CS 560: Design and Analysis of Algorithms

Chapter 4: Divide and Conquer

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.

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 $A[1 \,.\,.\, n]$

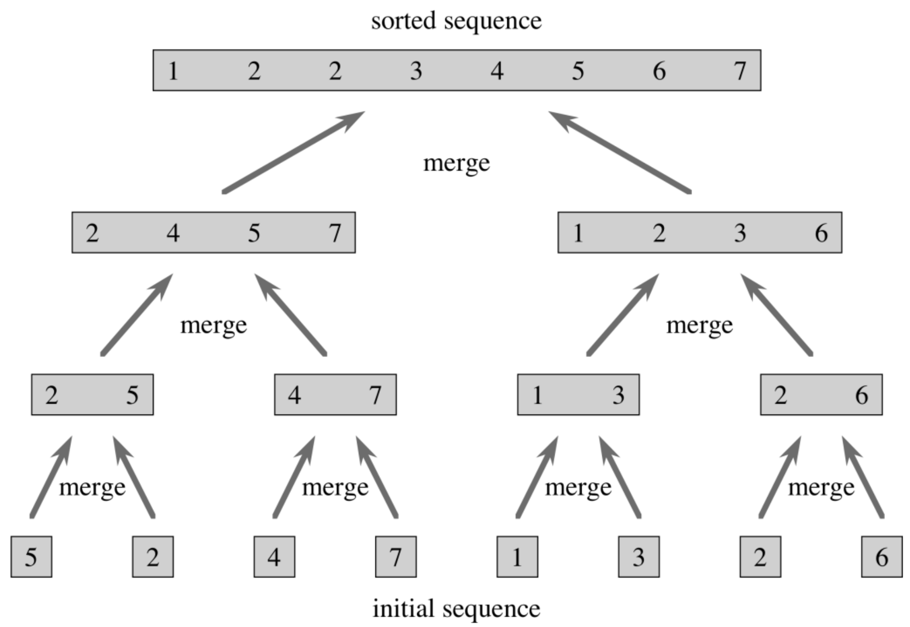
1. If $n = 1$, done.
2. Recursively sort $A[\, 1 \,.\,.\, \lceil n/2 \rceil \,]$ and $A[\, \lceil n/2 \rceil + 1 \,.\,.\, n \,]$.
3. "***Merge***" the 2 sorted lists.

Key subroutine: **MERGE**

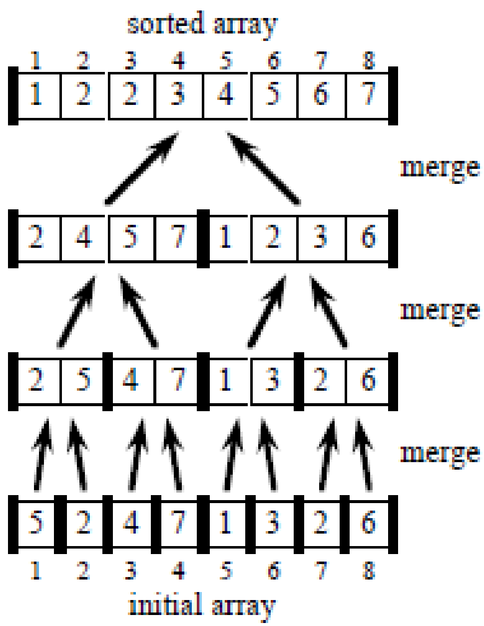
Merge Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

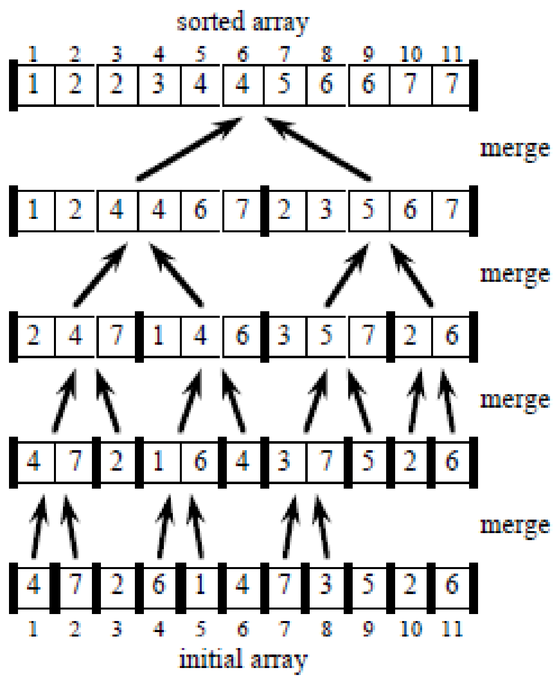
The book uses this notation. Note that both versions of Merge Sort are essentially the same



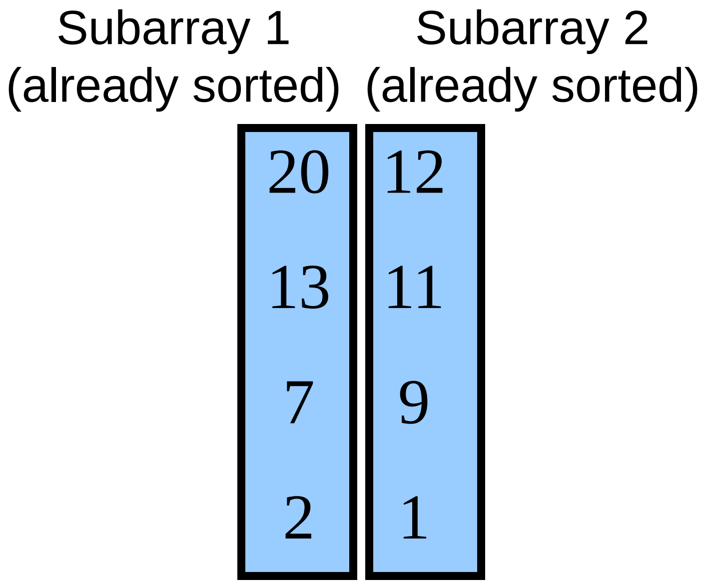
Merge Sort (n=8)



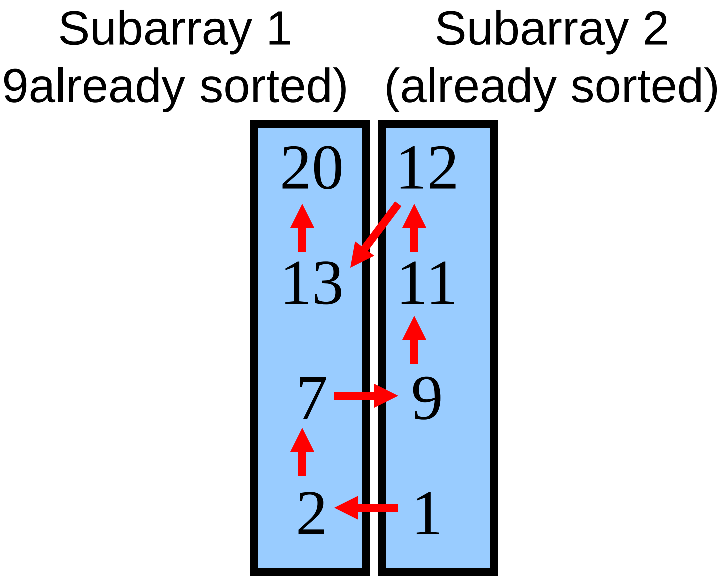
Merge Sort (n=11)



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

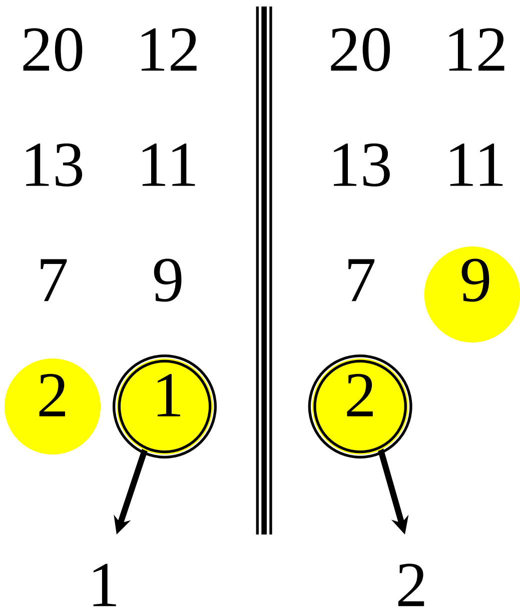
1

Merging two sorted arrays

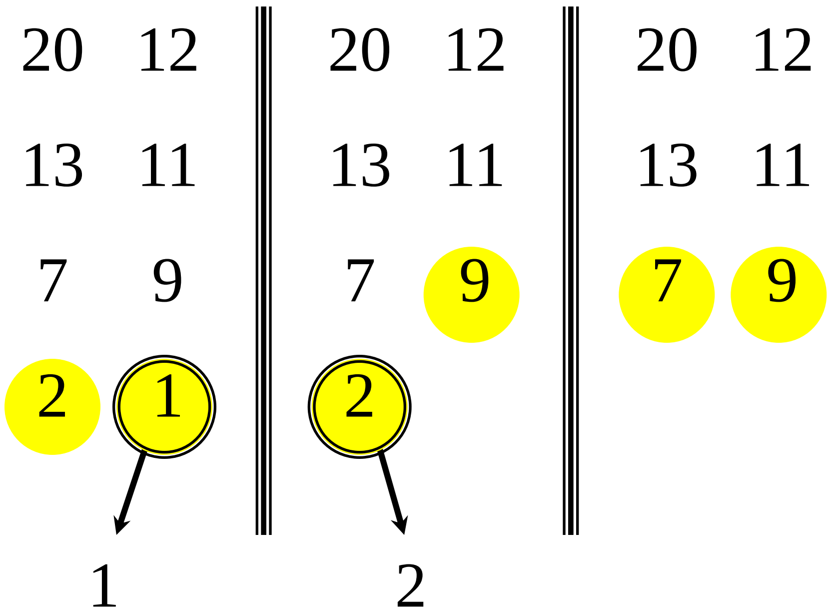
20	12		20	12
13	11		13	11
7	9		7	9
2	1		2	

1

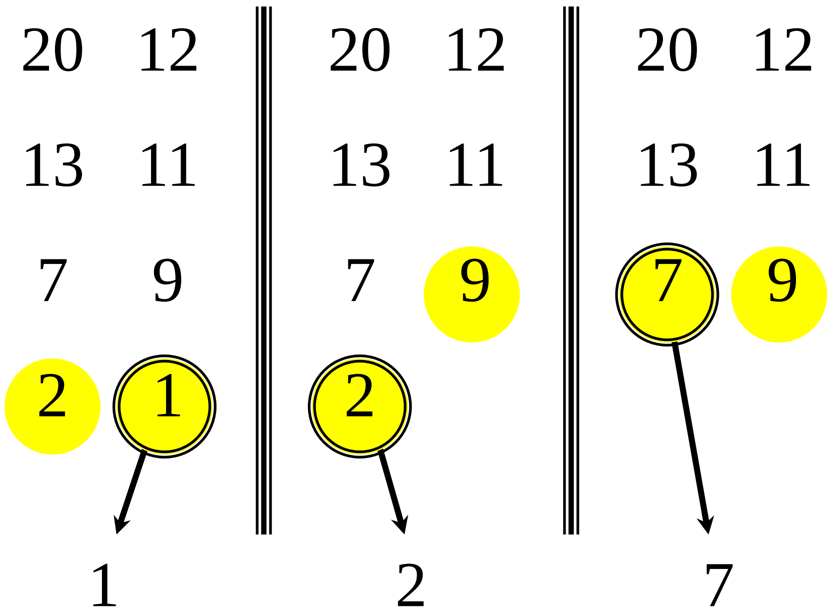
Merging two sorted arrays



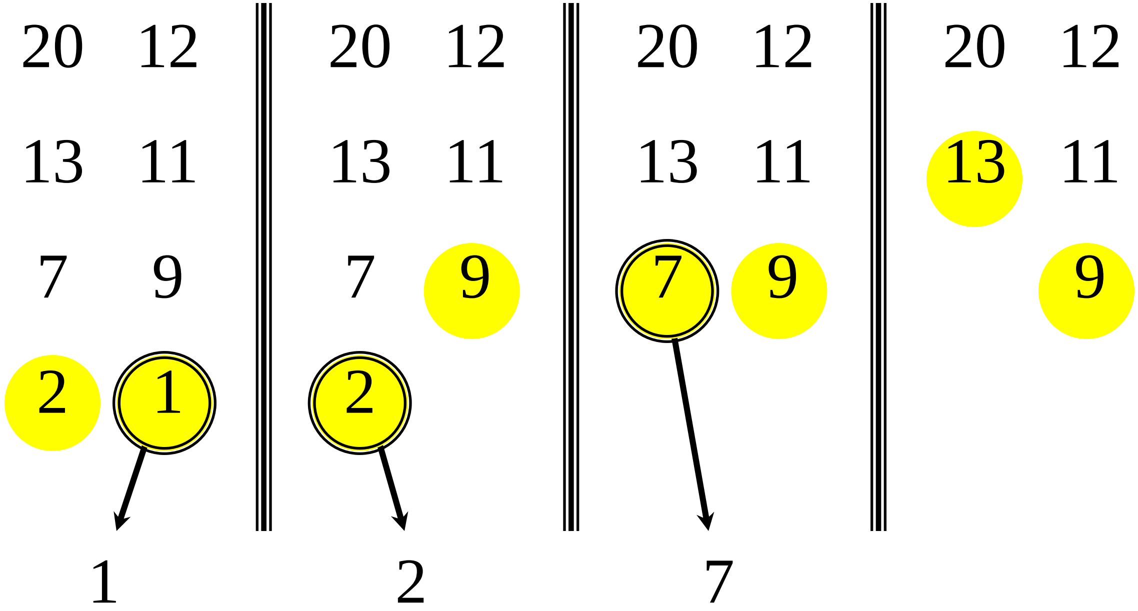
Merging two sorted arrays



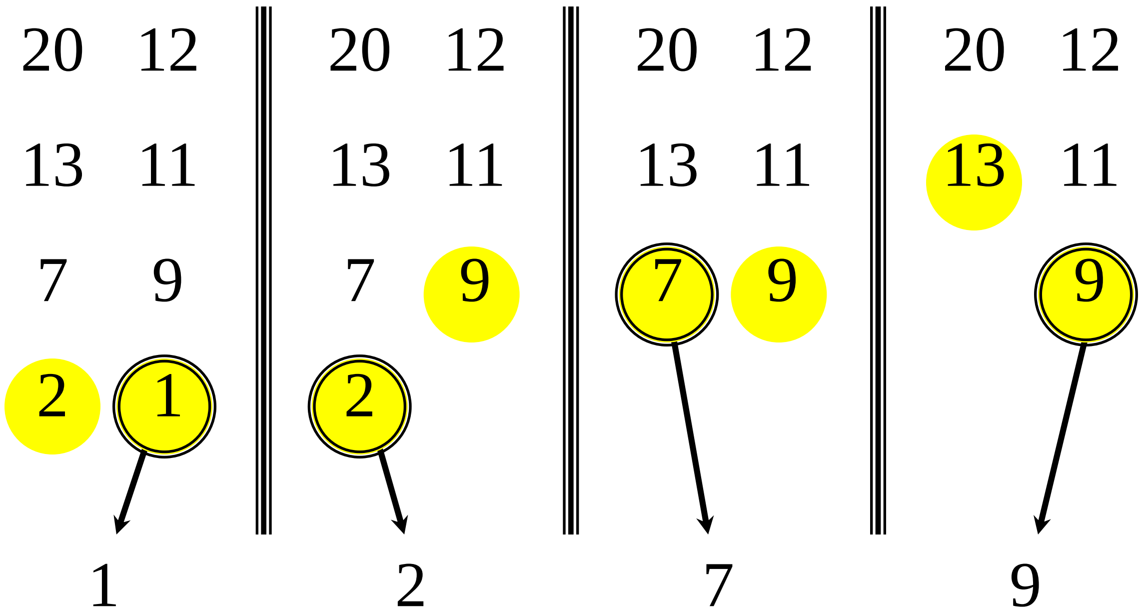
Merging two sorted arrays



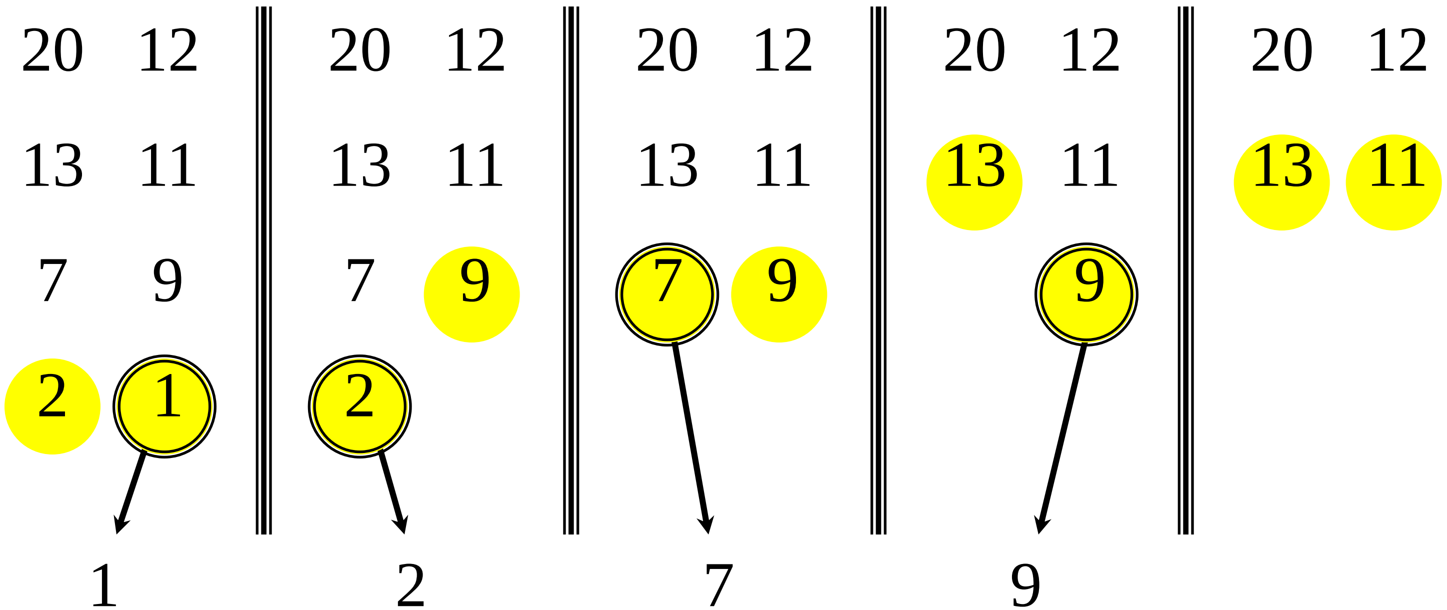
Merging two sorted arrays



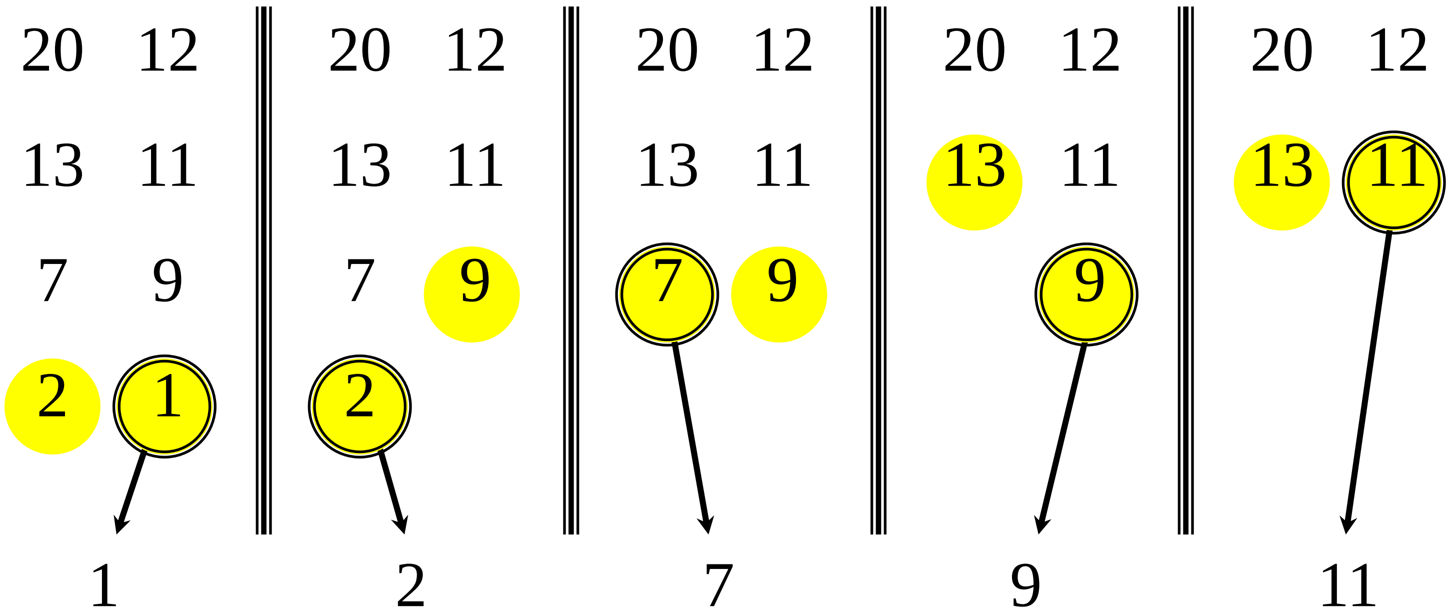
Merging two sorted arrays



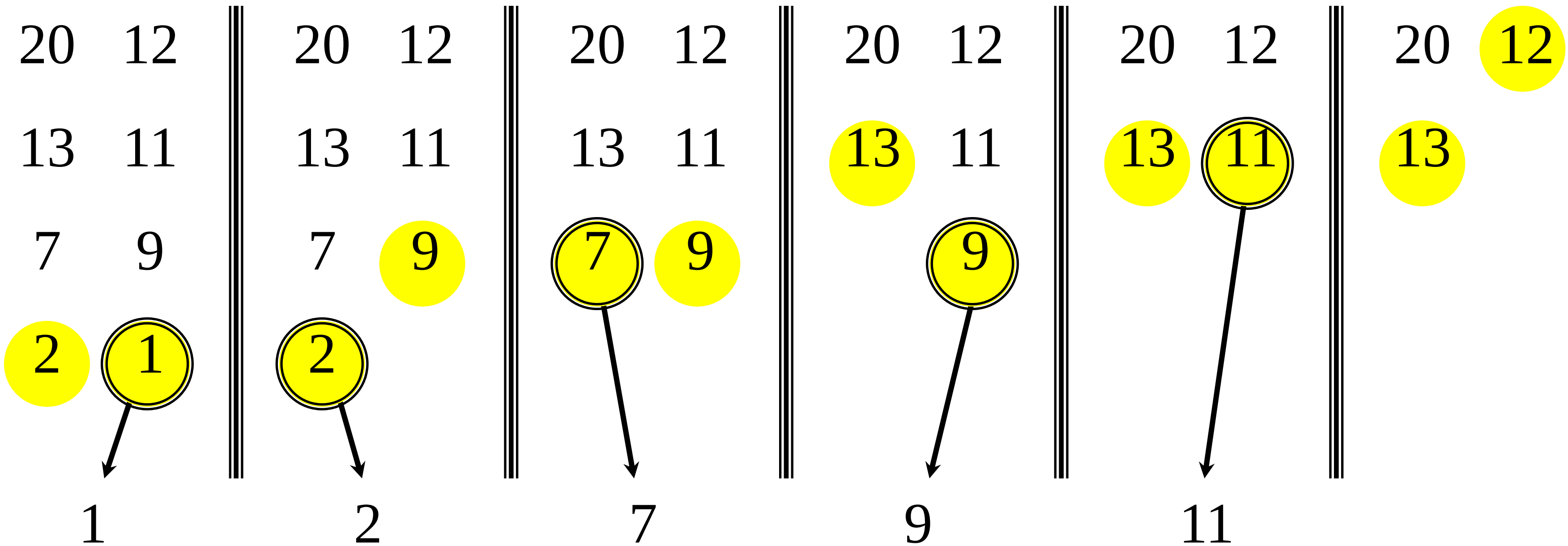
Merging two sorted arrays



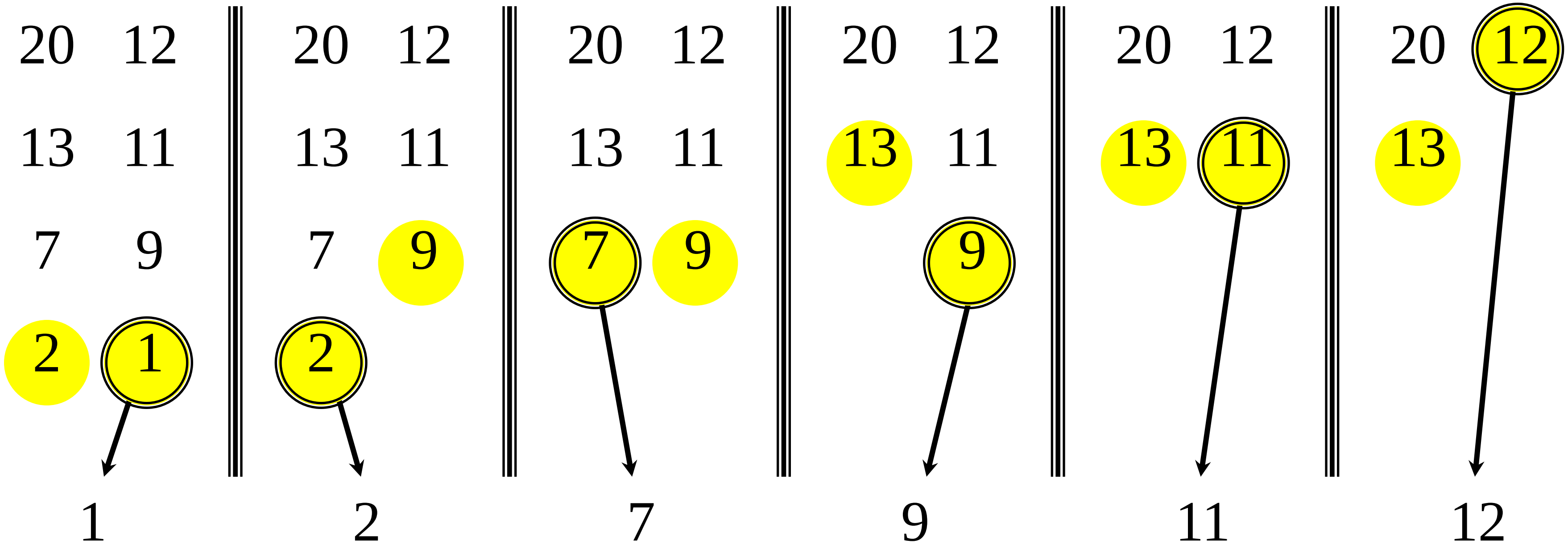
Merging two sorted arrays



Merging two sorted arrays



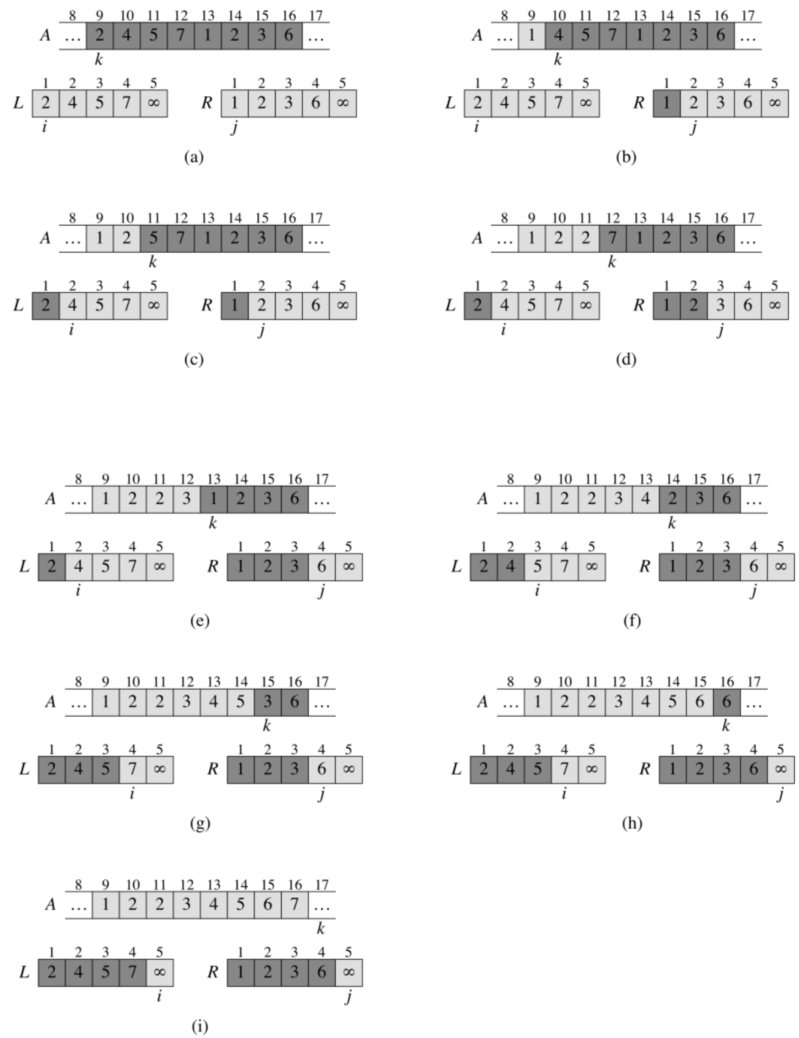
Merging two sorted arrays



Merge

```
MERGE(A, p, q, r)
n_1 = q - p + 1
n_2 = r - q
let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
for i = 1 to n_1
    L[i] = A[p + i - 1]
for j = 1 to n_2
    R[j] = A[q + j]
L[n_1 + 1] = ∞
R[n_2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Merge Operation



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] which is nothing but A[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.

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

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

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

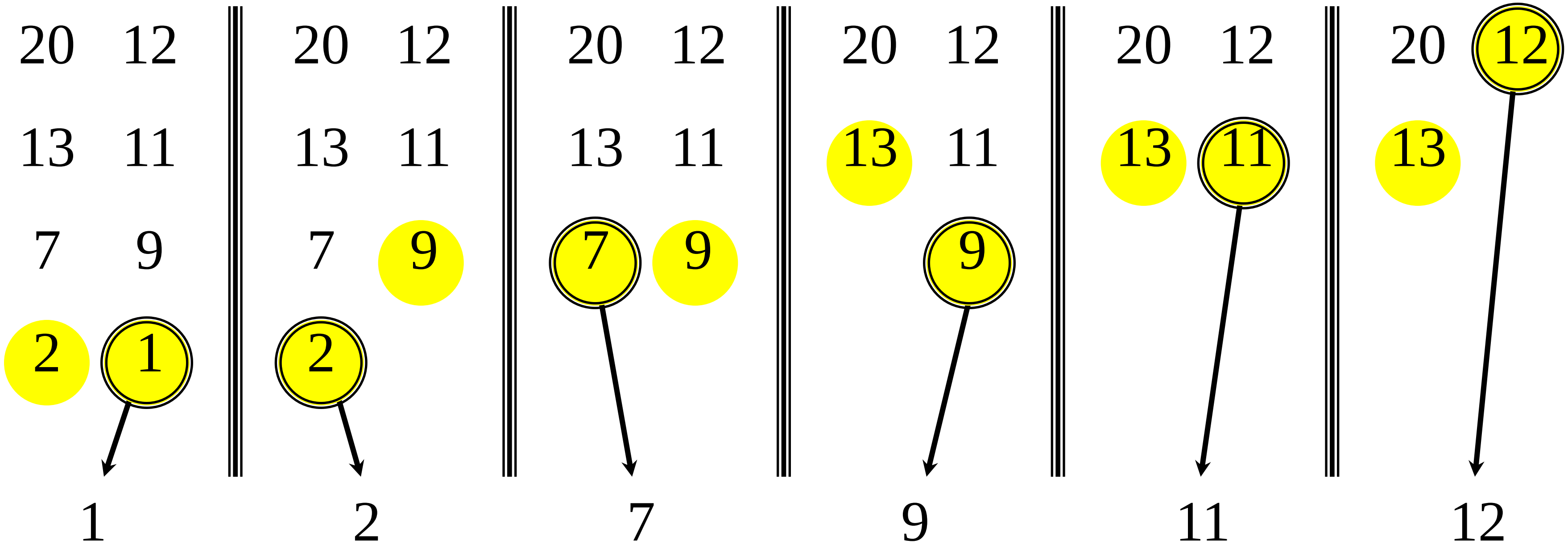
 # subproblems

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 …?

Binary Search

To find whether a query element is present in a sorted array, we

1. Check the middle element
2. If this middle element is same as query element , 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 whether a query element is present in a sorted array, we

1. Check the middle element
2. If this middle element is same as query element , 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 whether a query element is present in a sorted array, we

1. Check the middle element
2. If this middle element is same as query element , 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 whether a query element is present in a sorted array, we

1. Check the middle element
2. If this middle element is same as query element , 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 whether a query element is present in a sorted array, we

1. Check the middle element
2. If this middle element is same as query element , 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 whether a query element is present in a sorted array, we

1. Check the middle element
2. If this middle element is same as query element , 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], L, R, value) {
    if (L>R)
        return -1;              // not found
    mid = ⌊(L+R)/2 ⌋;
    if (A[mid] == value)
        return mid;             // found
    else if (A[mid] < value)
        return BinarySearch (A[1, N], mid+1, R,value)
    else
        return BinarySearch (A[1..N], L, mid-1,value);
}
```

What’s the recurrence relation for its running time?

Recurrence for binary search

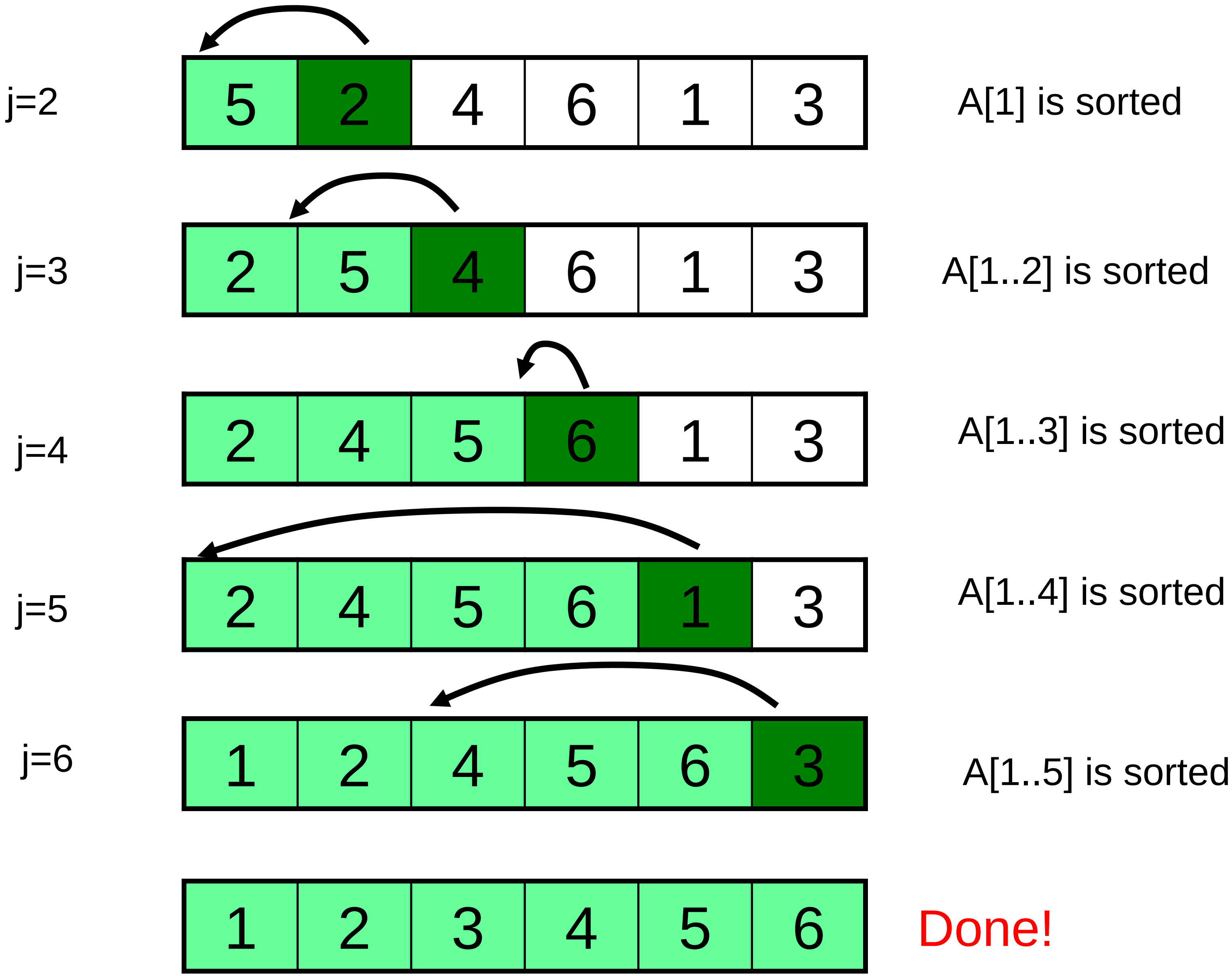
$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

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

Insertion Sort

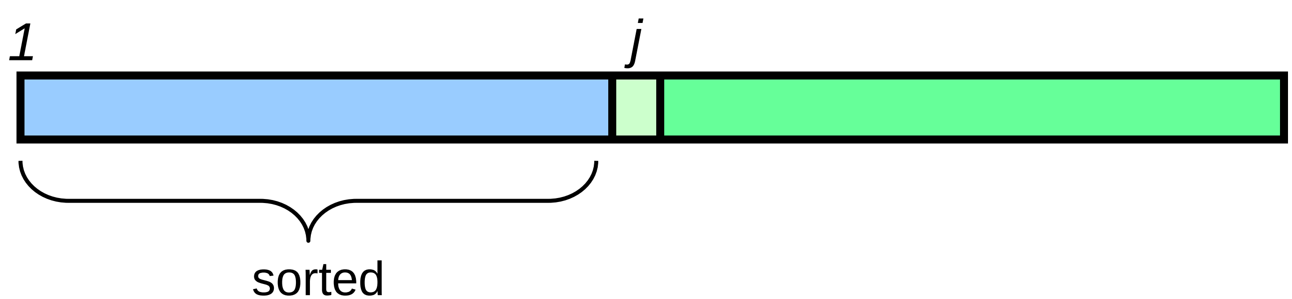
- For insertion sort we use an **incremental** approach
  - *Having sorted subarray A[1 .. j-1], we insert a single element A[j] into its proper place yielding the sorted subarray A[1 .. j]*
  - *Note that Insertion sort sorts "in place", meaning that it does not require any additional array for bookkeeping*
- *Insertion sort takes one parameter i.e., A*
  - *In this case we use A.length to denote number of elements present in A*
  - *Alternatively, it can take two parameters A and n, where n is the number of elements in A*

Example of insertion sort



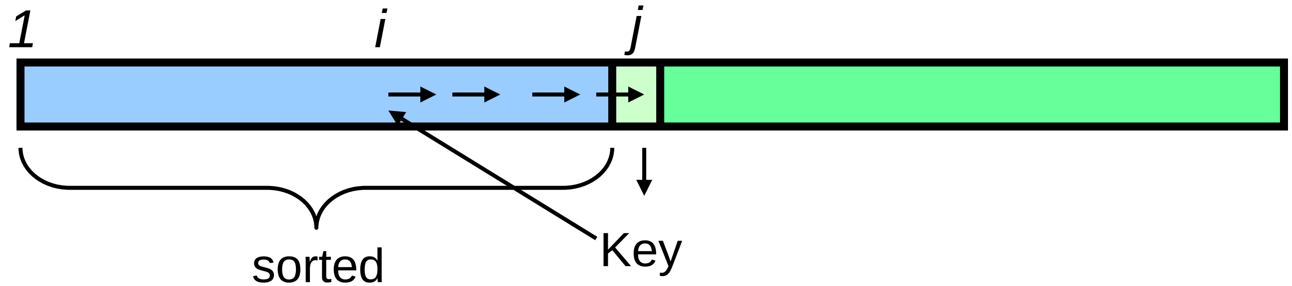
Insertion Sort

```
InsertionSort(A, n) {
  for j = 2 to n {
      ▹ Pre condition: A[1..j-1] is sorted

      1. Find position i in A[1..j-1] such that A[i] ≤ A[j] < A[i+1]
      2. Insert A[j] between A[i] and A[i+1]

      ▹ Post condition: A[1..j] is sorted
  }
}
```



Insertion Sort

```
InsertionSort(A, n) {
  for j = 2 to n {
    key = A[j];
    i = j - 1;
    while (i > 0) and (A[i] > key) {
        A[i+1] = A[i];
        i = i - 1;
    }
    A[i+1] = key
  }
}
```



Recursive Insertion Sort

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

Recursive Insertion Sort

Recursive_Insertion_Sort(A,n)

```
if n > 1
        Recursive_Insertion_Sort(A,n-1)
        key = A[n]
        i = n-1
        while i > 0 and A[i] > key
            A[i+1] = A[i]
            i = i - 1
        A[i + 1] = key
else
    do nothing
```

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.

Recurrence for computing factorial

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

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

- 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.

What do these 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 $T(n) = \Theta(n\lg n)$, 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 / iteration method
2. Substitution method
3. Master method

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 computing power

```
int pow (x, n)
    if(n==0) return 1;
    if(n==1) return x;
    return pow(x,⌊n/2⌋)*pow(x,⌈n/2⌉)
```

T(n) = ?

```
int pow (x, n)
    if(n==0) return 1;
    if(n==1) return x;
    if ((n % 2)==0)
        return pow(x*x, n/2);
    else
        return pow(x*x,⌊n/2 ⌋)*x;
```

T(n) = ?

Recurrence for computing power

```
int pow (x, n)
    if(n==0) return 1;
    if(n==1) return x;
    return pow(x,⌊n/2⌋)*pow(x,⌈n/2⌉)
```

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

```
int pow (x, n)
    if(n==0) return 1;
    if(n==1) return x;
    if ((n % 2)==0)
        return pow(x*x, n/2);
    else
        return pow(x*x, ⌊n/2 ⌋)*x;
```

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

Recurrence for merge sort

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

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.

$$
\begin{tikzcd}[column sep=small]
 & dn \arrow[dl, no head] \arrow[dr, no head] & \\
T(n/2) & & T(n/2)
\end{tikzcd}
$$

Recursion tree for merge sort

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

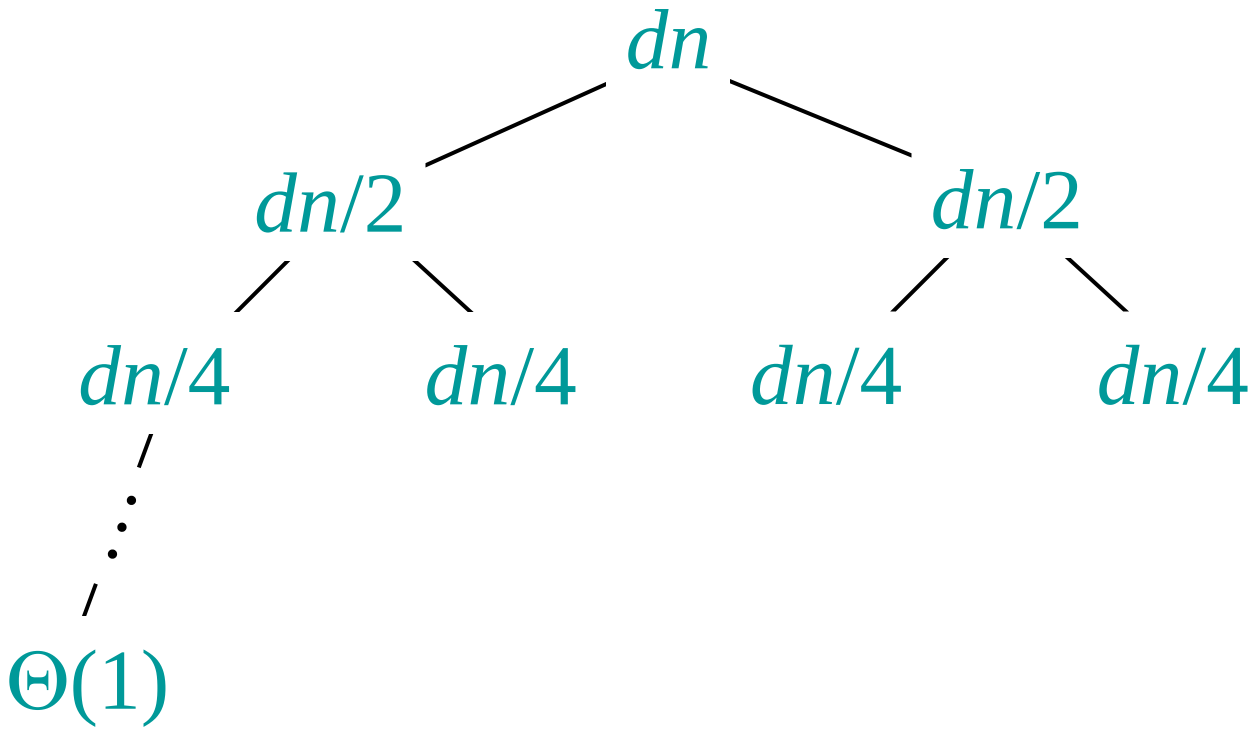
$$dn$$

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

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

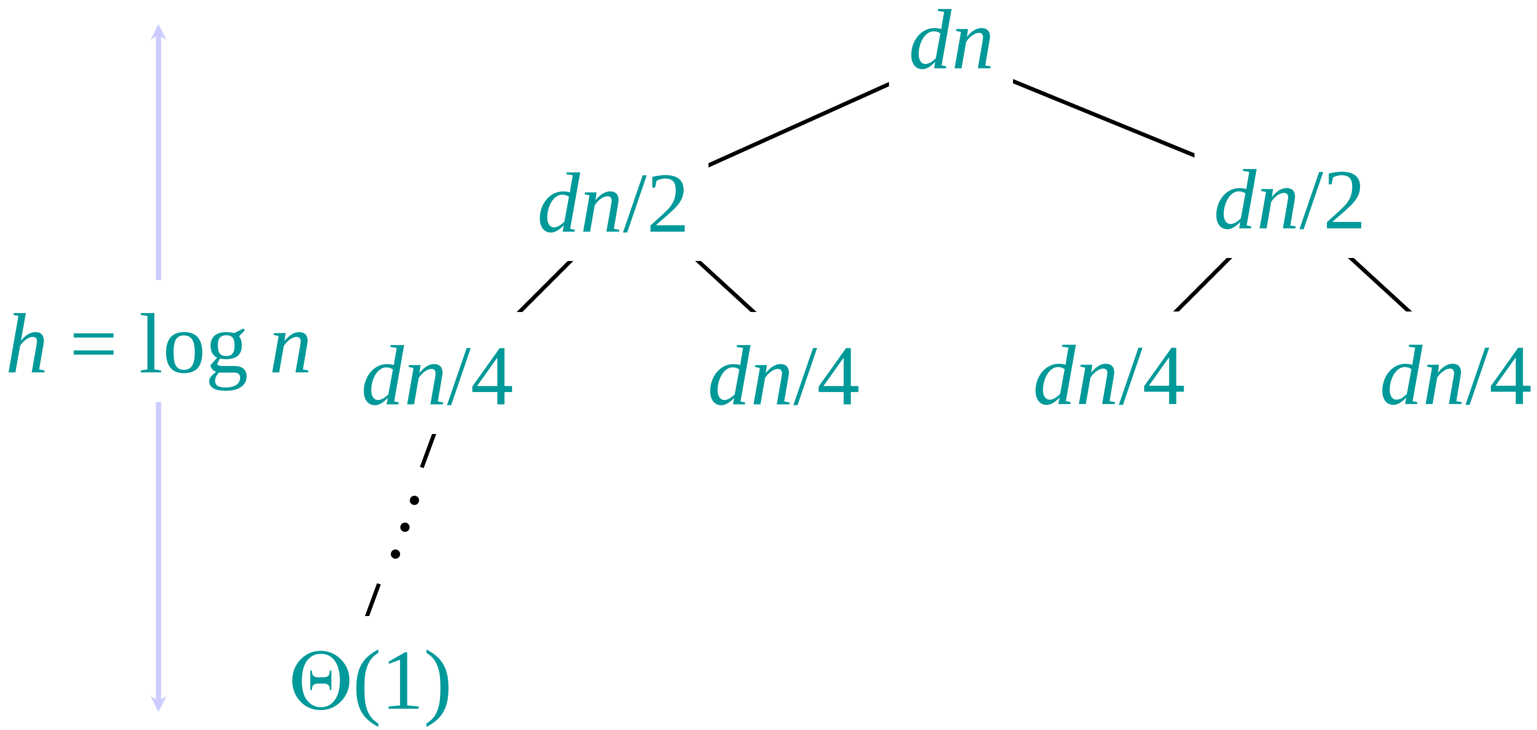
Recursion tree for merge sort

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



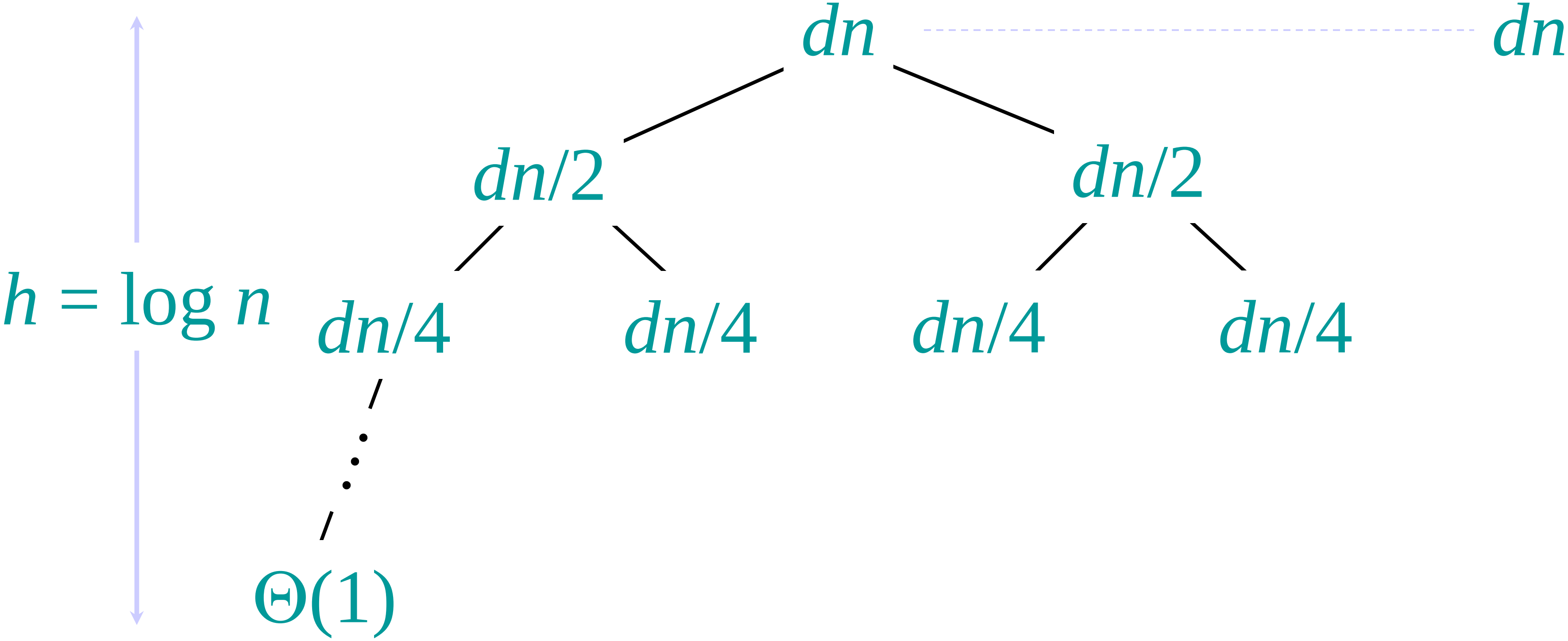
Recursion tree for merge sort

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



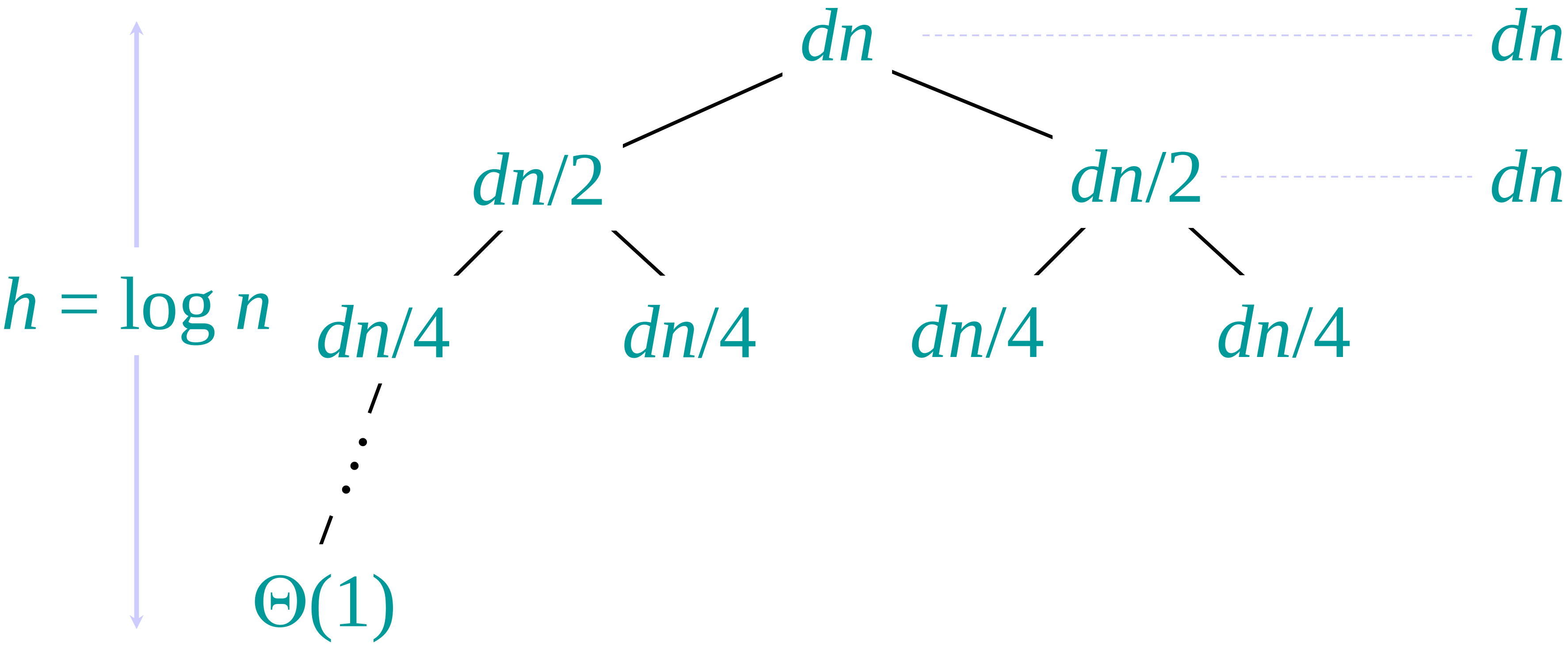
Recursion tree for merge sort

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



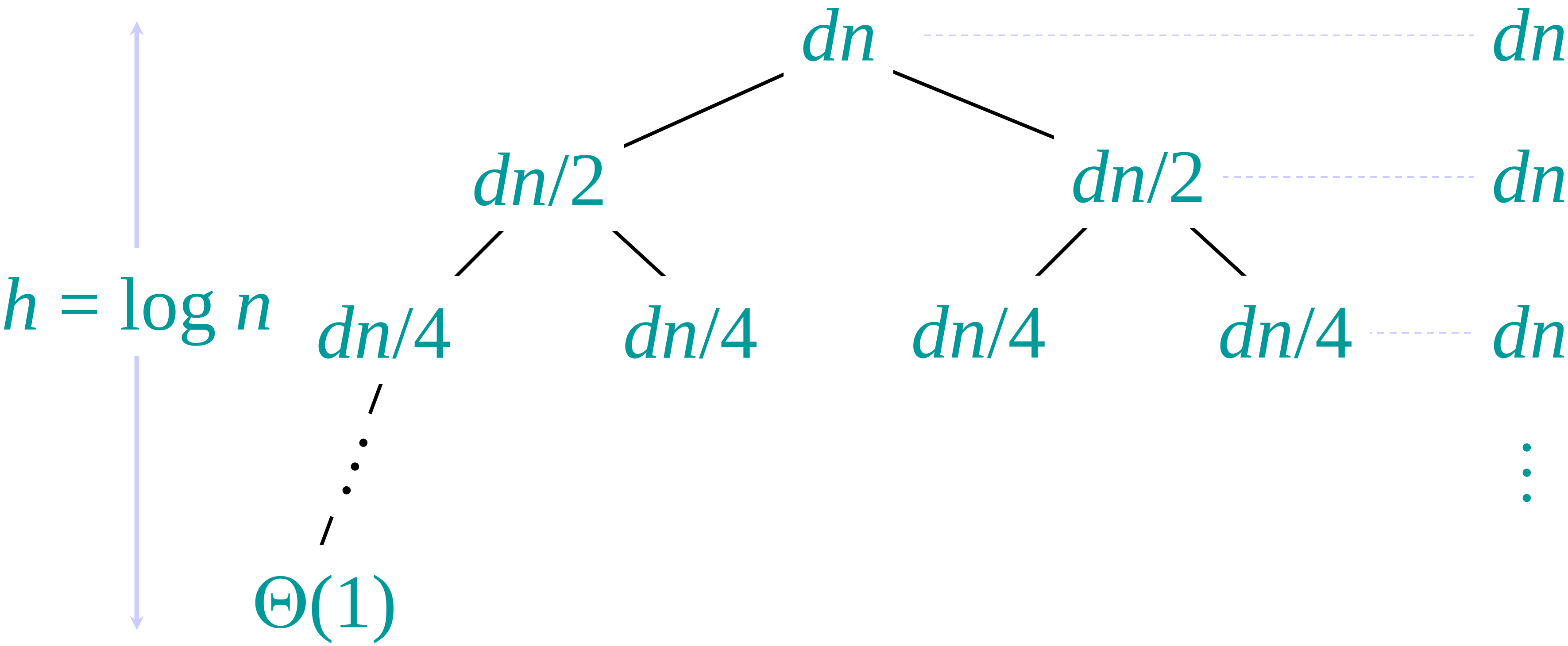
Recursion tree for merge sort

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



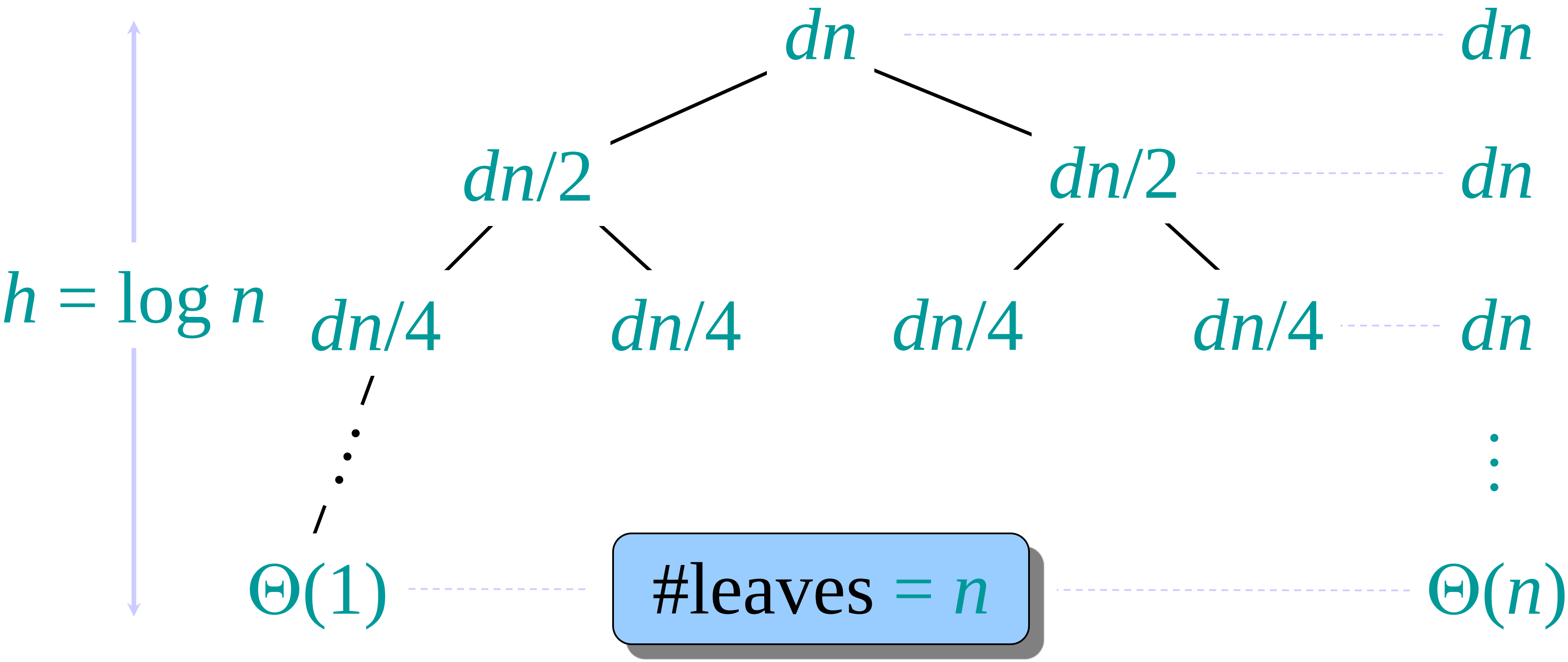
Recursion tree for merge sort

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



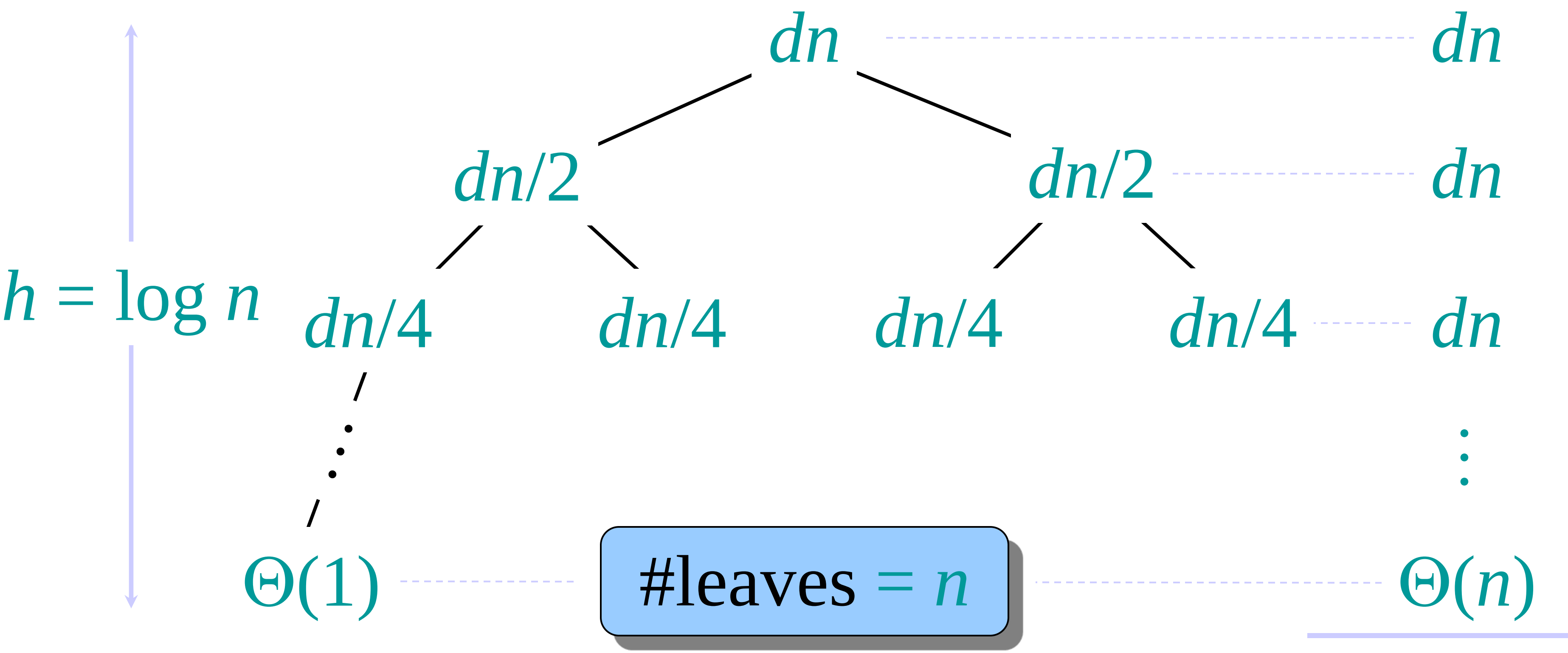
Recursion tree for merge sort

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)
    if(n==0) return 1;
    if(n==1) return x;
    return pow(x,⌊n/2⌋)*pow(x,⌈n/2⌉)
```

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

```
int pow (x, n)
    if(n==0) return 1;
    if(n==1) return x;
    if ((n % 2)==0)
        return pow(x*x, n/2);
    else
        return pow(x*x,⌊n/2 ⌋)*x;
```

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

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

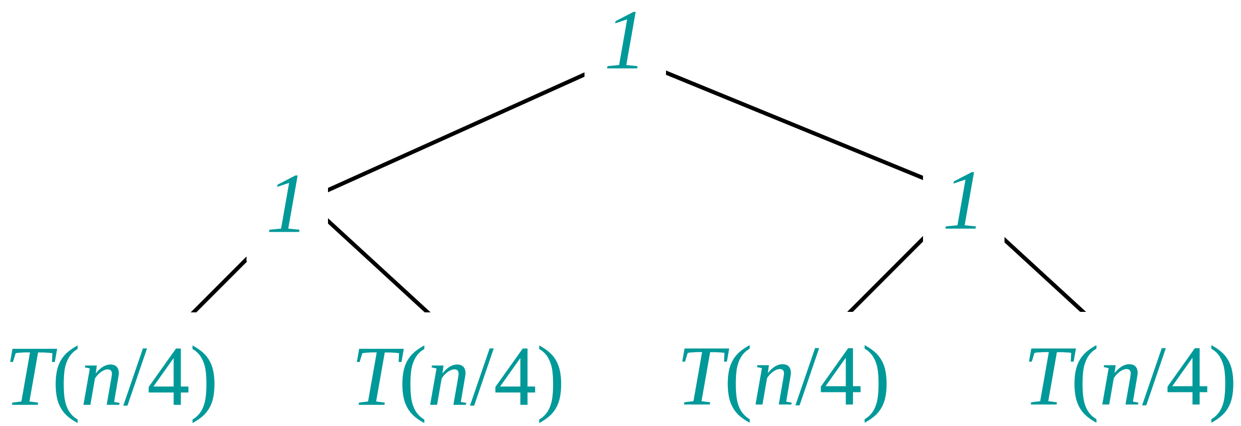
Solve $T(n) = 2T(n/2) + 1$.

$$1$$

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

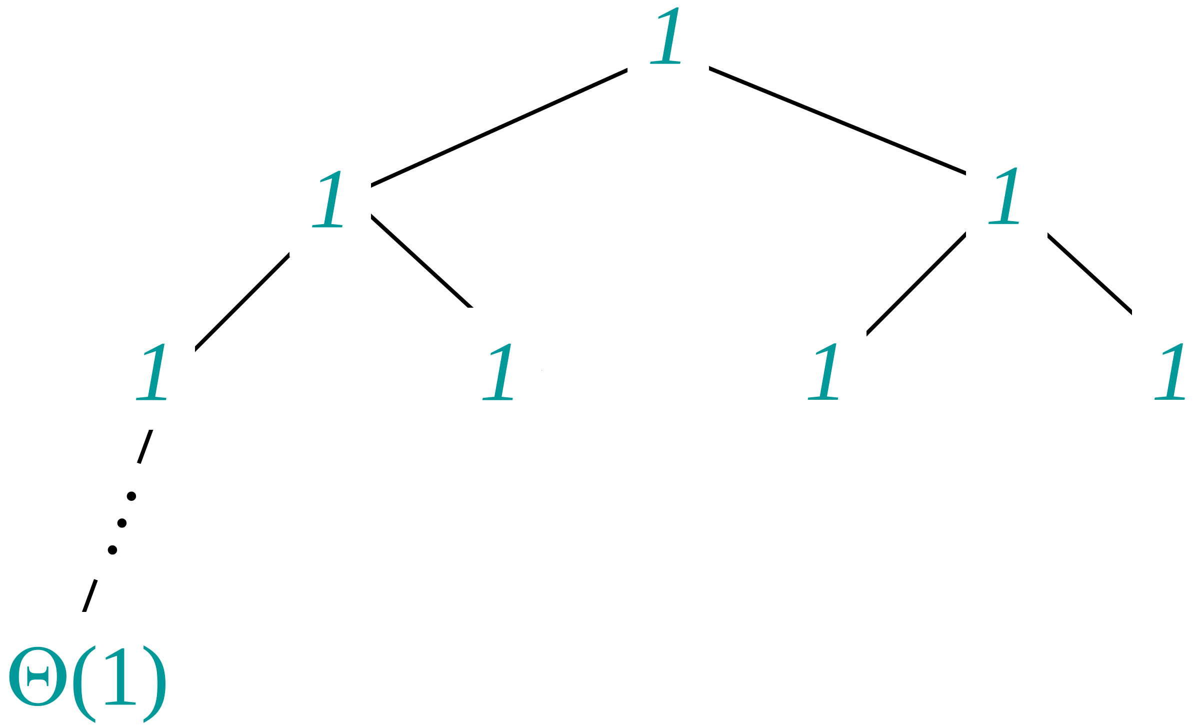
Time complexity for Alg2

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



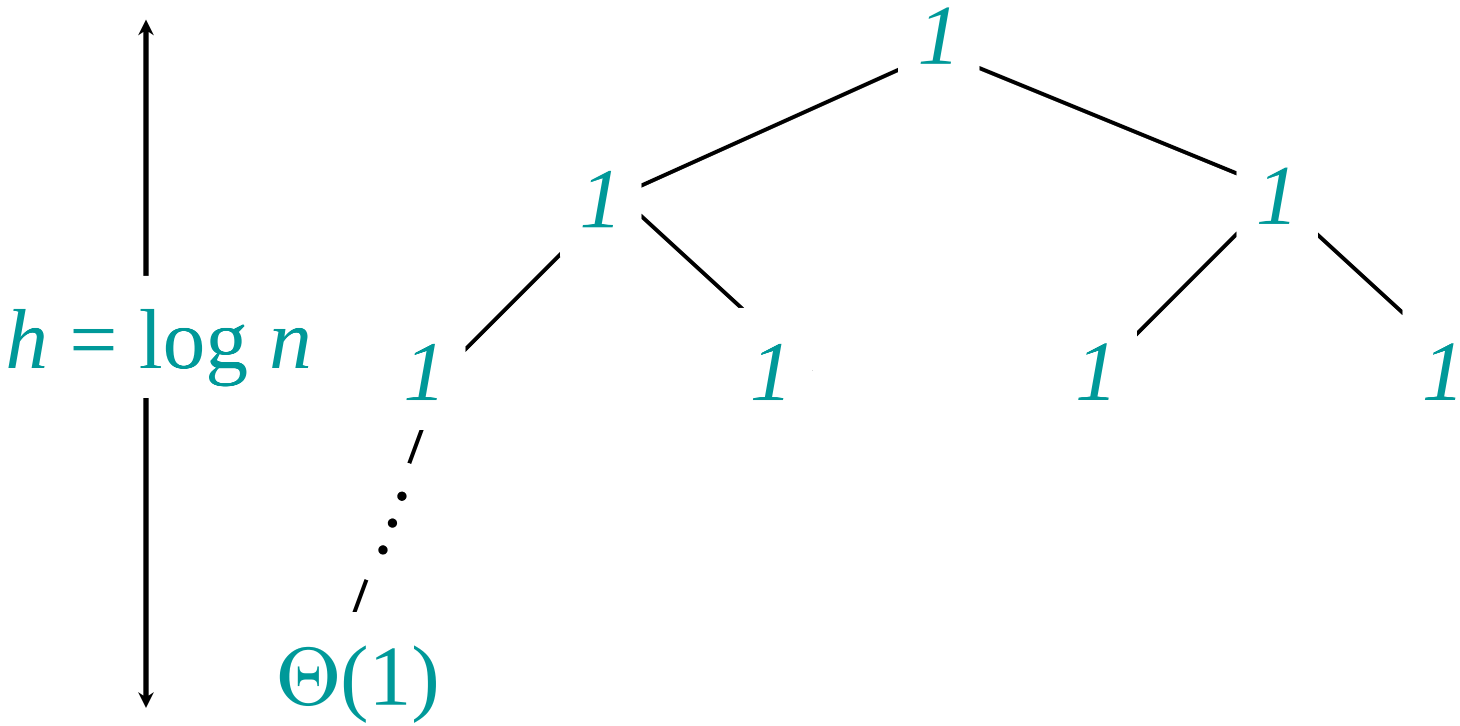
Time complexity for Alg2

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



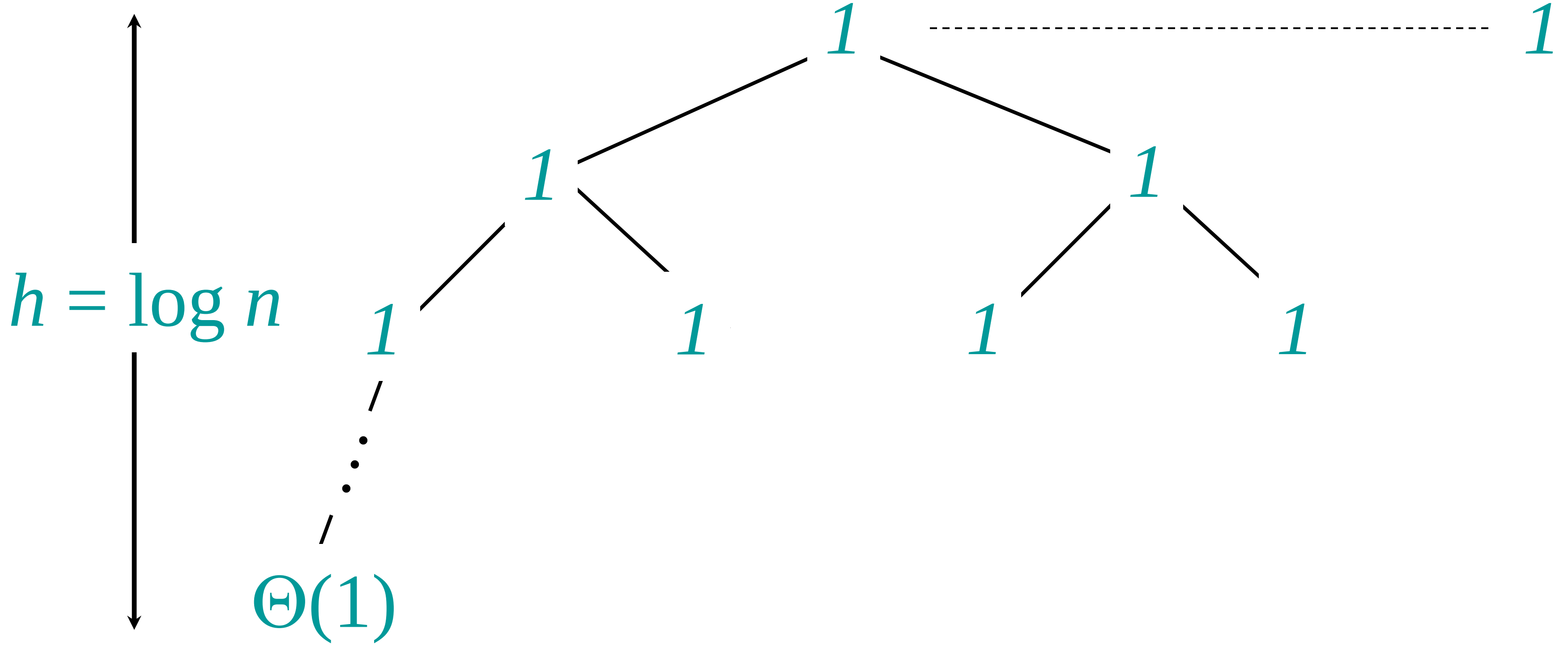
Time complexity for Alg2

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



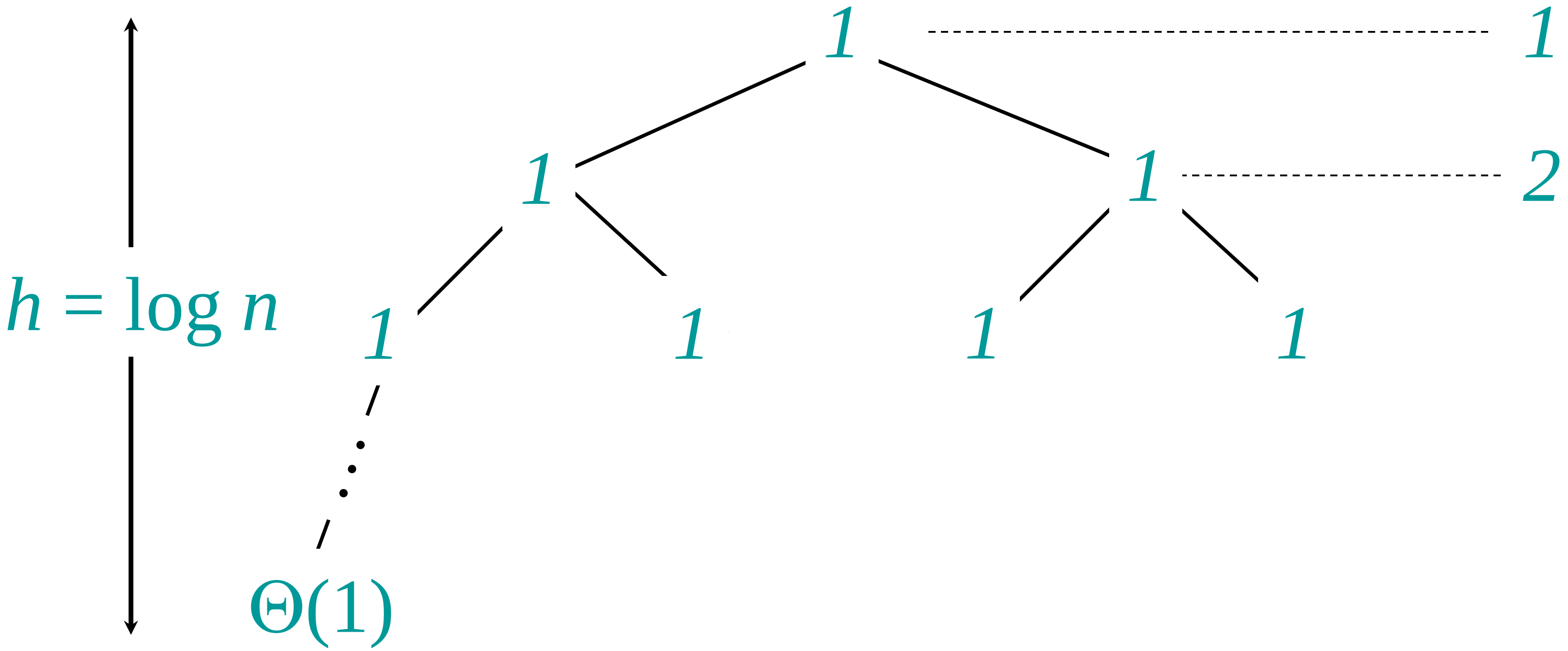
Time complexity for Alg2

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



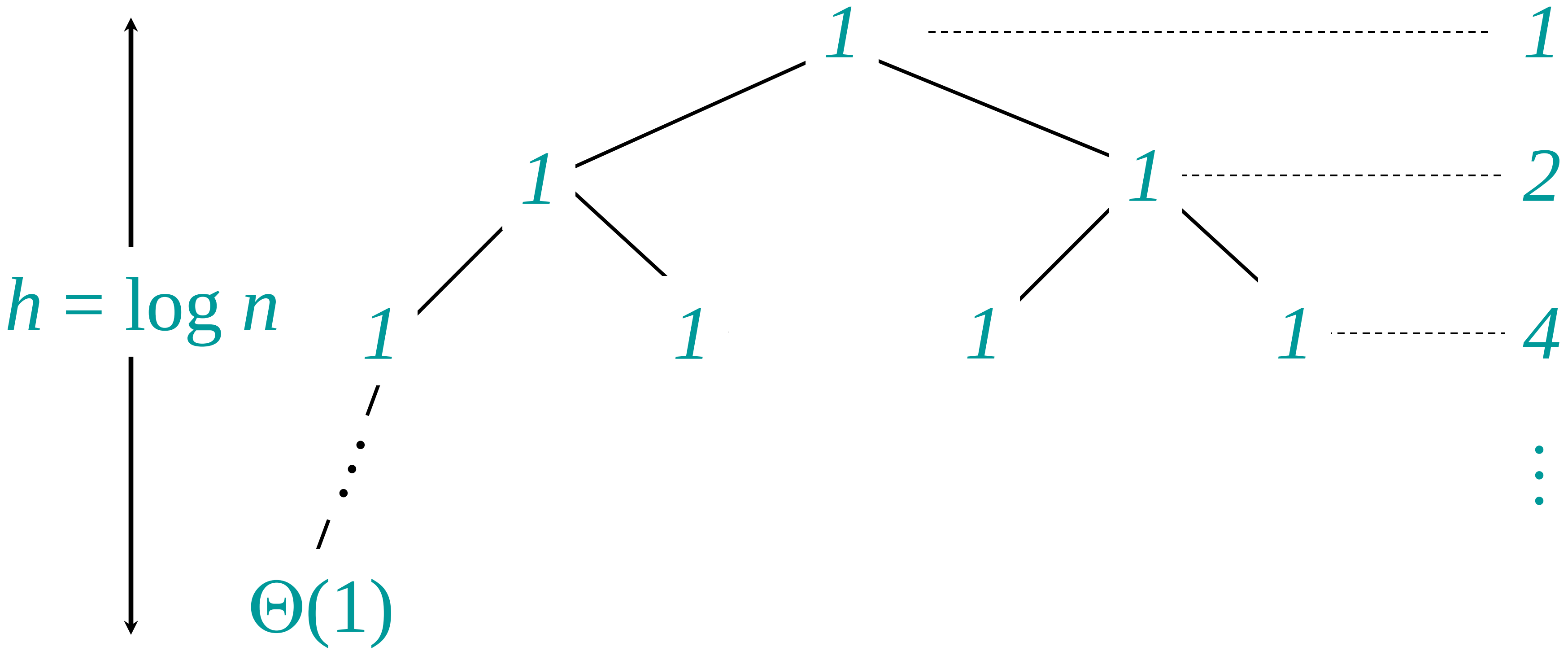
Time complexity for Alg2

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



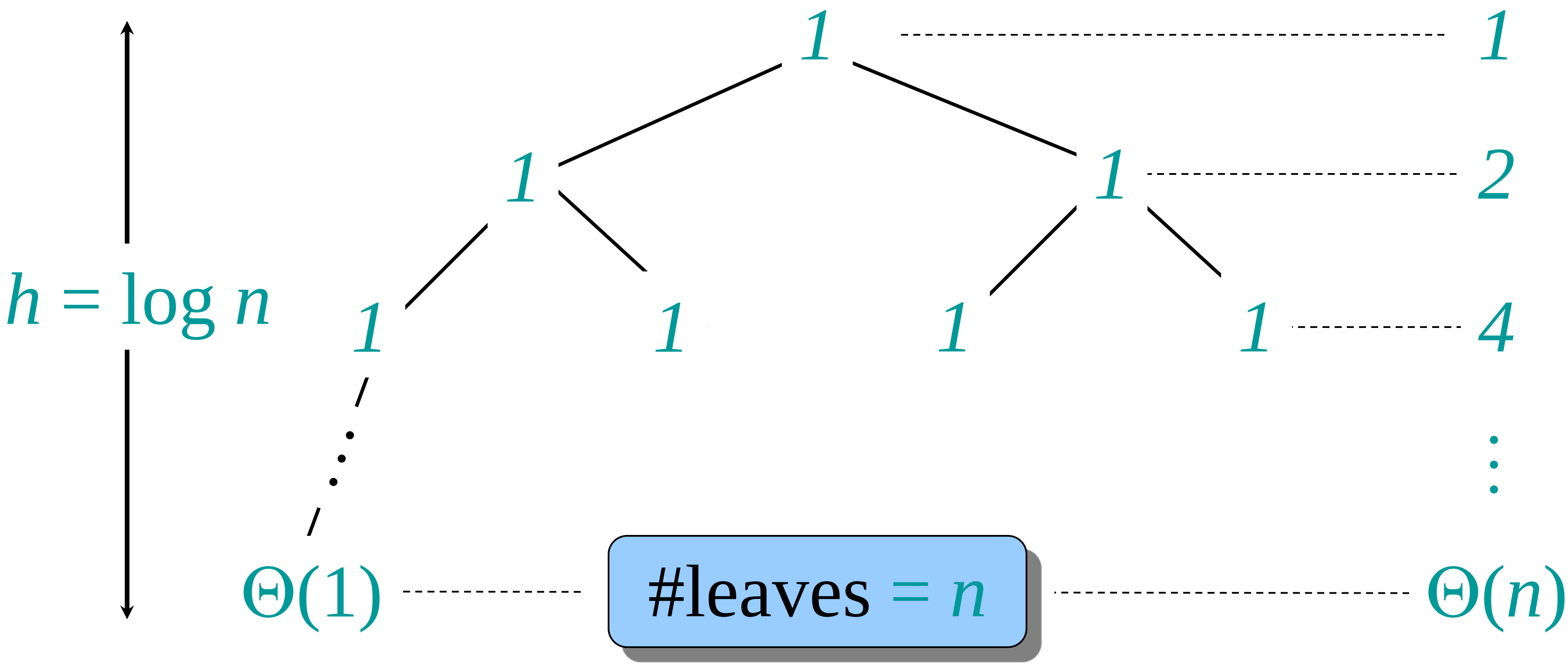
Time complexity for Alg2

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



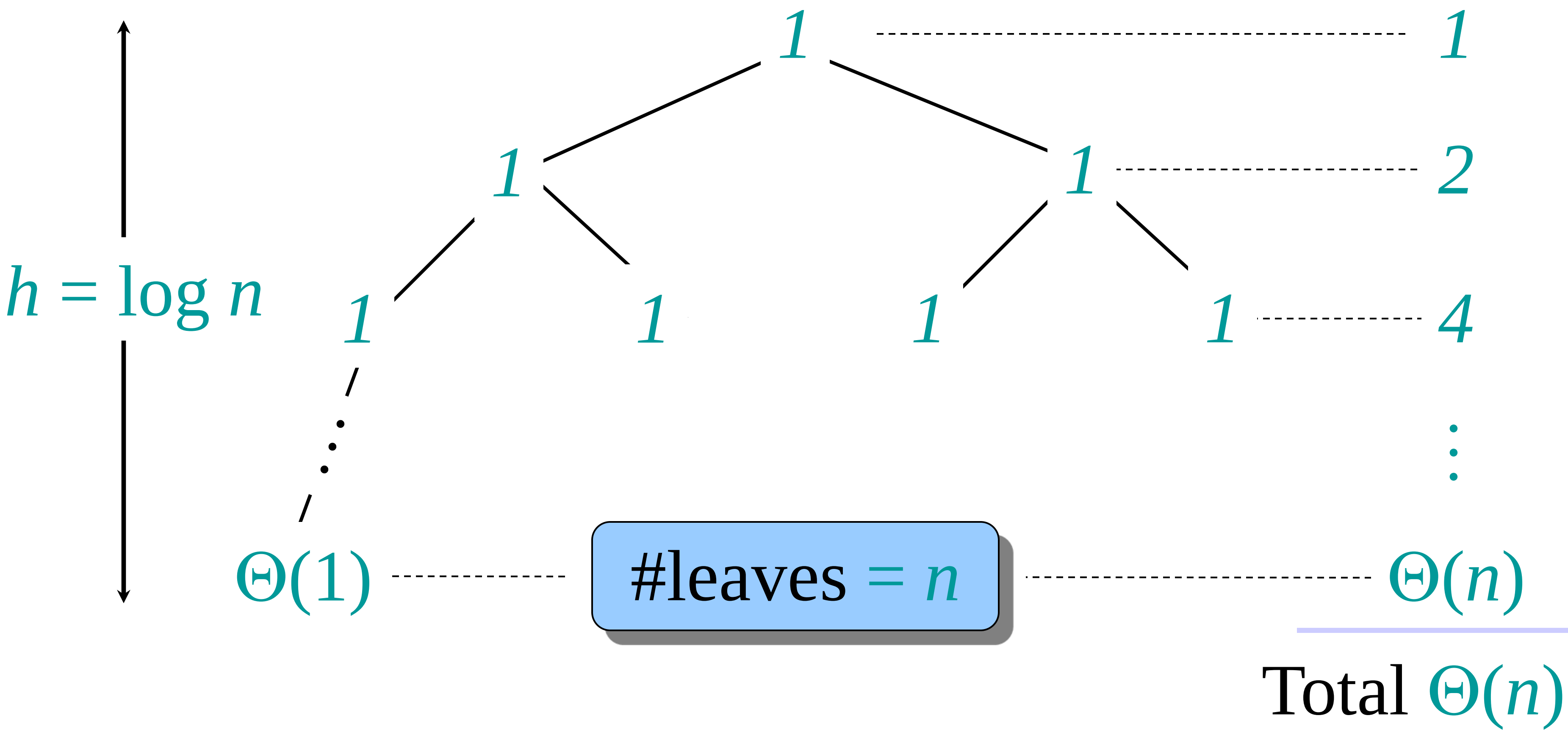
Time complexity for Alg2

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

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

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

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

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

  $= \Theta(n^2)$

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].
```

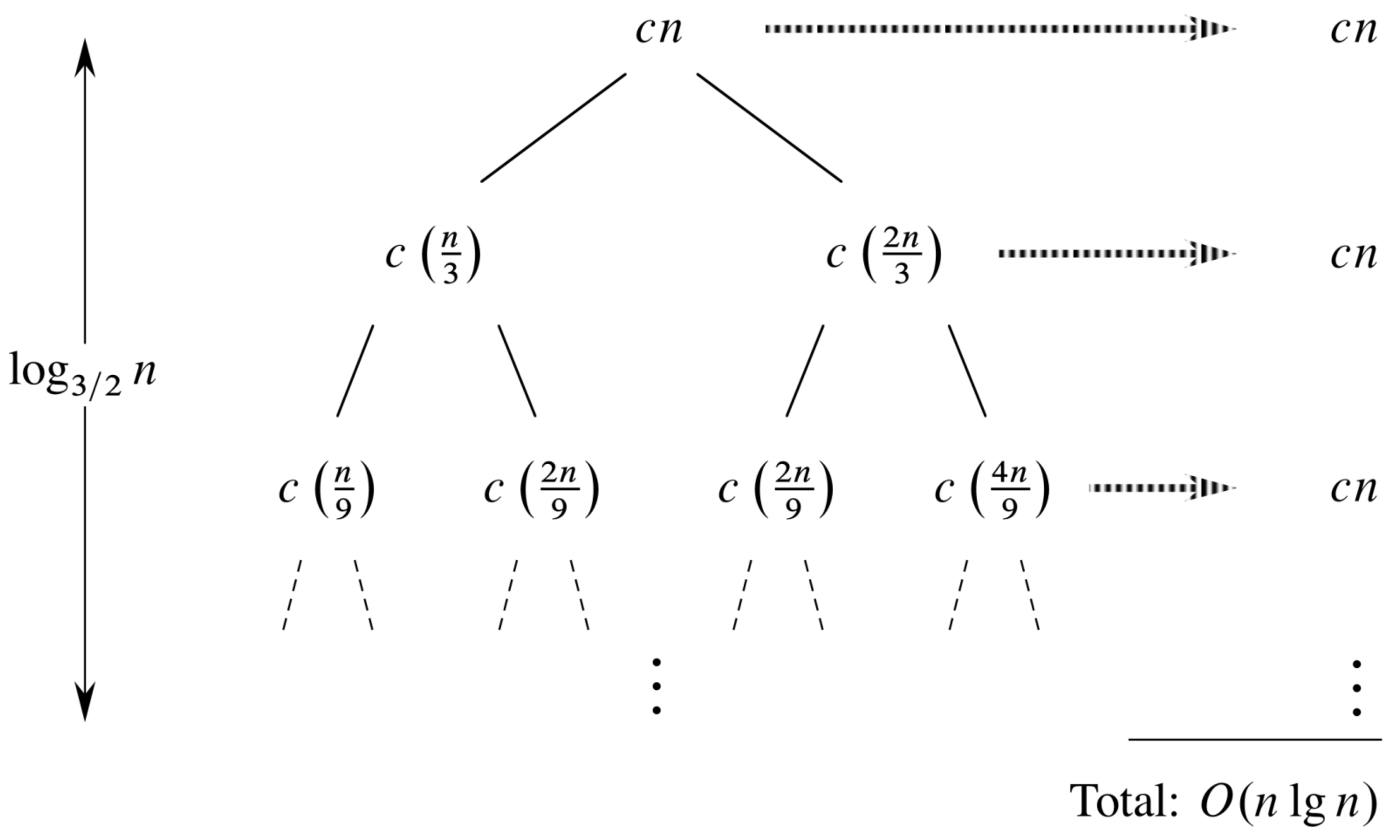
- Is this algorithm correct?
- What’s the recurrence function for the running time?
- What does the recurrence function solve to?

More recursion tree examples

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

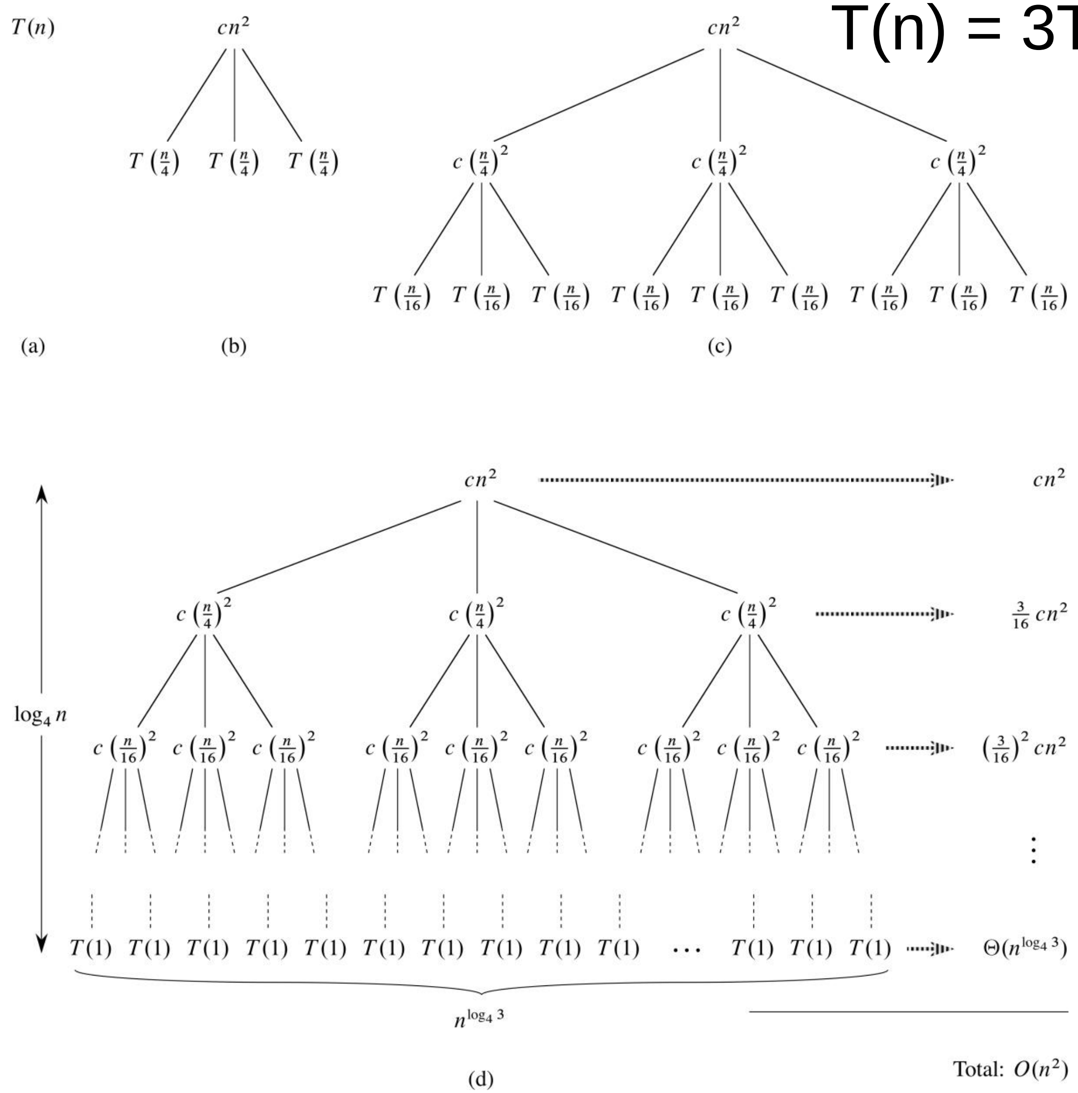
More recursion tree examples

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



More recursion tree examples

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



More recursion tree examples

- $T(n) = 3T(n/3) + n$ $\quad T(n) = \Theta(n\log n)$
- $T(n) = T(n/3) + T(2n/3) + n$ $\quad T(n) = \Theta(n\log n)$
- $T(n) = 3T(n/4) + n$ $\quad T(n) = \Theta(n)$
- $T(n) = 3T(n/4) + n^2$ $\quad T(n) = \Theta(n^2)$

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 $\cancel{a\,f(n/b) \le c\,f(n)}$

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^{ε} 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. $T(n) = T(n/2) + 1$ $\qquad \log_b a = 0$

$T(n) = 2\,T(n/2) + n$ $\qquad \log_b a = 1$

$T(n) = 4T(n/2) + n^2$ $\qquad \log_b a = 2$

$T(n) = 8T(n/2) + n^3$ $\qquad \log_b a = 3$

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^{ε} 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

- $a\,f(n/b) \le c\,f(n)$ 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?

- 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

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

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

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

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

Some tricks

- Changing variables

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

Changing variables

$$T(n) = 2T(n\text{-}1) + 1$$

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

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

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

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

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

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

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 $T(n) = T(2^m) = S(m)$

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

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

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

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

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

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

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

Fibonacci sequence

- Fibonacci sequence 1,1,2,3,5,8,13,21,34
  - Every number after the first two is sum of thye preceding two
  - How do we run a program to compute n-th Fibonacci number?
    - Iterative
    - recursive

Fibonacci sequence: Iterative

Fibonacci(n)

```
If ((n==1) or (n==2))
        return 1
else
        previous=1
        current=1
        for 3 to n
                next=previous+current
                previous=current
                current=next
        return current
```

- What is the running time T(n)?
  - T(n)=O(n)

Fibonacci sequence: Recursive

Fibonacci(n)

```
If ((n==1) or (n==2))
        return 1
else
        return (Fibonacci(n-1)+Fibonacci(n-2))
```

- What is the running time T(n)?
  - $T(n)=T(n-1)+T(n-2)+1$

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
    - Set $n=2^m$

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 Ω 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 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: $T(n) \leq c\, n \log (n)$
- Assume: $T(n/2) \leq cn/2 \log (n/2)$

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

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

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 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

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

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.

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

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

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

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

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

$\leq cn \log n$ (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.

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

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

$$
\begin{aligned}
T(n) &= T(n/3) + T(2n/3) + n \\
&\le cn/3 \log(n/3) + 2cn/3 \log(2n/3) + n \\
&\le cn \log n + n - cn (\log 3 - 2/3) \\
&\le cn \log n + n(1 - c\log 3 + 2c/3) \\
&\le cn \log n, \text{ for all } n > 0 \text{ (if } 1 - c \log 3 + 2c/3 \le 0)
\end{aligned}
$$

$$c \log 3 - 2c/3 \ge 1$$

$\Rightarrow c \ge 1 / (\log 3 - 2/3)$ > 0

Therefore, $T(n) = T(n/3) + T(2n/3) + n \le cn \log n$ for $c = 1 / (\log 3 - 2/3)$ 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.

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

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

$$\leq 3\, c\, n^2 / 16 + n^2$$

$$\leq (3c/16 + 1)\, n^2$$

$$\overset{?}{\leq} cn^2$$

$3c/16 + 1 \leq c$ 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 $T(n) \leq cn + n$
  – Our guess is wrong!! The correct answer is $T(n)= \Theta(n\log n)$

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 $T(n) = O(n – 1)$ and we know $O(n-1)=O(n)$

Details

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

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

$\leq cn + (1-2c)$

$\leq cn - (2c-1)$

$\leq cn$ (for any $c>=1/2$)

Another example

- Prove that $T(n) = 3T(n/3) + n^{0.5} = O(n)$
- Exercise in class

Details

- Prove that $T(n) = 3T(n/3) + n^{0.5} = O(n)$
- We need to prove that $T(n) <= cn$
- Assume that this holds for sub-problems of size n/3 and smaller.
  - That is, we assume $T(n/3) <= cn/3$
- Then,
- $T(n)=3T(n/3)+n^{0.5}$

  $<=3cn/3+n^{0.5}=cn+n^{0.5}$ This is not what we wanted!

Instead we will try to prove $T(n)=O(n-n^{0.5})$

Since know $O(n-n^{0.5})=O(n)$, that will be enough to argue $T(n)=O(n)$

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 Ω:

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.