Foundations of Algorithm SCS1020

Dr. Dinuni Fernando PhD

Senior Lecturer

UCSC

Goal

- Learn how to design and analyze recursive algorithms
- Learn when to use or not to use recursive algorithms
- Derive & solve recurrence equations to analyze recursive algorithms

Recursion

- What is the recursive definition of $n!$?

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \text{ or } 1 \\ n * ((n-1)!) & \text{otherwise} \end{cases}$$

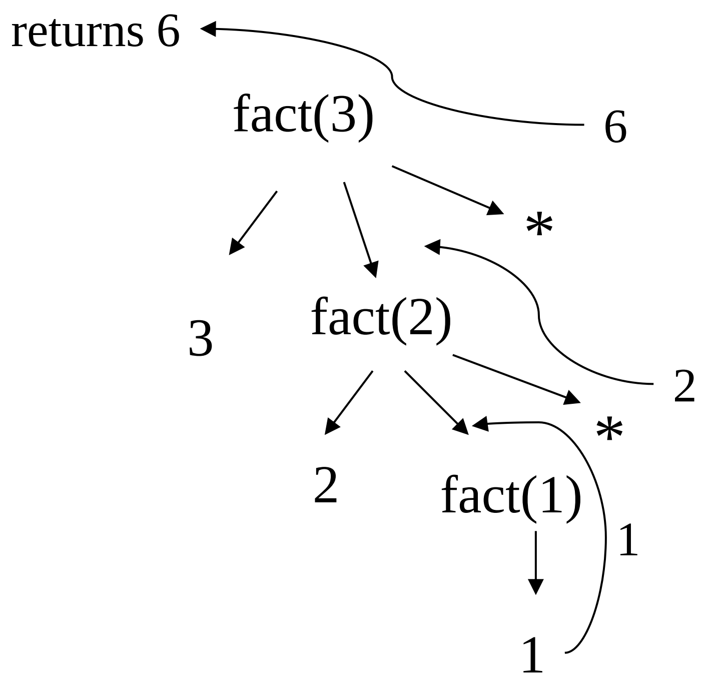
- Program

```
fact(n) {
   if (n<=1) return 1;
   else return n*fact(n-1);
}
// Note '*' is done after returning from fact(n-1)
```

Recursive algorithms

- A recursive algorithm typically contains recursive calls to the same algorithm
- In order for the recursive algorithm to terminate, it must contain code to directly solve some "base case(s)"with no recursive calls
- We use the following notation:
  - *DirectSolutionSize* is the "size" of the base case
  - *DirectSolutionCount* is the number of operations done by the "direct solution"

A Call Tree for fact(3)



```
int fact(int n) {
   if (n<=1) return 1;
else return n*fact(n-1);
}
```

The Run Time Environment

- When a function is called an activation records('ar') is created and pushed on the program stack.
- The activation record stores copies of local variables, pointers to other ‘ar’ in the stack and the return address.
- When a function returns the stack is popped.

Goal: Analyzing recursive algorithms

- Until now we have only analyzed (derived the count of) non-recursive algorithms.

- In order to analyze recursive algorithms, we must learn to:
    - Derive the recurrence equation from the code
    - Solve recurrence equations

Deriving a Recurrence Equation for a Recursive Algorithm

- Our goal is to compute the count (Time) $T(n)$ as a function of n, where n is the size of the problem
- We will first write a recurrence equation for $T(n)$

  For example, $T(n)=T(n-1)+1$ and $T(1)=0$

- Then we will solve the recurrence equation. What’s the solution to $T(n)=T(n-1)+1$ and $T(1)=0$?

Deriving a Recurrence Equation for a Recursive Algorithm

1. Determine the “size of the problem”. The count T is a function of this *size*

2. Determine *DirectSolSize*, such that for *size* ≤ *DirectSolSize* the algorithm computes a direct solution, with the *DirectSolCount(s).*

$$T(size) = \begin{cases} DirectSolCount & size \le DirectSolSize \\ GeneralCount & \text{otherwise} \end{cases}$$

Deriving a Recurrence Equation for a Recursive Algorithm

$$T(size) = \begin{cases} DirectSolCount & size \le DirectSolSize \\ GeneralCount & \text{otherwise} \end{cases}$$

To determine *GeneralCount*:

3. Analyze the total number of recursive calls, *k*, done by a single call of the algorithm and their counts,

   $T(n_1), \ldots, T(n_k) \rightarrow$ $RecursiveCallSum = \sum_{i=1}^{k} T(n_i)$

4. Determine the "non recursive count" $t(size)$ done by a single call of the algorithm, i.e., the amount of work, excluding the recursive calls done by the algorithm

$$T(size) = \begin{cases} DirectSolCount & size \le DirectSolSize \\ RecursiveCallSum + t(size) & \text{otherwise} \end{cases}$$

Deriving *DirectSolutionCount* for *Factorial*

$$T(size) = \begin{cases} DirectSolCount & size \le DirectSolSize \\ RecursiveCallSum + t(size) & \text{otherwise} \end{cases}$$

```
int fact(int n) {
  if (n<=1) return 1;
  else  return n*fact(n-1); }
```

1. *Size = n*
2. *DirectSolSize is* n<=1
3. *DirectSolCount* is θ(1)
   The algorithm does a small constant number of operations (comparing n to 1, and returning)

Deriving a *GeneralCount* for Factorial

$$T(size) = \begin{cases} DirectSolCount & size \le DirectSolSize \\ GeneralCount & \text{otherwise} \end{cases}$$

```
int fact(int n) {
    if (n<=1) return 1;
    // Note '*' is done after returning from fact(n-1)
    else
      return n * fact(n-1);
```

Operations counted in t(n)

The only recursive call, requiring $T(n - 1)$ operations

3. *RecursiveCallSum* = $T(n - 1)$

4. $t(n) = \theta$`(1) (if, *, -, return)`

$$\boxed{T(n) = \begin{cases} \Theta(1) & \text{for } n \le 1 \\ T(n-1) + \Theta(1) & \text{for } n > 1 \end{cases}}$$

Solving recurrence equations

- Techniques for solving recurrence equations:
  - *Recursion tree method*
  - *Substitution method*
  - *Iteration method*
  - *Master Theorem*
- We discuss these methods with examples.

Deriving the count using the recursion tree method

$$T(size) = \begin{cases} DirectSolCount & size \le DirectSolSize \\ GeneralCount & \text{otherwise} \end{cases}$$

- Recursion trees provide a convenient way to represent the unrolling of a recursive algorithm
- It is not a formal proof but a good technique to compute the count.
- Once the tree is generated, each node contains its “non recursive number of operations” $t(n)$ or $DirectSolutionCount$
- The count is derived by summing the “non recursive number of operations” of all the nodes in the tree
- For convenience, we usually compute the sum for all nodes at each given depth, and then sum these sums over all depths.

$$T(size) = \begin{cases} DirectSolCount & size \le DirectSolSize \\ GeneralCount & \text{otherwise} \end{cases}$$

Building the Recursion tree

- The initial recursion tree has a single node containing two fields:
  - The recursive call, (for example $Factorial(n)$) and
  - the corresponding count $T(n)$.
- The tree is generated by:
  - Unrolling the recursion of the node depth 0,
  - then unrolling the recursion for the nodes at depth 1, etc.
- The recursion is unrolled as long as the size of the recursive call is greater than $DirectSolutionSize$

$$T(size) = \begin{cases} DirectSolCount & size \le DirectSolSize \\ GeneralCount & \text{otherwise} \end{cases}$$

Building the Recursion tree

- When the "recursion is unrolled", each current leaf node is substituted by a subtree containing a root and a child for each recursive call done by the algorithm.
  - The root of the subtree contains the recursive call, and the corresponding "non recursive count".
  - Each child node contains a recursive call, and its corresponding count.
- The unrolling continues, until the "size" in the recursive call is *DirectSolutionSize*
- Nodes with a call of *DirectSolutionSize*, are not "unrolled", and their count is replaced by *DirectSolutionCount*

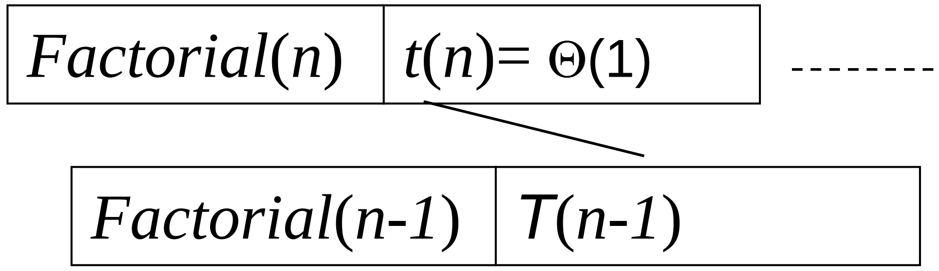
Example: Recursive factorial

Factorial(n)	$T(n)$

- Initially, the recursive tree is a node containing the call to *Factorial*(n), and count T(n).

- When we unroll the computation this node is replaced with a subtree containing a root and one child:
- The root of the subtree contains the call to *Factorial*(n) , and the "non recursive count" for this call $t(n)= \Theta(1)$.
- The child node contains the recursive call to *Factorial*(n-1), and the count for this call, $T(n\text{-}1)$.

After the first unrolling

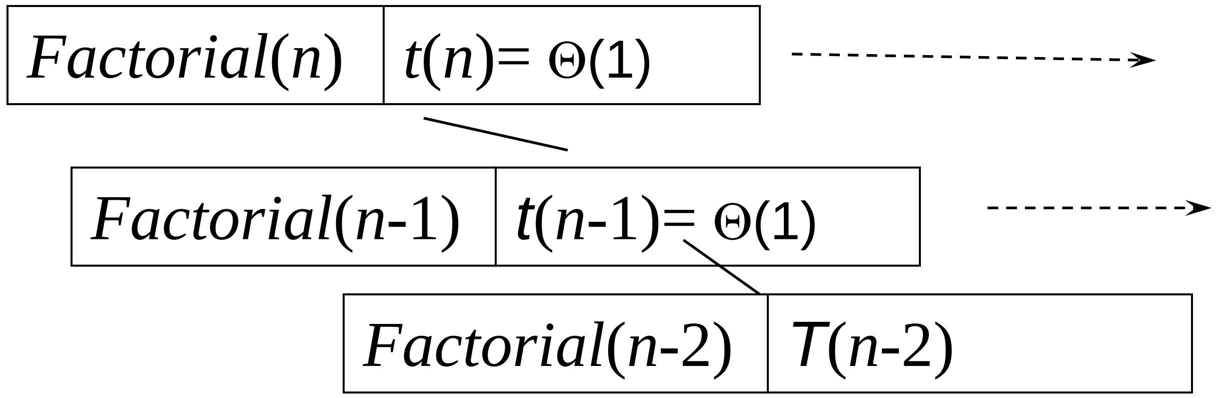


depth	*nd*	$T(n)$
0	1	$\Theta(1)$
1	1	$T(n-1)$

nd denotes the number of nodes at that depth

$$T(n) = \begin{cases} \Theta(1) & \text{for } n \le 1 \\ T(n-1) + \Theta(1) & \text{for } n > 1 \end{cases}$$

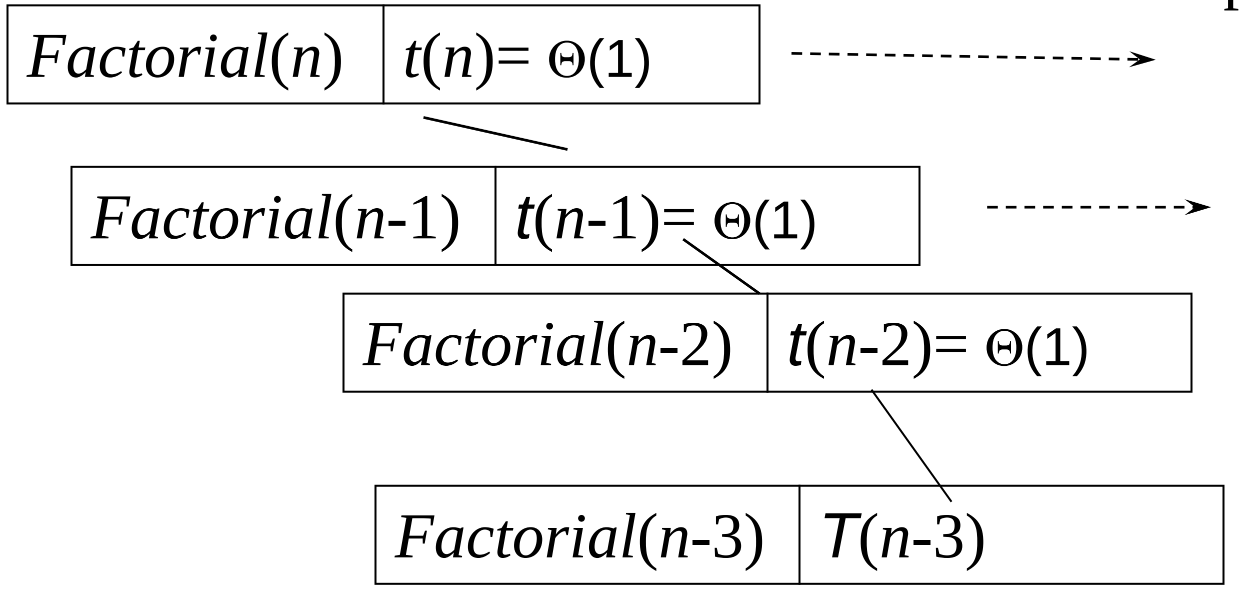
After the second unrolling



depth	nd	$T(n)$
0	1	$\Theta(1)$
1	1	$\Theta(1)$
2	1	$T(n-2)$

$$T(n) = \begin{cases} \Theta(1) & \text{for } n \le 1 \\ T(n-1) + \Theta(1) & \text{for } n > 1 \end{cases}$$

After the third unrolling



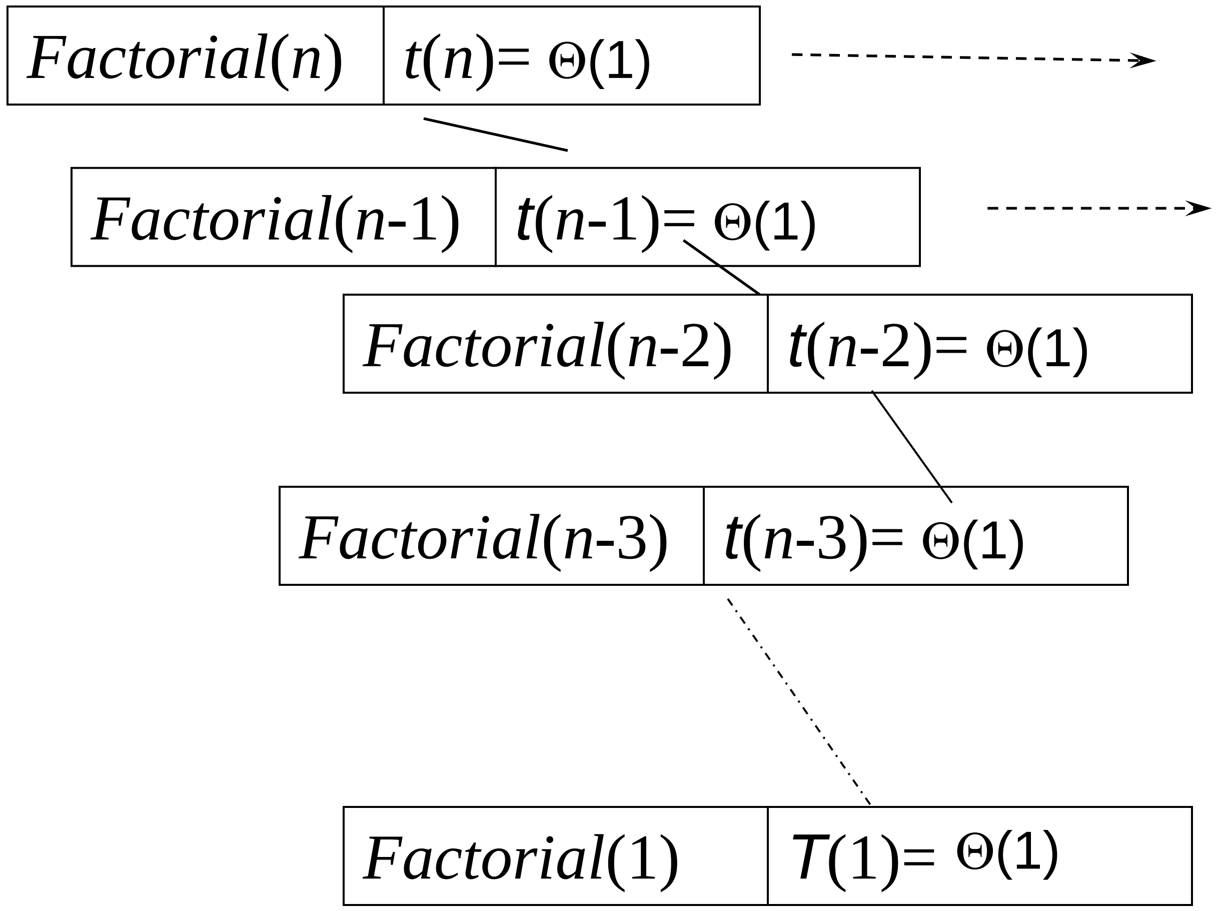
depth	*nd*	$T(n)$
0	1	$\Theta(1)$
1	1	$\Theta(1)$
2	1	$\Theta(1)$
3	1	$T(n-3)$

$$T(n) = \begin{cases} \Theta(1) & \text{for } n \le 1 \\ T(n-1) + \Theta(1) & \text{for } n > 1 \end{cases}$$

For *Factorial*

DirectSolutionSize = 1 and *DirectSolutionCount*= $\Theta(1)$

The recursion tree



Factorial(n) $t(n) = \Theta(1)$
Factorial(n-1) $t(n-1) = \Theta(1)$
Factorial(n-2) $t(n-2) = \Theta(1)$
Factorial(n-3) $t(n-3) = \Theta(1)$
Factorial(1) $T(1) = \Theta(1)$

depth	*nd*	$T(n)$
0	1	$\Theta(1)$
1	1	$\Theta(1)$
2	1	$\Theta(1)$
3	1	$\Theta(1)$
n-1	1	$T(1) = \Theta(1)$

The sum $T(n) = n^* \Theta(1) = \Theta(n)$

Binary search

- The problem is divided into 3 subproblems
    - x=S[mid], x∈ S[low,..,mid-1], x∈ S[mid+1,..,high]

- The first case x=S[mid] is easily solved

- The other cases
  x∈ S[low,..,mid-1], or x∈ S[mid+1,..,high] require a recursive call

- When the array is empty the search terminates with a “non-index value”

```
BinarySearch($S$, $x$, $low$, $high$)
  if low > high then
        return $NoSuchKey$
  else
        mid ← floor ((low+high)/2)
        if ($x$ == $S$[mid])
              return mid
        else if  ($x$ < $S$[mid]) then
              return BinarySearch($S$, x, low, mid-1)
        else
              return BinarySearch($S$, $x$,  mid+1, high)
```

Worst case analysis – Binary Search Tree

- A worst input (what is it?) causes the algorithm to keep searching until low>high
- Assume $2^k \leq$ n < 2^{k+1} k = ëlg nû

  - $T\,(n)$: worst case number of comparisons for the call to $BS(n)$

$$\boxed{T(n) = \begin{cases} 0 \text{ for } & n = 0 \\ 1 \text{ for } & n = 1 \\ 1 + T(\lfloor n/2 \rfloor) \text{ for } & n > 1 \end{cases}}$$

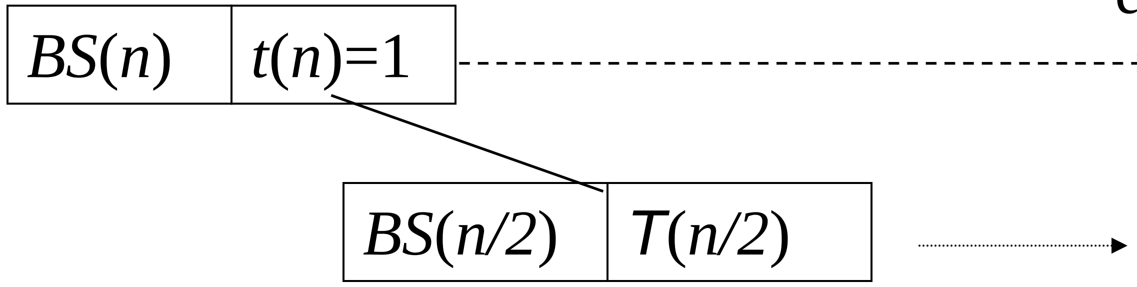
Recursion tree for BinarySearch (BS)

$BS(n)$	$T(n)$

- Initially, the recursive tree is a node containing the call to $BS(n)$, and total amount of work in the worst case, T(n).

- When we unroll the computation this node is replaced with a subtree containing a root and one child:
- The root of the subtree contains the call to $BS(n)$, and the "nonrecursive work" for this call $t(n)$.
- The child node contains the recursive call to $BS(n/2)$, and the total amount of work in the worst case for this call is$T(n/2)$.

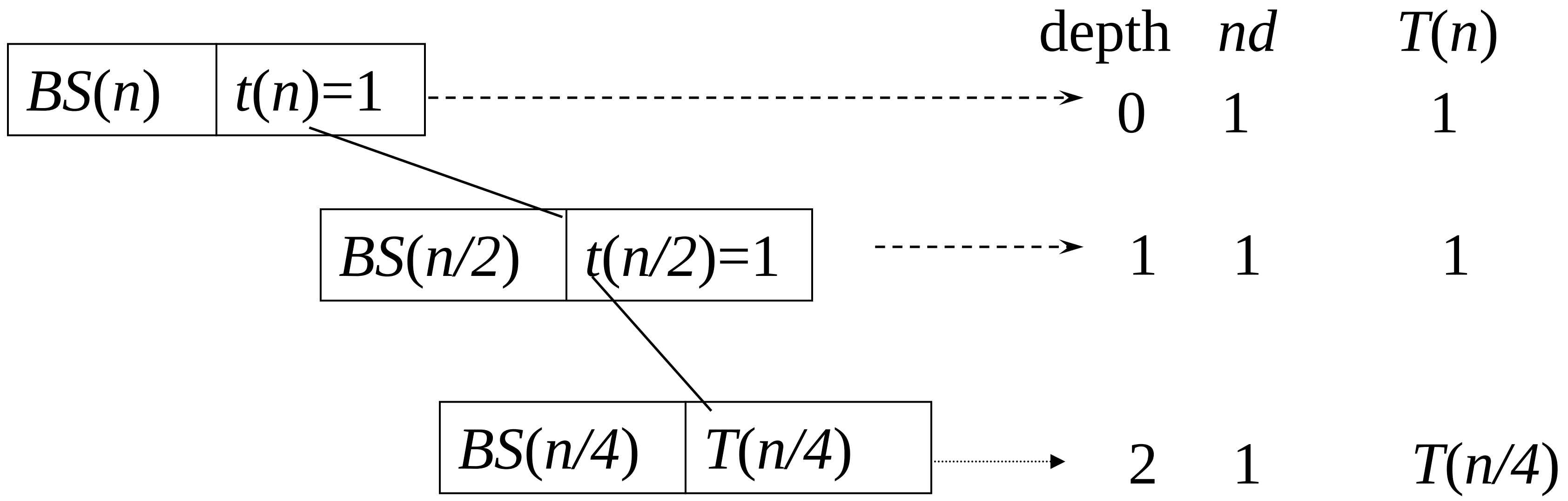
After first unrolling



depth	nd	$T(n)$
0	1	1
1	1	$T(n/2)$

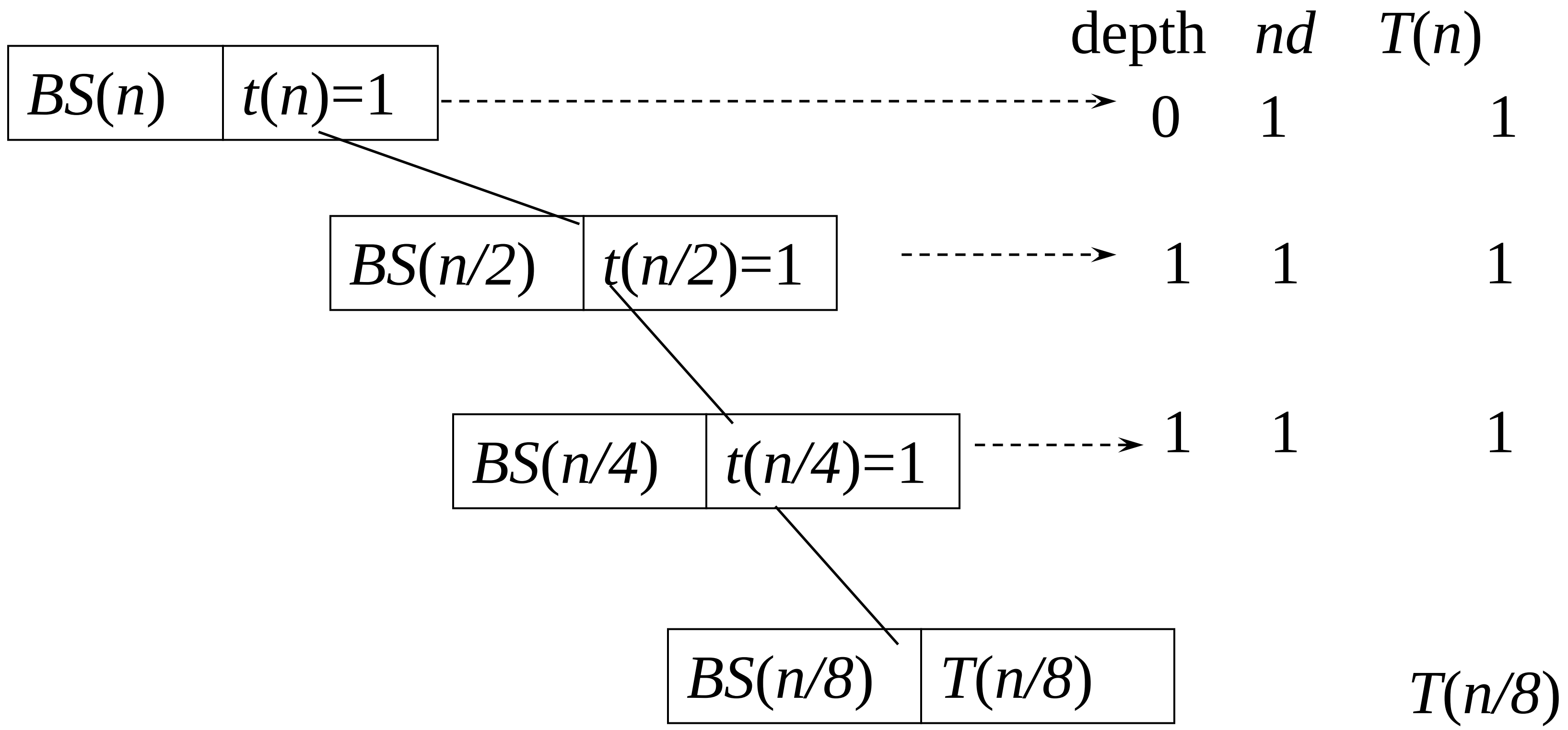
$$\boxed{T(n)=\begin{cases}0 \text{ for } & n=0\\ 1 \text{ for } & n=1\\ 1+T(\lfloor n/2 \rfloor) \text{ for } & n>1\end{cases}}$$

After second unrolling



$$T(n) = \begin{cases} 0 \text{ for } & n = 0 \\ 1 \text{ for } & n = 1 \\ 1 + T(\lfloor n/2 \rfloor) \text{ for } & n > 1 \end{cases}$$

After third unrolling

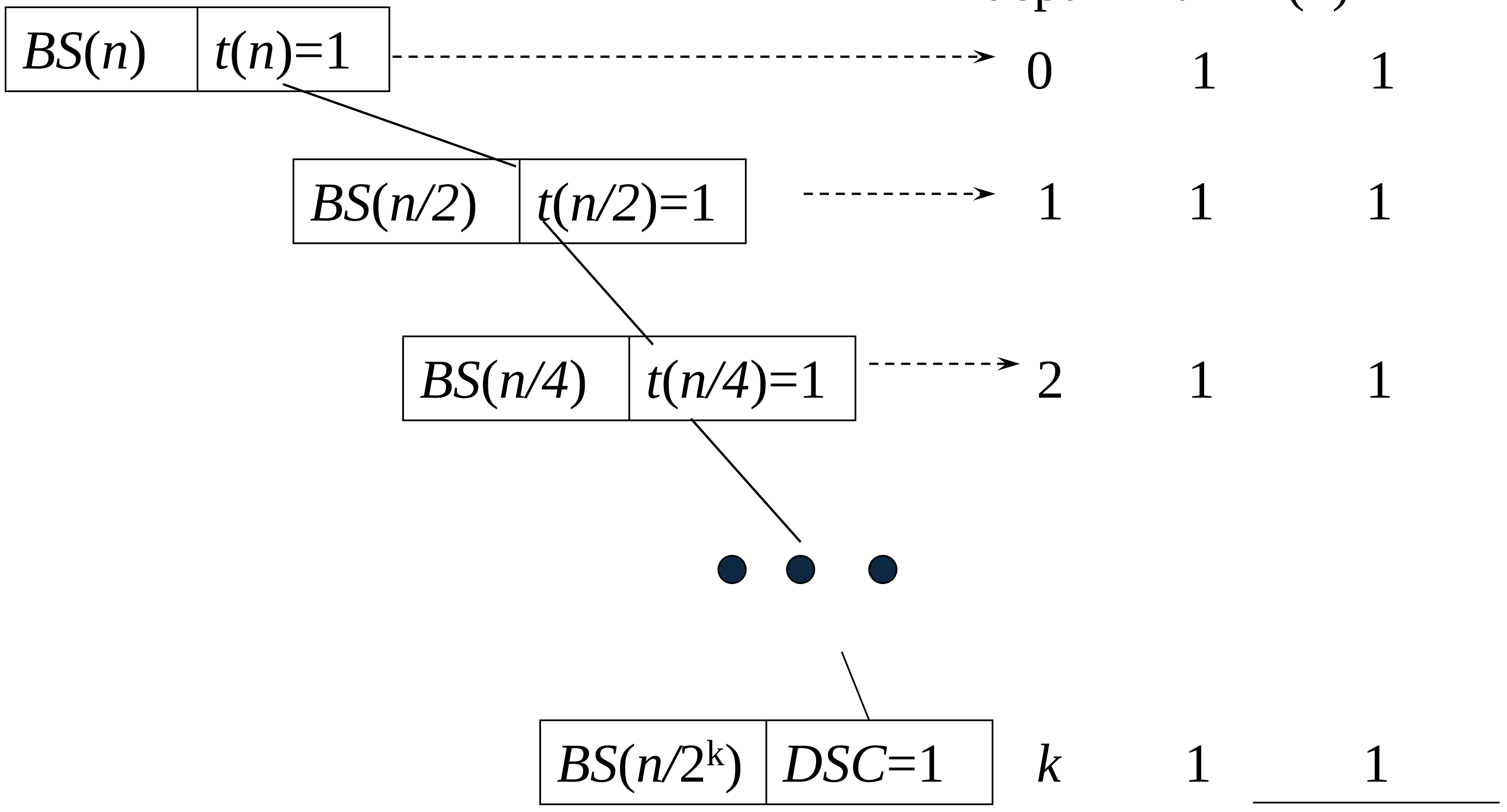


For *BinarySearch*, *DirectSolutionSize* = 0 or 1
and *DirectSolutionCount* = 0 for 0 and 1 for 1

Terminating the unrolling

- Let $2^{k} \leq n < 2^{k+1}$
- $k = \lfloor \lg n \rfloor$
- When a node has a call to $BS(n/2^{k})$, (or to $BS(n/2^{k+1})$):
  - The size of the list is *DirectSolutionSize* since $\lfloor n/2^{k} \rfloor = 1$, (or $\lfloor n/2^{k+1} \rfloor = 0$)
  - In this case the unrolling terminates, and the node is a leaf containing *DirectSolutionCount* (*DSC*) = 1, (or 0)

The recursion tree



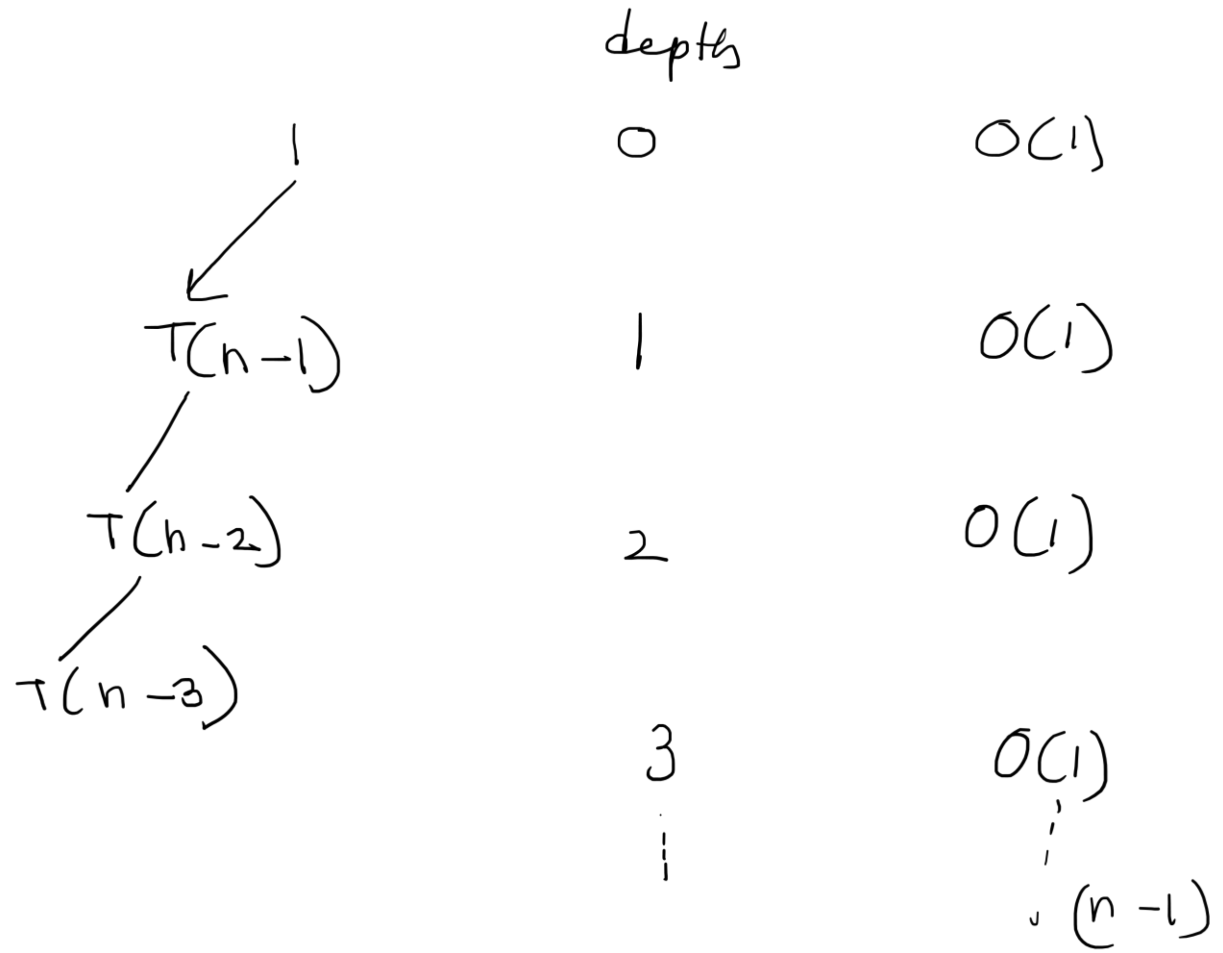
depth	nd	$T(n)$
0	1	1
1	1	1
2	1	1
k	1	1

$T(n)=k+1=\lfloor \lg n \rfloor +1$

Solving using Recursion Tree method

- $T(n) = T(n-1) + 1 : n > 0$
- $T(n) = 1 : n = 0$

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



$$\underbrace{[0 \ldots (n-1)]}_{n} * O(1)$$

$$\underline{\underline{O(n)}}$$

Solving using Substitution method

- $T(n) = T(n-1) + 1 : n > 0$
- $T(n) = 1 : n = 0$

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

$$T(n) = T(n-1) + 1$$
$$= T(n-2) + 1 + 1$$
$$= T(n-2) + 2$$
$$\vdots \quad k$$
$$= T(n-k) + k$$

assume $n - k = 0$;

$$= T(0) + n = 1 + n = \Theta(n)$$

Masters Theorem

- Let T(n) be <u>a monotonically increasing</u> function that satisfies

  $$T(n) = a\,T(n/b) + f(n)$$

  $$T(1) = c$$

  where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

When Master’s Theorem Does not Apply

- Master’s Theorem cannot be used in certain cases:

1. If f(n) is not polynomially related to $n^{log_b^a}$ ($if\ f(n) involves\ with\ irregular\ functions\ like\ logarithms\ or\ exponential\ terms$)
2. If the recurrence relation does not fit the required form.

Other methods such as recursion tree, substitution method can be used.

Master method examples

- Case 1:
  - $T(n) = 8T(n/4) + 5n^2$ for $n>1$, n is a a power of 4
  - $T(1) = 3$

→ $a=8$, $b=4$, $d=2$

→ As $a < b^d$ (i.e., $8 < 4^2$), $T(n) = \Theta(n^2)$

Master method examples

- Case 2:
  - $T(n) = 8T(n/2) + 5n^3$ for $n>64$, n is a power of 2
  - $T(64) = 200$

$\rightarrow$As $a = b^k$ (i.e., $8 = 2^3$), $T(n) = \Theta(n^3 \lg n)$

Master method examples

- Case 3:
  - T(n) = 9T(n/3) + 5n for n > 1, n is a power of 3
  - T(1) = 7

→a = 9, b = 3, d = 1

→Since a > b^d ,

$$T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$$

Thank you

See you next week with more problems