# Design and Analysis of Algorithms
# Lecture-3: Divide and Conquer

Prof. Eugene Chang

# Overview

- Analyzing recurrence
- Solving recurrence
- The master method
- Part of the slides are based on material from Prof. Jianhua Ruan, The University of Texas at San Antonio

# Analyzing recursive algorithms

# Recursive algorithms

- General idea:
  - Divide a large problem into smaller ones
    - By a constant ratio
    - By a constant or some variable
  - Solve each smaller one *recursively* or *explicitly*
  - Combine the solutions of smaller ones to form a solution for the original problem

Divide and Conquer

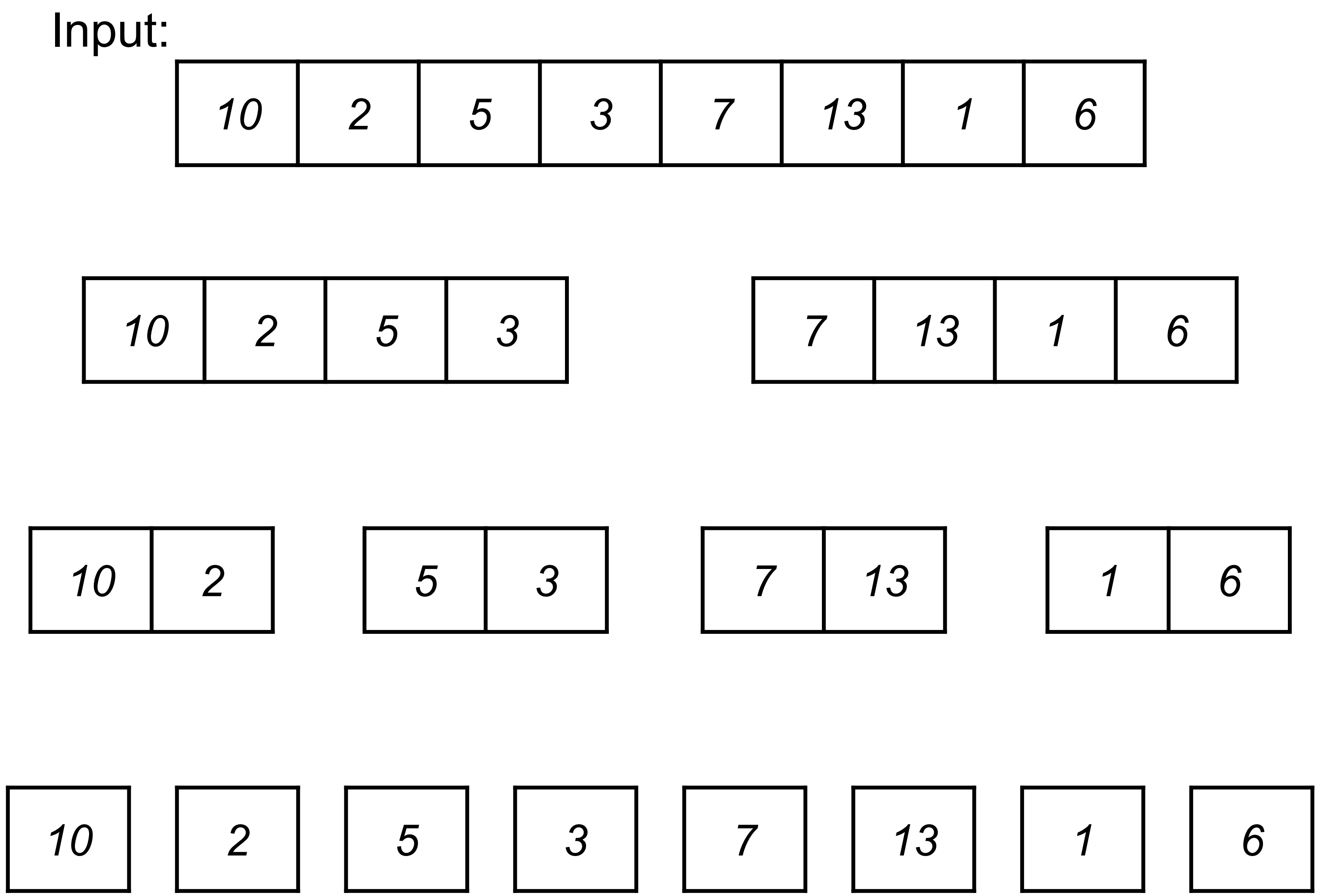
# Merge sort

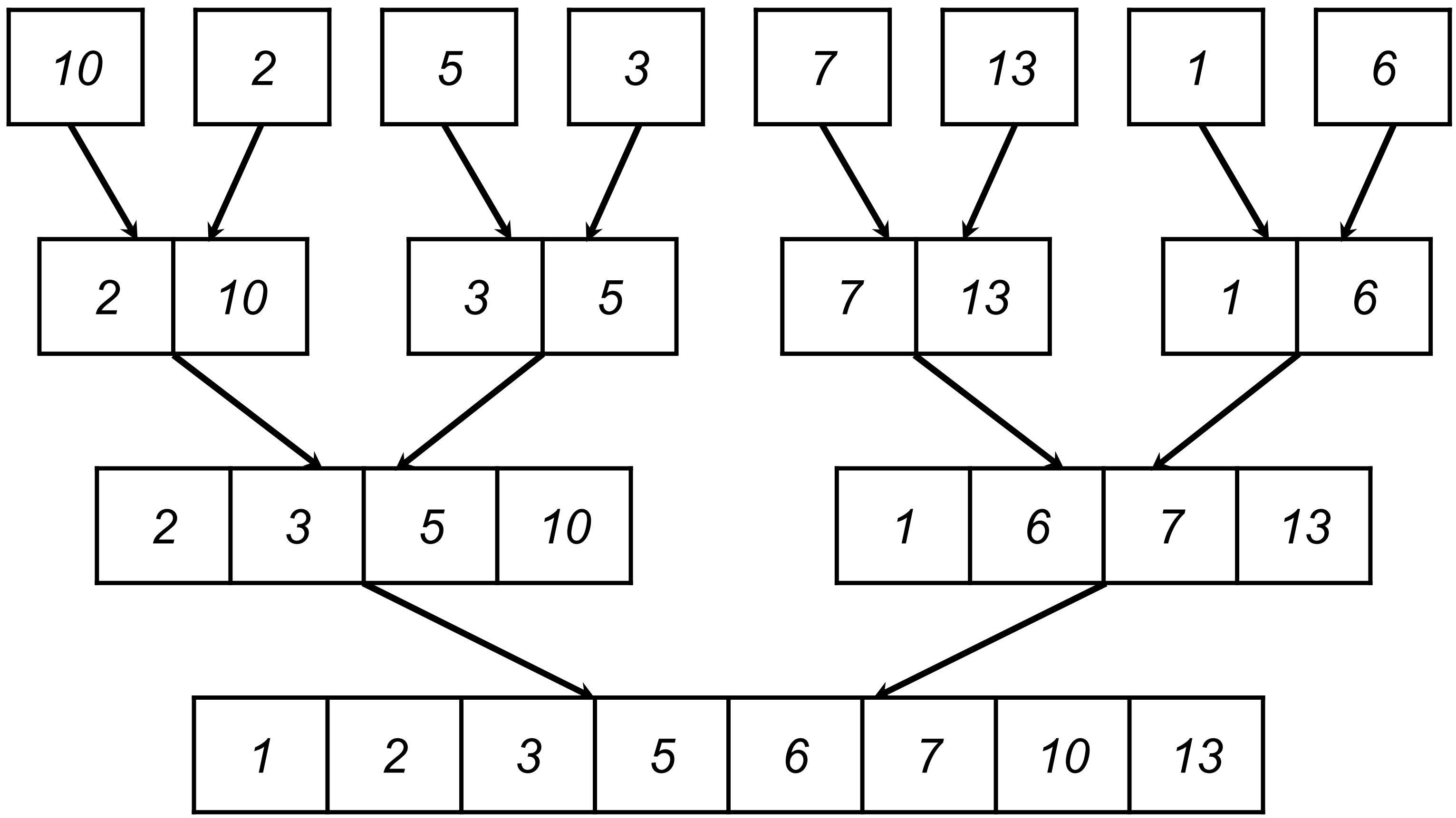
**MERGE-SORT** $A[1 \,.\,.\, n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$ .
3. "***Merge***" the 2 sorted lists.

***Key subroutine:*** **MERGE**

The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy.


Input:
10 2 5 3 7 13 1 6
10 2 5 3
7 13 1 6
10 2
5 3
7 13
1 6
10
2
5
3
7
13
1
6


10
2
5
3
7
13
1
6
2
10
3
5
7
13
1
6
2
3
5
10
1
6
7
13
1
2
3
5
6
7
10
13

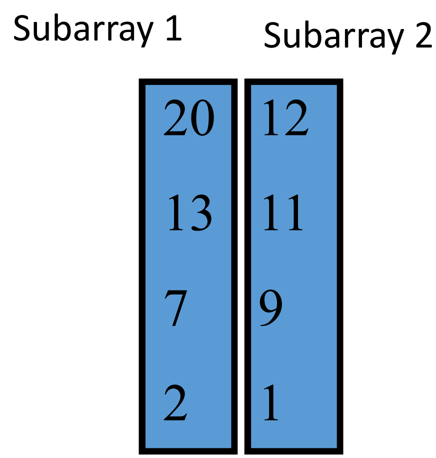
```
function mergesort(a[1...n])

if n > 1:
    return merge(mergesort(a[1...⌊n/2⌋]),
                 mergesort(a[⌊n/2⌋+ 1...n]))
else:
    return a
```

```
function merge(x[1...k], y[1...m])

if k = 0: return y[1...m]
if m = 0: return x[1...k]

if x[1] ≤ y[1]:
    return x[1] ✪ merge(x[2...k],
    y[1...m])
else:
    return y[1] ✪ merge(x[1...k],
    y[2...m])

    where ✪ is concatenation.
```

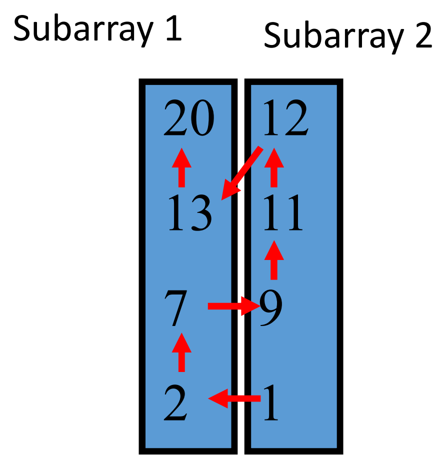
```
function merge(x[1...k], y[1...m])

if k = 0: return y[1...m]
if m = 0: return x[1...k]

if x[1] ≤ y[1]:
    return x[1] ✪ merge(x[2...k],
    y[1...m])
else:
    return y[1] ✪ merge(x[1...k],
    y[2...m])
```

This merge procedure does a constant amount of work per recursive call (provided the required array space is allocated in advance), for a total running time of $O(k + m)$.

# Merging two sorted arrays



# Merging two sorted arrays



# Merging two sorted arrays

| | |
|---|---|
| 20 | 12 |
| 13 | 11 |
| 7 | 9 |
| 2 | 1 |

# Merging two sorted arrays

| | |
|---|---|
| 20 | 12 |
| 13 | 11 |
| 7 | 9 |
| 2 | 1 |

# Merging two sorted arrays

| | |
|---|---|
| 20 | 12 |
| 13 | 11 |
| 7 | 9 |
| 2 | 1 |

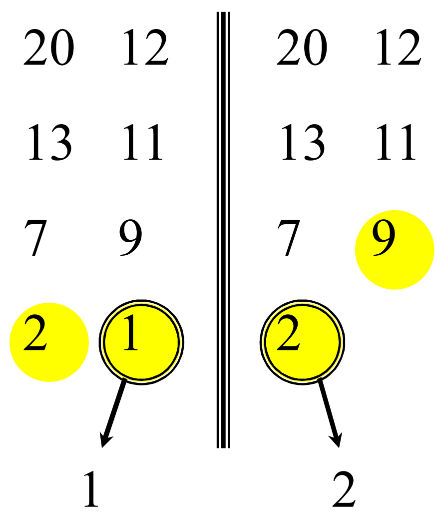
1

# Merging two sorted arrays

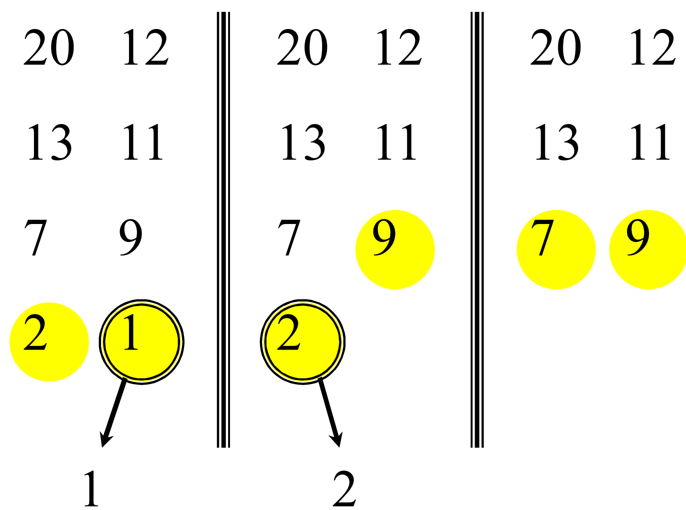
| | | | | |
|---|---|---|---|---|
| 20 | 12 | ‖ | 20 | 12 |
| 13 | 11 | ‖ | 13 | 11 |
| 7 | 9 | ‖ | 7 | 9 |
| 2 | 1 | ‖ | 2 | |

1

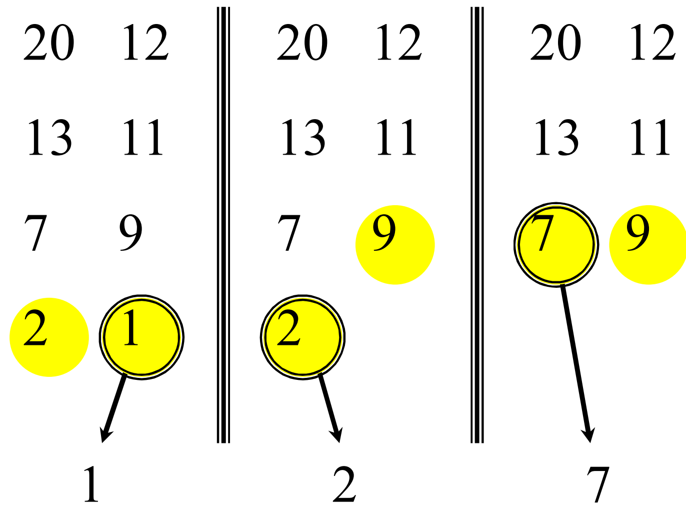
# Merging two sorted arrays



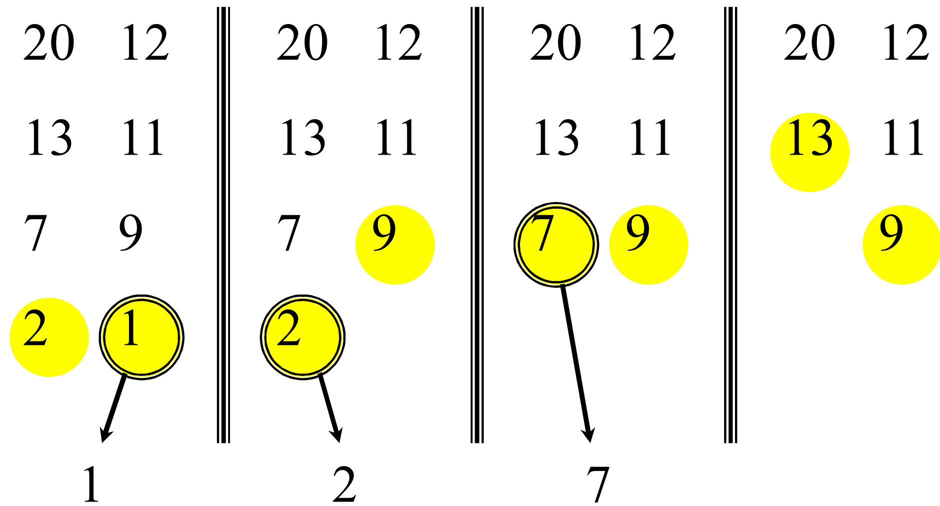
# Merging two sorted arrays



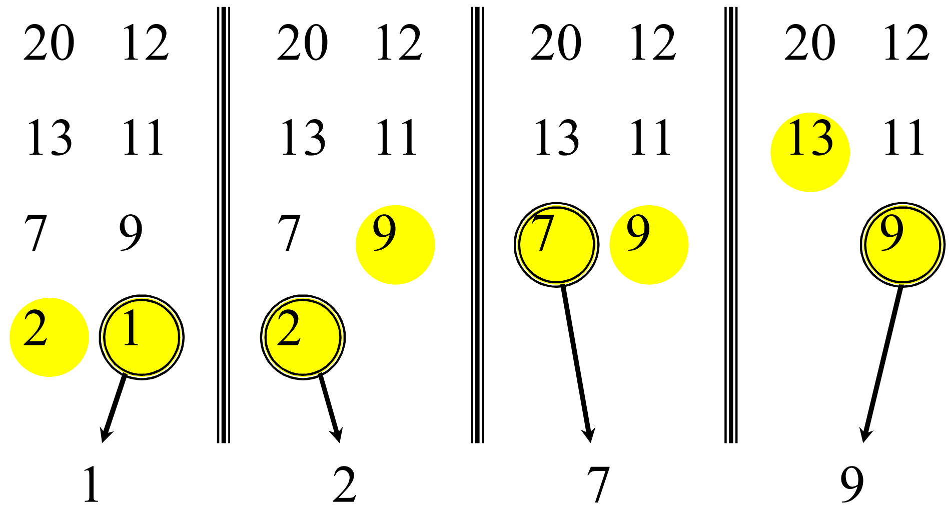
# Merging two sorted arrays



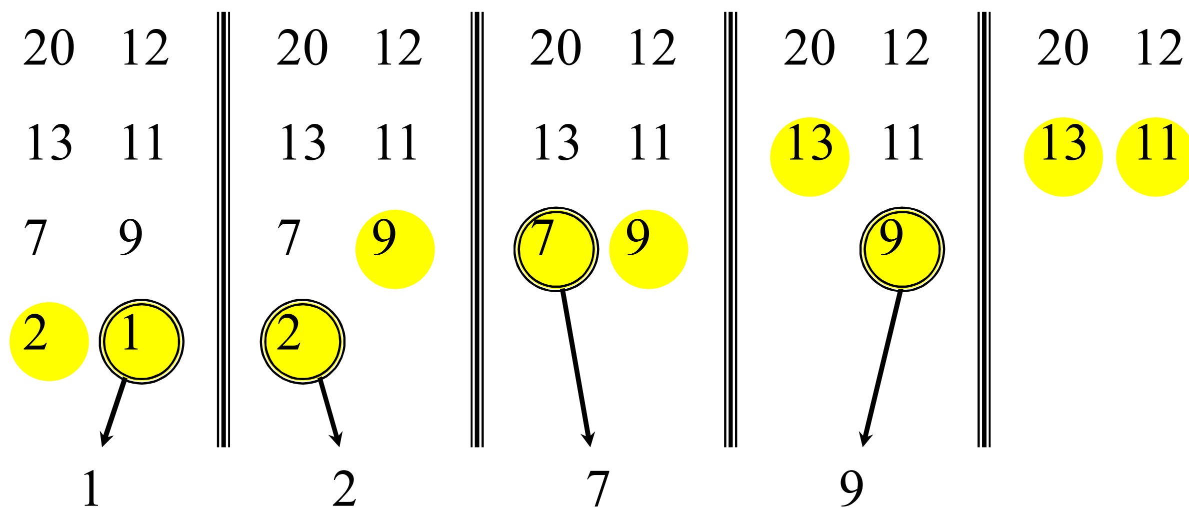
# Merging two sorted arrays



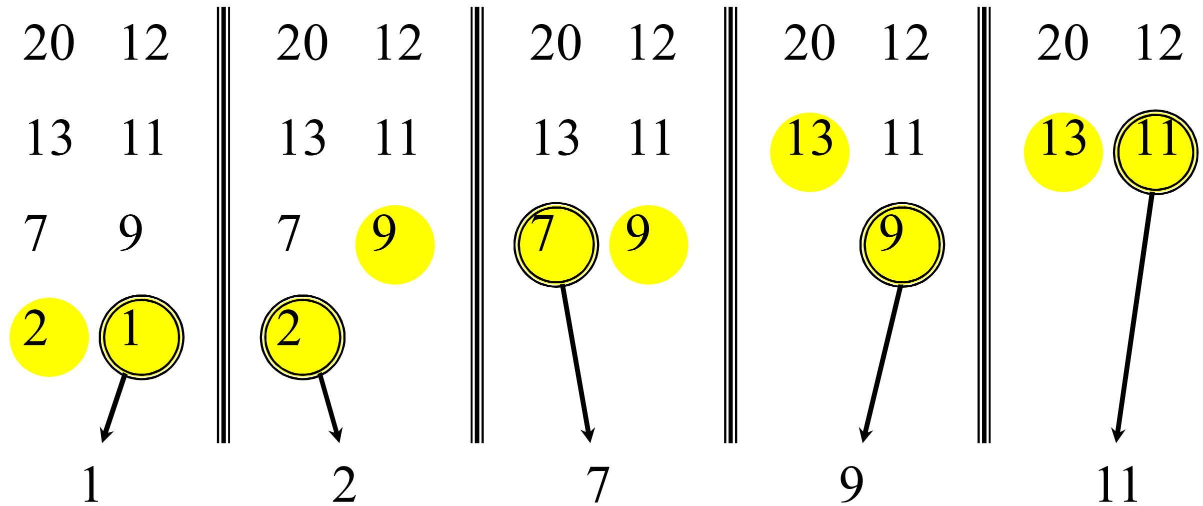
# Merging two sorted arrays



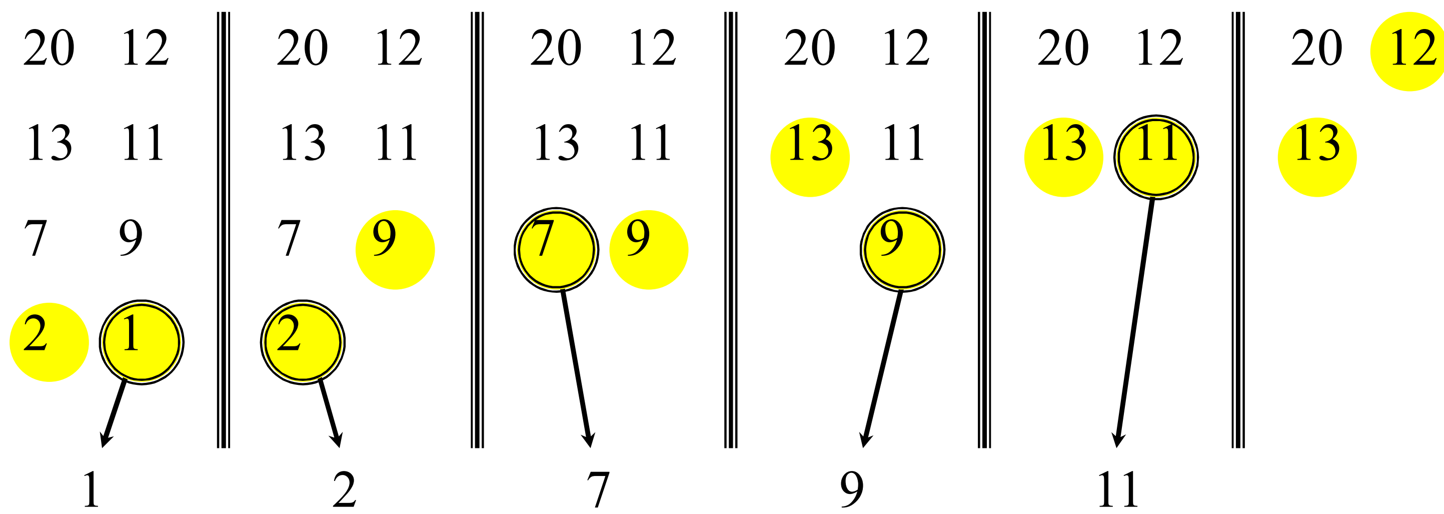
# Merging two sorted arrays



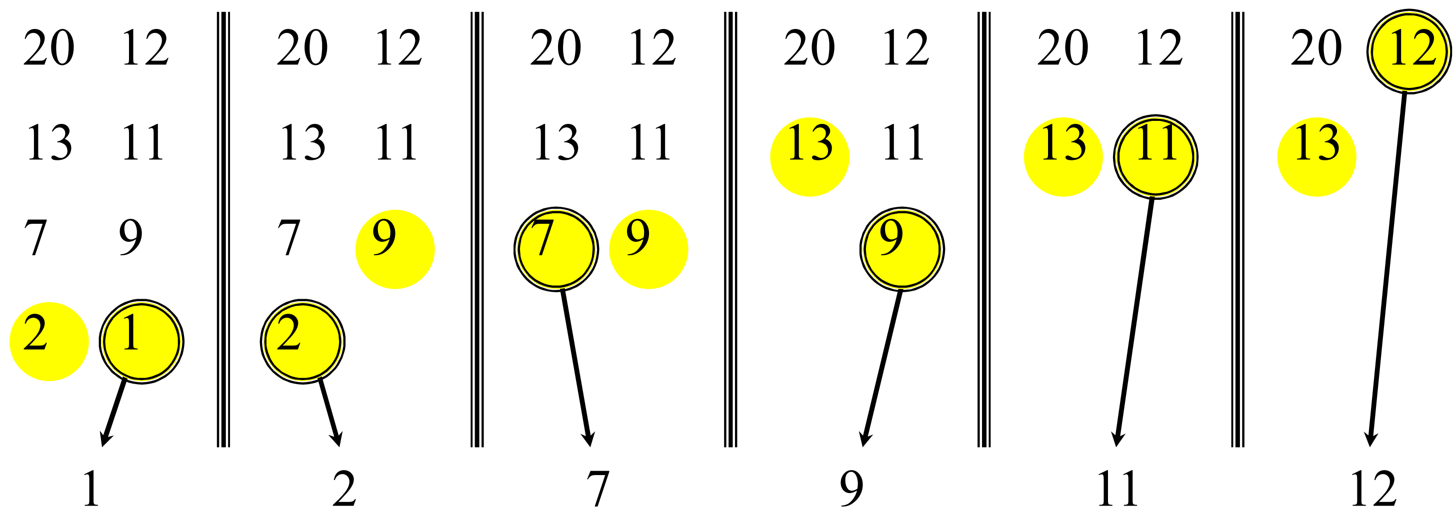
# Merging two sorted arrays



# Merging two sorted arrays



# Merging two sorted arrays



## How to show the correctness of a recursive algorithm?

- By induction:
  - Base case: prove it works for small examples
  - Inductive hypothesis: assume the solution is correct for all sub-problems
  - Step: show that, if the inductive hypothesis is correct, then the algorithm is correct for the original problem.

# Correctness of merge sort

**MERGE-SORT** $A[1 \,..\, n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$.
3. "***Merge***" the 2 sorted lists.

***Proof:***

1. Base case: if n = 1, the algorithm will return the correct answer because A[1..1] is already sorted.
2. Inductive hypothesis: assume that the algorithm correctly sorts $A[1..\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil+1..n]$.
3. Step: if $A[1..\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil+1..n]$ are both correctly sorted, the whole array $A[1..\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil+1..n]$ is sorted after merging.

# Analyzing merge sort

| | |
|---|---|
| $T(n)$ | **MERGE-SORT** $A[1 \,..\, n]$ |
| $\Theta(1)$ | 1. If $n = 1$, done. |
| $2T(n/2)$ | 2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$ . |
| $f(n)$ | ***3. "Merge"*** the 2 sorted lists |

***Sloppiness:*** Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$ , but it turns out not to matter asymptotically.

# Analyzing merge sort

1. ***Divide:*** Trivial.
2. ***Conquer:*** Recursively sort 2 subarrays.
3. ***Combine:*** Merge two sorted subarrays

$$T(n) = 2\,T(n/2) + f(n) + \Theta(1)$$

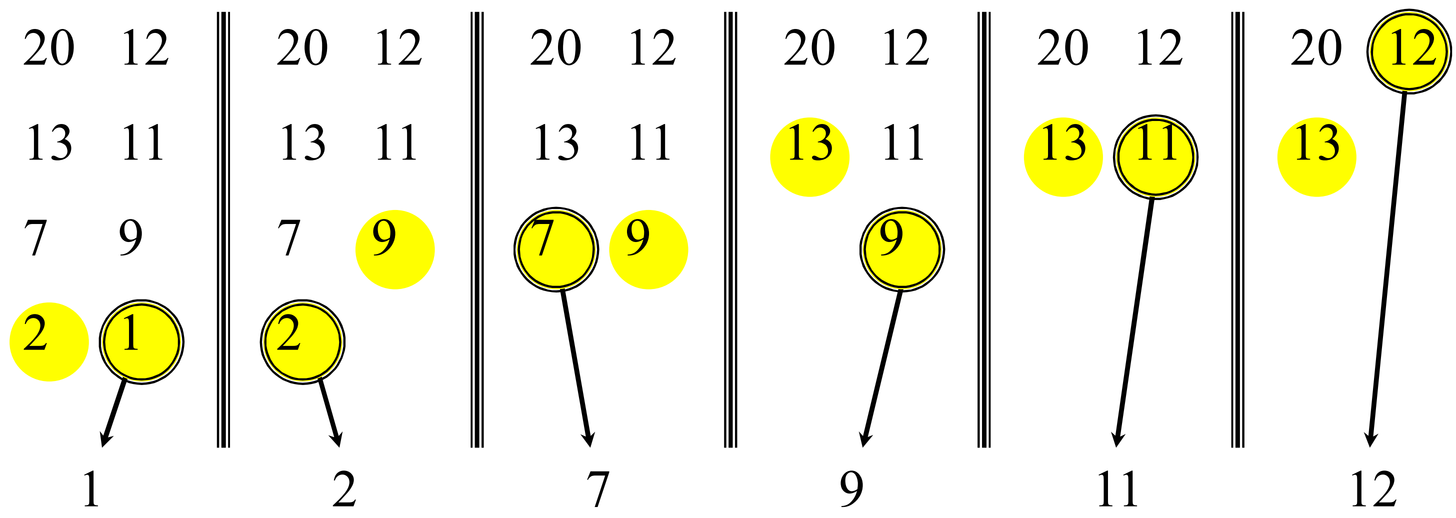
*# subproblems*

*subproblem size*

*Dividing and Combining*

1. What is the time for the base case? Constant
2. What is $f(n)$?
3. What is the growth order of $T(n)$?

# Merging two sorted arrays




$\Theta(n)$ time to merge a total of $n$ elements (linear time).

# Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- Later we shall often omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small $n$, but only when it has no effect on the asymptotic solution to the recurrence.

- But what does $T(n)$ solve to? I.e., is it $O(n)$ or $O(n^2)$ or $O(n^3)$ or …?

# How to analyze the time-efficiency of a recursive algorithm?

- Express the running time on input of size n as a function of the running time on smaller problems

*T(n) = 2T(n/2) + O(n).*

$$T(n) = a * T(n/b) + d * f(n)$$

*a = 2, b = 2, d = 1* ➠ *O(n log n)*

# Solving recurrence

1. Recursion tree / iteration method
2. Substitution method
3. Master method

# Binary Search

```
BinarySearch (A[1..N], value) {
   if (N == 0)
         return -1;              // not found
   mid = (1+N)/2;
   if (A[mid] == value)
         return mid;             // found
   else if (A[mid] > value)
         return BinarySearch (A[1..mid-1], value);
   else
         return BinarySearch (A[mid+1, N], value)
   }
```

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

# Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

***Example:*** Find 9

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 5 | 7 | 8 | 9 | 12 | 15 |

# Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

***Example:*** Find 9

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 5 | 7 | 8 | 9 | 12 | 15 |

# Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

***Example:*** Find 9

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 5 | 7 | 8 | 9 | 12 | 15 |

# Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

***Example:*** Find 9

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 5 | 7 | 8 | 9 | 12 | 15 |

# Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

***Example:*** Find 9

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 5 | 7 | 8 | 9 | 12 | 15 |

# Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

***Example:*** Find 9

3 5 7 8 9 12 15

# Binary Search

```
BinarySearch (A[1..N], value) {
   if (N == 0)
         return -1;                 // not found
   mid = (1+N)/2;
   if (A[mid] == value)
         return mid;                // found
   else if (A[mid] < value)
         return BinarySearch (A[mid+1, N], value)
   else
         return BinarySearch (A[1..mid-1], value);
}
```

What's the recurrence relation for its running time?

# Recurrence for binary search

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

$$T(1) = \Theta(1)$$

# Recursive Insertion Sort

***RecursiveInsertionSort***(A[1..n])

1. if (n == 1) do nothing;
2. ***RecursiveInsertionSort***(A[1..n-1]);
3. Find index *i* in A such that A[i] <= A[n] < A[i+1];
4. Insert A[n] after A[i];

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

# Recursive Insertion Sort

***RecursiveInsertionSort***(A[1..n])

1. if (n == 1) do nothing;
2. ***RecursiveInsertionSort***(A[1..n-1]);
3. Find index *i* in A such that A[i] <= A[n] < A[i+1];
4. Insert A[n] after A[i];

# Recurrence for insertion sort

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

$$T(1) = \Theta(1)$$

# Compute factorial

```
Factorial (n)
  if (n == 1) return 1;
  return n * Factorial (n-1);
```

- Note: here we use n as the size of the input. However, usually for such algorithms we would use log(n), i.e., the bits needed to represent n, as the input size.

# Compute factorial

***Factorial*** (n)

```
  if (n == 1) return 1;
  return n * Factorial (n-1);
```

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

- Note: here we use n as the size of the input. However, usually for such algorithms we would use log(n), i.e., the bits needed to represent n, as the input size.

# Recurrence for computing factorial

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

$$T(1) = \Theta(1)$$

- Note: here we use n as the size of the input. However, usually for such algorithms we would use log(n), i.e., the bits needed to represent n, as the input size.

# Power series

- How many multiplications do you need to compute $3^{16}$?

$3^{16} = 3 \times 3 \times 3 \ldots\ldots \times 3$

Answer: 15

$3^{16} = 3^8 \times 3^8$

$3^8 = 3^4 \times 3^4$

$3^4 = 3^2 \times 3^2$

$3^2 = 3 \times 3$

Answer: 4

# Pseudo code

```
int pow (b, n)    // compute b^n
  m = n >> 1;
  p = pow (b, m);
  p = p * p;
  if (n % 2)
        return p * b;
  else
        return p;
```

# Pseudo code

```
int pow (b, n)
   m = n >> 1;
   p = pow (b, m);
   p = p * p;
   if (n % 2)
        return p * b;
   else
        return p;
```

```
int pow (b, n)
  m = n >> 1;
  p = pow(b,m) * pow(b,m);
  if (n % 2)
        return p * b;
  else
        return p;
```

# Recurrence for computing power

```
int pow (b, n)
   m = n >> 1;
   p = pow (b, m);
   p = p * p;
   if (n % 2)
        return p * b;
   else
        return p;
```

T(n) = ?

```
int pow (b, n)
  m = n >> 1;
  p=pow(b,m)*pow(b,m);
  if (n % 2)
        return p * b;
  else
        return p;
```

T(n) = ?

# Recurrence for computing power

```
int pow (b, n)
   m = n >> 1;
   p = pow (b, m);
   p = p * p;
   if (n % 2)
        return p * b;
   else
        return p;
```

$T(n) = T(n/2)+\Theta(1)$

```
int pow (b, n)
   m = n >> 1;
   p=pow(b,m)*pow(b,m);
   if (n % 2)
        return p * b;
   else
        return p;
```

$T(n) = 2T(n/2)+\Theta(1)$

# What do they mean?

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

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

$$T(n) = T(n/2) + 1$$

$$T(n) = 2T(n/2) + 1$$

Challenge: how to solve the recurrence to get a closed form, e.g. $T(n) = \Theta(n^2)$ or $T(n) = \Theta(nlgn)$, or at least some bound such as $T(n) = O(n^2)$?

# Solving recurrence

- Running time of many algorithms can be expressed in one of the following two recursive forms

$$T(n) = aT(n-b) + f(n)$$

or

$$T(n) = aT(n/b) + f(n)$$

Both can be very hard to solve. We focus on relatively easy ones, which you will encounter frequently in many real algorithms (and exams…)

# Solving recurrence

1. Recursion tree or iteration method
   - Good for guessing an answer
2. Substitution method
   - Generic method, rigid, but may be hard
3. Master method
   - Easy to learn, useful in limited cases only
   - Some tricks may help in other cases

# Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ 2T(n/2) + \Theta(\mathrm{n}) & \text{if } n > 1. \end{cases}$$

We will usually ignore the base case, assuming it is always a constant (but not 0).

# Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.

# Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.

$$T(n)$$

# Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.

$$
\begin{array}{ccc}
 & dn & \\
\swarrow & & \searrow \\
T(n/2) & & T(n/2)
\end{array}
$$

# Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.

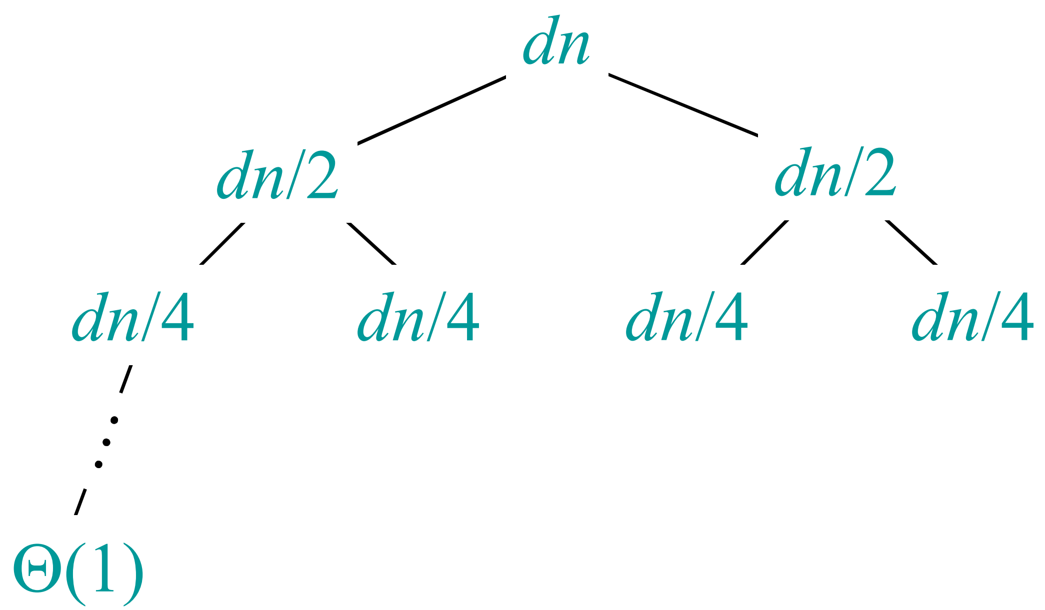
$$dn$$

$$dn/2 \qquad\qquad dn/2$$

$$T(n/4) \quad T(n/4) \quad T(n/4) \quad T(n/4)$$

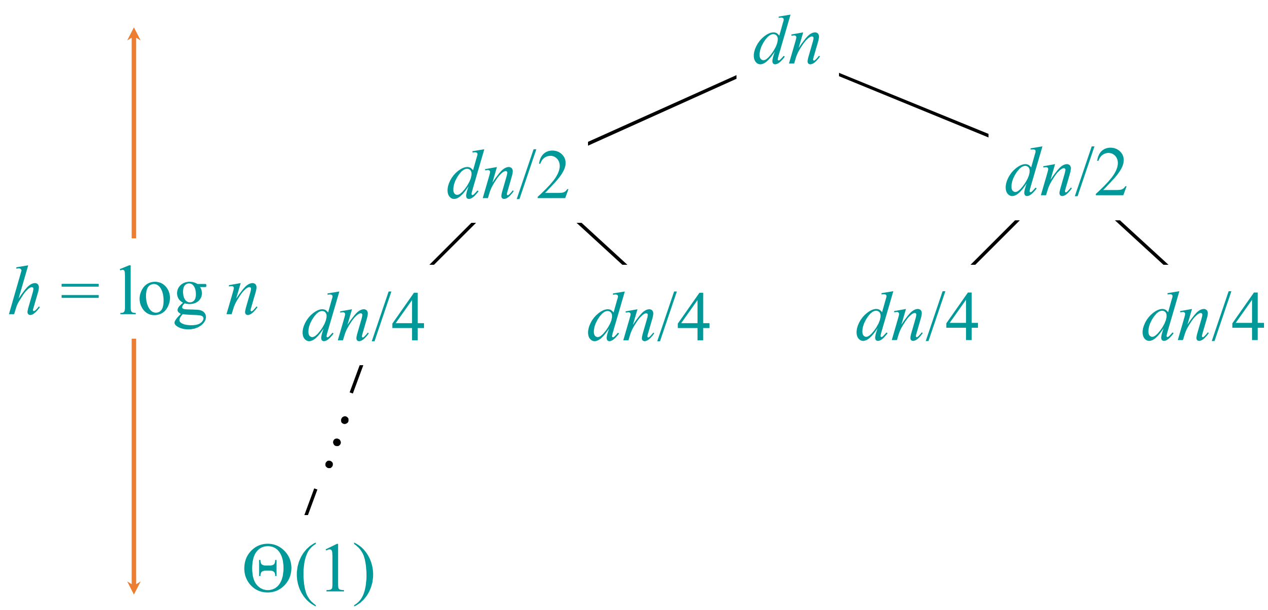
# Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.



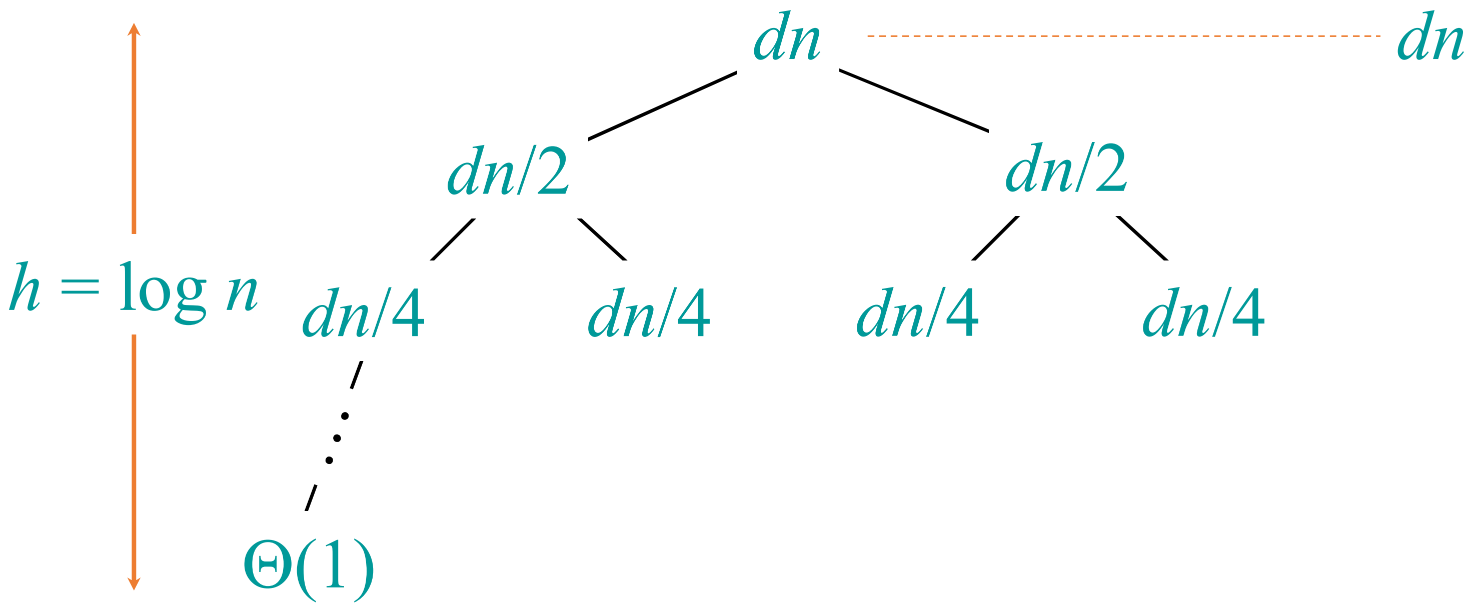
# Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.



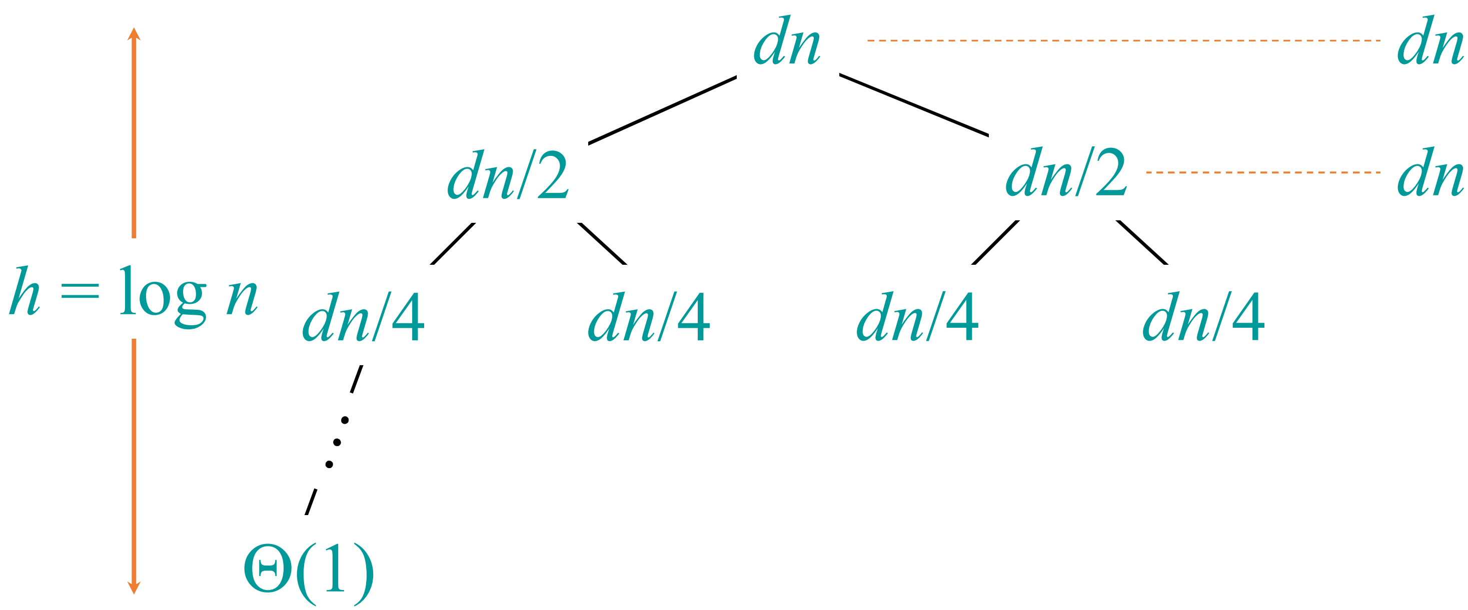
# Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.



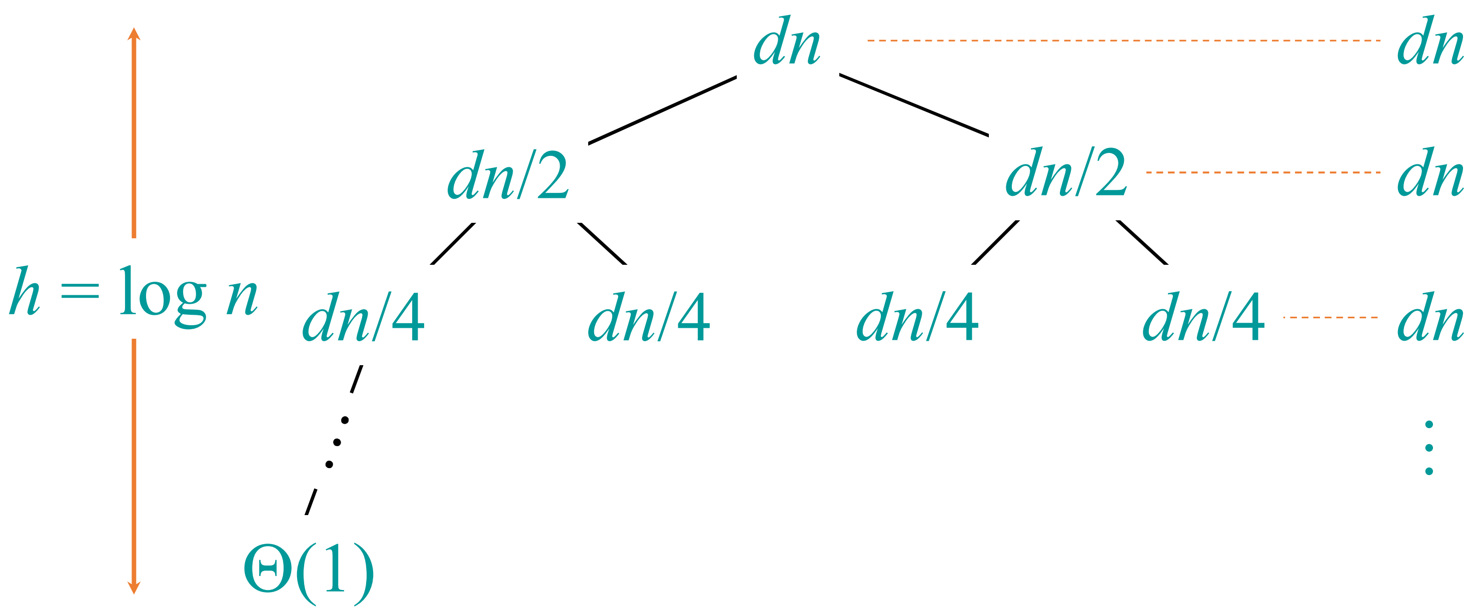
# Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.



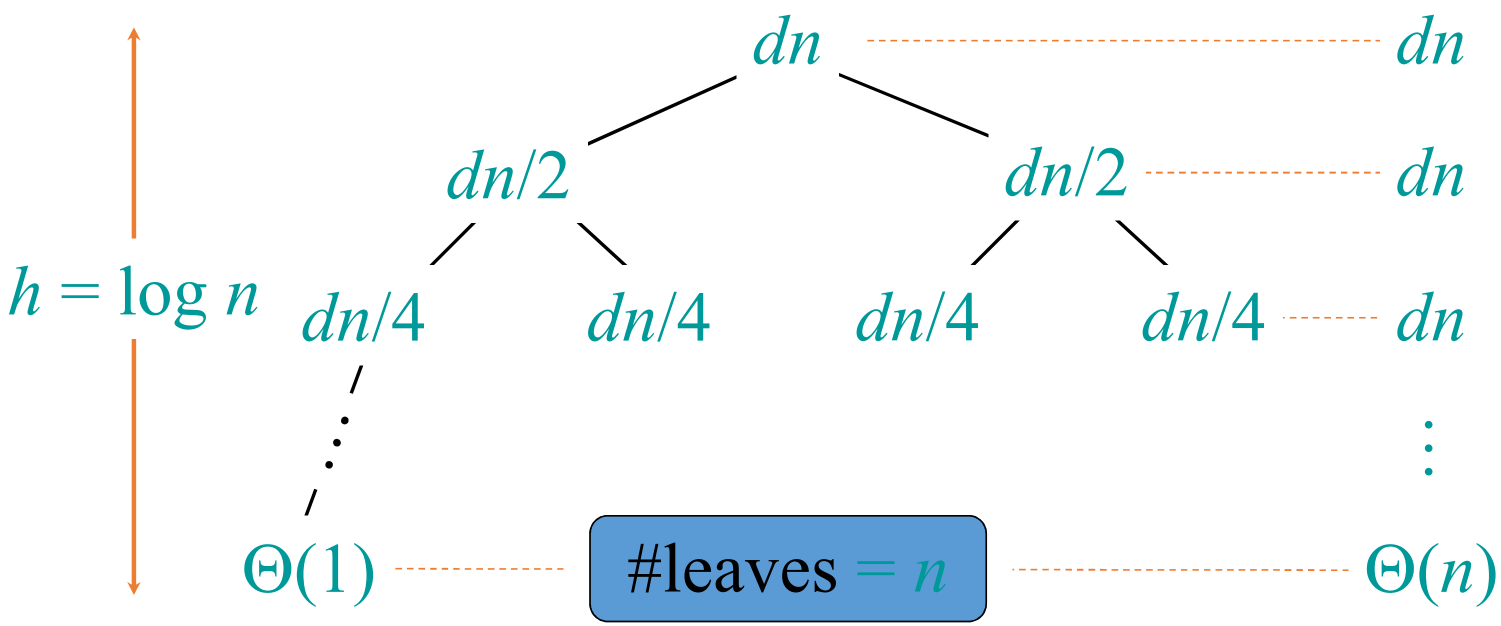
# Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.



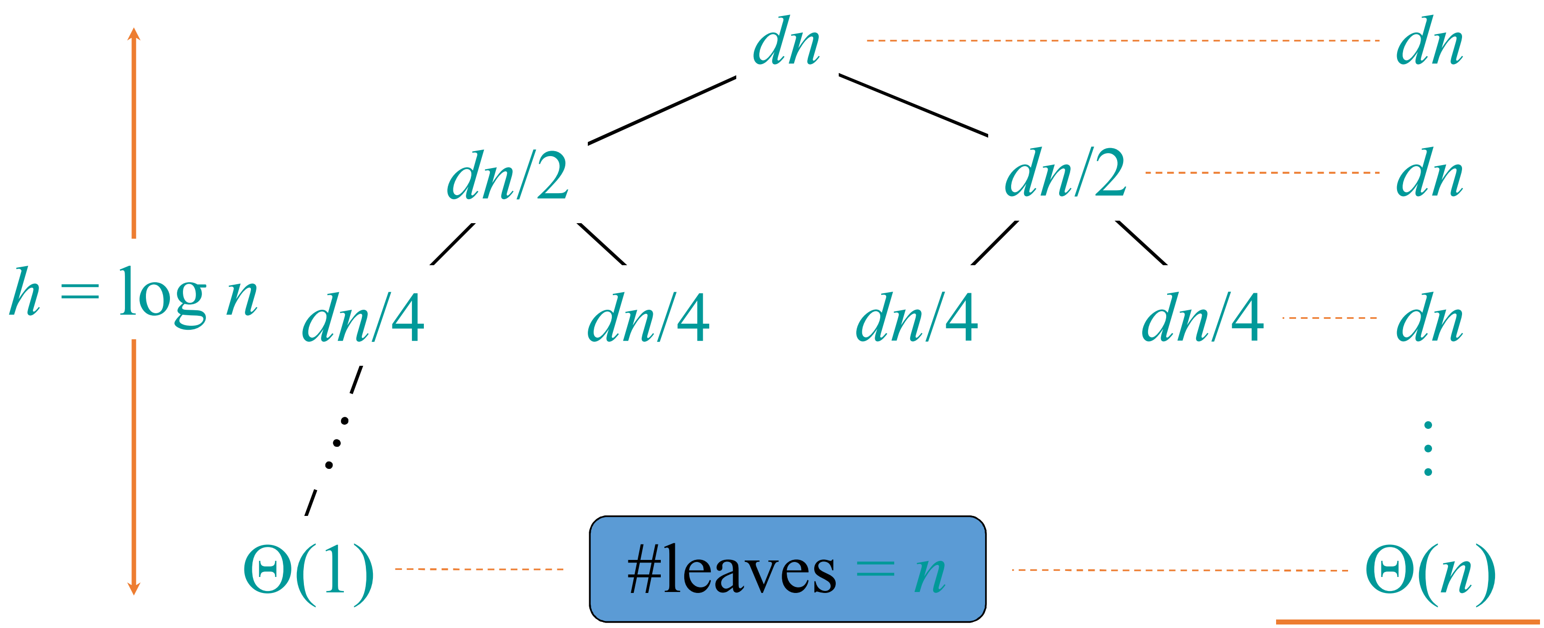
# Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.



# Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.




Total $\Theta(n \log n)$

Later we will usually ignore $d$

# Recurrence for computing power

```
int pow (b, n)
   m = n >> 1;
   p = pow (b, m);
   p = p * p;
   if (n % 2)
        return p * b;
   else
        return p;
```

$T(n) = T(n/2)+\Theta(1)$

```
int pow (b, n)
   m = n >> 1;
   p=pow(b,m)*pow(b,m);
   if (n % 2)
        return p * b;
   else
        return p;
```

$T(n) = 2T(n/2)+\Theta(1)$

Which algorithm is more efficient asymptotically?

# Time complexity for Alg1

Solve $T(n) = T(n/2) + 1$

- $T(n) = T(n/2) + 1$

$$= T(n/4) + 1 + 1$$

$$= T(n/8) + 1 + 1 + 1$$

$$= T(1) + \underbrace{1 + 1 + \ldots + 1}_{log(n)}$$

$$= \Theta\ (log(n))$$

Iteration method

# Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.

# Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.

$$T(n)$$

# Time complexity for Alg2

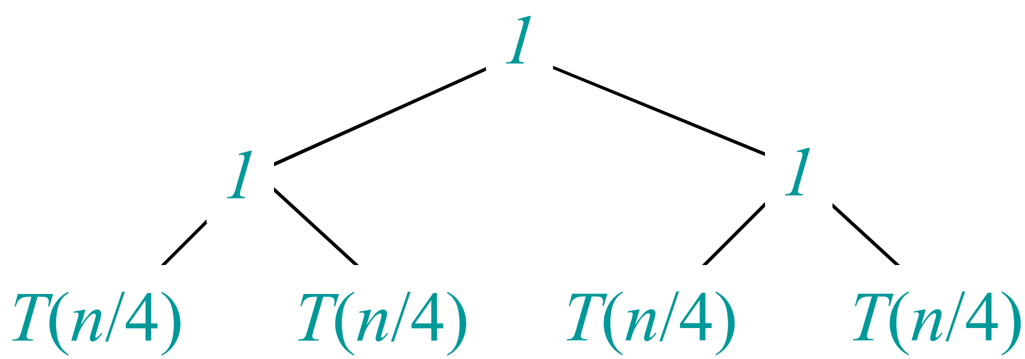
Solve $T(n) = 2T(n/2) + 1$.

$$1$$

$$T(n/2) \qquad\qquad T(n/2)$$

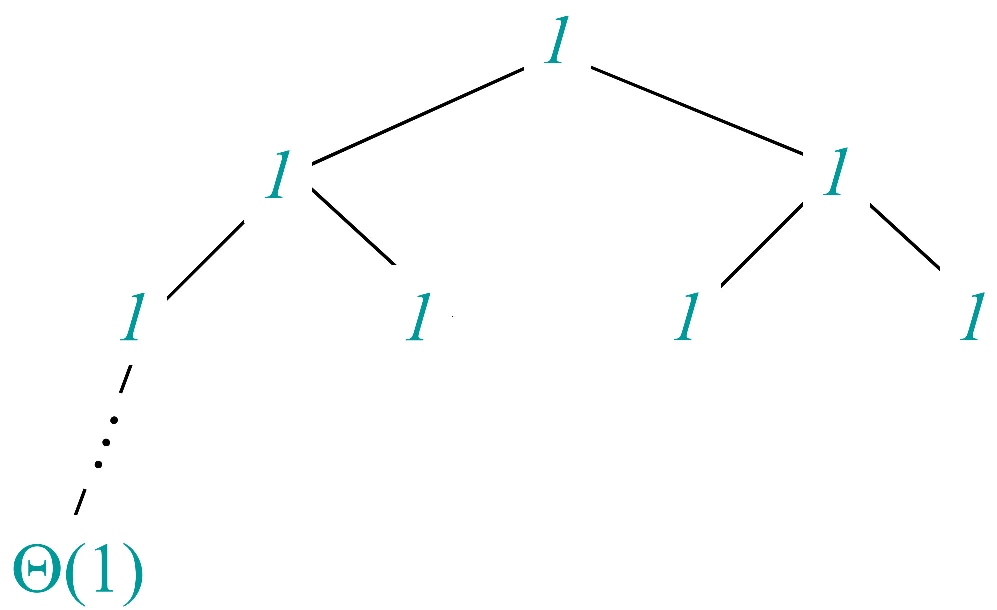
# Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



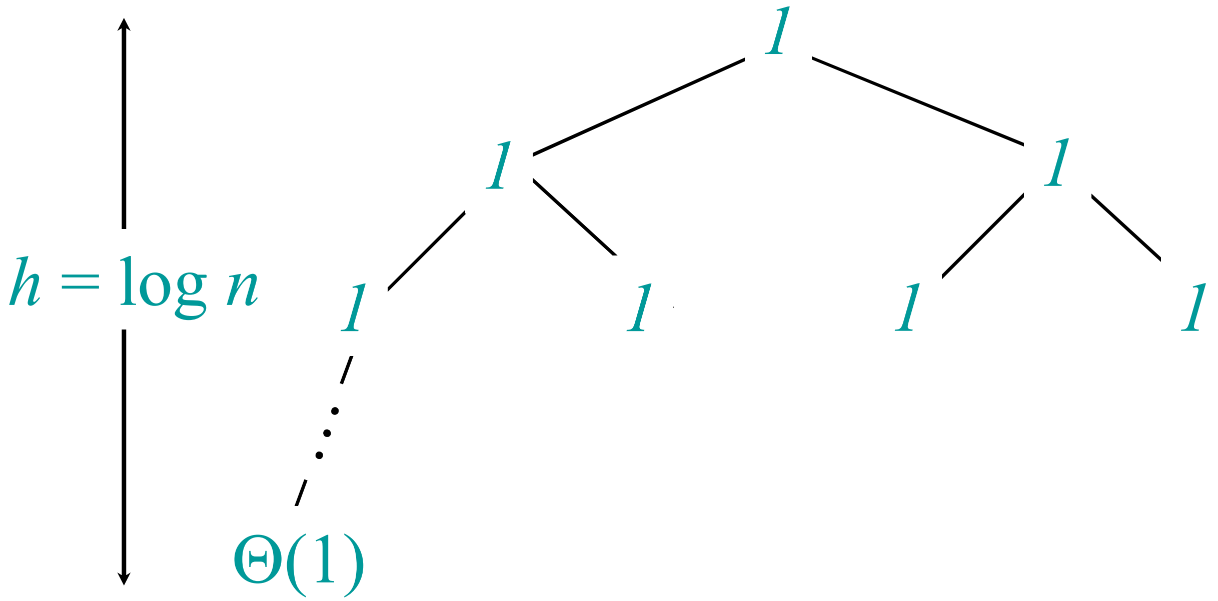
# Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



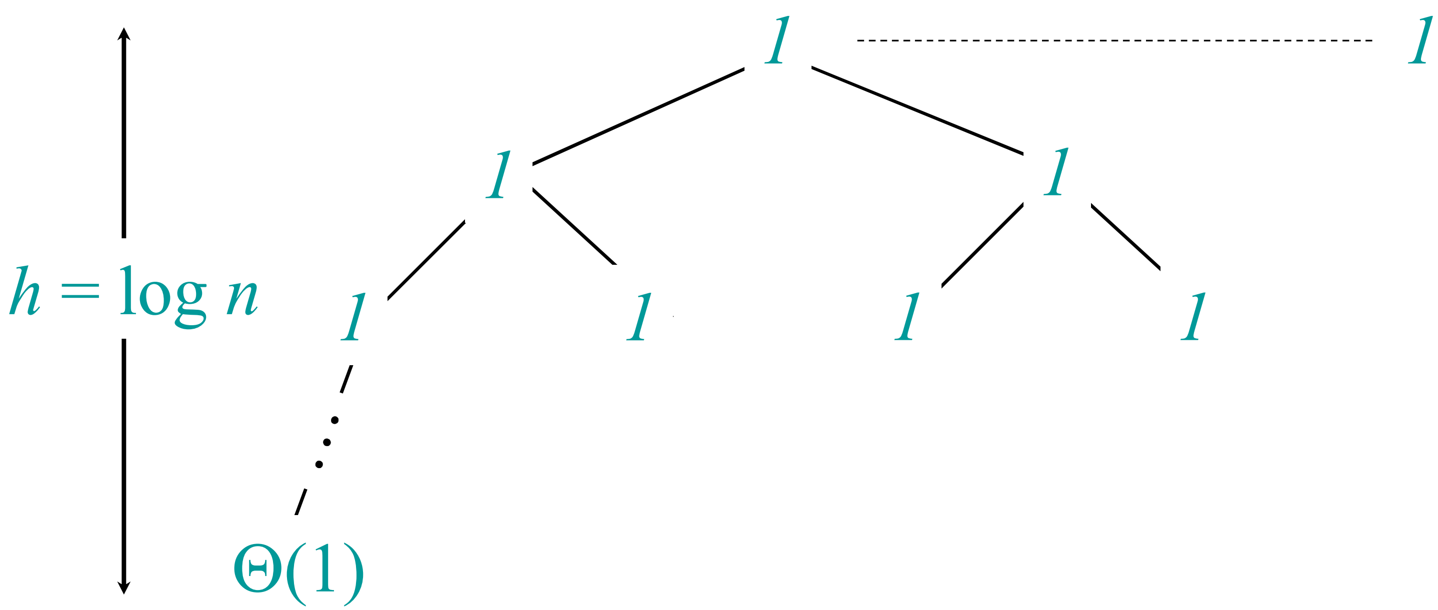
# Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



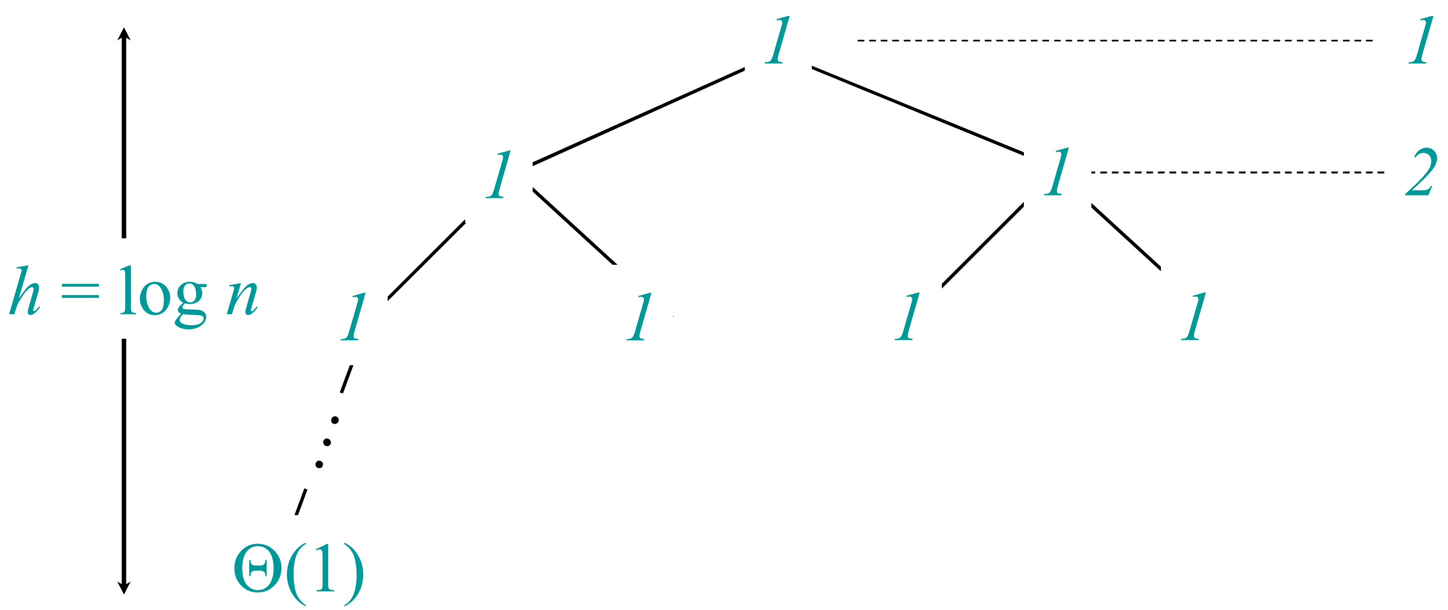
# Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



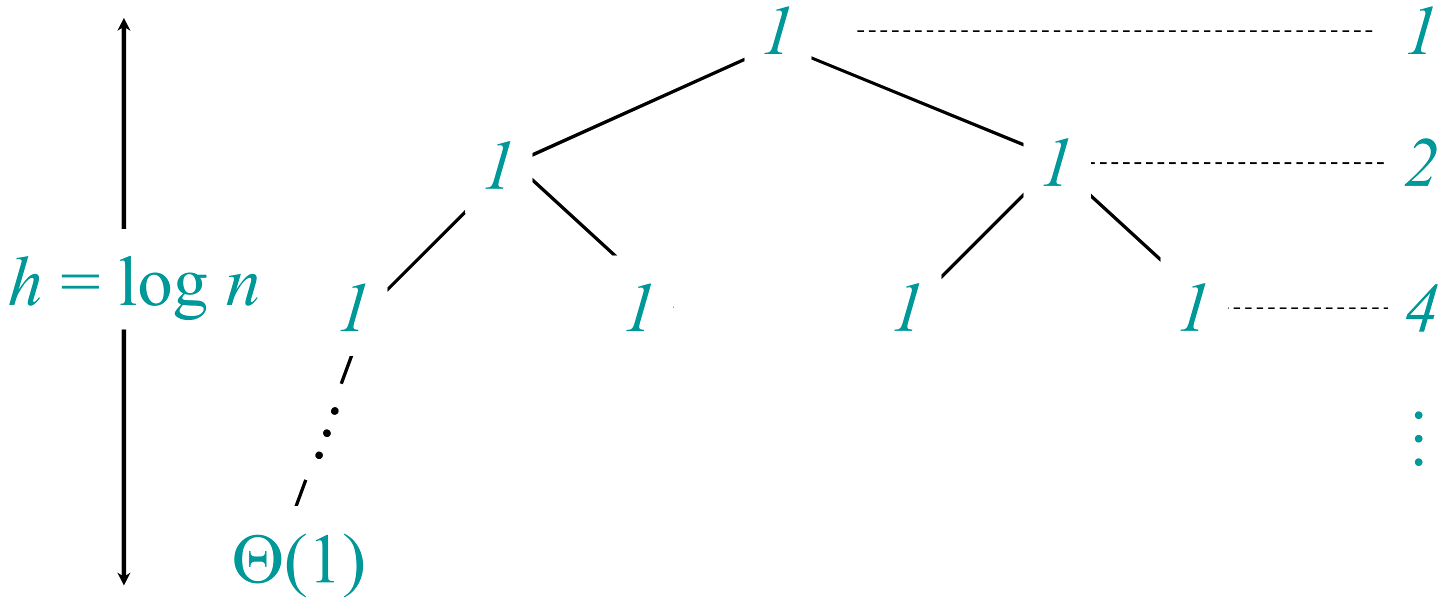
# Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



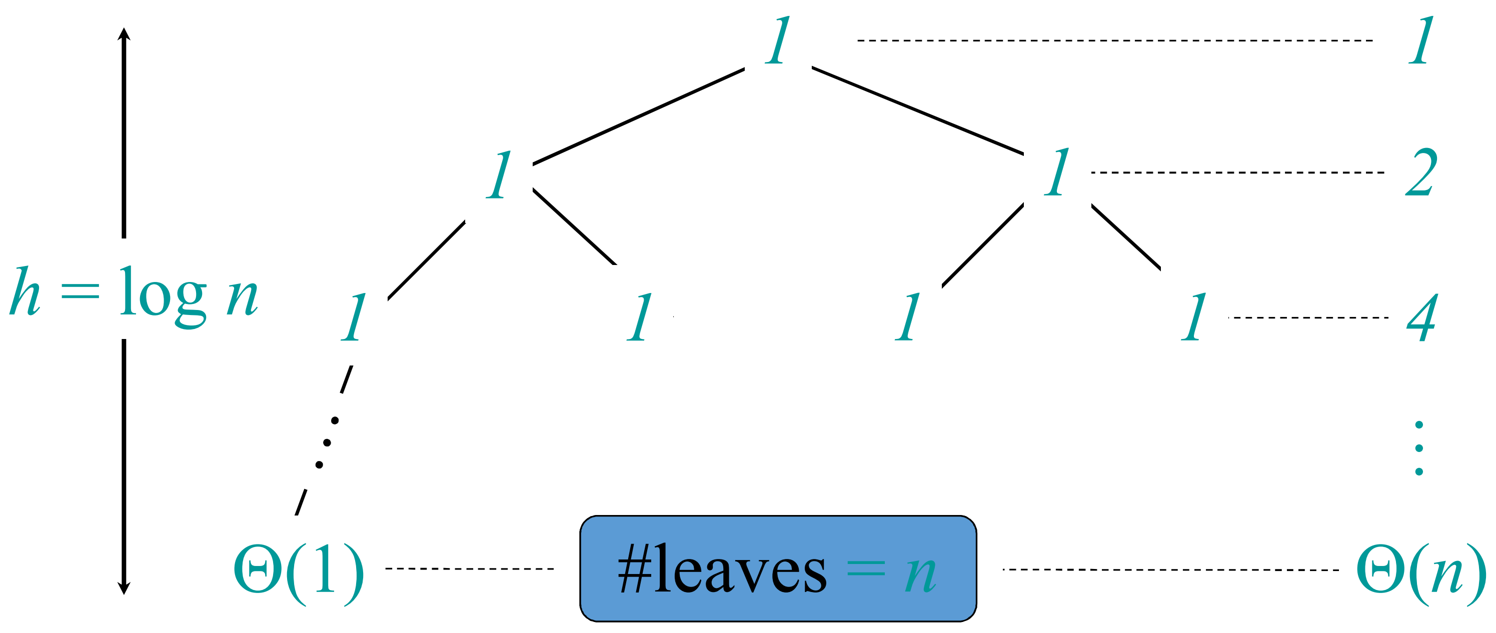
# Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



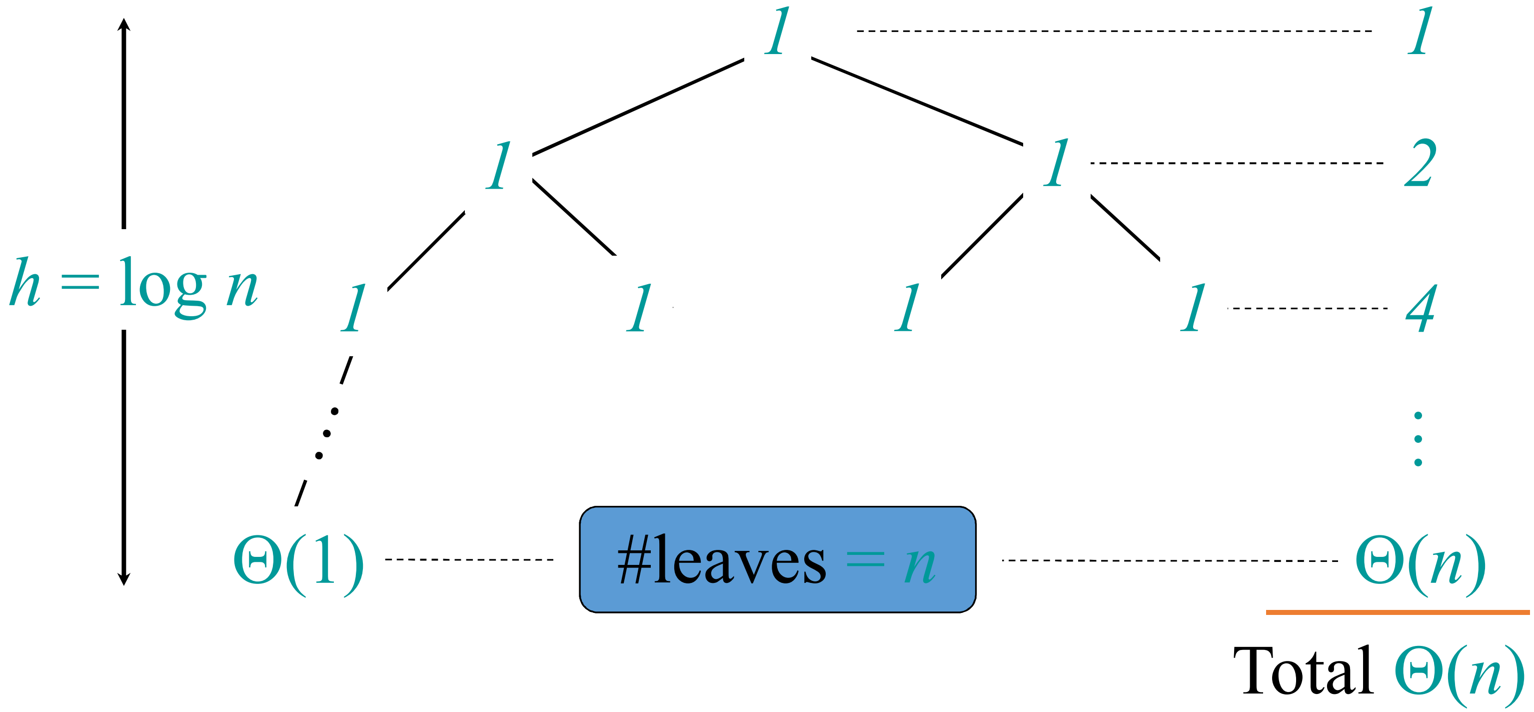
# Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



# Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.



# More iteration method examples

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

$$= T(n-2) + 1 + 1$$

$$= T(n-3) + 1 + 1 + 1$$

$$= T(1) + \underbrace{1 + 1 + \ldots + 1}_{n - 1}$$

$$= \Theta(n)$$

# More iteration method examples

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

  $= T(n-2) + (n-1) + n$

  $= T(n-3) + (n-2) + (n-1) + n$

  $= T(1) + 2 + 3 + \ldots + n$

  $= \Theta(n^2)$

# Recursive definition of sum of series

- $T(n) = \sum_{i=0..n} i$ is equivalent to:

$$\begin{cases} T(n) = T(n-1) + n \\ T(0) = 0 \end{cases}$$

Recurrence relation

Boundary condition

- $T(n) = \sum_{i=0..n} a^i$ is equivalent to:

$$\begin{cases} T(n) = T(n-1) + a^n \\ T(0) = 1 \end{cases}$$

Recursive definition is often intuitive and easy to obtain. It is very useful in analyzing recursive algorithms, and some non-recursive algorithms too.

# 3-way-merge-sort

```
3-way-merge-sort (A[1..n])
    If (n <= 1) return;
    3-way-merge-sort(A[1..n/3]);
    3-way-merge-sort(A[n/3+1..2n/3]);
    3-way-merge-sort(A[2n/3+1.. n]);
    Merge A[1..n/3] and A[n/3+1..2n/3];
    Merge A[1..2n/3] and A[2n/3+1..n];
```

- Is this algorithm correct?
- What's the recurrence function for the running time?
- What does the recurrence function solve to?

# Unbalanced-merge-sort

```
ub-merge-sort (A[1..n])
    if (n<=1) return;
    ub-merge-sort(A[1..n/3]);
    ub-merge-sort(A[n/3+1.. n]);
    Merge A[1.. n/3] and A[n/3+1..n].
```

- Is this algorithm correct?
- What's the recurrence function for the running time?
- What does the recurrence function solve to?

# More recursion tree examples (1)

- $T(n) = 3T(n/3) + n$
- $T(n) = T(n/3) + T(2n/3) + n$
- $T(n) = 2T(n/4) + n$
- $T(n) = 2T(n/4) + n^2$
- $T(n) = 3T(n/2) + n$
- $T(n) = 3T(n/2) + n^2$

# More recursion tree examples (2)

- $T(n) = T(n-2) + n$
- $T(n) = T(n-2) + 1$
- $T(n) = 2T(n-2) + n$
- $T(n) = 2T(n-2) + 1$

# Solving recurrence

1. Recursion tree / iteration method
   - Good for guessing an answer
2. Substitution method
   - Generic method, rigid, but may be hard
3. Master method
   - Easy to learn, useful in limited cases only
   - Some tricks may help in other cases

# The master method

The master method applies to recurrences of the form

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

**where $a \geq 1$, $b > 1$, and $f$ is asymptotically positive**.

1. ***Divide*** the problem into $a$ subproblems, **each** of size $n/b$
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions

   Divide + combine takes $f(n)$ time.

# Master theorem

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

***Key:*** compare $f(n)$ with $n^{\log_b a}$

**CASE 1**: $f(n) = O(n^{\log_b a - \varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$

**CASE 2**: $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$

**CASE 3**: $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $a\,f(n/b) \le c\,f(n)$

Regularity Condition

$$\Rightarrow T(n) = \Theta(f(n))$$

# Case 1

$f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.

Alternatively: $n^{\log_b a} / f(n) = \Omega(n^{\varepsilon})$

Intuition: $f(n)$ grows polynomially slower than $n^{\log_b a}$

Or: $n^{\log_b a}$ dominates $f(n)$ by an $n^{\varepsilon}$ factor for some $\varepsilon > 0$

***Solution:*** $T(n) = \Theta(n^{\log_b a})$

$T(n) = 4T(n/2) + n$
$b = 2,\ a = 4,\ f(n) = n$
$\log_2 4 = 2$
$f(n) = n = O(n^{2-\varepsilon})$, or
$n^2 / n = n^1 = \Omega(n^{\varepsilon})$, for $\varepsilon = 1$
$\therefore T(n) = \Theta(n^2)$

$T(n) = 2T(n/2) + n/\log n$
$b = 2,\ a = 2,\ f(n) = n / \log n$
$\log_2 2 = 1$
$f(n) = n/\log n \notin O(n^{1-\varepsilon})$, or
$n^1 / f(n) = \log n \notin \Omega(n^{\varepsilon})$, for any $\varepsilon > 0$
$\therefore$ CASE 1 does not apply

# Case 2

$f(n) = \Theta\ (n^{\log_b a})$.

*Intuition:* $f(n)$ and $n^{\log_b a}$ have the same asymptotic order.

***Solution:*** $T(n) = \Theta(n^{\log_b a} \log n)$

| | |
|---|---|
| e.g. $T(n) = T(n/2) + 1$ | $\log_b a = 0$ |
| $T(n) = 2\,T(n/2) + n$ | $\log_b a = 1$ |
| $T(n) = 4T(n/2) + n^2$ | $\log_b a = 2$ |
| $T(n) = 8T(n/2) + n^3$ | $\log_b a = 3$ |

# Case 3

$f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.

Alternatively: $f(n) / n^{\log_b a} = \Omega(n^{\varepsilon})$

Intuition: $f(n)$ grows polynomially faster than $n^{\log_b a}$

Or: $f(n)$ dominates $n^{\log_b a}$ by an $n^{\varepsilon}$ factor for some $\varepsilon > 0$

***Solution:*** $T(n) = \Theta(f(n))$

$T(n) = T(n/2) + n$
$b = 2, a = 1, f(n) = n$
$n^{\log_2 1} = n^0 = 1$
$f(n) = n = \Omega(n^{0+\varepsilon})$, or
$n / 1 = n = \Omega(n^{\varepsilon})$
$\therefore T(n) = \Theta(n)$

$T(n) = T(n/2) + \log n$
$b = 2, a = 1, f(n) = \log n$
$n^{\log_2 1} = n^0 = 1$
$f(n) = \log n \notin \Omega(n^{0+\varepsilon})$, or
$f(n) / n^{\log_2 1} / = \log n \notin \Omega(n^{\varepsilon})$
$\therefore$ *CASE 3 does not apply*

# Regularity condition

- $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n
- This is needed for the master method to be mathematically correct.
  - to deal with some non-converging functions such as sine or cosine functions
- For most *f(n)* you'll see (e.g., polynomial, logarithm, exponential), you can safely ignore this condition, because it is implied by the first condition $f(n) = \Omega(n^{\log_b a + \varepsilon})$

# Examples

$T(n) = 4T(n/2) + n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n.$

**Case 1**: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

$\therefore\ T(n) = \Theta(n^2).$

$T(n) = 4T(n/2) + n^2$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2.$

**Case 2**: $f(n) = \Theta(n^2).$

$\therefore\ T(n) = \Theta(n^2 \log n).$

# Examples

$T(n) = 4T(n/2) + n^3$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^3.$

**Case 3**: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

***and*** $4(n/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore\ T(n) = \Theta(n^3).$

$T(n) = 4T(n/2) + n^2/\log n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2/\log n.$

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\log n)$.

# Examples

$T(n) = 4T(n/2) + n^{2.5}$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^{2.5}.$

**CASE 3**: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 0.5$

***and*** $4(n/2)^{2.5} \le cn^{2.5}$ (reg. cond.) for $c = 0.75$.

$\therefore\ T(n) = \Theta(n^{2.5})$.

$T(n) = 4T(n/2) + n^2 \log n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2 \log n.$

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\log n)$.

How do I know which case to use? Do I need to try all three cases one by one?

# Master theorem

- Compare $f(n)$ with $n^{\log_b a}$

check if $n^{\log_b a} / f(n) \in \Omega(n^{\varepsilon})$

- $f(n) \in \begin{cases} o(n^{\log_b a}) & \text{Possible CASE 1} \\ \Theta(n^{\log_b a}) & \text{CASE 2} \\ \omega(n^{\log_b a}) & \text{Possible CASE 3} \end{cases}$

check if $f(n) / n^{\log_b a} \in \Omega(n^{\varepsilon})$

# Examples

a. $T(n) = 4T(n/2) + n;$ $\log_b a = 2$. $n = o(n^2)$ => Check case 1

b. $T(n) = 9T(n/3) + n^2;$ $\log_b a = 2$. $n^2 = \Theta(n^2)$ => case 2

c. $T(n) = 6T(n/4) + n;$ $\log_b a = 1.3$. $n = o(n^{1.3})$ => Check case 1

d. $T(n) = 2T(n/4) + n;$ $\log_b a = 0.5$. $n = \omega(n^{0.5})$ => Check case 3

e. $T(n) = T(n/2) + n \log n;$ $\log_b a = 0$. $n\log n = \omega(n^0)$ => Check case 3

f. $T(n) = 4T(n/4) + n \log n.$ $\log_b a = 1$. $n\log n = \omega(n)$ => Check case 3

# More examples

$$T(n) = nT(n/2) + n$$

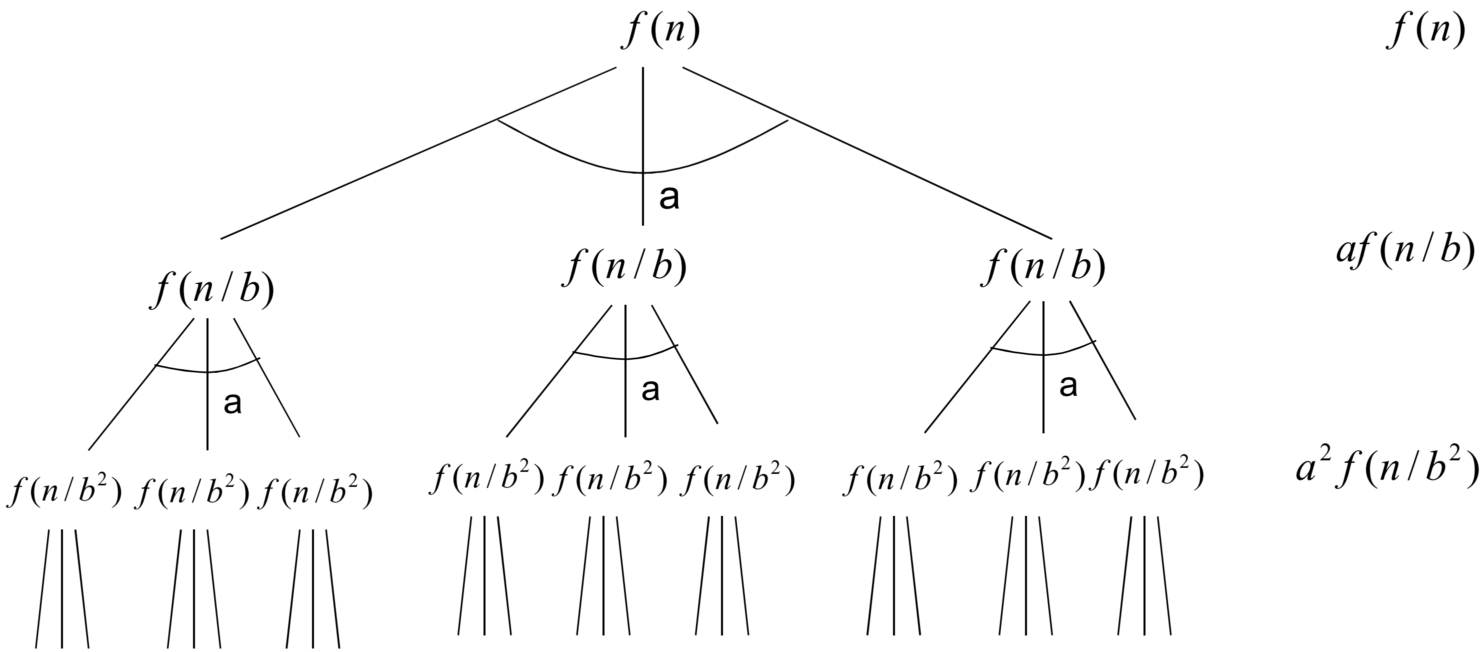
$$T(n) = 0.5T(n/2) + n\log n$$

$$T(n) = 3T(n/3) - n^2 + n$$

$$T(n) = T(n/2) + n(2 - \cos n)$$

# Why does the master method work?

$$T(n) = aT(n/b) + f(n)$$



# What is the depth of the tree?

At each level, the size of the data is divided by b

$$\frac{n}{b^d} = 1$$

$$\log\left(\frac{n}{b^d}\right) = 0$$

$$\log n - \log 4^b = 0$$

$$d \log b = \log n$$

$$d = \log_b n$$

# How many leaves?

How many leaves are there in a complete *a*-ary tree of depth *d*?

$$a^{d} = a^{\log_b n}$$

$$= n^{\log_b a}$$

# Total cost

$$\text{if } f(n) = O(n^{\log_b a-\varepsilon}) \text{ for } \varepsilon > 0, \text{then } T(n) = \Theta(n^{\log_b a})$$

$$\text{if } f(n) = \Theta(n^{\log_b a}), \text{then } T(n) = \Theta(n^{\log_b a} \log n)$$

$$\text{if } f(n) = \Omega(n^{\log_b a+\varepsilon}) \text{ for } \varepsilon > 0 \text{ and } af(n/b) \le cf(n) \text{ for } c < 1$$

$$\text{then } T(n) = \Theta(f(n))$$

$$T(n) = cf(n) + af(n/b) + a^2 f(n/b^2) + ... + a^{n-1} f(n/b^{n-1}) + \Theta(n^{\log_b a3})$$

$$= \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) + \Theta(n^{\log_b a})$$

Case 1: cost is dominated by the cost of the leaves

$$= \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) < \Theta(n^{\log_b a})$$

# Total cost

if $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$

if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for $\varepsilon > 0$ and $af(n/b) \leq cf(n)$ for $c < 1$
then $T(n) = \Theta(f(n))$

$$T(n) = cf(n) + af(n/b) + a^2 f(n/b^2) + \ldots + a^{n-1} f(n/b^{n-1}) + \Theta(n^{\log_b a3})$$

$$= \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) + \Theta(n^{\log_b a})$$

Case 2: cost is evenly distributed across tree

As we saw with mergesort, log *n* levels to the tree and at each level *f(n)* work

## Total cost

$$\text{if } f(n) = O(n^{\log_b a - \varepsilon}) \text{ for } \varepsilon > 0, \text{ then } T(n) = \Theta(n^{\log_b a})$$

$$\text{if } f(n) = \Theta(n^{\log_b a}), \text{ then } T(n) = \Theta(n^{\log_b a} \log n)$$

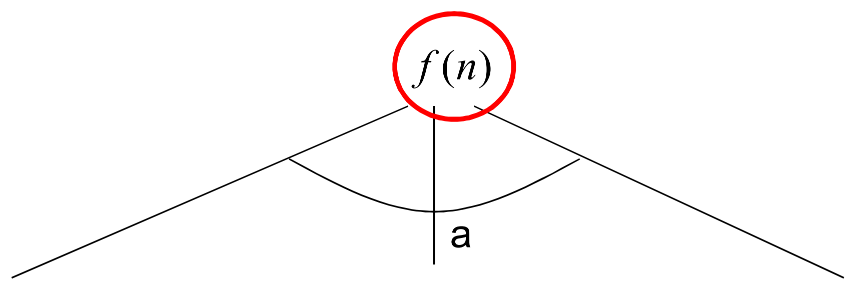
$$\text{if } f(n) = \Omega(n^{\log_b a + \varepsilon}) \text{ for } \varepsilon > 0 \text{ and } af(n/b) \le cf(n) \text{ for } c < 1$$

$$\text{then } T(n) = \Theta(f(n))$$

$$T(n) = cf(n) + af(n/b) + a^2 f(n/b^2) + \ldots + a^{d-1} f(n/b^{d-1}) + \Theta(n^{\log_b a3})$$

$$= \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) + \Theta(n^{\log_b a})$$

Case 3: cost is dominated by the cost of the root



# Some tricks

- Changing variables

- Obtaining upper and lower bounds
  - Make a guess based on the bounds
  - Prove using the substitution method

# Changing variables

$$T(n) = 2T(n-1) + 1$$

- Let $n = \log m$, i.e., $m = 2^n$

$\Rightarrow T(\log m) = 2\, T(\log (m/2)) + 1$

- Let $S(m) = T(\log m) = T(n)$

$\Rightarrow S(m) = 2S(m/2) + 1$ **CASE 1**

$\Rightarrow S(m) = \Theta(m)$

$\Rightarrow T(n) = S(m) = \Theta(m) = \Theta(2^n)$

# Changing variables

$$T(n) = T(\sqrt{n}) + 1$$

- Let n $=2^m$

=> sqrt(n) $= 2^{m/2}$

- We then have $T(2^m) = T(2^{m/2}) + 1$
- Let $T(n) = T(2^m) = S(m)$

=> $S(m) = S(m/2) + 1$ **CASE 2**

$\Rightarrow S(m) = \Theta(\log m) = \Theta(\log \log n)$

$\Rightarrow T(n) = \Theta(\log \log n)$

# Changing variables

- $T(n) = 2T(n-2) + 1$
- Let $n = \log m$, i.e., $m = 2^n$

$\Rightarrow T(\log m) = 2\,T(\log m/4) + 1$

- Let $S(m) = T(\log m) = T(n)$

$\Rightarrow S(m) = 2S(m/4) + 1$ **CASE 1**

$\Rightarrow S(m) = \Theta(m^{1/2})$

$\Rightarrow T(n) = S(m) = \Theta((2^n)^{1/2}) = \Theta((\mathrm{sqrt}(2))^n) \approx \Theta(1.4^n)$

# Obtaining bounds

*Solve the Fibonacci sequence:*

$$T(n) = T(n-1) + T(n-2) + 1$$

- $T(n) >= 2T(n-2) + 1$ [1]
- $T(n) <= 2T(n-1) + 1$ [2]

- Solving [1], we obtain $T(n) >= 1.4^n$
- Solving [2], we obtain $T(n) <= 2^n$
- Actually, $T(n) \approx 1.62^n$

# Obtaining bounds

- $T(n) = T(n/2) + \log n$
- $T(n) \in \Omega(\log n)$
- $T(n) \in O(T(n/2) + n^{\varepsilon})$
- Solving $T(n) = T(n/2) + n^{\varepsilon}$,

  we obtain $T(n) = O(n^{\varepsilon})$, for any $\varepsilon > 0$
- So: $T(n) \in O(n^{\varepsilon})$ for any $\varepsilon > 0$
  - T(n) is unlikely polynomial
  - Actually, $T(n) = \Theta(\log^2 n)$ by extended case 2

# Extended Case 2

**CASE 2**: $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$.

**Extended CASE 2**: (k >= 0)

$f(n) = \Theta(n^{\log_b a}\, log^k n) \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

# Solving recurrence

1. Recursion tree / iteration method
   - Good for guessing an answer
   - Need to **verify**
2. Substitution method
   - Generic method, rigid, but may be hard
3. Master method
   - Easy to learn, useful in **limited cases** only
   - Some tricks may help in other cases

# Substitution method

*The most general method* to solve a recurrence (prove O and $\Omega$ separately):

1. ***Guess*** the form of the solution
   (e.g. by recursion tree / iteration method)
2. ***Verify*** by induction (inductive step).
3. ***Solve*** for O-constants $n_0$ and $c$ (base case of induction)

# Substitution method

- Recurrence: $T(n) = 2T(n/2) + n$.
- Guess: $T(n) = O(n \log n)$. (eg. by recursion tree method)
- To prove, have to show $T(n) \le c\, n \log n$ for some $c > 0$ and for all $n > n_0$
- Proof by induction: assume it is true for $T(n/2)$, prove that it is also true for $T(n)$. This means:
- Given: $T(n) = 2T(n/2) + n$
- Need to Prove: $T(n) \le c\, n \log (n)$
- Assume: $T(n/2) \le cn/2 \log (n/2)$

# Proof

- Given: $T(n) = 2T(n/2) + n$
- Need to Prove: $T(n) \leq c\, n \log (n)$
- Assume: $T(n/2) \leq cn/2 \log (n/2)$
- *Proof:*

  Substituting $T(n/2) \leq cn/2 \log (n/2)$ into the recurrence, we get

  $T(n) = 2\, T(n/2) + n$

  $\leq cn \log (n/2) + n$

  $\leq c\, n \log n - c\, n + n$

  $\leq c\, n \log n - (c - 1)\, n$

  $\leq c\, n \log n$ for all n > 0 *(if $c \geq 1$).*

Therefore, by definition, T(n) = O(n log n).

# Substitution method – example 2

- Recurrence: $T(n) = 2T(n/2) + n$.
- Guess: $T(n) = \Omega(n \log n)$.
- To prove, have to show $T(n) \geq c\, n \log n$ for some $c > 0$ and for all $n > n_0$
- Proof by induction: assume it is true for $T(n/2)$, prove that it is also true for $T(n)$. This means:

- Given: $T(n) = 2T(n/2) + n$
- Need to Prove: $T(n) \geq c\, n \log (n)$
- Assume: $T(n/2) \geq cn/2 \log (n/2)$

# Proof

- Given: $T(n) = 2T(n/2) + n$
- Need to Prove: $T(n) \geq c\ n \log(n)$
- Assume: $T(n/2) \geq cn/2 \log(n/2)$
- *Proof:*

  Substituting $T(n/2) \geq cn/2 \log(n/2)$ into the recurrence, we get

$$
\begin{aligned}
T(n) &= 2\ T(n/2) + n \\
&\geq cn \log(n/2) + n \\
&\geq c\ n \log n - c\ n + n \\
&\geq c\ n \log n + (1 - c)\ n \\
&\geq c\ n \log n \quad \text{for all } n > 0 \text{ (if } c \leq 1\text{).}
\end{aligned}
$$

Therefore, by definition, $T(n) = \Omega(n \log n)$.

# More substitution method examples (1)

- Prove that $T(n) = 3T(n/3) + n = O(n\log n)$
- Need to show that $T(n) \leq c\, n \log n$ for some c, and sufficiently large n
- Assume above is true for $T(n/3)$, i.e.

  $T(n/3) \leq cn/3 \log (n/3)$

# examples

$T(n) = 3\,T(n/3) + n$

$\leq 3\,cn/3 \log(n/3) + n$

$\leq cn \log n - cn \log 3 + n$

$\leq cn \log n - (cn \log 3 - n)$

$\leq cn \log n$ (if $cn \log 3 - n \geq 0$)

$cn \log 3 - n \geq 0$

=> $c \log 3 - 1 \geq 0$ (for $n > 0$)

=> $c \geq 1/\log 3$

=> $c \geq \log_3 2$

Therefore, $T(n) = 3\,T(n/3) + n \leq cn \log n$ for $c = \log_3 2$ and $n > 0$. By definition, $T(n) = O(n \log n)$.

# More substitution method examples (2)

- Prove that $T(n) = T(n/3) + T(2n/3) + n = O(n\log n)$
- Need to show that $T(n) \leq c\, n \log n$ for some c, and sufficiently large n
- Assume above is true for $T(n/3)$ and $T(2n/3)$, i.e.

  $T(n/3) \leq cn/3 \log (n/3)$

  $T(2n/3) \leq 2cn/3 \log (2n/3)$

# examples

$T(n) = T(n/3) + T(2n/3) + n$

$\leq cn/3 \log(n/3) + 2cn/3 \log(2n/3) + n$

$\leq cn \log n + n - cn (\log 3 - 2/3)$

$\leq cn \log n + n(1 - c\log3 + 2c/3)$

$\leq cn \log n$, for all $n > 0$ (if $1 - c \log3 + 2c/3 \leq 0$)

$c \log3 - 2c/3 \geq 1$

$\Rightarrow c \geq 1 / (\log3-2/3) > 0$

Therefore, $T(n) = T(n/3) + T(2n/3) + n \leq cn \log n$ for $c = 1 / (\log3-2/3)$ and $n > 0$. By definition, $T(n) = O(n \log n)$.

# More substitution method examples (3)

- Prove that $T(n) = 3T(n/4) + n^2 = O(n^2)$
- Need to show that $T(n) \leq c\, n^2$ for some c, and sufficiently large n
- Assume above is true for $T(n/4)$, i.e.

  $T(n/4) \leq c(n/4)^2 = cn^2/16$

# examples

$$T(n) = 3T(n/4) + n^2$$
$$\leq 3\,c\,n^2 / 16 + n^2$$
$$\leq (3c/16 + 1)\,n^2$$
$$\leq cn^2$$

?

$3c/16 + 1 \leq c$ implies that $c \geq 16/13$

Therefore, $T(n) = 3(n/4) + n^2 \leq cn^2$ for $c = 16/13$ and all n. By definition, $T(n) = O(n^2)$.

# Avoiding pitfalls

- Guess $T(n) = 2T(n/2) + n = O(n)$
- Need to prove that $T(n) \leq c\ n$
- Assume $T(n/2) \leq cn/2$

- $T(n) \leq 2 * cn/2 + n = cn + n = O(n)$

- What's wrong?

- Need to prove $T(n) \leq cn$, not $T(n) \leq cn + n$

# Subtleties

- Prove that $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 = O(n)$
- Need to prove that $T(n) \leq cn$
- Assume above is true for $T(\lfloor n/2 \rfloor)$ & $T(\lceil n/2 \rceil)$

$T(n) <= c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1$

$\leq cn + 1$

Is it a correct proof?

No! has to prove $T(n) <= cn$

However we can prove $T(n) = O(n – 1)$

Details skipped.

# Making good guess

$T(n) = 2T(n/2 + 17) + n$

When n approaches infinity, n/2 + 17 are not too different from n/2

Therefore can guess $T(n) = \Theta(n \log n)$

Prove $\Omega$:

Assume $T(n/2 + 17) \geq c\,(n/2+17) \log (n/2 + 17)$

Then we have

$$
\begin{aligned}
T(n) &= n + 2T(n/2+17) \\
&\geq n + 2c\,(n/2+17) \log (n/2 + 17) \\
&\geq n + c\,n \log (n/2 + 17) + 34\,c \log (n/2+17) \\
&\geq c\,n \log (n/2 + 17) + 34\,c \log (n/2+17) \\
&\;\;\ldots.
\end{aligned}
$$

Maybe can guess $T(n) = \Theta((n-17) \log (n-17))$ (trying to get rid of the +17).

Details skipped.