Design and Analysis of Algorithms

CS3230

Week 3

Iteration, Recursion, and Divide-and-Conquer

Warut Suksompong

Iterative algorithms

Iterative algorithms

- Algorithms which have one or multiple loops, sequentially processing input elements

```
NDaysOfChristmas(gifts[2..n]):
  for i ← 1 to n
      Sing "On the ith day of Christmas, my true love gave to me"
      for j ← i down to 2
          Sing "j gifts[j],"
      if i > 1
          Sing "and"
      Sing "a partridge in a pear tree."
```

- Our running example in this lecture: **insertion sort**.

The problem of sorting

- ***Input:*** sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.
- ***Output:*** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- Example:
  - ***Input:*** 8 2 4 9 3 6
  - ***Output:*** 2 3 4 6 8 9

Insertion Sort

Runtime of $\Theta(n^2)$ already argued in last lecture slides

```
INSERTION-SORT(A[1..n])
for j = 2 to n
    key = A[j]
    // Insert A[j] into sorted seq A[1 .. j-1]
    i = j –1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i –1
    A[i+1] = key
```

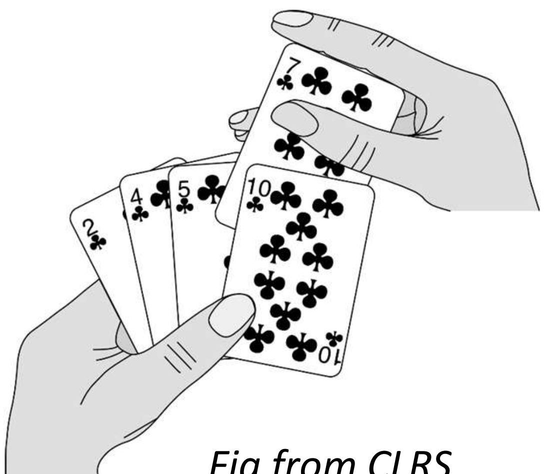


Fig from CLRS

Example run of Insertion Sort

Consider the iteration of the **for** loop where $j = 5$.

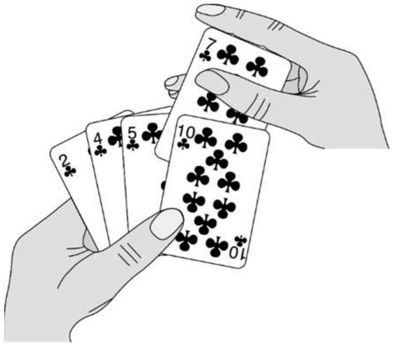
Suppose the array A at this stage is [1, 4, 6, 9, 2, 7, 3].

i	
4	1 4 6 9 2 7 3

0^{th} round of while loop

```
INSERTION-SORT(A[1..n])
for j = 2 to n
    key = A[j]
    // Insert A[j] into sorted seq A[1 .. j-1]
    i = j –1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i –1
    A[i+1] = key
```

Example run of Insertion Sort



Consider the iteration of the **for** loop where $j = 5$.

Suppose the array A at this stage is [1, 4, 6, 9, 2, 7, 3].

i		
4	1 4 6 9 2 7 3	0^{th} round of while loop
3	1 4 6 9 9 7 3	1^{st} round of while loop

```
INSERTION-SORT(A[1..n])
for j = 2 to n
    key = A[j]
    // Insert A[j] into sorted seq A[1 .. j-1]
    i = j –1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i –1
    A[i+1] = key
```

Example run of Insertion Sort

Consider the iteration of the **for** loop where $j = 5$.

Suppose the array A at this stage is [1, 4, 6, 9, 2, 7, 3].

i		
4	1 4 6 9 2 7 3	0th round of while loop
3	1 4 6 9 9 7 3	1st round of while loop
2	1 4 6 6 9 7 3	2nd round of while loop

```
INSERTION-SORT(A[1..n])
for j = 2 to n
    key = A[j]
    // Insert A[j] into sorted seq A[1 .. j-1]
    i = j –1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i –1
    A[i+1] = key
```

Example run of Insertion Sort

Consider the iteration of the **for** loop where $j = 5$.

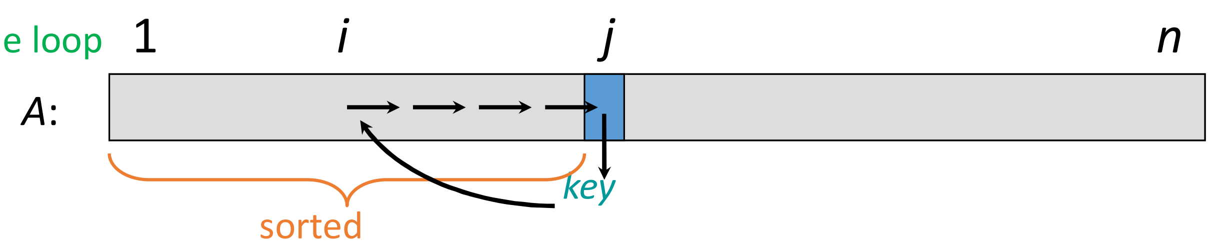
Suppose the array A at this stage is [1, 4, 6, 9, 2, 7, 3].

i		
4	1 4 6 9 2 7 3	0^{th} round of while loop
3	1 4 6 9 $\boxed{9}$ 7 3	1^{st} round of while loop
2	1 4 6 $\boxed{6\ 9}$ 7 3	2^{nd} round of while loop
1	1 4 $\boxed{4\ 6\ 9}$ 7 3	3^{rd} round of while loop

End of while loop

INSERTION-SORT(*A[1..n]*)

```
for j = 2 to n
    key = A[j]
    // Insert A[j] into sorted seq A[1 .. j-1]
    i = j –1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i –1
    A[i+1] = key
```



Correctness of Iterative Algorithms

The key step in the reasoning about the correctness of iterative algorithms is finding a:

Loop invariant

- True before the first iteration
- If true before an iteration, then remains true at the beginning of the next iteration
- If true at the end, then it implies algorithm's correctness

See review of induction in supplementary material!

Example: Step 1 (for loop) of Insertion srot

j		
	8 2 4 9 3 6	0th round of for loop
2	2 8 4 9 3 6	1st round of for loop
3	2 4 8 9 3 6	2nd round of for loop
4	2 4 8 9 3 6	3rd round of for loop
5	2 3 4 8 9 6	4th round of for loop
6	2 3 4 6 8 9	6th round of for loop

```
INSERTION-SORT(A[1..n])
1.  for j = 2 to n
2.      key = A[j]
3.      // Insert A[j] into sorted seq A[1 .. j-1]
4.      i = j –1
5.      while i > 0 and A[i] > key
6.          A[i+1] = A[i]
7.          i = i –1
8.      A[i+1] = key
```

By inspection, the invariant is “A[1..j-1] is the sorted list of elements originally in A[1..j-1]”.

How to use invariant to show the correctness of an iterative algorithm?

To understand the correctness of an algorithm using an invariant, we need to show three things:

- **Initialization:** The invariant is true before the first iteration of the loop
- **Maintenance:** If the invariant is true before an iteration, it remains true before the next iteration
- **Termination:** When the algorithm terminates, the invariant provides a useful property for showing correctness.

Invariant: the subarray A[1 .. j-1] consists of the elements originally in A[1 .. j-1], but in sorted order

- **Initialization:** Before the start of the first iteration, *j* has been initialized to 2. The subarray *A*[1 .. *j*-1] is just *A*[1], which is trivially sorted.

INSERTION-SORT(*A[1..n]*)

```
for j = 2 to n
    key = A[j]
    // Insert A[j] into sorted seq A[1 .. j-1]
    i = j –1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i –1
    A[i+1] = key
```

Invariant: the subarray A[1 .. j-1] consists of the elements originally in A[1 .. j-1], but in sorted order

- **Maintenance:** (Sketch) By the property of the invariant, *A*[1 .. *j*-1] is sorted.
  - Line 2 assigns *A*[*j*] to *key*.
  - The **while** loop ensures that all array entries in *A*[1 .. *j*-1] larger than *key* is shifted one place to the right.
  - Line 8 assigns *key* to location created by shifts.
  - Then, A[1..j] is sorted!

```
INSERTION-SORT(A[1..n])
1. for j = 2 to n
2.     key = A[j]
3.     // Insert A[j] into sorted seq A[1 .. j-1]
4.     i = j –1
5.     while i > 0 and A[i] > key
6.         A[i+1] = A[i]
7.         i = i –1
8.     A[i+1] = key
```

Invariant: the subarray A[1 .. j-1] consists of the elements originally in A[1 .. j-1], but in sorted order

- **Termination:** Array length is *n* and after the final loop, *j* is incremented to *n*+1. From the invariant, we have *A*[1 .. *j*-1] being sorted. Substituting *j*, the whole array is sorted.

```
INSERTION-SORT(A[1..n])
for j = 2 to n
    key = A[j]
    // Insert A[j] into sorted seq A[1 .. j-1]
    i = j –1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i –1
    A[i+1] = key
```

Invariant of Iterative Algorithm

- Recap:
  - Invariant is a condition that is true at the beginning of every iteration
  - To show an invariant is true, we need to show that the invariant is true at initialization, is correctly maintained, and implies correctness with termination condition.

Question 1

The pseudo-code for selection sort is given below:

```
SELECTION-SORT(A)
n = A.length
for j = 1 to n-1
    smallest = j
    for i = j+1 to n
        if A[i] < A[smallest]
            smallest = i
    exchange A[j] with A[smallest]
```

What is a suitable loop invariant for the outer loop?

- The array A is sorted.
- The array A[1..j-1] is sorted.
- The array A[1..j-1] contains the j-1 smallest elements of the array A[1..n]
- The array A[1..j-1] is sorted and contains the j-1 smallest elements of the array A[1..n].

Answer to Question 1

Answer: The array $A[1..j\text{-}1]$ is sorted and contains the j-1 smallest elements of the array $A[1..n]$

Termination: The invariant needs to imply a sorted array when the loop terminates.

- At termination, $j == n$, so by the invariant, $A[1..n\text{-}1]$ is sorted and does not have elements larger than $A[n]$, implying that the whole array is sorted.

Answer to Question 1

Initialization: $A[1..j\text{-}1]$ is empty, so invariant is true.
Maintenance: Need invariant for inner loop stating that $A[smallest]$ is the smallest element in $A[j .. i\text{-}1]$.
When inner loop terminate, $i == n + 1$, so $A[smallest]$ is the smallest element in $A[j .. n]$.
If outer invariant is true before loop, it will be true after swapping on last line and incrementing j.

Divide-and-conquer algorithms

Divide-and-Conquer

Divide the problem into a number of subproblems that are smaller instances of the same problem.

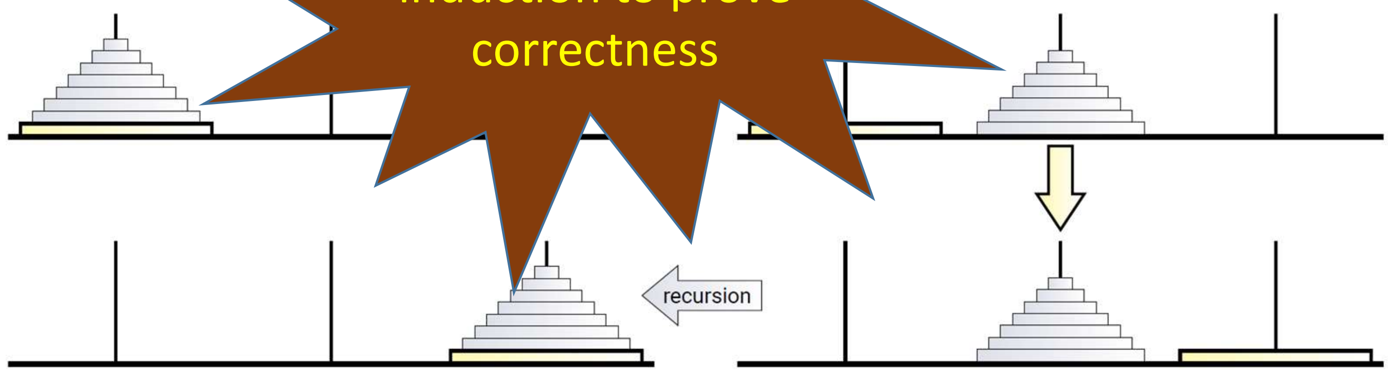
Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

(CLRS, pg. 86)

Tower-of-Hanoi

Use the obvious induction to prove correctness



(*Algorithms*. Erickson, pg. 25)

<u>HANOI(n, src, dst, tmp):</u>
    if $n > 0$
        HANOI($n-1$, src, tmp, dst)   ⟨⟨*Recurse!*⟩⟩
        move disk n from src to dst
        HANOI($n-1$, tmp, dst, src)   ⟨⟨*Recurse!*⟩⟩

Warning: don't try to unroll recursion. Head will explode!

Merge Sort

Input is an array $A[1, , \ldots, r]$

MERGE-SORT$(A, 1, r)$

if $p < r$

$q = \lfloor (1 + r)/2 \rfloor$

MERGE-SORT$(A, 1, q)$

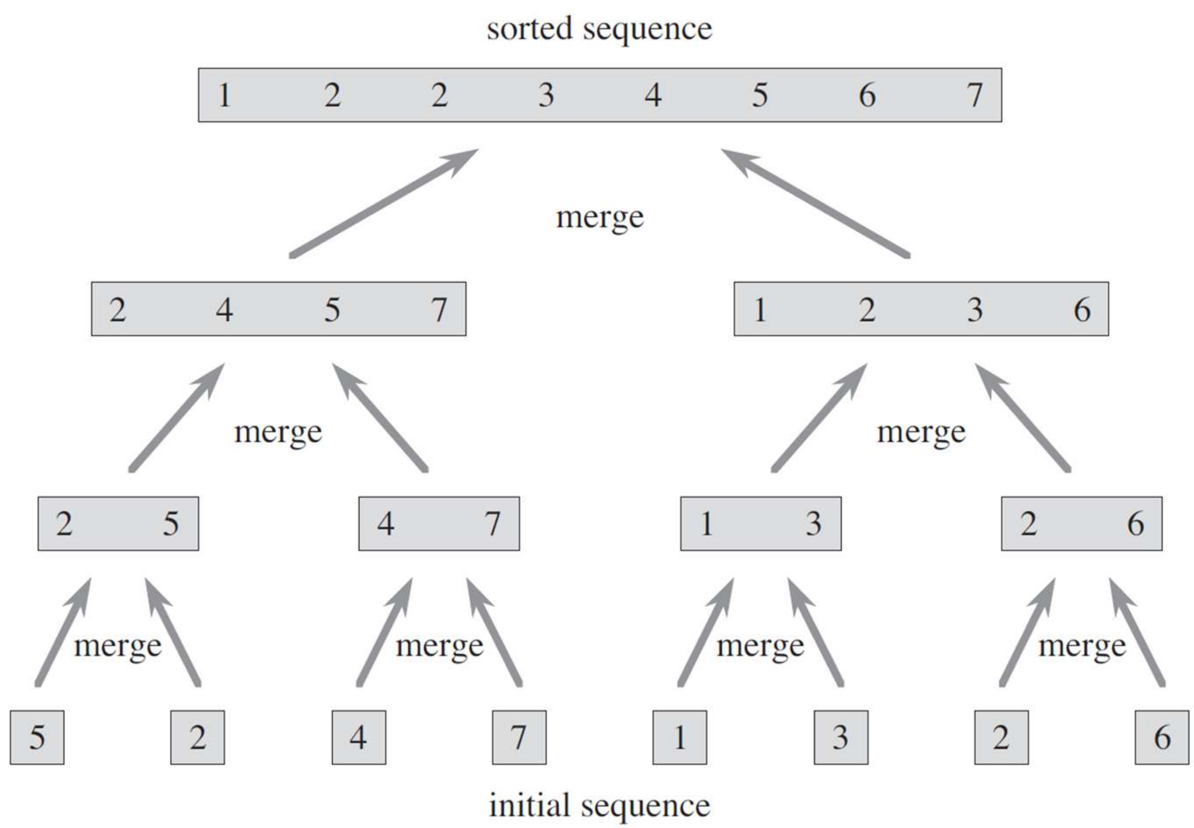
MERGE-SORT$(A, q + 1, r)$

MERGE$(A, 1, q, r)$

Use the obvious induction to prove correctness

Merges two sorted $A[1, .., q]$ and $A[q + 1, .., r]$ into one sorted

The recursion tree



(CLRS, pg. 35)

Correctness of Recursive Algorithms

- Use strong induction
- Prove base cases
- Show algorithm works correctly assuming algorithm works correctly for all smaller cases

How to analyze the running time of a recursive algorithm?

1. Derive a recurrence
   - Already seen one example (Recursive algorithm for Fibonacci number)

2. Solve the recurrence

Analyzing Tower-Of-Hanoi

$T(n)$	<u>$\textsc{Hanoi}(n, src, dst, tmp)$:</u>
	if $n > 0$
$T(n-1)$	$\quad\textsc{Hanoi}(n-1, src, tmp, dst)$
1	$\quad$move disk n from src to dst
$T(n-1)$	$\quad\textsc{Hanoi}(n-1, tmp, dst, src)$

Recurrence: $T(1) = 1$ and $T(n) = 2 \cdot T(n-1) + 1$.

Claim: $T(n) = 2^n - 1$.

Proof: By induction. Base case is $n = 1$ which holds. Assuming induction hypothesis, $T(n+1) = 2 \cdot (2^n - 1) + 1 = 2^{n+1} - 1$.

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 $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$.
$\Theta(n)$	3. ***"Merge"*** the 2 sorted lists

Unspecified constants

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

- See lecture notes for how to argue this formally for this recurrence
- For a general result showing that floors and ceilings don't matter, check out the SODA '21 paper posted on LumiNUS.

Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & if\ n > 1 \end{cases}$$

- We <u>usually omit stating the base case when $T(n) = \Theta(1)$</u> for sufficiently small n.
- Next, we describe a few ways to solve the recurrence to find a good upper bound on $T(n)$.

Recurrences for Divide-and-Conquer

Divide, **conquer**, **combine**.
Consider the recurrence

$$T(n) = \boldsymbol{a}T(\boldsymbol{n/b}) + \boldsymbol{f(n)}$$

#sub-problems

sub-problem size

time to divide and combine

How to solve a recurrence?

- Recursion tree
- Master method
- Substitution method

Recursion tree

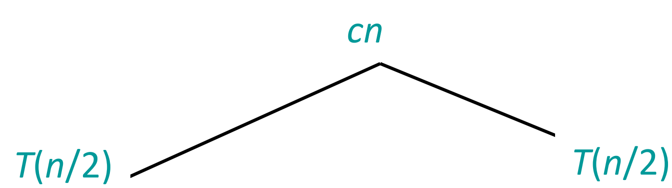
Recursion tree

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

$T(n)$

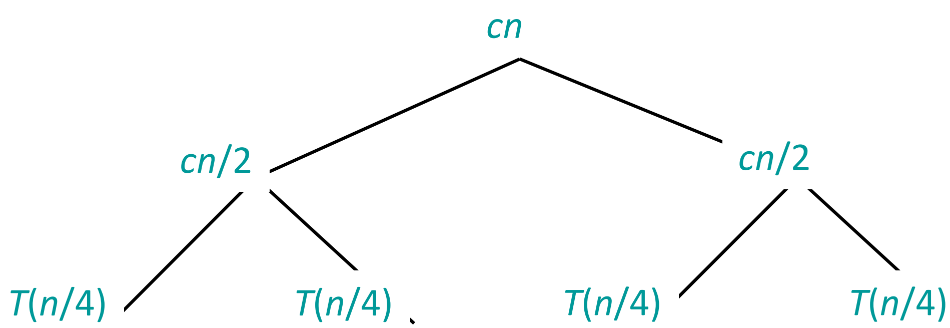
Recursion tree

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



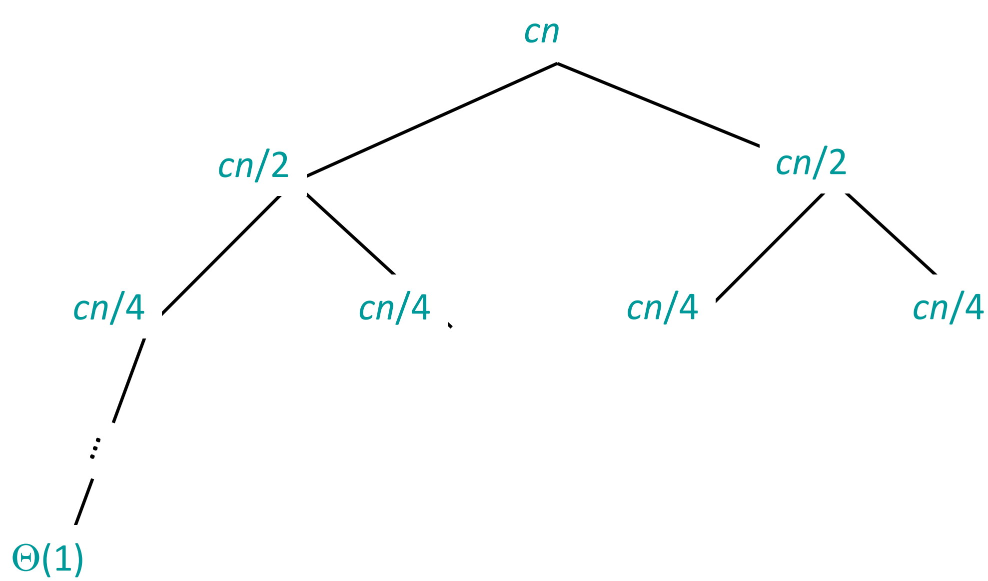
Recursion tree

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



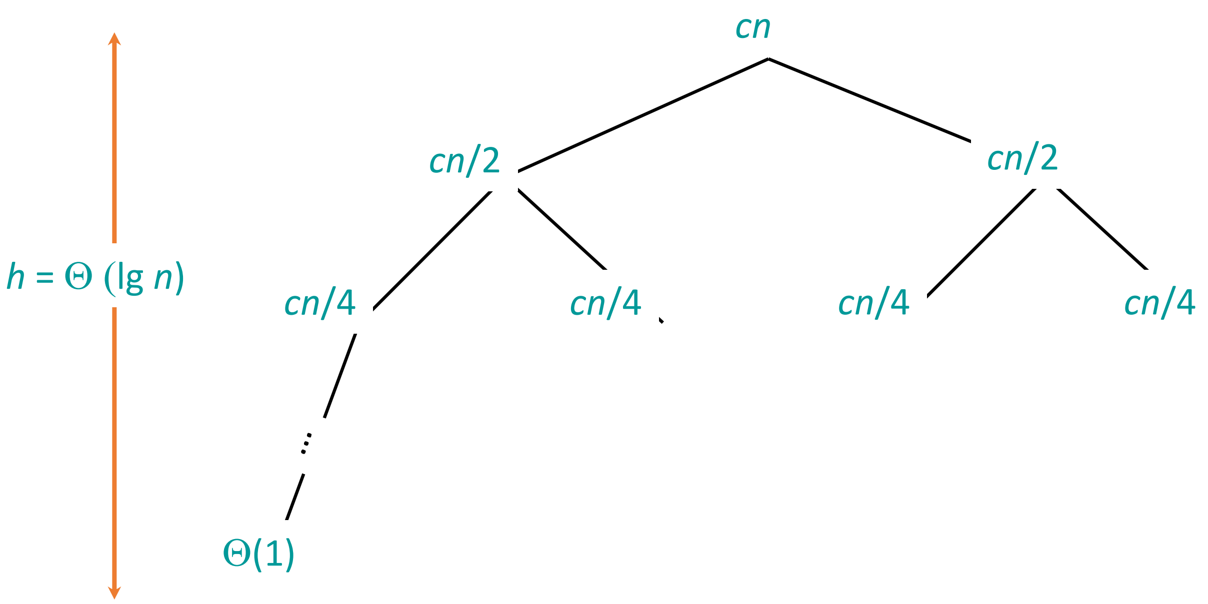
Recursion tree

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



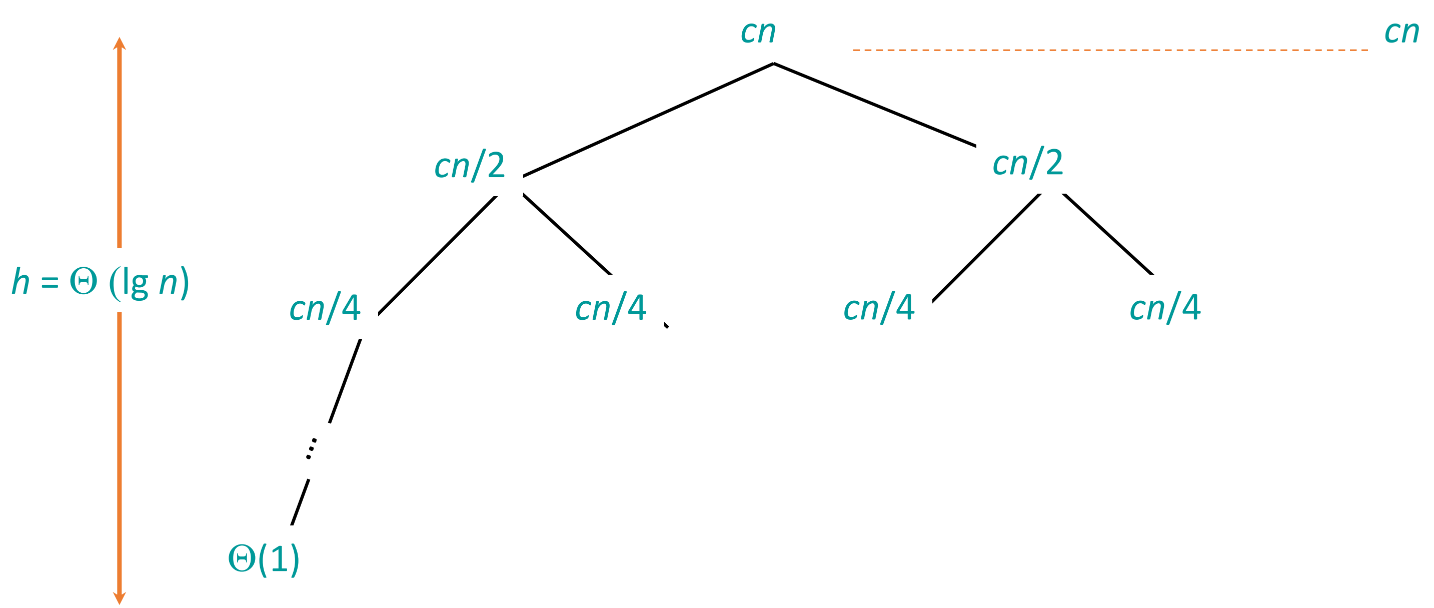
Recursion tree

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



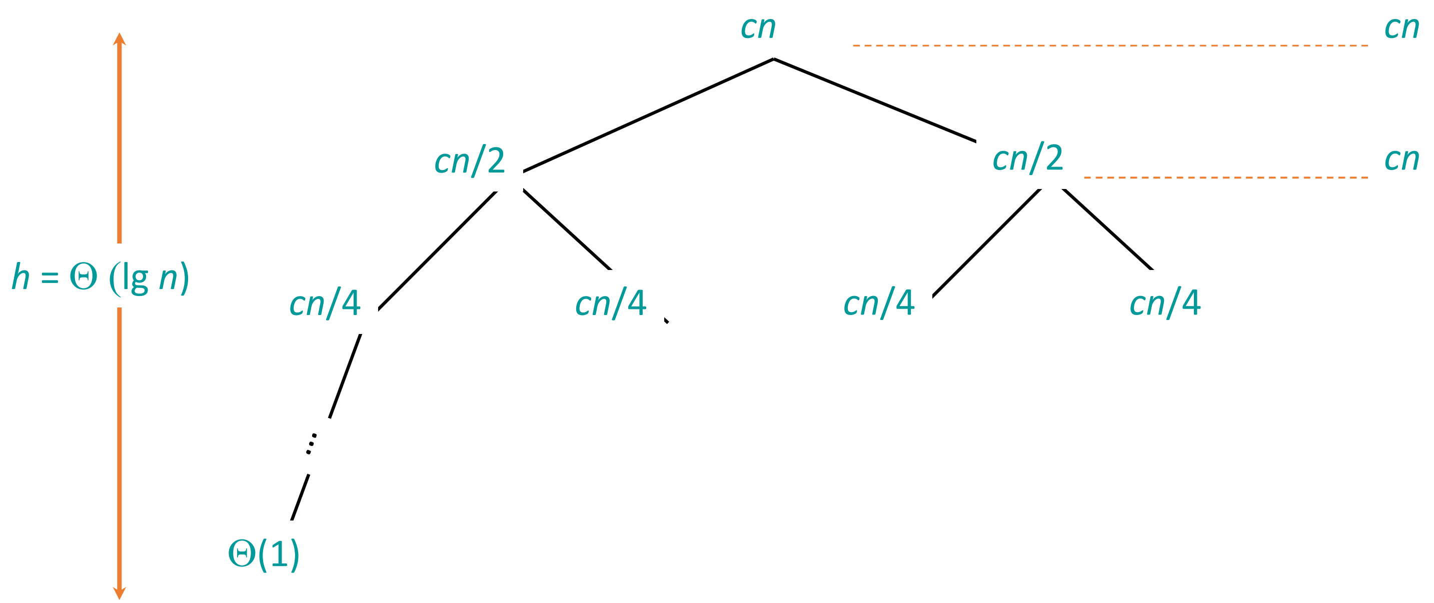
Recursion tree

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



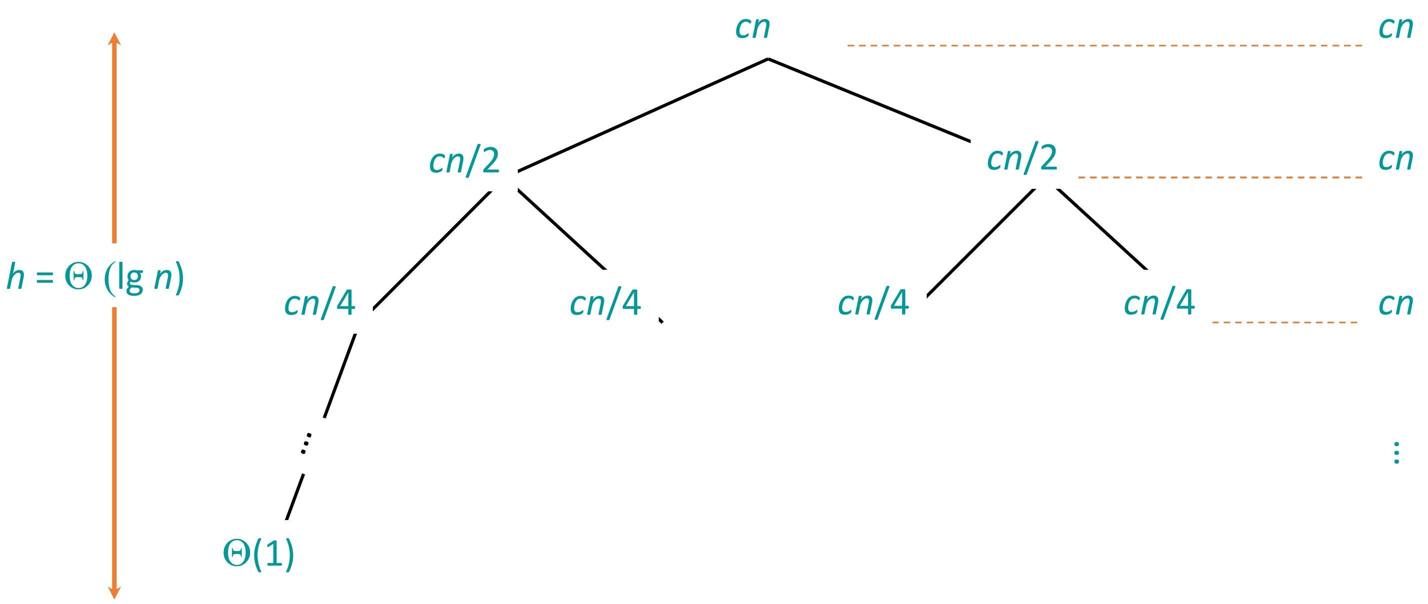
Recursion tree

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



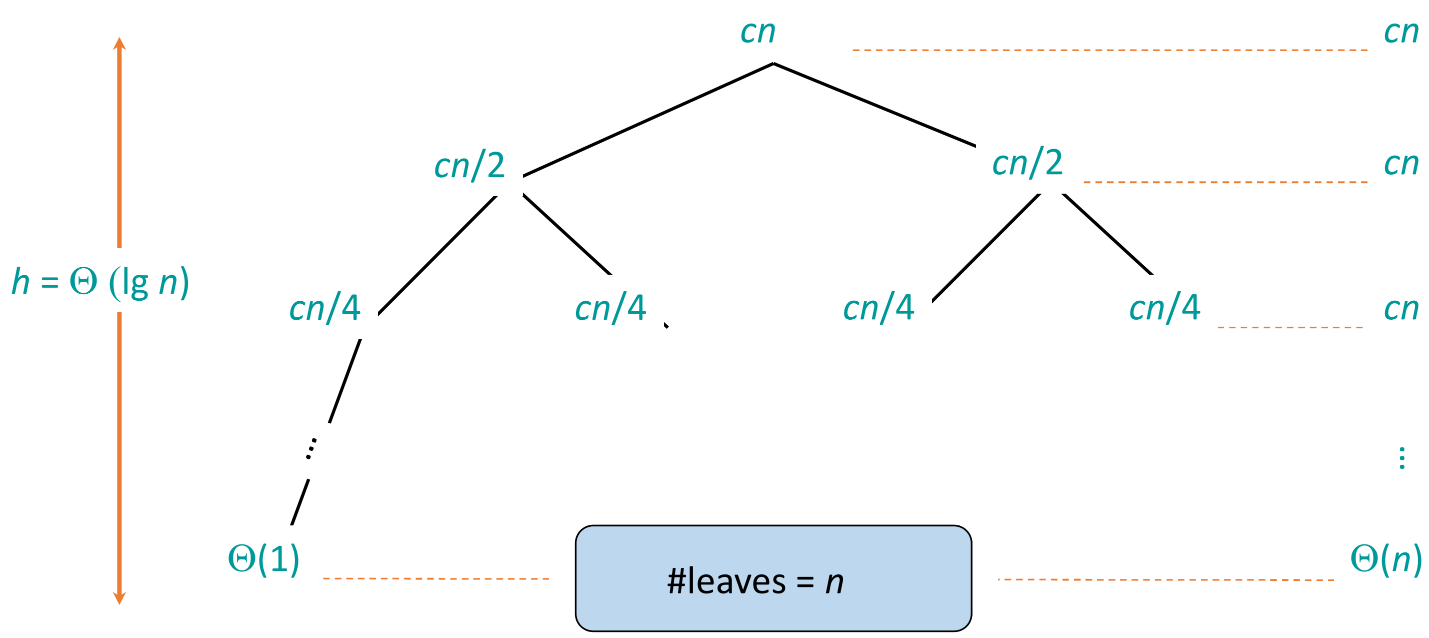
Recursion tree

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



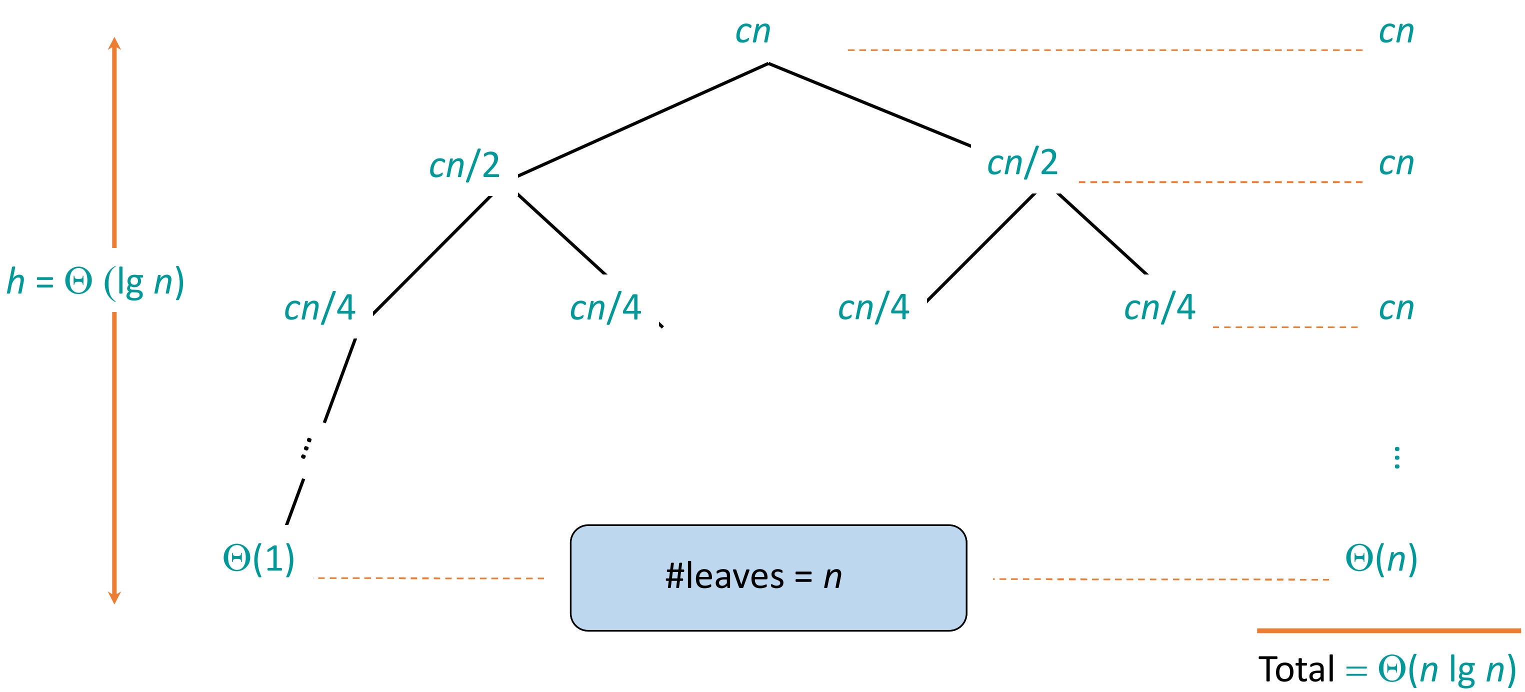
Recursion tree

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



Recursion tree

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



Question 2

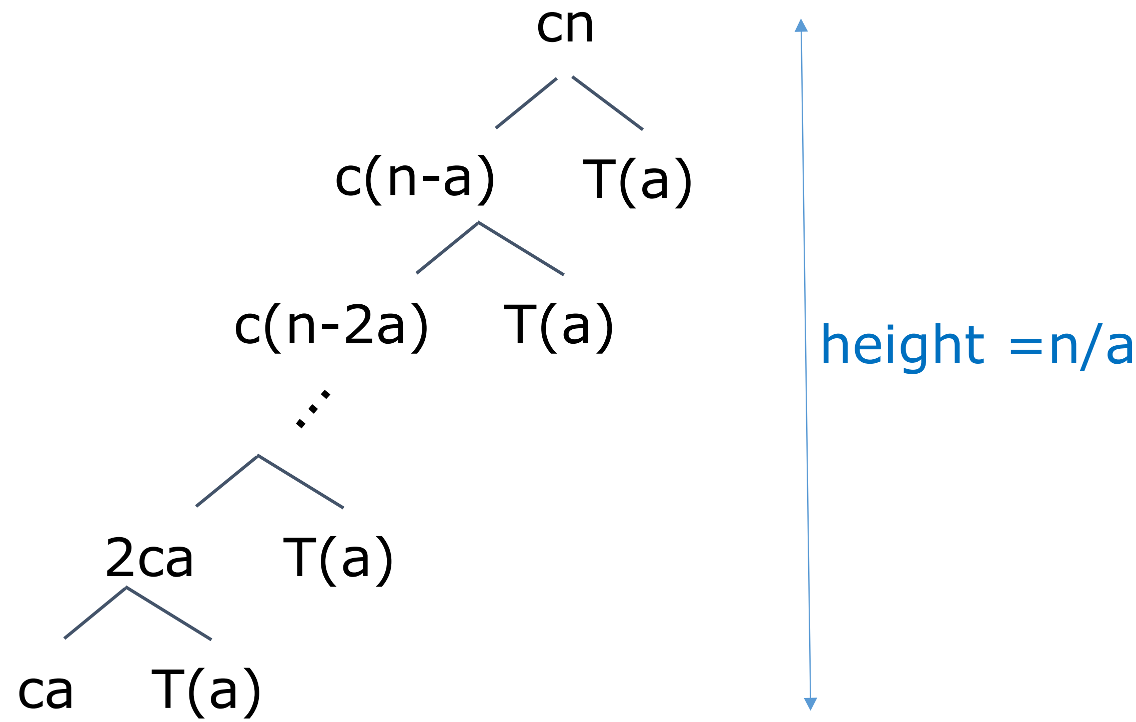
Use a recursion tree to give an asymptotically tight upper bound to the recurrence $T(n) = T(n-a) + T(a) + cn$ where $a \geq 1$ and $c > 0$, and $T(a)$ is a constant.

- $T(n) \leq C\frac{n}{a}\log n$ for some constant C
- $T(n) \leq Cn/a$ for some constant C
- $T(n) \leq Cn^2/a$ for some constant C
- $T(n) \leq Ca\log n$ for some constant C

Answer of Question 2

$T(n) = T(n\text{-}a) + T(a) + cn$

Answer: $T(n) \leq Cn^2/a$



$T(n) = T(n-a) + T(a) + cn$

Recursion tree height is n/a
At depth k, computation:
$T(a) + c(n-ka)$.
Summed over height, we get

Arithmetic Series

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n$$
$$= \frac{1}{2}n(n+1) = \Theta(n^2)$$

$$T(a)\frac{n}{a} + ac + 2ac + 3ac + \cdots + \frac{n}{a}ac$$
$$= T(a)\frac{n}{a} + \frac{ac}{2}\frac{n}{a}\left(\frac{n}{a} + 1\right) \leq C\left(\frac{n^2}{a}\right)$$

Master method

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.

Three common cases

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

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

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

Three common cases

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

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

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

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Three common cases (cont.)

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

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

   and $f(n)$ satisfies the ***regularity condition*** that $a f(n/b) \le c f(n)$ for some constant $c < 1$.

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

The regularity condition guarantees the sum of subproblems is smaller than f(n)

Summary: Master Theorem

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

Case 1:

$$f(n) = O(n^{\log_b a - \epsilon})$$

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

← If ε=0, it is case 2.

Case 2:

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

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

Case 3:

$$f(n) = \Omega(n^{\log_b a + \epsilon})$$

$$af(n/b) \leq cf(n), c < 1$$

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

Examples

Ex. $T(n) = 4T(n/2) + n$
$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n.$
Case 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.
$\therefore\ T(n) = \Theta(n^2).$

Ex. $T(n) = 4T(n/2) + n^2$
$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2.$
Case 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.
$\therefore\ T(n) = \Theta(n^2 \lg n).$

Question 3

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

1. $T(n) = \Theta(n^2)$.
2. $T(n) = \Theta(n^3)$.
3. $T(n) = \Theta(n \log n)$.
4. $T(n) = \Theta(n^2 \log n)$.

Answer of Question 3

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

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

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

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

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

Examples

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

$n^2/\lg n \notin O(n^{2-\varepsilon})$ → Not case 1
- *Reason:* for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\lg n)$.

$n^2/\lg n \notin \Theta\left(n^2 \log^k n\right)$ for any $k \geq 0$ → Not case 2
$n^2/\lg n \notin \Omega(n^{2+\varepsilon})$ → Not case 3
Master method does not apply.

Summary: Master Theorem

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

Case 1:

$$f(n) = O(n^{\log_b a - \epsilon})$$

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

← If ε=0, it is case 2.

Case 2:

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

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

Case 3:

$$f(n) = \Omega(n^{\log_b a + \epsilon})$$

$$af(n/b) \leq cf(n), c < 1$$

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

Substitution method

- The most general method:

1. Guess the form of the solution
2. Verify by induction

Example: Solve $T(n) = 4\,T(n/2) + n$

- [Assume $T(1)=q$ where q is a constant.]
- **Step 1:** Guess $T(n) = O(n^3)$.
  - I.e. there exists a constant c such that $T(n) \leq c \cdot n^3$, for $n \geq n_0$.
- **Step 2:** Verify by induction.
  - Set $c=\max\{2,q\}$ and $n_0=1$.
  - <u>Base case </u>($n=n_0=1$): $T(1) = q \leq c(1)^3$.
  - <u>Recursive case </u>($n>1$):
    - By strong induction, assume $T(k) \leq c \cdot k^3$ for $n > k \geq 1$.
    - $T(n) = 4\,T(n/2) + n \leq 4\,c\,(n/2)^3 + n = (c/2)\,n^3 + n \leq c\,n^3$.
  - Hence, $T(n) \leq c\,n^3$ for $n \geq 1$.
- Conclusion: $T(n) = O(n^3)$.

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

- Is $T(n) = O(n^3)$ a tight bound?

- **Answer:** No.
- The tight bound is $T(n) = O(n^2)$

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

- A possible solution to prove that $T(n) = O(n^2)$.
  - i.e. we show that $T(n) \le c\, n^2$ for $n \ge n_0$.

- Set $c=\max\{2,q\}$ and $n_0=1$.
- Base case ($n=1$): $T(1) = q \le c(1)^2$.
- Recursive case ($n>1$):
  - By strong induction, assume $T(k) \le c \cdot k^2$ for $n > k \ge 1$.
  - $T(n) = 4\,T(n/2) + n$
  - $\le 4\, c \cdot (n/2)^2 + n$
  - $= c\, n^2 + n$
  - $= O(n^2)$. ← This is not correct! You need to show $T(n) \le c\, n^2$!

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

- [Assume $T(1)=q$ where q is a constant.]
- Correct solution: Show that, for $n \geq n_0$, $T(n) \leq c_1\, n^2 - c_2\, n$.
- Set $c_1 = q+1$ and $c_2=1$ and $n_0=1$.
- <u>Base case</u> ($n=1$): $T(1) = q \leq (q+1)\,(1)^2 - (1)(1)$.
- <u>Recursive case</u> ($n>1$):
  - By strong induction, assume $T(k) \leq c_1 \cdot k^2 - c_2 \cdot k$ for $n > k \geq 1$.
  - $T(n) = 4\,T(n/2) + n = 4\,(c_1\,(n/2)^2 - c_2\,(n/2)\,) + n = c_1\, n^2 - 2\,c_2\, n + n$
    $= c_1\, n^2 - c_2\, n + (1 - c_2)\, n$
  - Since $(1 - c_2) = 0$, $T(n) \leq c_1\, n^2 - c_2\, n$.

Summary for substitution method

- Guess the time complexity and verify that it is correct by induction.

- Sometimes, the verification is a bit tricky.

- Sometimes, guessing the correct expression is also difficult and need experience. So I will <u>suggest not to use this method</u> as a beginner (unless you feel comfortable).

Powering a number

Powering a number

- **Problem:** Compute $f(n,m) = a^n \pmod{m}$ for any integer n,m.

(Assume each of n,m fits into one/constantly many words)

- **Observation:** $f(x+y,m) = f(x,m)*f(y,m) \pmod{m}$.

- Naïve solution:

1. Divide: Trivial.
2. Conquer: Recursively compute $f(n-1,m)$ and $f(1,m)$.
3. Combine: $f(n-1,m)*f(1,m) \pmod{m}$.

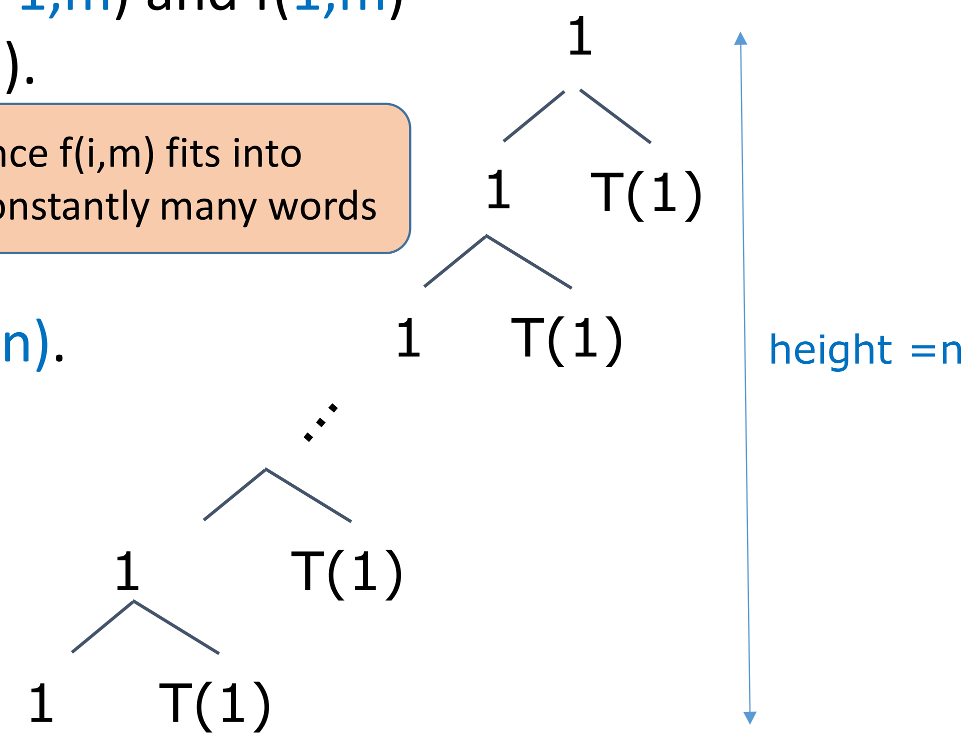
Running time of naïve solution

1. Divide: Trivial.
2. Conquer: Recursively compute f(n-1,m) and f(1,m)
3. Combine: f(n-1,m)*f(1,m) (mod m).

Not affected by m

Since f(i,m) fits into one/constantly many words

- $T(n) = T(n-1) + T(1) + \Theta(1)$
- By recursion tree, we have $T(n) = \Theta(n)$.



A better algorithm for powering a number

1. Divide: Trivial.
2. Conquer: Recursively compute $f(\lfloor n/2 \rfloor, m)$
3. Combine: $f(n, m) = f(\lfloor n/2 \rfloor, m)^2$ (mod m) if n is <u>even</u>;
   $f(n, m) = f(1, m) * f(\lfloor n/2 \rfloor, m)^2$ (mod m) if n is <u>odd</u>.

- $T(n) = T(n/2) + \Theta(1)$.
- By master theorem, we have $T(n) = \Theta(\log n)$.

… # Acknowledgement

- The slides are modified from
  - the slides from Prof. Erik D. Demaine and Prof. Charles E. Leiserson
  - the slides from Prof. Leong Hon Wai
  - the slides from Prof. Lee Wee Sun
  - the slides from Prof. Ken Sung
  - the slides from Prof. Diptarka Chakraborty
  - the slides from Prof. Arnab Bhattacharyya