INTRODUCTION TO RECURSIVE FORMULATIONS FOR ALGORITHM DESIGN: III

Partha P Chakrabarti

Indian Institute of Technology Kharagpur

Algorithm Design by Recursion Transformation

- Algorithms and Programs
- Pseudo-Code
- Algorithms + Data Structures = Programs
- Initial Solutions + Analysis + Solution Refinement + Data Structures = Final Algorithm
- Use of Recursive Definitions as Initial Solutions
- Recurrence Equations for Proofs and Analysis
- Solution Refinement through Recursion Transformation and Traversal
- Data Structures for saving past computation for future use

1. Initial Solution
   a. Recursive Definition – A set of Solutions
   b. Inductive Proof of Correctness
   c. Analysis Using Recurrence Relations
2. Exploration of Possibilities
   a. Decomposition or Unfolding of the Recursion Tree
   b. Examination of Structures formed
   c. Re-composition Properties
3. Choice of Solution & Complexity Analysis
   a. Balancing the Split, Choosing Paths
   b. Identical Sub-problems
4. Data Structures & Complexity Analysis
   a. Remembering Past Computation for Future
   b. Space Complexity
5. Final Algorithm & Complexity Analysis
   a. Traversal of the Recursion Tree
   b. Pruning
6. Implementation
   a. Available Memory, Time, Quality of Solution, etc

Piṅgala's Numbers (3rd Century BC)

- The Chandaḥśāstra presents the first known description of a binary numeral system in connection with the systematic enumeration of meters with fixed patterns of short and long syllables. The discussion of the combinatorics of meter corresponds to the binomial theorem. **Halāyudha's** commentary includes a presentation of the Pascal's triangle (called ***meruprastāra***). Piṅgala's work also contains the Fibonacci numbers, called ***mātrāmeru***. (later Bharata Muni (100 BC), Virahanka (700 AD), Hemachandra (1150 AD) – all before Fibonacci 1200 AD) $f(n) = f(n-1) + f(n-2)$
- Use of zero is sometimes ascribed to Piṅgala due to his discussion of binary numbers, usually represented using 0 and 1 in modern discussion, but Piṅgala used light (*laghu*) and heavy (*guru*) rather than 0 and 1 to describe syllables. As Piṅgala's system ranks binary patterns starting at one (four short syllables—binary "0000"—is the first pattern), the nth pattern corresponds to the binary representation of n-1 (with increasing positional values). Piṅgala is thus credited with using binary numbers in the form of short and long syllables (the latter equal in length to two short syllables), a notation similar to Morse code.
- Piṅgala used the Saṁskṛta word ***śūnya*** explicitly to refer to zero.

Mātrāmeru or Fibonacci Numbers

$$f(n) = 0 \text{ if } n = 0 \ (n \le 0)$$
$$= 1 \text{ if } n = 1$$
$$= f(n-1) + f(n-2), \text{ if } n > 1$$

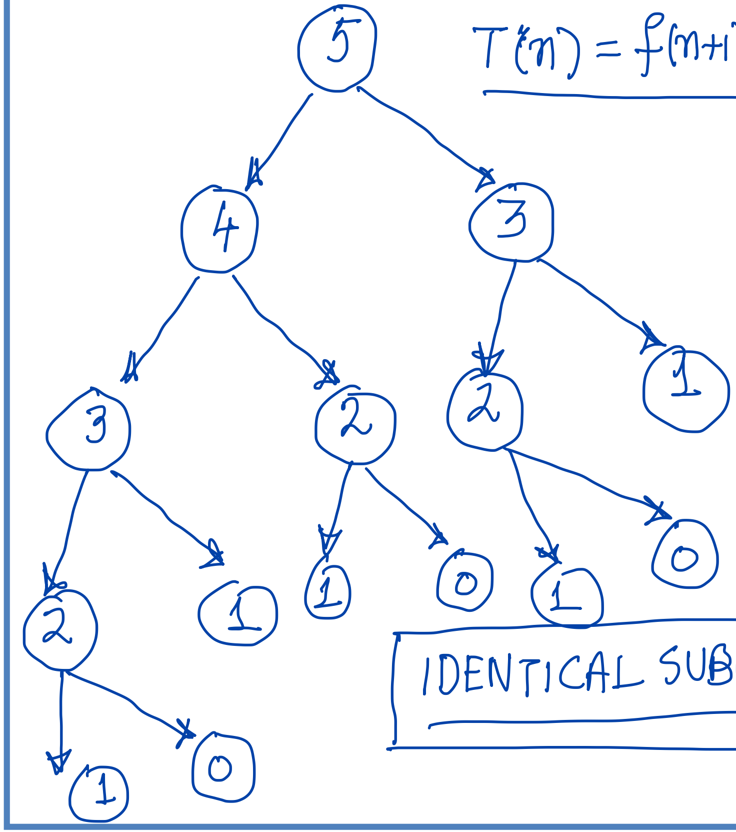
$$f(n) = \left\{ \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} \right.$$

0, 1, 1, 2, 3, 5, 8, ...

```
fib(n)
{ if (n ≤ 0) return (0)
  if (n = 1) return (1)
  m = fib(n-1) + fib(n-2)
  return (m)
}
```

$$T(n) = 0 \text{ if } n \le 1$$
$$= T(n-1) + T(n-2) + 1$$
$$= \boxed{f(n+1) - 1}$$

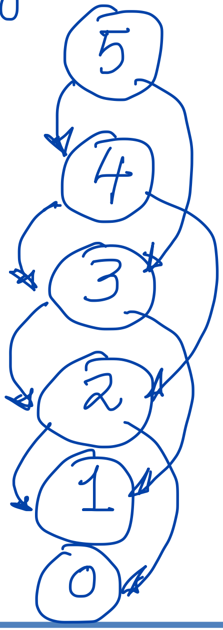
Analyzing the Recursion Structure



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

IDENTICAL SUBPROBLEMS

We would like to compute the value of $f(n)$ only once for every n and reuse the same



Data Storage

To store the required past computations

MEMOIZATION

$T(n) = O(n)$

Memoization

FIB[ ] , FIB[0] = 0   FIB[1] = 1

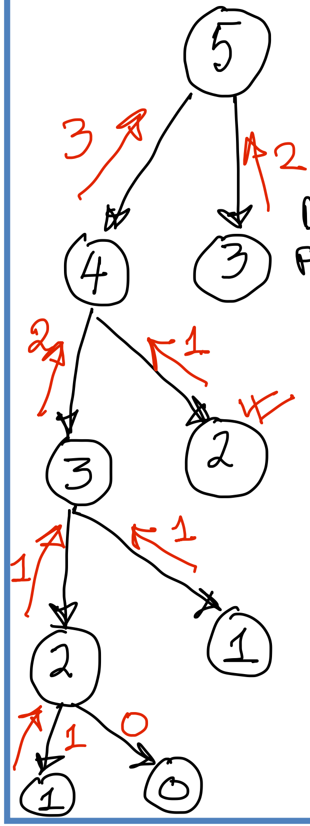
Top-down algorithm

Done[ ],   Done[0] = 1 , Done[1] = 1

All others are 0.

```
fib2(n)
{ if (Done[n] = 1) return
                   (FIB[n]);

   m = fib2(n-1) + fib2(n-2)
   Done[n] = 1
   FIB[n] = m
   return (m)
}
```



Done[0] = 1   FIB[0] = 0

Done[1] = 1   FIB[1] = 1

	0	1	2	3	4	5
Done	1	1	1	1	1	1
FIB	0	1	1	2	3	5

$T(n) = O(n)$

Space Requirement

$O(n)$

Finalizing the Algorithm

```
fib3(n)                FIB[0]=0
                        FIB[1]=1
{ for i=2 to n do
     FIB[i]= FIB[i-1] +
                FIB[i-2]
}
```

0	1	2	3	4	5	
0	1	1	2	3	5	. . . . .

BOTTOM-UP EVALUATION

```
fib4(n)
{  x1=1   // FIB[1]
   x2=0   // FIB[0]
   for i=2 to n do
   {  m = x1+x2
      x2= x1
      x1= m
   }
   return (m)
}
```

fib5(n, - -)

Using TAIL RECURSION

Variations

1. $f(n) = f(n-1) + f(n-157)$

2. $f(n) = f(n-1) + f\left(n - \frac{n}{2}\right)$

3. $f(n) = f(n+1) + f(n+3)$ if n is even

   $f\left(\frac{n-1}{2}\right)$ if n is odd

4. $f(n) = f(g(n)) + f(h(n))$

   ↳ possibility of cyclic dependencies

Evaluation of Fibonacci-like Recurrences

$$\boxed{\begin{aligned} f(n) &= r(n) \quad \text{if } c(n) \text{ is true} \\ &= f(g(n)) + f(h(n)) \quad \text{if } c(n) \text{ is false} \end{aligned}}$$

where $c(n)$ is the base condition. Also $c(n)$, $r(n)$, $g(n)$, $h(n)$ may be assumed to be deterministic non-recursive functions (some well-defined algorithm)

Write an algorithm to evaluate $f(n)$

1. Detect and flag cyclic dependencies
2. Avoid re-solving identical sub-problems
3. optimal memory usage

Evaluation Algorithm

Data Structure for Dynamically storing F[ ] and Done[ ]

```
eval-f(n)
{ if (c(n) = true) return(r(n))
    x = g(n)
    y = h(n)
    z = f(x) + f(y)
    return (z)
}
```

F[ ] to store value

Done[i] —
- 0 if it has not yet been called
- 1 if it has been called
- 2 if value is computed

```
eval-f(n)
{ If (c(n) = true)
     { Done[n] = 2 ; F[n] = r(n) ;
          return (F[n]); }
   if (Done[n] = 1) { print ("CYCLE")
                        exit
                      }
   if (Done[n] = 2)   return (F[n]);
   Done[n] = 1
    x = g(n) ;  y = h(n) ;
    z = eval-f(x) + eval-f(y).
    F[n] = z
    Done[n] = 2
    return (F[n])
}
```

We could also memoize g(n) & h(n)

Memoization Data Structure

Data Structure

Define operations :-

find (n)
insert (n)

Balanced BST

create, get_val, assign_val

Overview of Algorithm Design

1. Initial Solution
   a. Recursive Definition – A set of Solutions
   b. Inductive Proof of Correctness
   c. Analysis Using Recurrence Relations
2. Exploration of Possibilities
   a. Decomposition or Unfolding of the Recursion Tree
   b. Examination of Structures formed
   c. Re-composition Properties
3. Choice of Solution & Complexity Analysis
   a. Balancing the Split, Choosing Paths
   b. Identical Sub-problems
4. Data Structures & Complexity Analysis
   a. Remembering Past Computation for Future
   b. Space Complexity
5. Final Algorithm & Complexity Analysis
   a. Traversal of the Recursion Tree
   b. Pruning
6. Implementation
   a. Available Memory, Time, Quality of Solution, etc

1. Core Methods
   a. Divide and Conquer
   b. Greedy Algorithms
   c. Dynamic Programming
   d. Branch-and-Bound
   e. Analysis using Recurrences
   f. Advanced Data Structuring
2. Important Problems to be addressed
   a. Sorting and Searching
   b. Strings and Patterns
   c. Trees and Graphs
   d. Combinatorial Optimization
3. Complexity & Advanced Topics
   a. Time and Space Complexity
   b. Lower Bounds
   c. Polynomial Time, NP-Hard
   d. Parallelizability, Randomization

Thank you

Any Questions?