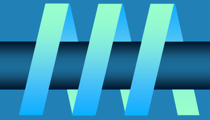
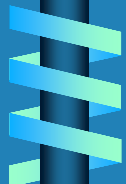
Lecture 3

The Analysis of Recursive Algorithm Efficiency

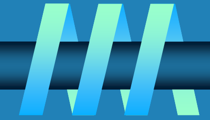
What we have learned



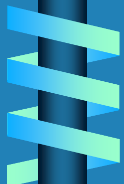
- What is algorithm
- Why study algorithm
- The time and space efficiency of algorithm
- The analysis framework of time efficiency
- Asymptotic orders of growth: O, Θ and Ω
- Analyze the time complexity of non-recurrent relation



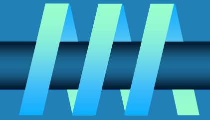
Outline



- Some examples on the time complexity of non-recurrence relation
- How to analyze the time complexity of recurrent algorithms
  - Substitution
  - Recursion Tree
  - Master Theorem

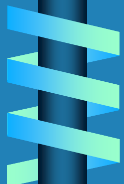


Time Complexity of Non-recursive algorithms



General Plan for Analysis

- **Decide on parameter *n* indicating *input size***
- **Identify algorithm's *basic operation***
- **Determine *worst*, *average*, and *best* cases for input of size *n***
- **Set up a sum for the number of times the basic operation is executed**
- **Simplify the sum using standard formulas and rules (see Appendix A)**



Example 1: Insertion Sort

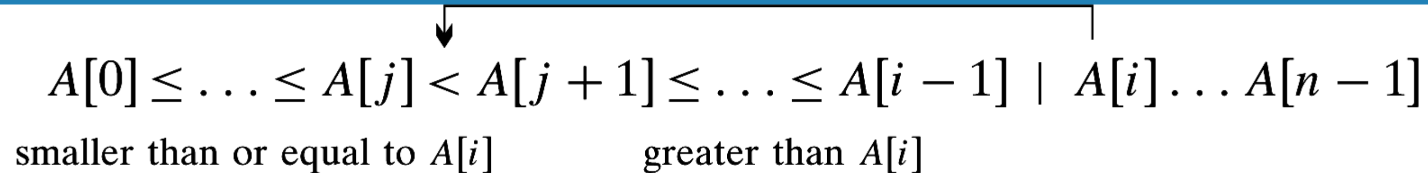


FIGURE 5.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

89 | **45** 68 90 29 34 17

45 89 | **68** 90 29 34 17

45 68 89 | **90** 29 34 17

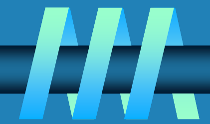
45 68 89 90 | **29** 34 17

29 45 68 89 90 | **34** 17

29 34 45 68 89 90 | **17**

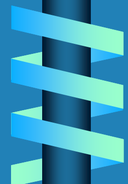
17 29 34 45 68 89 90

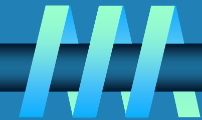
FIGURE 5.4 Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.



ALGORITHM *InsertionSort*$(A[0..n-1])$

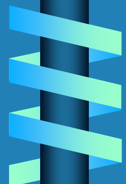
//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$



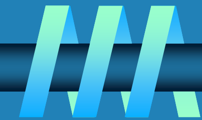


For some algorithms efficiency depends on form of input:

- **Worst case:** $C_{worst}(n)$ – **maximum over inputs of size** n
- **Best case:** $C_{best}(n)$ – **minimum over inputs of size** n
- **Average case:** $C_{avg}(n)$ – **"average" over inputs of size** n



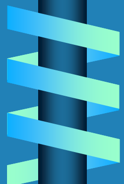
Example 2: Bubble Sort

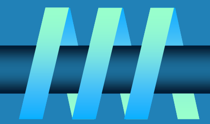


89	$\overset{?}{\leftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\leftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\leftrightarrow}$	90	$\overset{?}{\leftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\leftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\leftrightarrow}$	17
45		68		89		29		34		17	|	90
45	$\overset{?}{\leftrightarrow}$	68	$\overset{?}{\leftrightarrow}$	89	$\overset{?}{\leftrightarrow}$	29		34		17	|	90
45		68		29		89	$\overset{?}{\leftrightarrow}$	34		17	|	90
45		68		29		34		89	$\overset{?}{\leftrightarrow}$	17	|	90
45		68		29		34		17	|	89		90

etc.

FIGURE 3.2 First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.





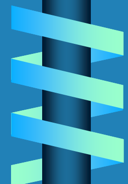
ALGORITHM *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

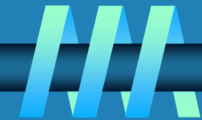
Worst case

Best case

Average case



Example: Sequential search

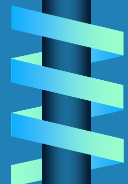


ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//           or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

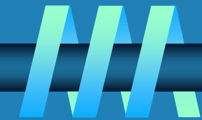
- **Worst case**

- **Best case**



- **Average case**

Useful summation formulas and rules



$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le u} 1 = n - 1 + 1 = n \in \Theta(n)$

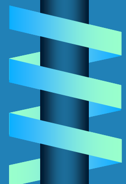
$\Sigma_{1 \le i \le n} i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

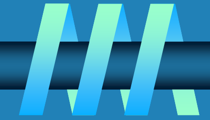
$\Sigma_{0 \le i \le n} a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \ne 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

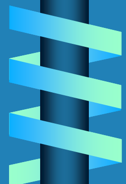
$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$ $\quad \Sigma c a_i = c \Sigma a_i$ $\quad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$



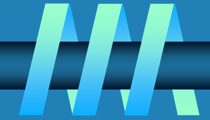
Plan for Analysis of Recursive Algorithms



- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.



Example 1: Recursive evaluation of n!



Definition: $n\,! = 1 * 2 * \ldots *(n\text{-}1) * n$ **for** $n \geq 1$ **and** $0! = 1$

Recursive definition of n**!:** $F(n) = F(n\text{-}1) * n$ **for** $n \geq 1$ **and**

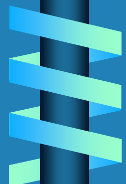
$$F(0) = 1$$

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

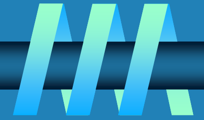
Size:

Basic operation:



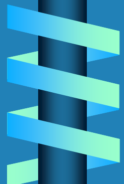
Recurrence relation:

Solving the recurrence for M(n)

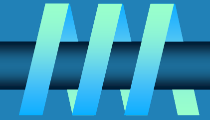


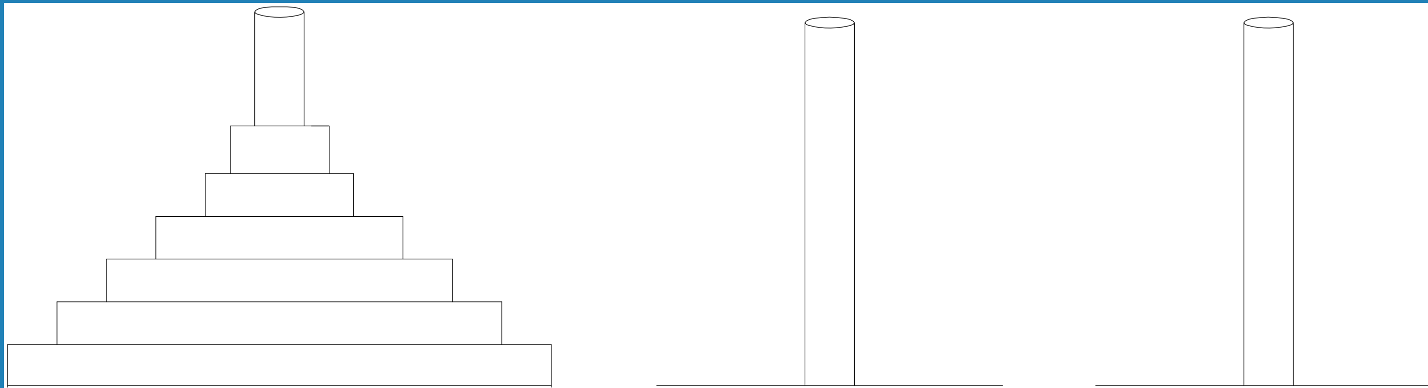
$T(n) = T(n-1) + 1$, $T(0) = 0$ (M is the number of execution of multiplication)

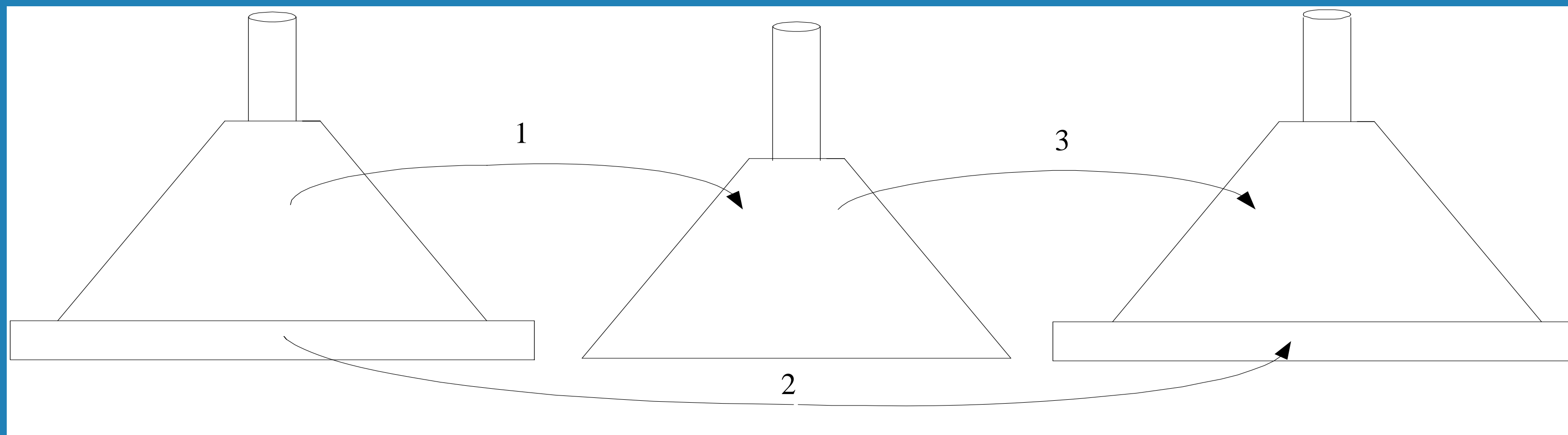
T(n) = ?



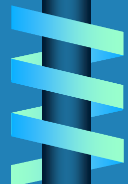
Example 2: The Tower of Hanoi Puzzle



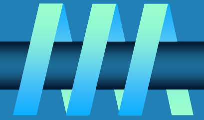




Recurrence for number of moves:

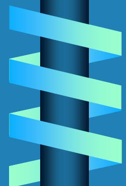


Solving recurrence for number of moves

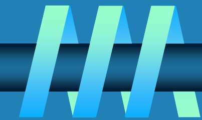


$T(n) = 2T(n\text{-}1) + 1, \; T(1) = 1$

$T(n) =?$



Substitution method

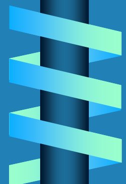


The most general method:

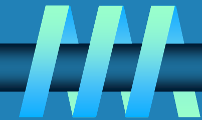
1. *Guess* the form of the solution.
2. *Verify* by induction.
3. *Solve* for constants.

Example: $T(n) = 4T(n/2) + 100n$

- [Assume that $T(1) = \Theta(1)$.]
- Guess $O(n^3)$. (Prove O and Ω separately.)
- Assume that $T(k) \leq ck^3$ for $k < n$.
- Prove $T(n) \leq cn^3$ by induction.

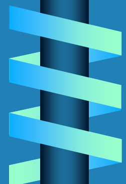


Example of substitution

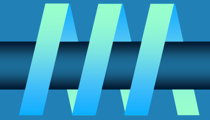


$$
\begin{aligned}
T(n) &= 4T(n/2) + 100n \\
&\le 4c(n/2)^3 + 100n \\
&= (c/2)n^3 + 100n \\
&= cn^3 - ((c/2)n^3 - 100n) \quad \leftarrow \textit{desired} - \textit{residual} \\
&\le cn^3 \quad \leftarrow \textit{desired}
\end{aligned}
$$

whenever $(c/2)n^3 - 100n \ge 0$, for example, if $c \ge 200$ and $n \ge 1$.

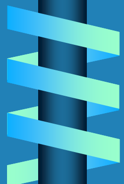


Example (continued)

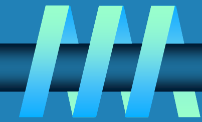


- We must also handle the initial conditions, that is, ground the induction with base cases.
- ***Base:*** $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant.
- For $1 \le n < n_0$, we have "$\Theta(1)$" $\le cn^3$, if we pick c big enough.

This bound is not tight!



A tighter upper bound?

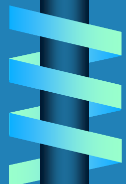


We shall prove that $T(n) = O(n^2)$.

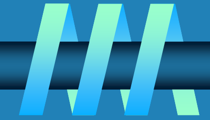
Assume that $T(k) \le ck^2$ for $k < n$:

$$T(n) = 4T(n/2) + 100n$$
$$\le cn^2 + 100n$$
$$\le cn^2$$

for ***no*** choice of $c > 0$. Lose!



A tighter upper bound!



IDEA: Strengthen the inductive hypothesis.

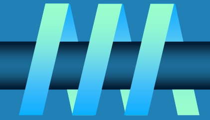
- ***Subtract*** a low-order term.

Inductive hypothesis: $T(k) \le c_1 k^2 - c_2 k$ for $k < n$.

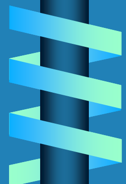
$$
\begin{aligned}
T(n) &= 4T(n/2) + 100n \\
&\le 4(c_1(n/2)^2 - c_2(n/2)) + 100n \\
&= c_1 n^2 - 2c_2 n + 100n \\
&= c_1 n^2 - c_2 n - (c_2 n - 100n) \\
&\le c_1 n^2 - c_2 n \quad \text{if } c_2 > 100.
\end{aligned}
$$

Pick c_1 big enough to handle the initial conditions.

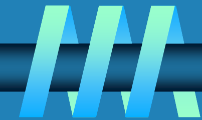
Recursion-tree method



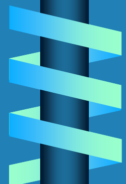
- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion tree method is good for generating guesses for the substitution method.
- The recursion-tree method promotes intuition, however.



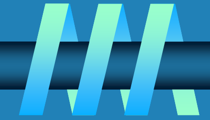
Example of recursion tree



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

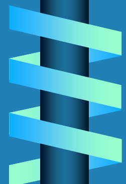


Example of recursion tree

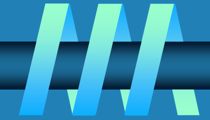


Solve $T(n) = T(n/4) + T(n/2) + n^2$:

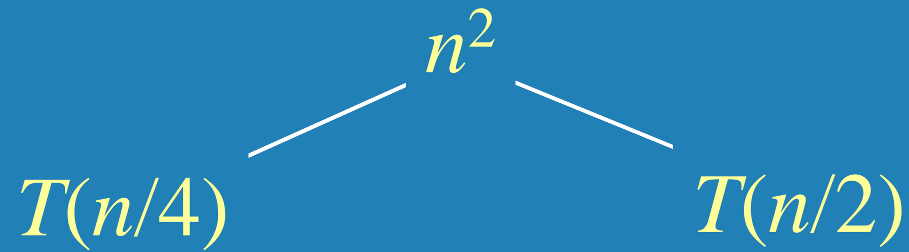
$$T(n)$$

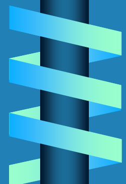


Example of recursion tree

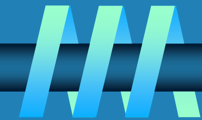


Solve $T(n) = T(n/4) + T(n/2) + n^2$:

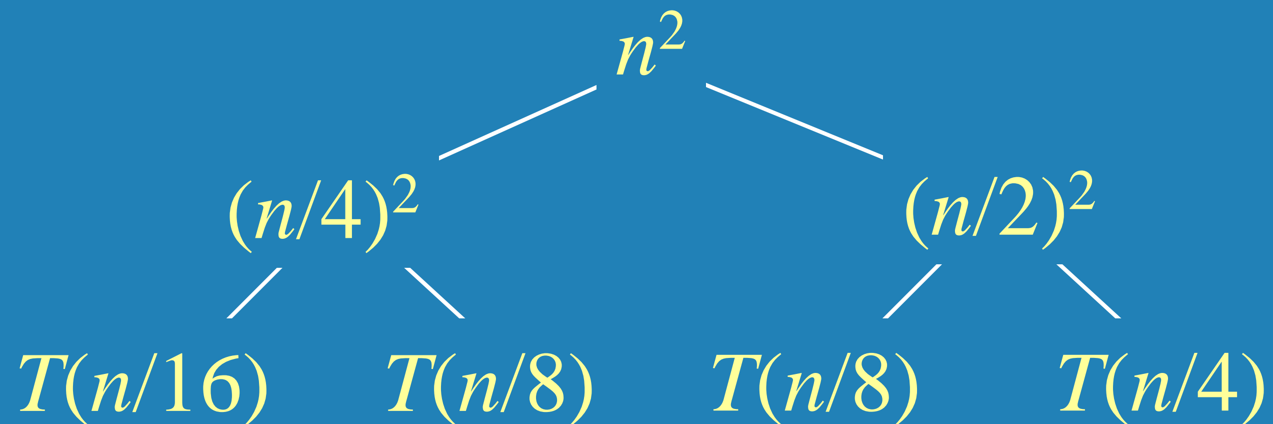


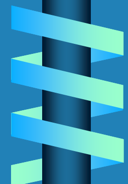


Example of recursion tree

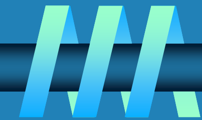


Solve $T(n) = T(n/4) + T(n/2) + n^2$:

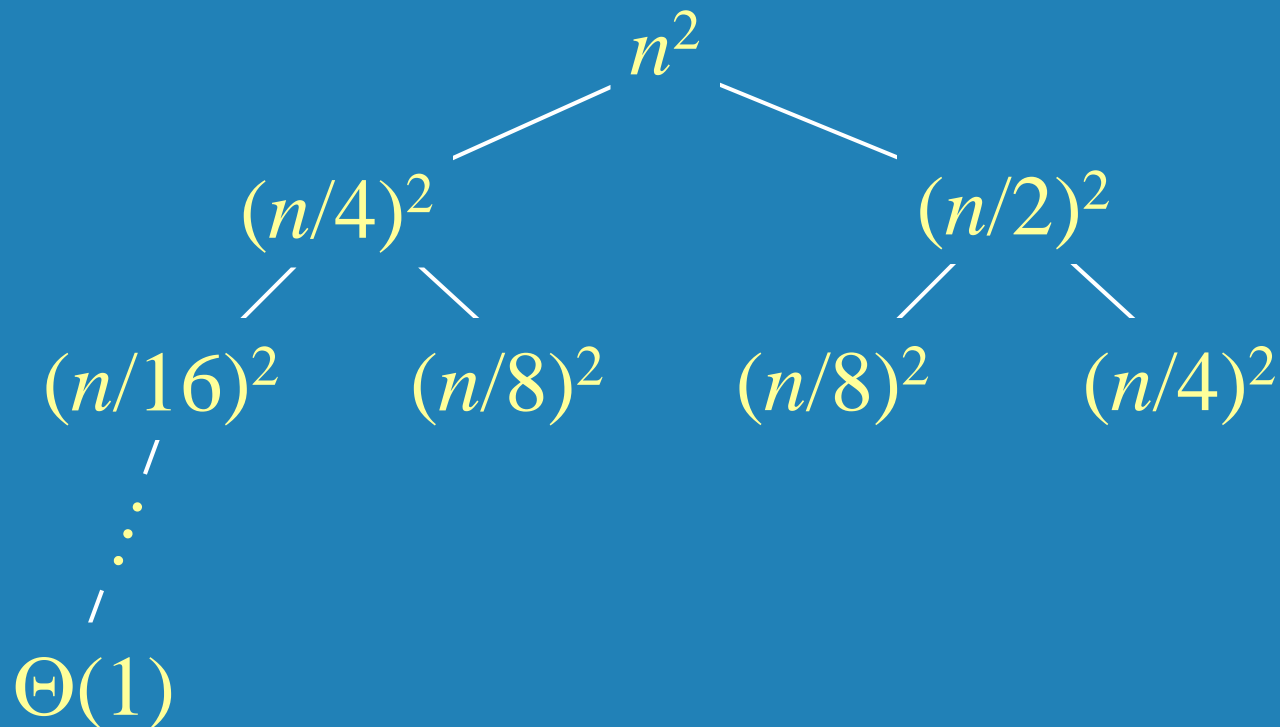


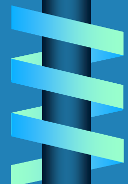


Example of recursion tree

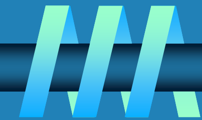


Solve $T(n) = T(n/4) + T(n/2) + n^2$:

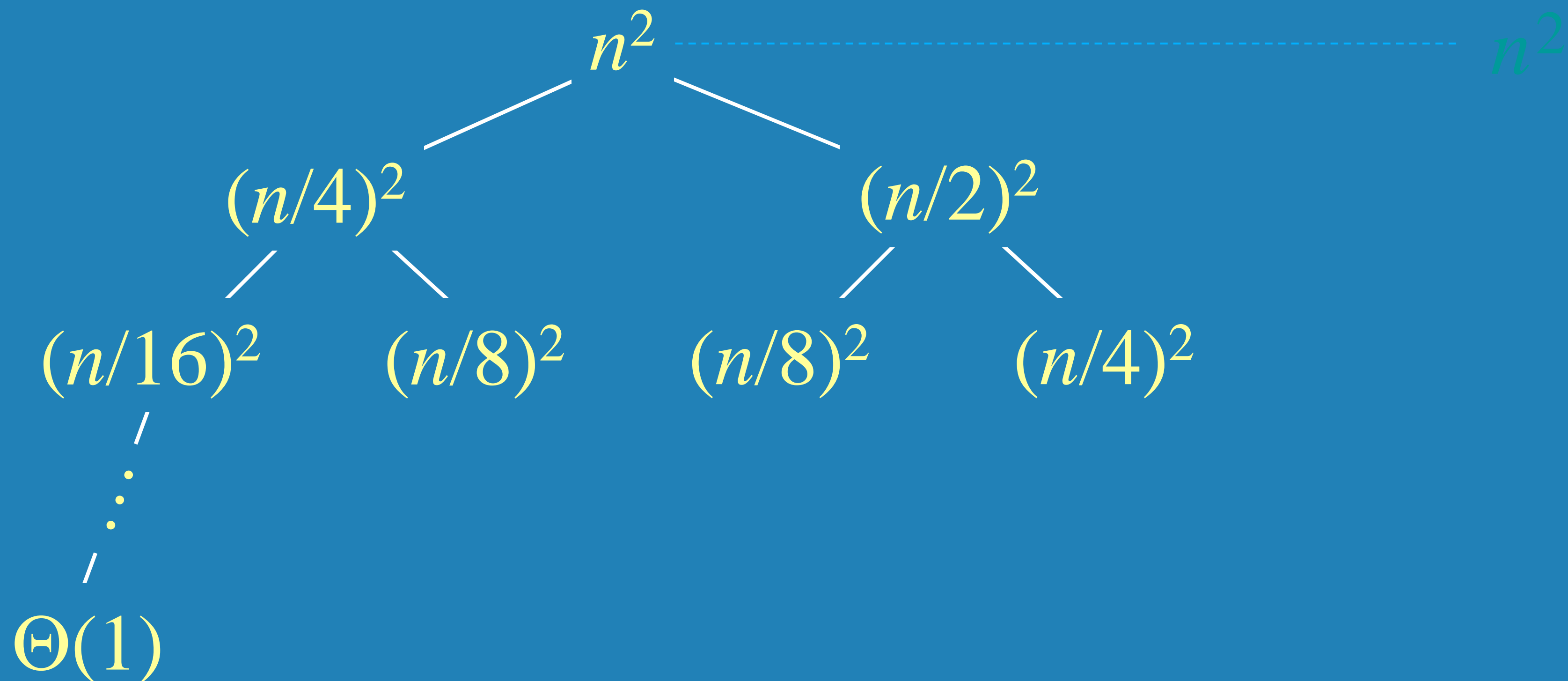


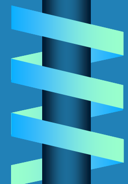


Example of recursion tree

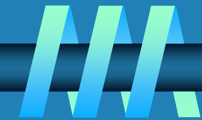


Solve $T(n) = T(n/4) + T(n/2) + n^2$:

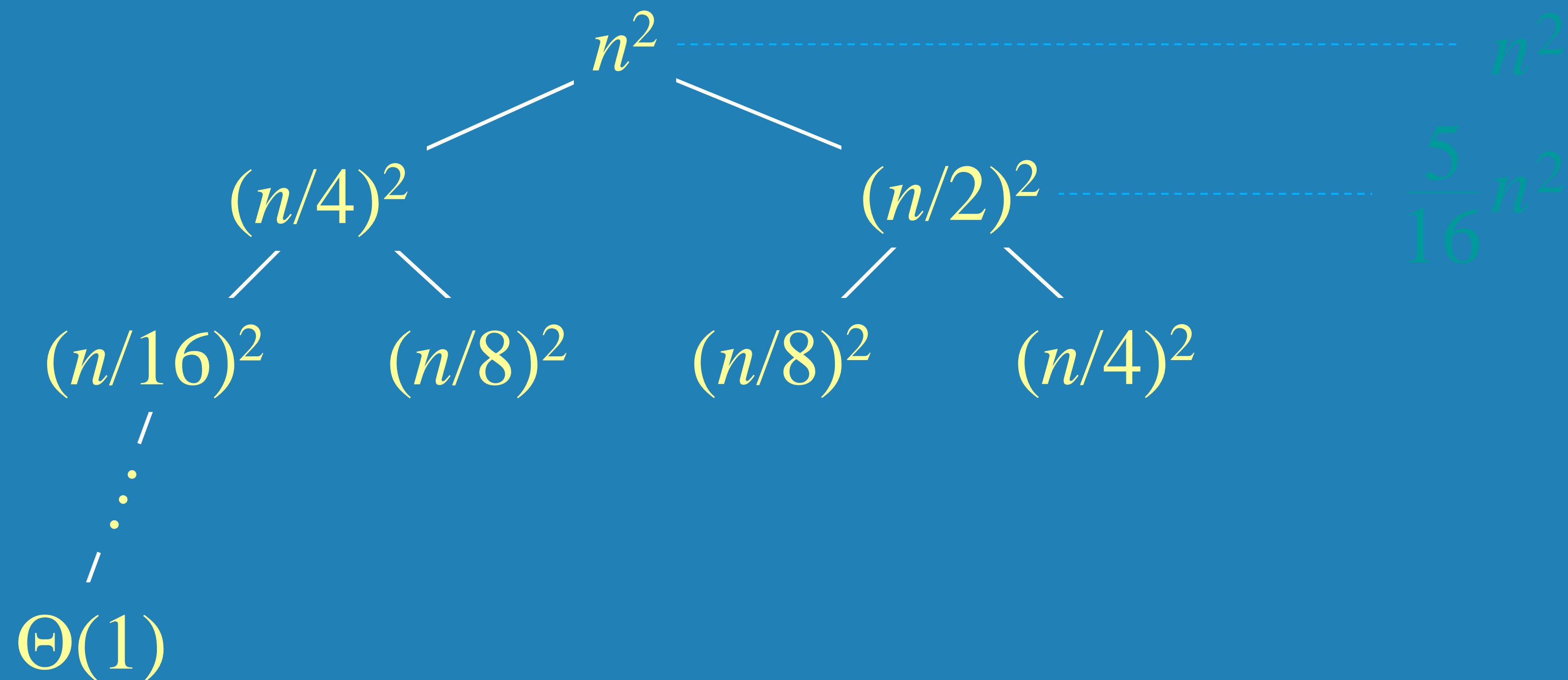


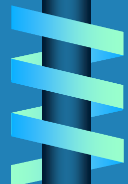


Example of recursion tree



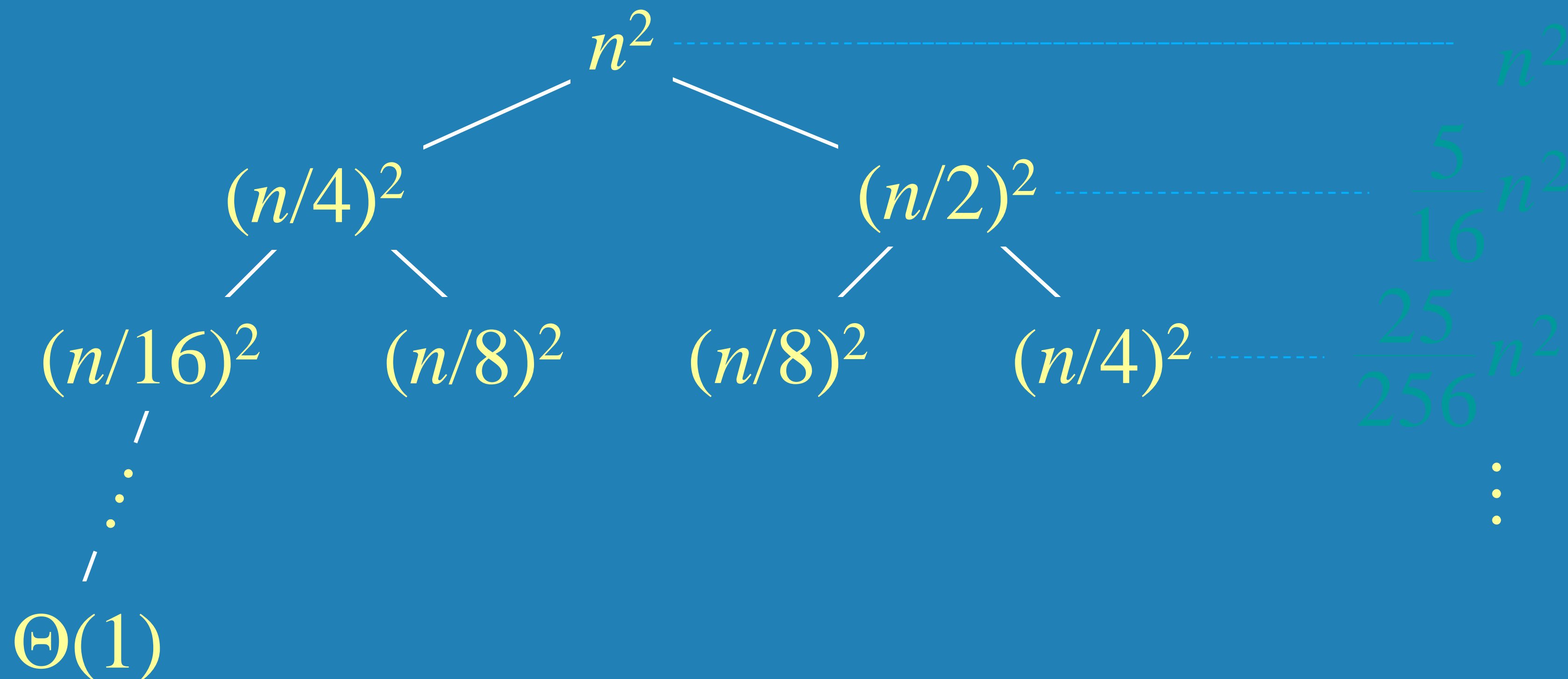
Solve $T(n) = T(n/4) + T(n/2) + n^2$:

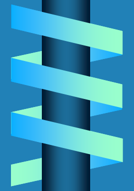




Example of recursion tree

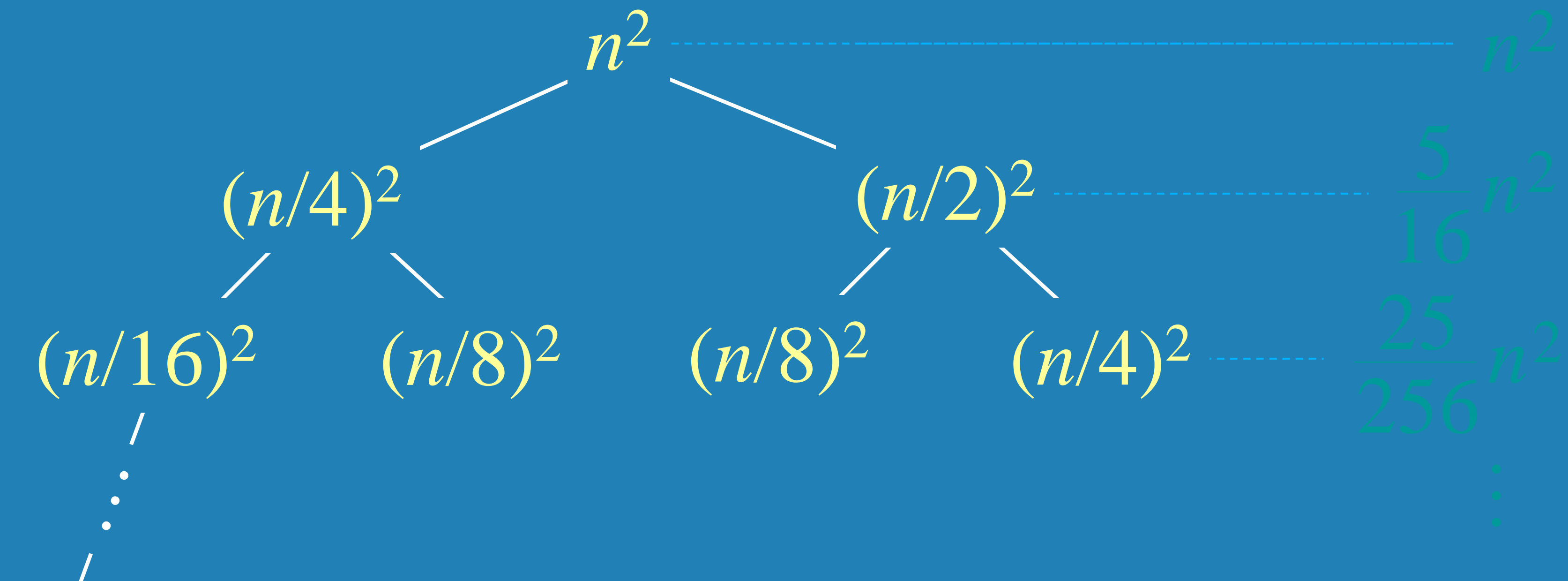
Solve $T(n) = T(n/4) + T(n/2) + n^2$:





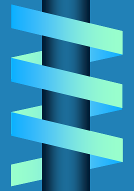
Example of recursion tree

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

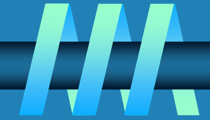


Total $= n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$

$= \Theta(n^2)$ *geometric series*



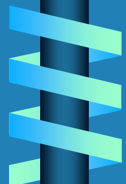
The master method



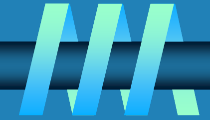
The master method applies to recurrences of the form

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

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



Idea of master theorem



Recursion tree:

$h = \log_b n$

$f(n)$ .... $f(n)$

a

$f(n/b)$ $f(n/b)$ $\cdots$ $f(n/b)$ .... $af(n/b)$

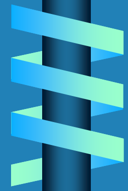
a

$f(n/b^2)$ $f(n/b^2)$ $\cdots$ $f(n/b^2)$ .... $a^2 f(n/b^2)$

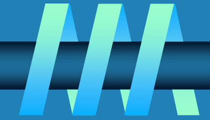
$\vdots$

$T(1)$ $\qquad$ $n^{\log_b a} T(1)$

$$\#\text{leaves} = a^h = a^{\log_b n} = n^{\log_b a}$$



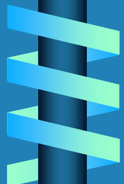
Three common cases



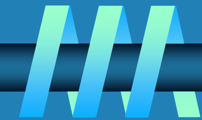
Compare $f(n)$ with $n^{\log_b a}$:

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



Idea of master theorem



Recursion tree:

$$f(n) \quad \cdots\cdots \quad f(n)$$

a

$$f(n/b) \quad f(n/b) \quad \cdots \quad f(n/b) \quad \cdots\cdots \quad af(n/b)$$

a

$$f(n/b^2) \quad f(n/b^2) \quad \cdots \quad f(n/b^2) \quad \cdots\cdots \quad a^2 f(n/b^2)$$

$h = \log_b n$

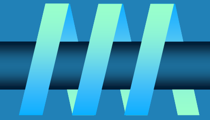
$T(1)$

$$n^{\log_b a} T(1)$$

CASE 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

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

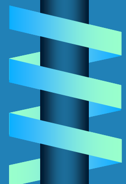
Three common cases



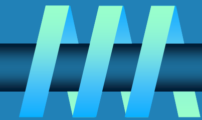
Compare $f(n)$ with $n^{\log_b a}$:

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



Idea of master theorem



Recursion tree:

$$f(n) \quad \cdots\cdots \quad f(n)$$

$$f(n/b) \quad f(n/b) \quad \cdots \quad f(n/b) \quad \cdots\cdots \quad a f(n/b)$$

$$f(n/b^2) \quad f(n/b^2) \quad \cdots \quad f(n/b^2) \quad \cdots\cdots \quad a^2 f(n/b^2)$$

$$\vdots$$

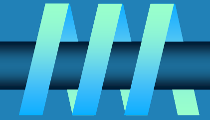
$$T(1) \qquad n^{\log_b a} T(1)$$

$h = \log_b n$

CASE 2: ($k = 0$) The weight is approximately the same on each of the $\log_b n$ levels.

$$\Theta(n^{\log_b a} \lg n)$$

Three common cases (cont.)

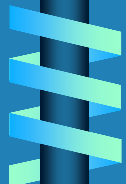


Compare $f(n)$ with $n^{\log_b a}$:

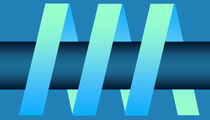
3. $f(n) - \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

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

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



Idea of master theorem



Recursion tree:

$$f(n)$$

a

$$f(n/b) \quad f(n/b) \quad \cdots \quad f(n/b)$$

a

$h = \log_b n$

$$f(n/b^2) \quad f(n/b^2) \quad \cdots \quad f(n/b^2)$$

$$T(1)$$

Level sums:

$$f(n)$$

$$a f(n/b)$$

$$a^2 f(n/b^2)$$

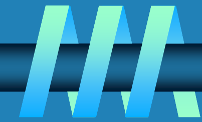
$$\vdots$$

$$n^{\log_b a} T(1)$$

$$\Theta(f(n))$$

CASE 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

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 $\varepsilon = 1$.

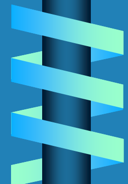
$\therefore\ T(n) = \Theta(n^2).$

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

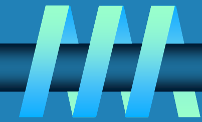
$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2.$

CASE 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.

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



Examples



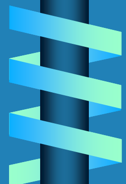
Ex. $T(n) = 4T(n/2) + n^3$

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

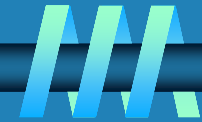
CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

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

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

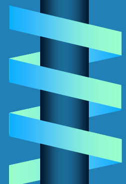


Summarization

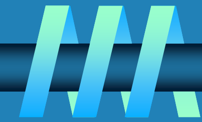


How to analyze the time complexity of recurrent algorithms

1. Substitution
2. Recursion Tree
3. Master Theorem



Exercises



Exercises 2.3 3, 5, 11

Exercises 2.4 5, 8, 9, 11