Foundations of Algorithm SCS1020

Dr. Dinuni Fernando PhD

Senior Lecturer

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

What is n! ?

- Can you compute 5! ?

- Can you write a simple code to compute n! ?

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

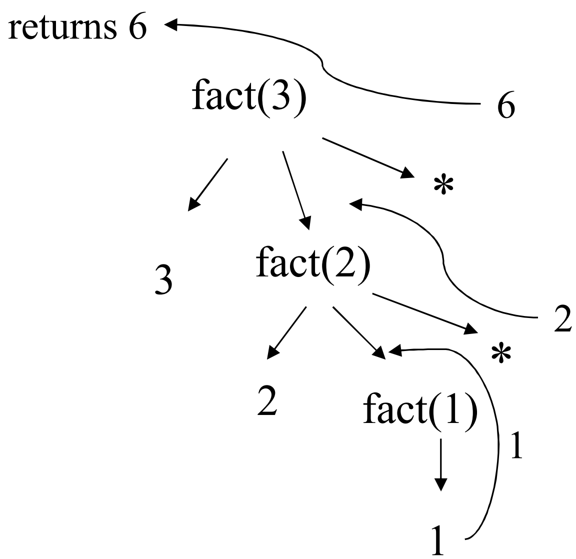
```
fact(n) {
 if (n<=1) return 1;
 else return n*fact(n-1);
}
```

// 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).*

#operations done by base case

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

Size of the base case

Recursive + non-recursive operation count

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*

$$T(size) = \begin{cases} DirectSolCount & size \leq DirectSolSize \\ \mathrm{Re}cursiveCallSum + t(size) & \text{otherwise} \end{cases}$$

```
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 $\theta(1)$

   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)* = θ`(1) (if, *, -, return)`

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

Reminder: Pitfall

- Is the complexity of recursive factorial $\Theta(n)$? Is it a linear time algorithm then?
- No! Recall for algorithms handling arbitrarily big numbers, we need to consider the input size in terms of the number of bits needed to express the input
- In recursive factorial, we have *only one input data, n, but the counts are proportional to the magnitude of n ≥ 2^{s-1} where s is the number of bits used to express n*
- So, it has *exponential* time complexity! $(s = \lfloor \lg n \rfloor + 1)$

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, $T(n\text{-}1)$.

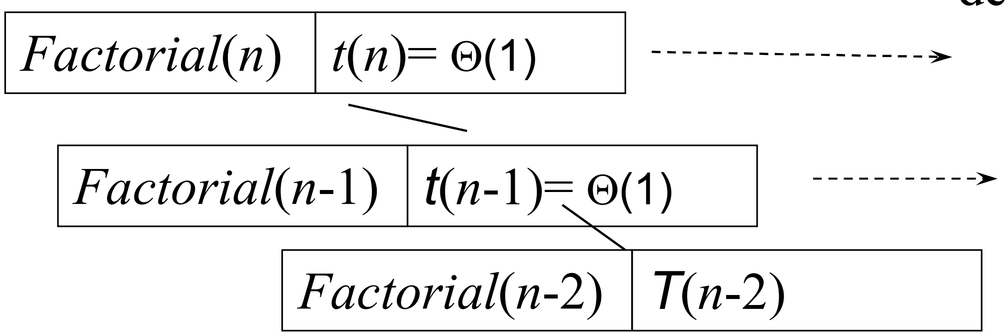
After the first unrolling

Factorial(n)	$t(n) = \Theta(1)$	depth	*nd*	$T(n)$
		0	1	$\Theta(1)$
Factorial(n-1)	$T(n\text{-}1)$	1	1	$T(n\text{-}1)$

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	$T(n\text{-}2)$

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

After the third unrolling

		depth	nd	$T(n)$
Factorial(n)	$t(n) = \Theta(1)$	0	1	$\Theta(1)$
Factorial(n-1)	$t(n\text{-}1) = \Theta(1)$	1	1	$\Theta(1)$
Factorial(n-2)	$t(n\text{-}2) = \Theta(1)$	2	1	$\Theta(1)$
Factorial(n-3)	$T(n\text{-}3)$	3	1	$T(n\text{-}3)$

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

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

The recursion tree

		depth	nd	$T(n)$
$Factorial(n)$	$t(n)=\Theta(1)$	0	1	$\Theta(1)$
$Factorial(n\text{-}1)$	$t(n\text{-}1)=\Theta(1)$	1	1	$\Theta(1)$
$Factorial(n\text{-}2)$	$t(n\text{-}2)=\Theta(1)$	2	1	$\Theta(1)$
$Factorial(n\text{-}3)$	$t(n\text{-}3)=\Theta(1)$	3	1	$\Theta(1)$
$Factorial(1)$	$T(1)=\Theta(1)$	n-1	1	$T(1)=\Theta(1)$

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

Iteration for binary search

$$W(n) = 1 + W(\lfloor n/2 \rfloor)$$
$$= 1 + (1 + W(\lfloor \lfloor n/2 \rfloor / 2 \rfloor)) = 2 + W(\lfloor n/4 \rfloor)$$
$$= 2 + (1 + W(\lfloor n/8 \rfloor)) = 3 + W(\lfloor n/8 \rfloor)$$
$$\ldots$$
$$= k + W(\lfloor n/2^k \rfloor) = k + W(1) = k + 1$$
$$= \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$$

Divide and Conquer

- Basic idea: divide a problem into smaller portions, solve the smaller portions and combine the results (if necessary).
- Name some algorithms you already know that employ this technique.
- D&C is a **top down** approach. We often use recursion to implement D&C algorithms.
- The following is an "outline" of a divide and conquer algorithm

Divide and Conquer

- Let $size(I) = n$
- $DirectSolutionCount = DS(n)$
- $t(n) = D(n) + C(n)$ *where:*
  - $D(n)$ = instruction counts for dividing problem into subproblems
  - $C(n)$ = instruction counts for combining solutions

$$T(n) = \begin{cases} DS(n) & \text{for } n \le DirectSolutionSize \\ \sum_{i=1}^{k} T(n_i) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Divide and Conquer

- Main advantages
  - Code: simple
  - Algorithm: efficient
  - Implementation
    - Parallel computation is possible, e.g., parallel quick sort, parallel merge sort, parallel convex hull, etc.
    - Parallel algorithm is an advanced topic. If we have time, we can discuss a few representative parallel algorithms in the end of the semester

Binary search

- Assumption: The list $S[low\ldots high]$ is sorted, and x is the search key
- If the search key x is in the list, x == S[i], and the index i is returned.
- If x is not in the list a *NoSuchKey* is returned

Binary search

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

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

- The other cases
  $x\in S[low,..,mid-1]$, or $x\in S[mid+1,..,high]$ 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 $2^k \leq n < 2^{k+1}$ $k = \lg n$

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

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

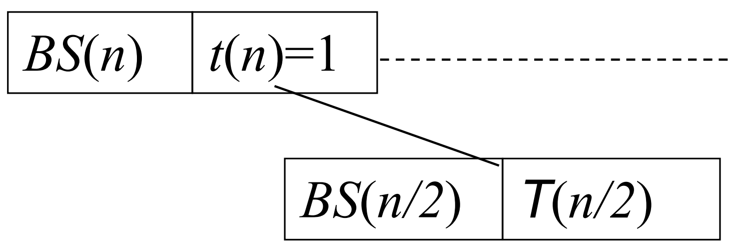
Recursion tree for BinarySearch (BS)

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

- 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$T(n/2)$.

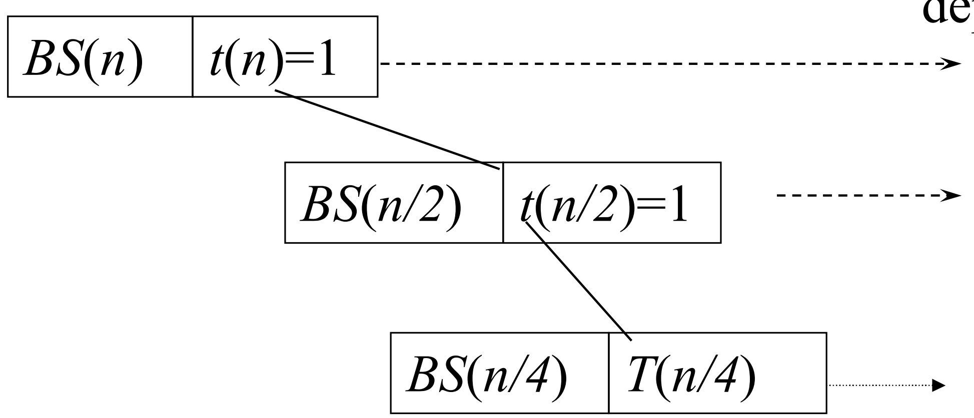
After first unrolling



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

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

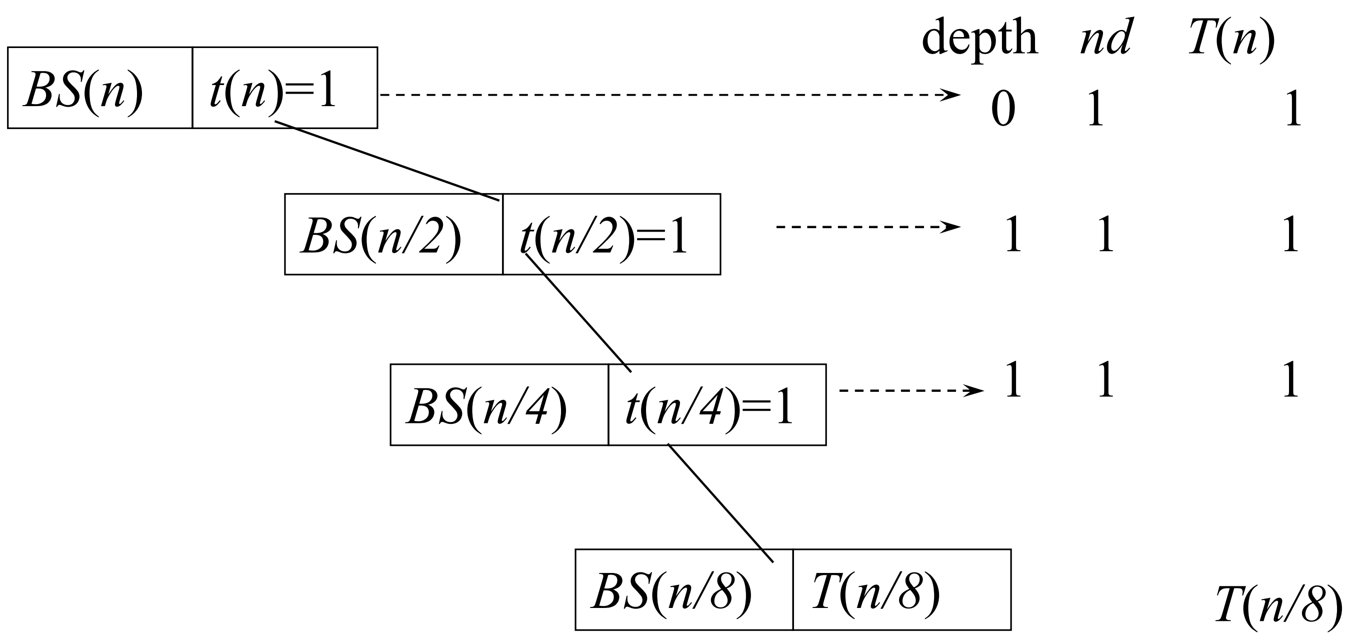
After second unrolling



depth	nd	$T(n)$
0	1	1
1	1	1
2	1	$T(n/4)$

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

After third unrolling

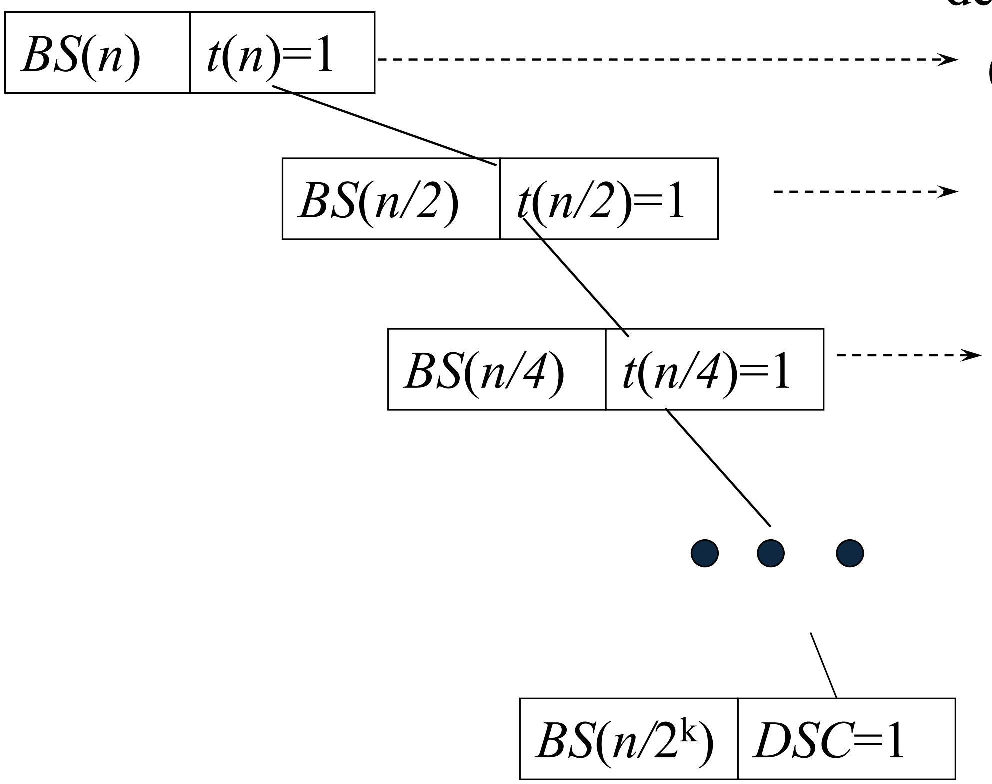


For *BinarySearch, DirectSolutionSize* = 0 or 1
and *DirectSolutionCount* = 0 for 0 and 1 for 1

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

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

Merge Sort

Input: S of size n.
Output: a permutation of S, such that if $i > j$ then $S[\, i\,] \geq S[\, j\,]$
Divide: If S has at least 2 elements, divide it into S_1 *and* S_2. S_1 contains the the first $\lceil n/2 \rceil$ elements of S. S_2 has the last $\lfloor n/2 \rfloor$ elements of S.

Recursion: Recursively sort S_1 and S_2.

Conquer: Merge sorted S_1 and S_2 into S.

Merge Sort Example

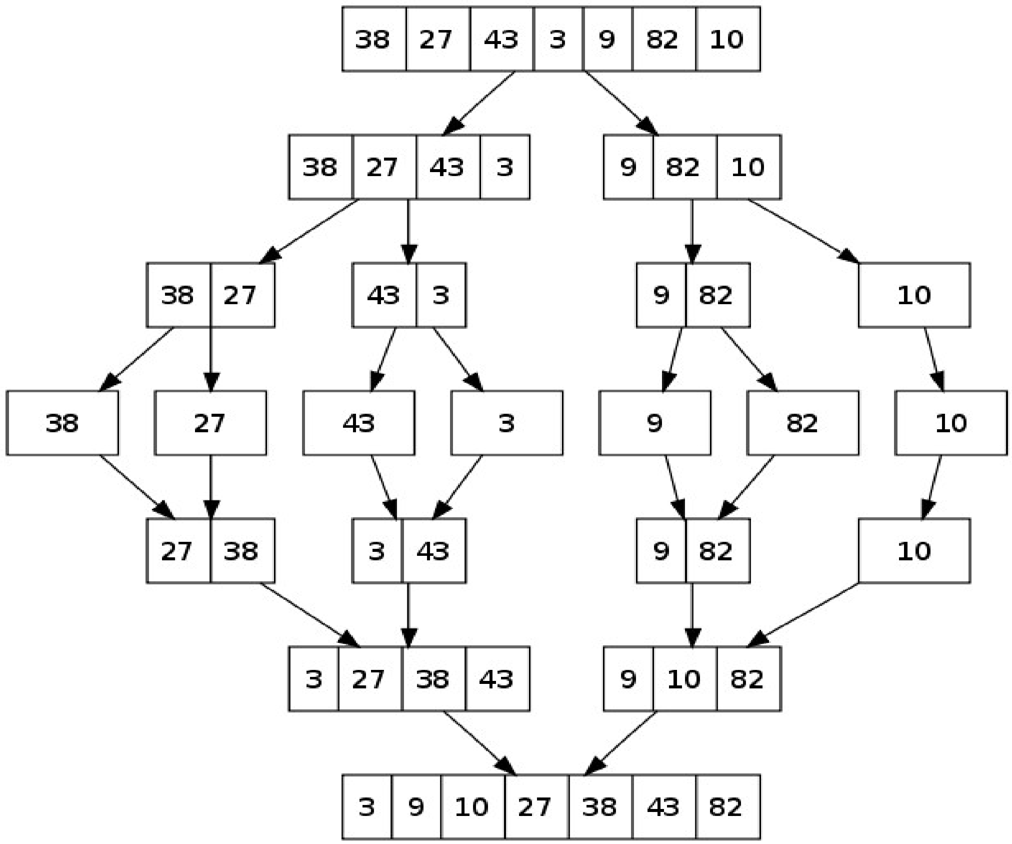


Image source: http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg

Deriving a recurrence equation for Merge Sort

```
Sort(S)
   if (n ≥ 2)
        // Divide S into S1 and S2
        Sort(S1)                 // recursion

        Sort(S2)                 // recursion
        Merge( S1, S2, S )       // conquer
```

DirectSolutionSize is n < 2

DirectSolutionCount is $\theta(1)$

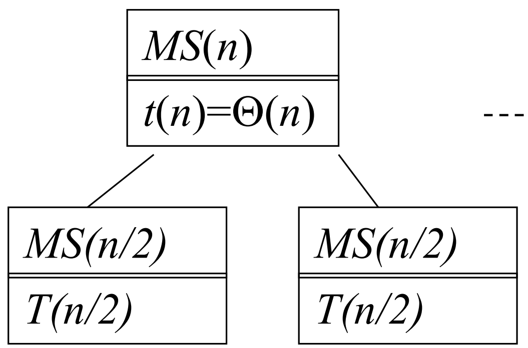
RecursiveCallSum is $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$

The non-recursive count $t(n) = \theta(n)$ ➔ Merge

Recurrence Equation (cont'd)

- The cost of division is O(1) and *merge* is $\Theta(n)$. So, the total cost for dividing and merging is $\Theta(n)$.
- The recurrence relation for the run time of MergeSort is:
  $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$.
  $T(0) = T(1) = \Theta(1)$
- The solution is $T(n) = \Theta(n \lg n)$

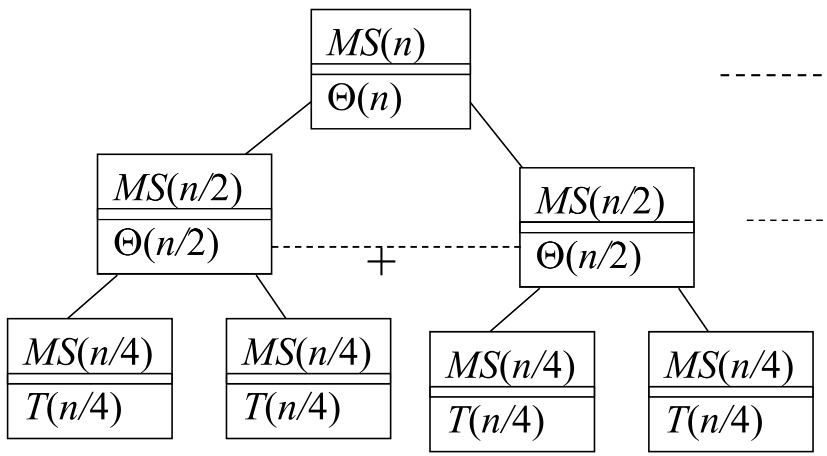
After first unrolling of mergeSort



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

$$T(n)=\begin{cases}1 \text{ for } n\le 1\\ 2T(n/2)+\theta(n) \text{ for } n>1\end{cases}$$

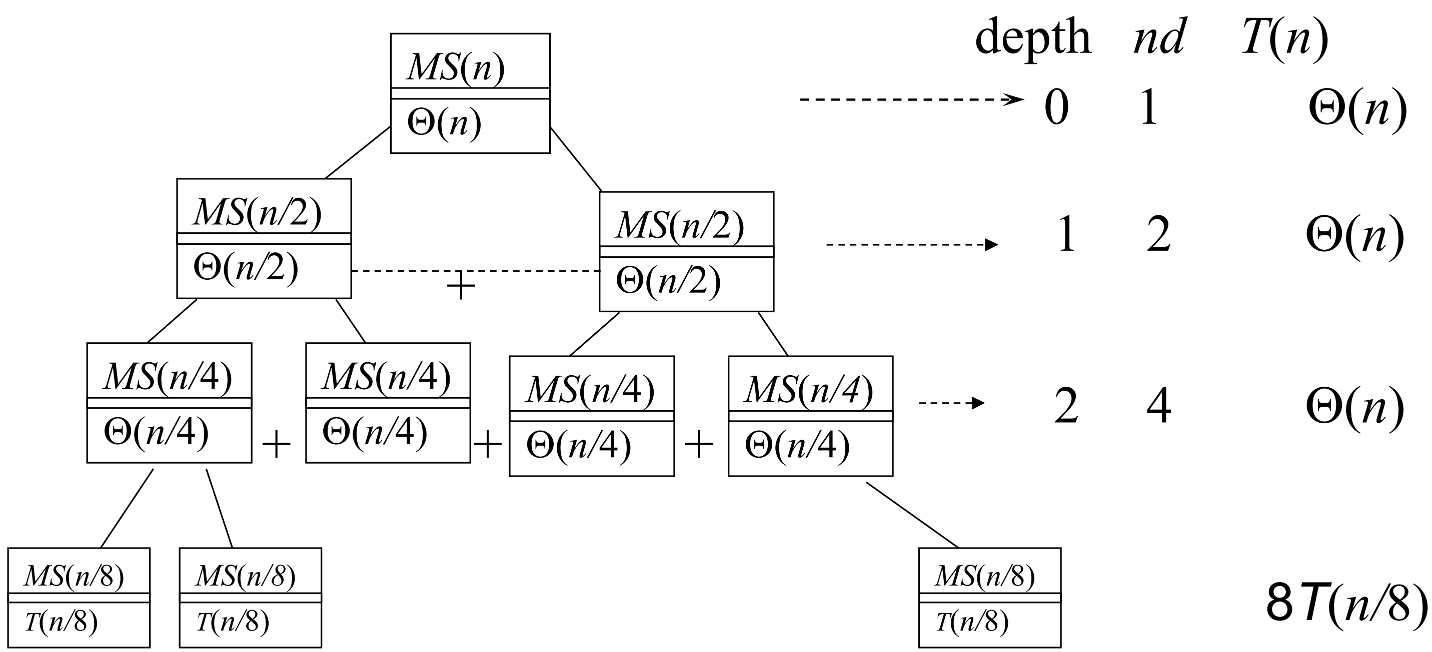
After second unrolling



depth	nd	$T(n)$
0	1	$\Theta(n)$
1	2	$\Theta(n)$
		$4T(n/4)$

$$T(n) = \begin{cases} 1 \ \text{ for } \ n \le 1 \\ 2T(n/2) + \theta(\mathrm{n}) \quad \text{for} \quad n > 1 \end{cases}$$

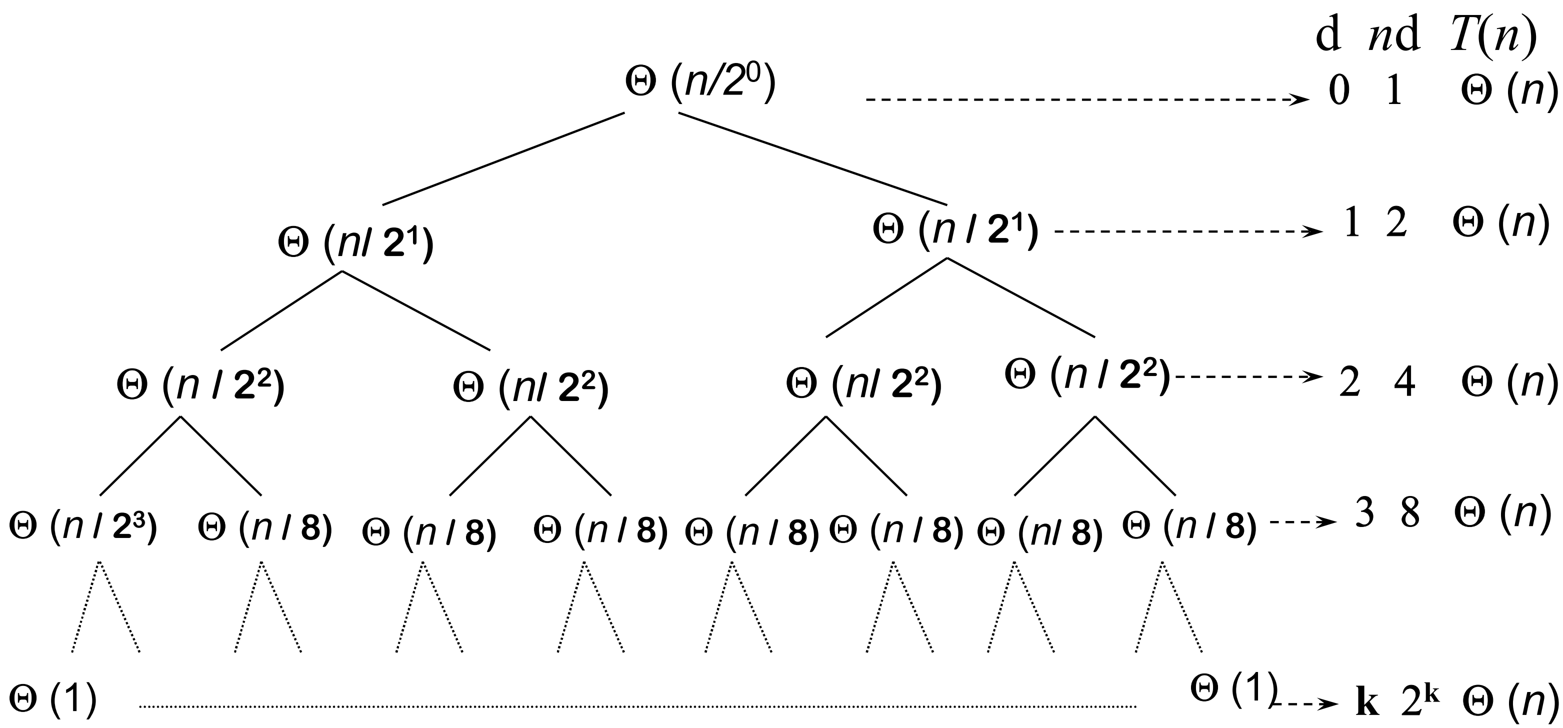
After third unrolling



Terminating the unrolling

- For simplicity let $n = 2^k$
- $\lg n = k$
- When a node has a call to *MergeSort*$(n/2^k)$:
  - The size of the list to merge sort is *DirectSolutionSize* since $n/2^k = 1$
  - In this case the unrolling terminates, and the node is a leaf containing *DirectSolutionCount* = $\Theta(1)$

The recursion tree



$$T(n) = (k+1)\,\Theta(n) = \Theta(n \lg n)$$

Thank you!