Dynamic Programming

- Computing Fibonacci Numbers
- Introduction to Dynamic Programming
- Optimal Substructure
- Memoization

Imdad ullah Khan

Algorithm Design Paradigms

- Greedy Algorithms
  - Build up a solution incrementally
  - Myopically and locally optimizing some local criterion
- Divide and Conquer
  - Break up a problem into (independent) sub-problems
  - Solve each sub-problem independently
  - Combine solution to sub-problems to form solution to original problem
- Dynamic programming = planning over time
  - More general and powerful than divide and conquer
  - Break up a problem into (in)(dependent) sub-problems
  - Generally there is a sequence of problems
  - Identify the optimal substructure: when optimal solution to a problem is made up of optimal solution to smaller subproblems
  - Build up solution to larger and larger subproblems
  - Identify redundancy and repetitions
  - Use memoization or build up memo on the run

Fibonacci Sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \dots$$

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

For $n \geq 8 \quad F_n > 2^{n/2}$ ▷ Prove it by induction

Recursive F_n computation

Implementing the recursive definition of F_n

Algorithm Recursive F_n computation

```
function FIB1(n)
    if n = 0 then
        return 0
    else if n = 1 then
        return 1
    else
        return FIB1(n − 1) + FIB1(n − 2)
```

It's correctness follows from the definition

How much time it takes to compute F_n?

Recursive F_n computation

Algorithm Recursive F_n computation

```
function FIB1(n)
    if n = 0 then
        return 0
    else if n = 1 then
        return 1
    else
        return FIB1(n − 1) + FIB1(n − 2)
```

Let $T(n)$ be the number of operations on input n

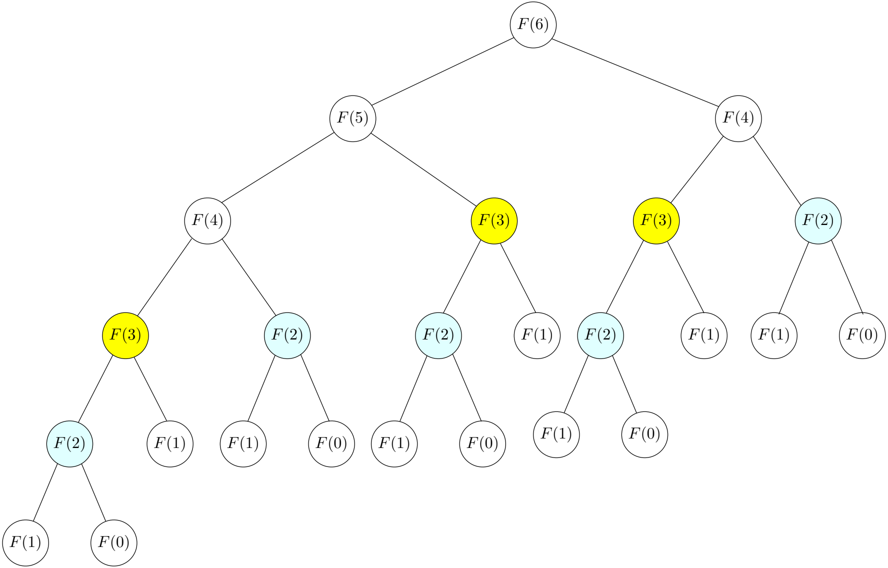
$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ T(n-1) + T(n-2) + 3 & \text{if } n \geq 2 \end{cases}$$

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

For $n \geq 8$, $T(n) > F_n \geq 2^{n/2}$ ▷ **exponential** in n

Problem is unnecessarily repeated recursive calls

Recursive F_n computation



Recursive F_n computation

Algorithm Recursive F_n computation

```
function FIB1(n)
    if n = 0 then
        return 0
    else if n = 1 then
        return 1
    else
        return FIB1(n − 1) + FIB1(n − 2)
```

$$\text{For } n \geq 8, \quad T(n) > F_n \geq 2^{n/2}$$

Problem is unnecessarily repeated recursive calls

Memoization: Save results of subproblems in a memo

Use the memo when needed instead of recomputing

F_n computation with Memoization

Algorithm F_n computation with memoization

$F[0 \dots n] \leftarrow \text{NEGONES}(n+1)$
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
function $\text{FIB2}(n)$
  if $F[n-1] = -1$ **then**
    $F[n-1] \leftarrow \text{FIB2}(n-1)$ ▷ Call FIB2 function only if $F[n-1] = -1$
  if $F[n-2] = -1$ **then**
    $F[n-2] \leftarrow \text{FIB2}(n-2)$
  return $F[n-1] + F[n-2]$

F_n computation with Memoization

Algorithm Compute F_n with memo

$F[0 \dots n] \leftarrow \textsc{negOnes}(n+1)$
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
function $\textsc{Fib2}(n)$
  if $F[n-1] = -1$ **then**
    $F[n-1] \leftarrow \textsc{Fib2}(n-1)$
  if $F[n-2] = -1$ **then**
    $F[n-2] \leftarrow \textsc{Fib2}(n-2)$
  return $F[n-1] + F[n-2]$

Let $T_2(n)$ be runtime of $\textsc{fib2}(n)$

- Count number of calls
- Only calls if $F[\cdot] = -1$
- Total calls $n+1$
- $O(1)$ operations per call

$$T_2(n) = O(n)$$

▷ Compare with $T(n) = O(2^n)$

F_n computation Bottom Up Approach

Algorithm Bottom-Up F_n Computation

$F[0 \dots n] \leftarrow \textsc{negones}(n+1)$
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i = 2$ to n **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

- No recursion overhead
- Analyze time needed to fill up memo
- Total number of updates to memo is $n+1$
- Total runtime $T_3(n) = O(n)$

▷ Compare with $T(n) = O(2^n)$