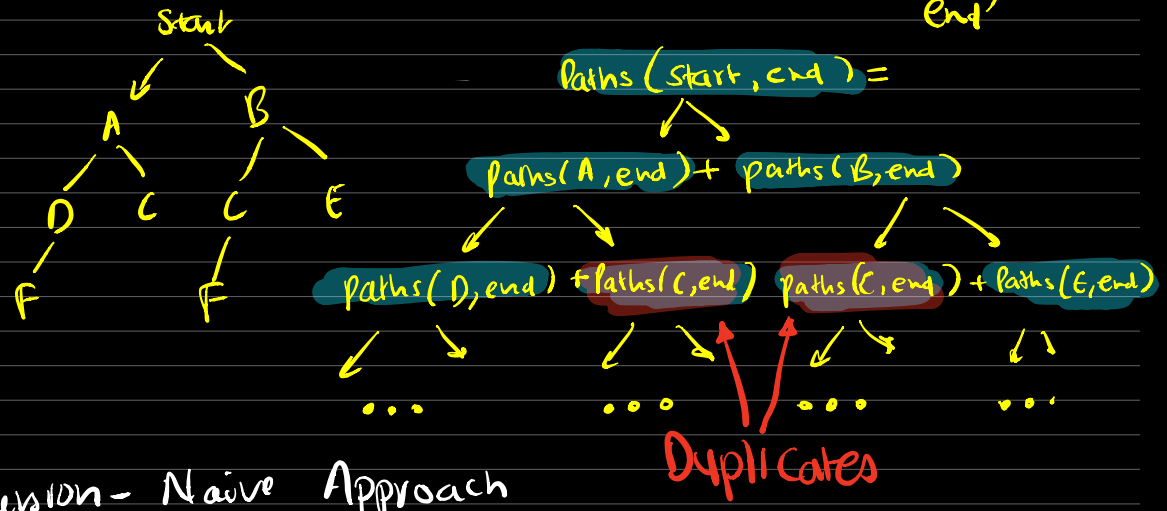
Paths(C, end)

Paths(C, end) + Paths(D, end)

Paths(e, end)

Paths(e, end), Paths(f, end)




## Recursion - Naive Approach

```
int countPaths( bool grid[][], int row, int col ) {
    if (!validSquare (grid, row, col)) return 0;
    if(isAtEnd (grid, row, col)) return 1;
    return countPaths(grid, row+1, col) + countPaths(grid, row, col+1);
}
```

$O(2^{n^2})$

## Memoization ( Top-down Approach)

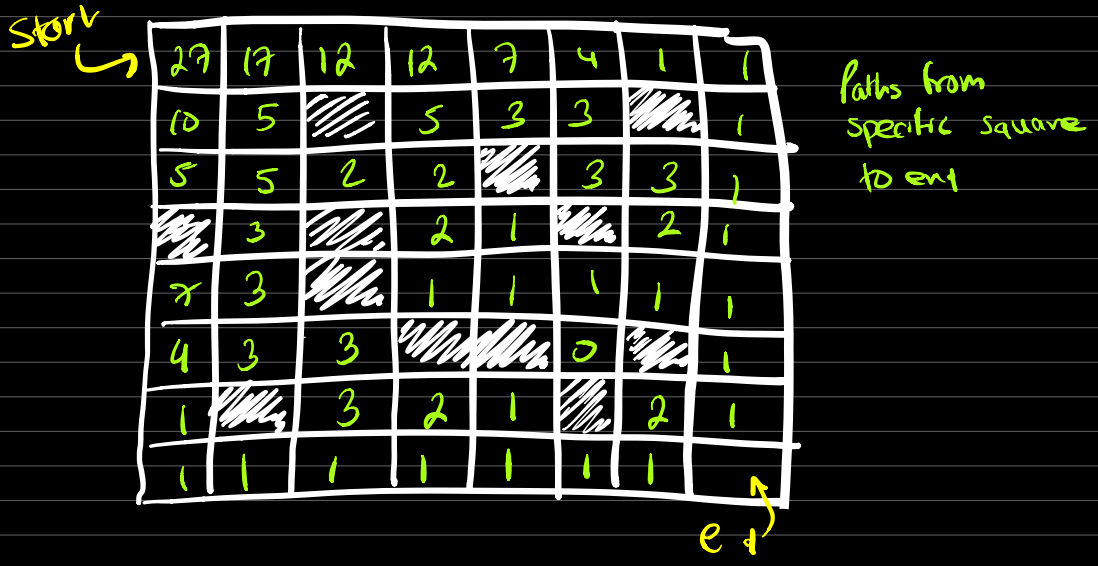
```
int countPaths( bool grid[][], int row, int col, int[][] memo){
    if( !validSquare (grid, row, col)) return 0;
    if( isAtEnd (grid, row, col) return 1;
    if( memo[row][col] == 0) {
```

```
    memo[row][col] = countPaths(grid, row+1, col) + countPaths(grid, row, col+1)
}
return memo[row][col];
```

$O(n^2)$

- Compute each square once

# bottom-up Approach




bottom-up doesn't require call-stack

∴ 27 paths to reach end