Data Structures

Nahian Salsabil (NNS)

Slides Adapted from Prantik Paul

Topics Covered So Far

- Array
- Linked List
- Stack
- Queue
- Basic Recursion

Outline

- Advanced Recursion

Problems

- Inefficient Recursion
- Space for Activation Frames
- Infinite Recursion

Solution

- Top-Down Approach
  - Memoization
- Bottom-Up Approach
  - Dynamic Programming

Recursive Programming (Steps)

1. Write down the recursion,
2. Implement the recursive solution,
3. Memoize it,
4. Transform into an iterative solution, and finally
5. Make further improvements.

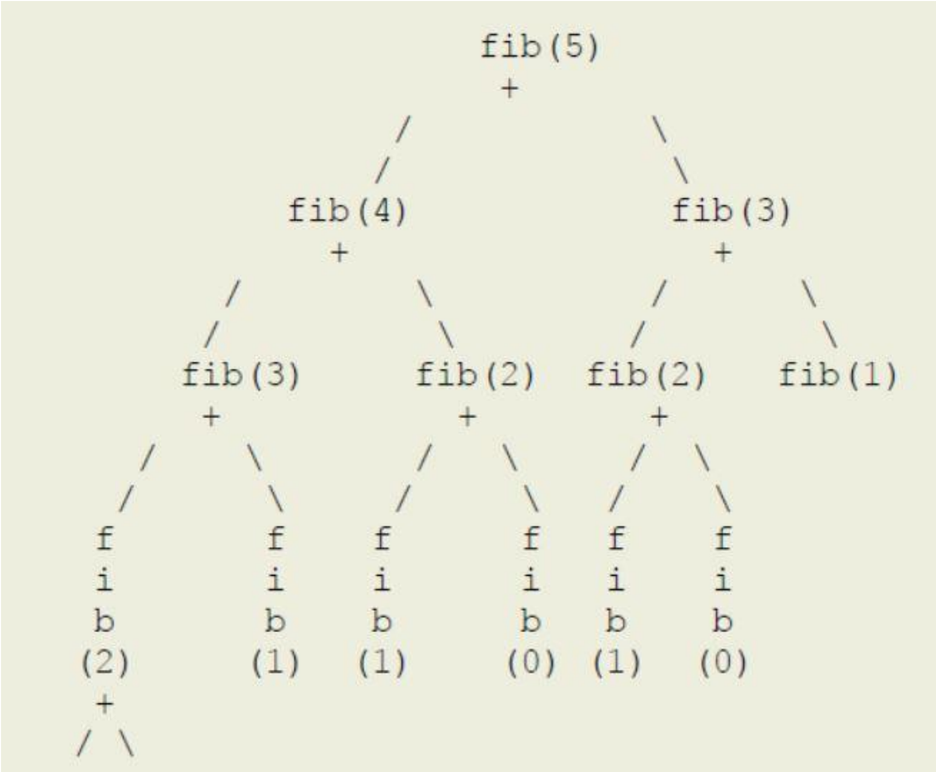
Recursive Programming (Fibonacci : Step 1)

```
                 -
                 | n                                 if n < 2
fib(n)    =      |
                 | fib(n-1) + fib(n-2)               n ≥ 2
                 -
```

Recursive Programming (Fibonacci : Step 2)

```
def fib(n):
  if n < 2:
    return n     #base case
  else
    return fib(n-1) + fib(n-2)     #recursive part
```

Recursive Programming (Fibonacci : Step 3)



Recursive Programming (Fibonacci : Step 3)

memoization trades space for time

Linear Array. Size : n+1

Memoization (Fibonacci)

```
def M_fib(n):
  # Assume that we have some "global" array (also called a table)
  # F with n+1 capacity. Why can F not be a local variable?

  if n < 2:
    return n    #base case
  else:

    #Compute (and save) if it's not already computed

    if (F[n] is empty)  # <<<< NOTE PSEUDOCODE!
      F[n] = M_fib(n-1) + M_fib(n-2)    #recursive part

    # Now just return the computed (and saved) value.
    return F[n]
```

Memoization (Fibonacci)

```
def fib(n):
  # Create and initialize the table.
  F = [-1]*(n+1)

  #Now we can call M-Fib with this extra parameter "F".
  return M_fib(n, F)

def M_fib(n, F):
  #The table "F" is being passed as a parameter

  if n < 2:
    return n    #base case

  else:
    #Compute (and save) if it's not already computed.
    if F[n] == -1:
      F[n] = M_fib(n-1, F) + M_fib(n-2, F)    #recursive part

    #Now just return the computed (and saved) value.
    return F[n]
```

Recursive Programming (Fibonacci : Step 4)

```
def fib(n):
  F=[None]*(n+1)  #The table to store computed values
  F[0]=0          #The base case for n = 0
  F[1]=1          #The base case for n = 1

  for i in range(2,n+1):
    F[i]=F[i-1]+F[i-2]
  return F[n]
```

Recursive Programming (Fibonacci : Step 5)

```
def fib(n):
  f_2 = 0     #The (n-2)th value
  f_1 = 1     #The (n-1)th value
  f = n
  #The result f is initialized to n (why? So that it works when n is 0 or 1).

  for i in range(2, n+1):
    f = f_1 + f_2

    #Now update the f_1 and f_2 for the next iteration (if any).
    f_2 = f_1
    f_1 = f

  return f
```