

Divide and Conquer

A thick, horizontal yellow brushstroke underline that spans most of the width of the slide, positioned directly beneath the title.

Divide and Conquer Paradigm



- An important general technique for designing algorithms:
 - divide problem into subproblems
 - recursively solve subproblems
 - combine solutions to subproblems to get solution to original problem
- Use recurrences to analyze the running time of such algorithms

Mergesort



Example: Mergesort



- DIVIDE the input sequence in half
- RECURSIVELY sort the two halves
 - basis of the recursion is sequence with 1 key
- COMBINE the two sorted subsequences by merging them



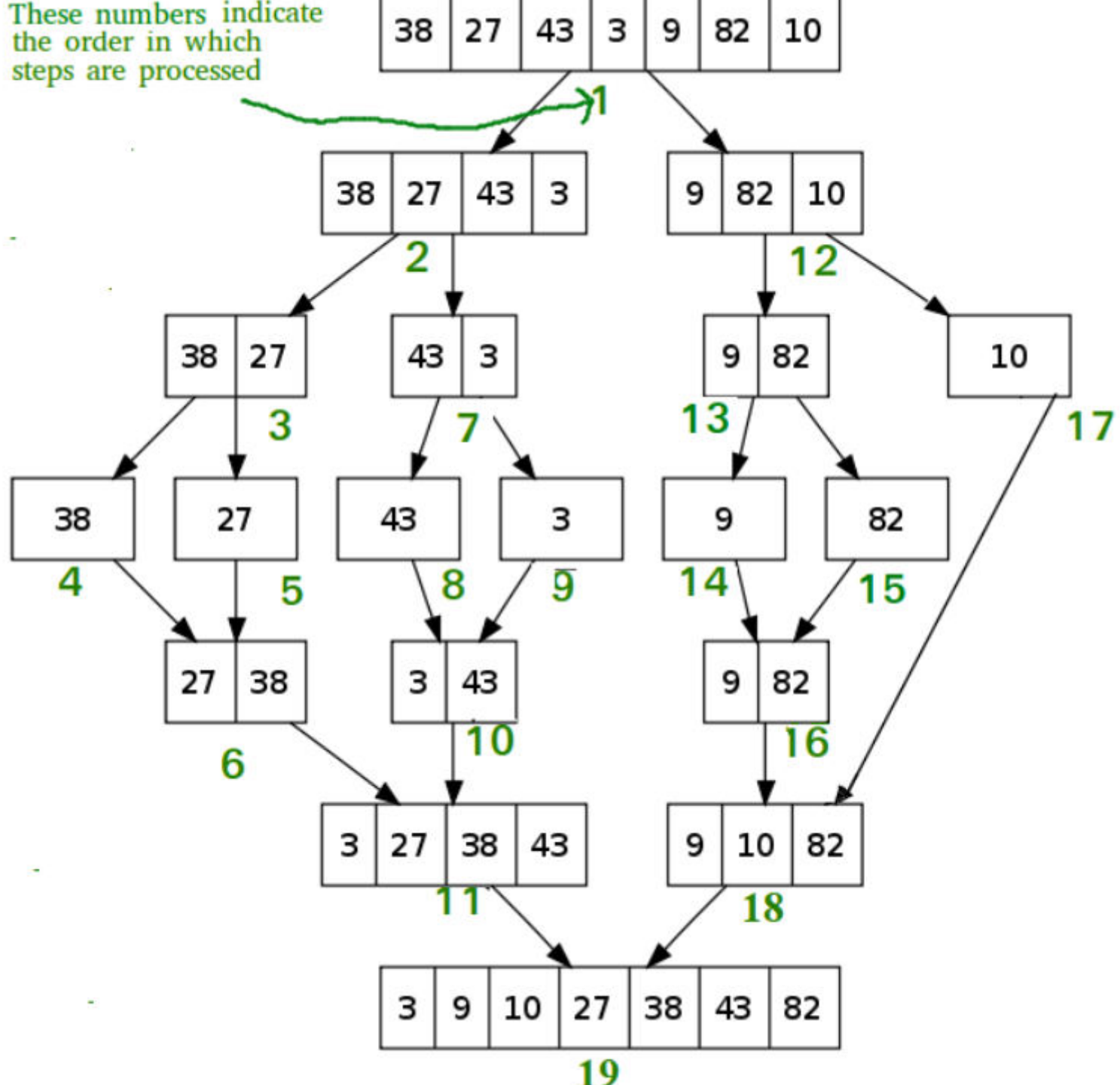
Merge sort

MERGE-SORT $A[1 \dots n]$

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

Key subroutine: **MERGE**

These numbers indicate the order in which steps are processed



Algorithm:

step 1: start

step 2: declare array and left, right, mid
variable

step 3: perform merge function.

if left > right

return

mid = (left + right) / 2

mergesort(array, left, mid)

mergesort(array, mid + 1, right)

merge(array, left, mid, right)

step 4: Stop

Analyzing merge sort

$T(n)$

$\Theta(1)$

$2T(n/2)$

$\Theta(n)$

use

MERGE-SORT $A[1 \dots n]$

1. If $n = 1$, done.
2. Recursively sort $A[1 \dots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \dots n]$.
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.

Recurrence for merge sort

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

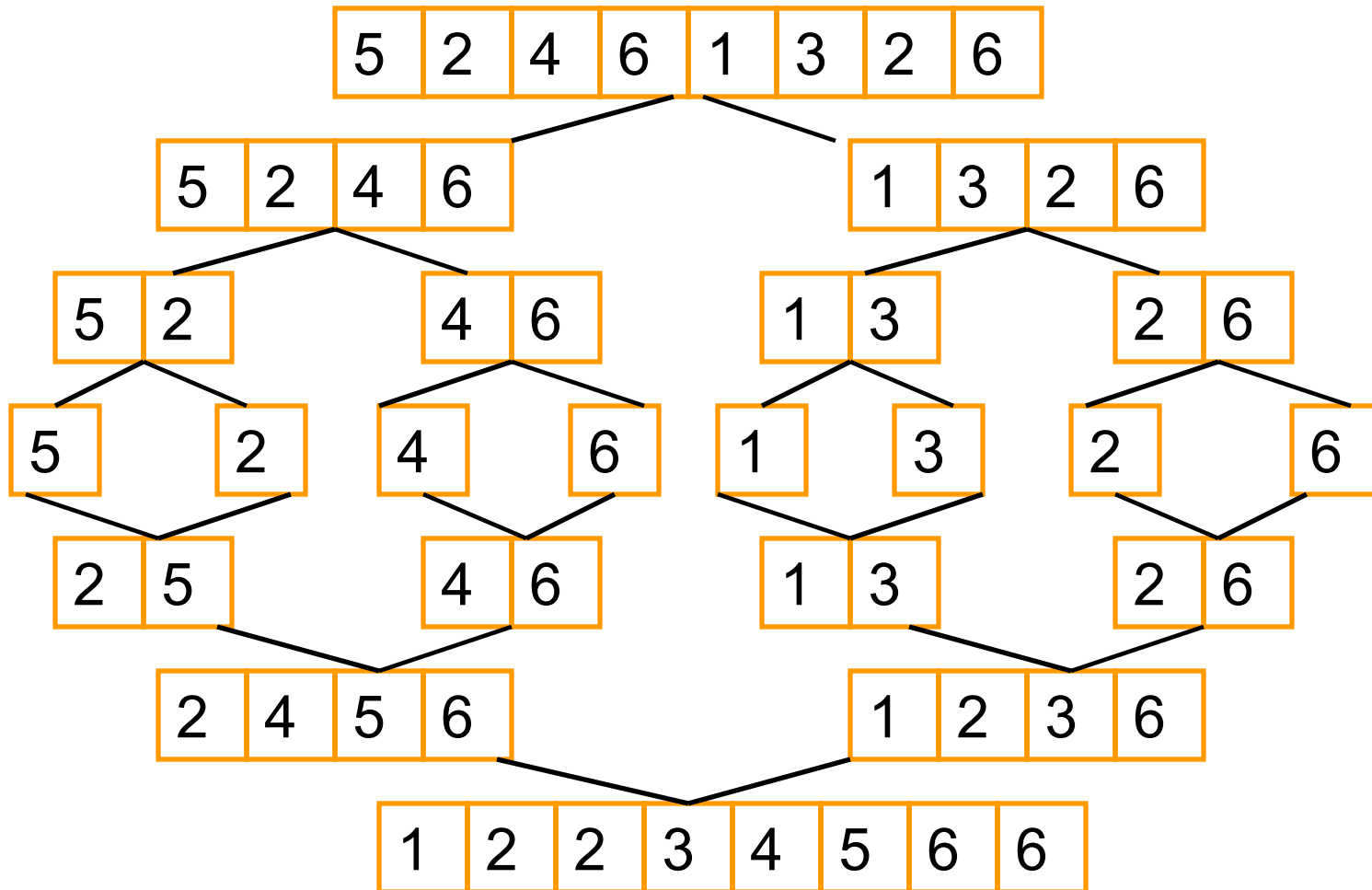
- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n , but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS and Lecture 2 provide several ways to find a good upper bound on $T(n)$.

Recurrence Relation for Mergesort



- Let $T(n)$ be worst case time on a sequence of n keys
- If $n = 1$, then $T(n) = \Theta(1)$ (constant)
- If $n > 1$, then $T(n) = 2 T(n/2) + \Theta(n)$
 - two subproblems of size $n/2$ each that are solved recursively
 - $\Theta(n)$ time to do the merge

Mergesort Example



Time Complexity: $O(N \log(N))$, Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.


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

Quick Sort

1. Correct position in final, sorted array
2. Items to the left are smaller
3. Items to the right are larger

2	6	5	3	8	7	1	0
---	---	---	---	---	---	---	---

2	6	5	3	8	7	1	0
---	---	---	---	---	---	---	---



1. **itemFromLeft** that is larger than pivot

2	6	5	0	8	7	1	3
---	---	---	---	---	---	---	---

1. **itemFromLeft** that is larger than pivot
2. **itemFromRight** that is smaller than pivot

2	6	5	0	8	7	1	3
---	---	---	---	---	---	---	---

↑
itemFromLeft

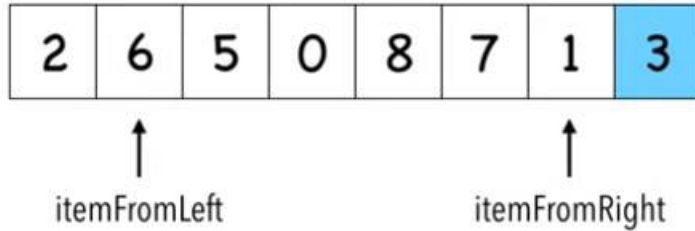
1. **itemFromLeft** that is larger than pivot
2. **itemFromRight** that is smaller than pivot

2	6	5	0	8	7	1	3
---	---	---	---	---	---	---	---

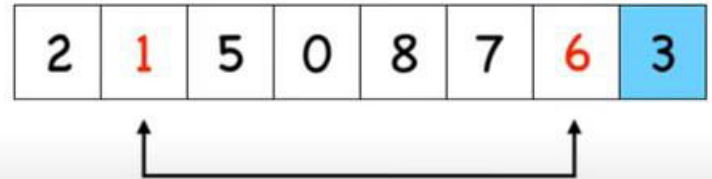
↑
itemFromLeft

↑
itemFromRight

1. **itemFromLeft** that is larger than pivot
2. **itemFromRight** that is smaller than pivot

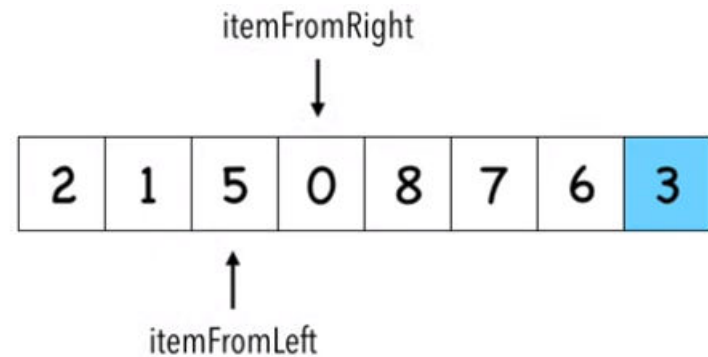
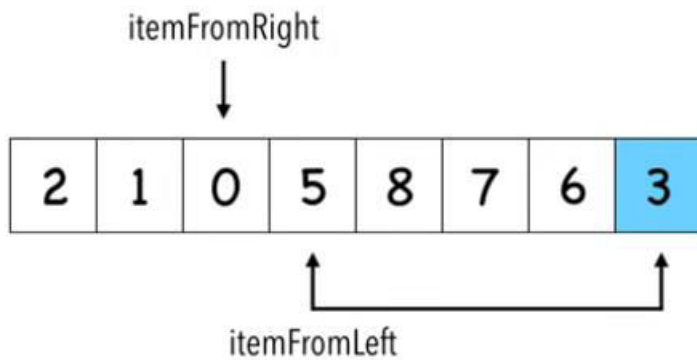


1. **itemFromLeft** that is larger than pivot
2. **itemFromRight** that is smaller than pivot

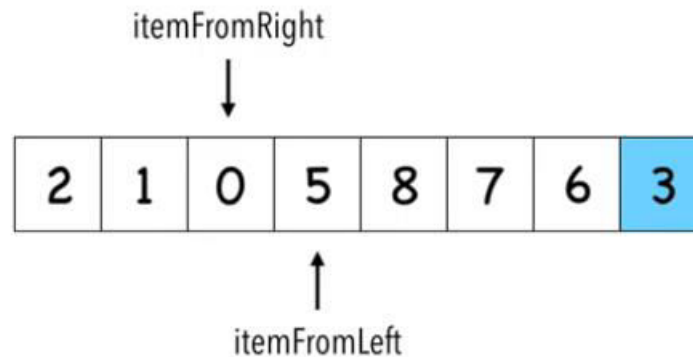
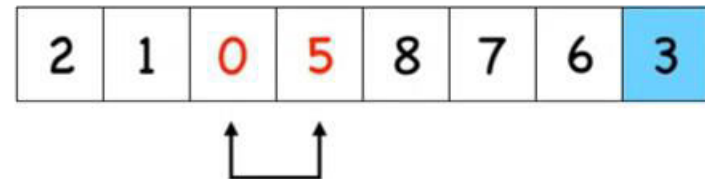


1. **itemFromLeft** that is larger than pivot
2. **itemFromRight** that is smaller than pivot





Swap **itemFromLeft** and **pivot**



Stop when index of **itemFromLeft** > index of **itemFromRight**

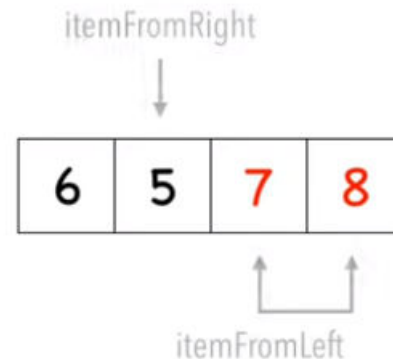
1. Correct position in final, sorted array
2. Items to the left are smaller
3. Items to the right are larger

2	1	0	3	8	7	6	5
---	---	---	---	---	---	---	---

8	7	6	5
---	---	---	---

1. **itemFromLeft** that is larger than pivot
2. **itemFromRight** that is smaller than pivot

8	5	6	7
---	---	---	---



How to choose pivot:
Median position element:

Worst Case Complexity: $O(n^2)$
Average Case Complexity: $\Theta(n \log n)$

Recurrence Relations

A thick, horizontal yellow brushstroke underline that spans most of the width of the slide, positioned directly beneath the title.

How To Solve Recurrences



- Ad hoc method:
 - expand several times
 - guess the pattern
 - can verify with proof by induction
- Master theorem
 - general formula that works if recurrence has the form $T(n) = aT(n/b) + f(n)$
 - a is number of subproblems
 - n/b is size of each subproblem
 - $f(n)$ is cost of non-recursive part

Master Theorem

Consider a recurrence of the form

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

with $a \geq 1$, $b > 1$, and $f(n)$ eventually positive.

a) If $f(n) = O(n^{\log_b(a) - \epsilon})$, then $T(n) = \Theta(n^{\log_b(a)})$.

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

c) If $f(n) = \Omega(n^{\log_b(a) + \epsilon})$ and $f(n)$ is regular, then
 $T(n) = \Theta(f(n))$

[$f(n)$ regular iff eventually $af(n/b) \leq cf(n)$ for some constant $c < 1$]

Excuse me, what did it say???

Essentially, the Master theorem compares the function $f(n)$ with the function $g(n)=n^{\log_b(a)}$.

Roughly, the theorem says:

- a) If $f(n) \ll g(n)$ then $T(n)=\Theta(g(n))$.
- b) If $f(n) \approx g(n)$ then $T(n)=\Theta(g(n)\log(n))$.
- c) If $f(n) \gg g(n)$ then $T(n)=\Theta(f(n))$.

Now go back and memorize the theorem!

Master Theorem

Consider a recurrence of the form

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

with $a \geq 1$, $b > 1$, and $f(n)$ eventually positive.

a) If $f(n) = O(n^{\log_b(a) - \epsilon})$, then $T(n) = \Theta(n^{\log_b(a)})$.

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

c) If $f(n) = \Omega(n^{\log_b(a) + \epsilon})$ and $f(n)$ is regular, then
 $T(n) = \Theta(f(n))$

[$f(n)$ regular iff eventually $af(n/b) \leq cf(n)$ for some constant $c < 1$]

If the recurrence is of the form $T(n) = aT(n/b) + \Theta(n^k \log^p n)$ where $a \geq 1, b > 1, k \geq 0$ and p is a real number, then:

- 1) If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$
- 2) If $a = b^k$
 - a. If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
 - b. If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log \log n)$
 - c. If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$
- 3) If $a < b^k$
 - a. If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
 - b. If $p < 0$, then $T(n) = O(n^k)$

Masters Theorem

- $T(n) = aT(n/b) + f(n)$,
- where, n = size of input
- a = number of subproblems in the recursion
- n/b = size of each subproblem. All subproblems are assumed to have the same size.
- $f(n)$ = cost of the work done outside the recursive call, which includes the cost of dividing the problem and the cost of merging the solutions.
- Here, $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is an asymptotically positive function. An asymptotically positive function means that for a sufficiently large value of n , we have $f(n) > 0$.

Masters Theorem also represented as

Consider a recurrence of the form

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

with $a \geq 1$, $b > 1$, and $f(n)$ eventually positive.

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

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

b) If $f(n) = \Theta(n^{\log_b(a)})$, then

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

c) If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ and $f(n)$ is regular, then $T(n) = \Theta(f(n))$

Master's method is a quite useful method for solving recurrence equations because it directly gives us the cost of an algorithm with the help of the type of a recurrence equation and it is applied when the recurrence equation is in the form of:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a \geq 1$, $b > 1$ and $f(n) > 0$.

For example,

$$c + T\left(\frac{n}{2}\right) \rightarrow a = 1, b = 2 \text{ and } f(n) = c,$$

$$n + 2T\left(\frac{n}{2}\right) \rightarrow a = 2, b = 2 \text{ and } f(n) = n, \text{ etc.}$$

$T(n) = aT(n/b) + f(n)$, where, n = size of input a = number of subproblems in the recursion n/b = size of each subproblem. All subproblems are assumed to have the same size.

Recurrence relation helps in finding the subsequent term (next term) dependent upon the preceding term (previous term).

Master's Theorem

Taking an equation of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a \geq 1$, $b > 1$ and $f(n) > 0$

The Master's Theorem states:

- **CASE 1** - if $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
- **CASE 2** - if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
- **CASE 3** - if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$.

By the use of these three cases, we can easily get the solution of a recurrence equation of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

Problems

Problem-1 $T(n) = 3T(n/2) + n^2$

Solution: $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.a)

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

Solution: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 \log n)$ (Master Theorem Case 2.a)

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

Solution: $T(n) = T(n/2) + n^2 \Rightarrow \Theta(n^2)$ (Master Theorem Case 3.a)

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

Solution: $T(n) = 2^n T(n/2) + n^n \Rightarrow$ Does not apply (a is not constant)

Problem-5 $T(n) = 16T(n/4) + n$

Solution: $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-6 $T(n) = 2T(n/2) + n \log n$

Solution: $T(n) = 2T(n/2) + n \log n \Rightarrow T(n) = \Theta(n \log^2 n)$ (Master Theorem Case 2.a)

Nothing is perfect...



The Master theorem does not cover all possible cases. For example, if

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

then we lie between cases 2) and 3), but the theorem does not apply.

There exist better versions of the Master theorem that cover more cases, but these are even harder to memorize.