COMP2013
Data Structures and Algorithms

Lecture 2
Analysis of Algorithms

Dr. Jieming Shi
COMP2013

Recap

- Problems, Problem Instances, and Algorithms
- Insertion Sort Algorithm, Pseudocode, Complexity
- Methods to Design Algorithms
  - Divide and Conquer
  - Merge sort

Methods to Design Algorithms

- Incremental approach
  - Build a solution into a larger solution
    - E.g., insertion sort: we have a sorted subarray *A[1:i–1]*, then append an item to obtain a sorted subarray *A[1:i]*
- Divide and conquer approach
  - Reduce the problem into smaller subproblems to solve it
  - E.g., merge sort
- (More methods in subsequent lectures)

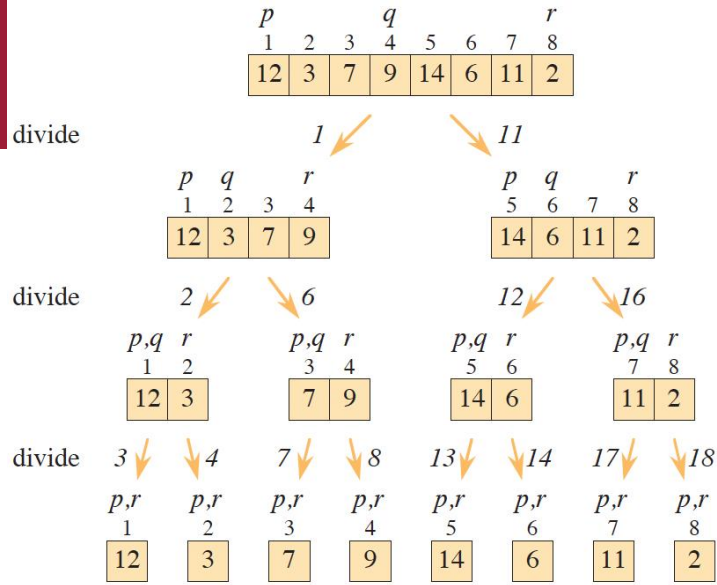
Divide and Conquer

- Divide
  - Break a problem into 1 or more subproblems (smaller instances of the same problem)
- Conquer
  - Solve the subproblems recursively
    - May recurse several levels
- Combine
  - Combine subproblem solutions to form a solution to the original problem

- Many algorithms are recursive in structure

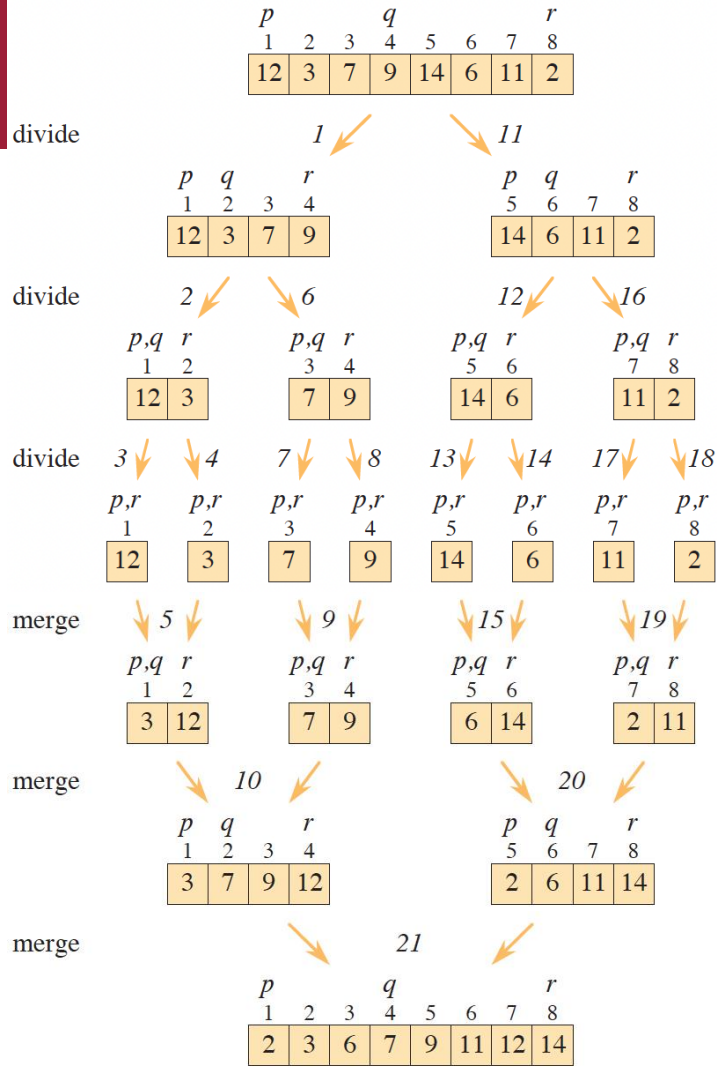
Merge Sort

- Divide
  - (Sub)array $A[p:r]$ to be sorted into two adjacent subarrays, each of half the size, $A[p:q]$ and $A[q:r]$, where q is the midpoint of $A[p:r]$
- Conquer
  - Sort subarrays $A[p:q]$ and $A[q+1:r]$ recursively via merge sort
- Combine
  - The sorted subarrays $A'[p:q]$ and $A'[q+1:r]$ back to get $A'[p:r]$



Merge Sort

- **Base case of merge sort**: a subarray with only 1 element
  - It is always sorted
  - Base case: a simple case that does not recursively call the function itself
- Which operations perform the sorting?
  - The key operation of merge sort occurs in the *combine* step
  - Different algorithms vary

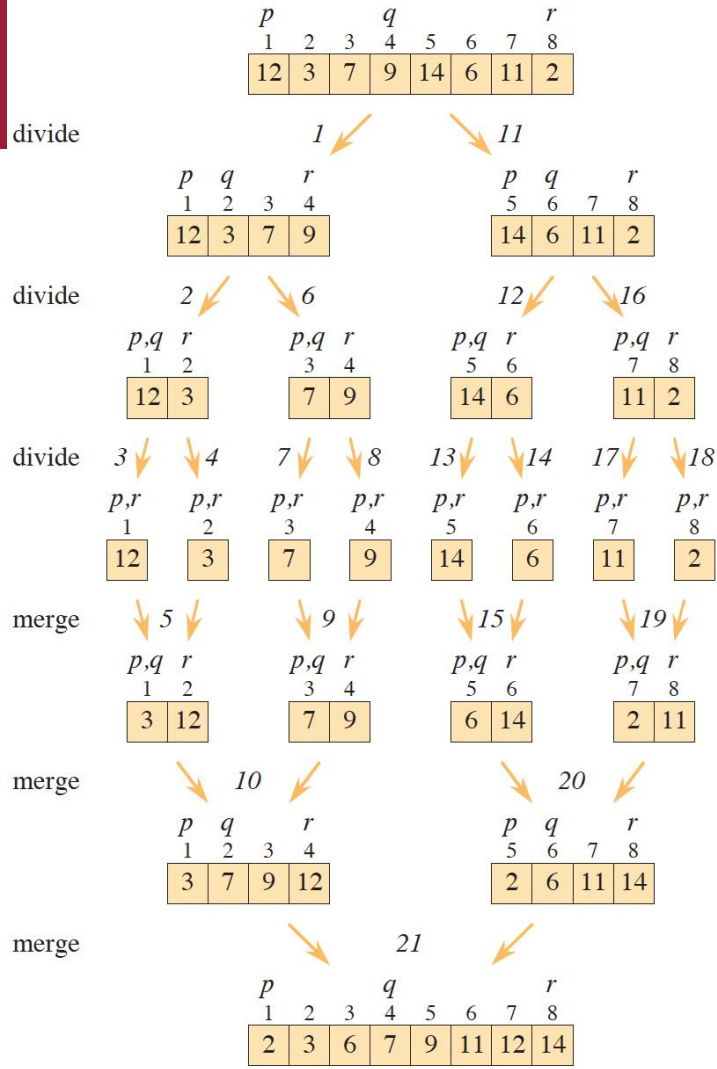


Merge Sort Pseudocode

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

A[p:q] and A[q+1:r] are sorted right before Line 7

Merge function combines two sorted arrays to be 1 sorted array

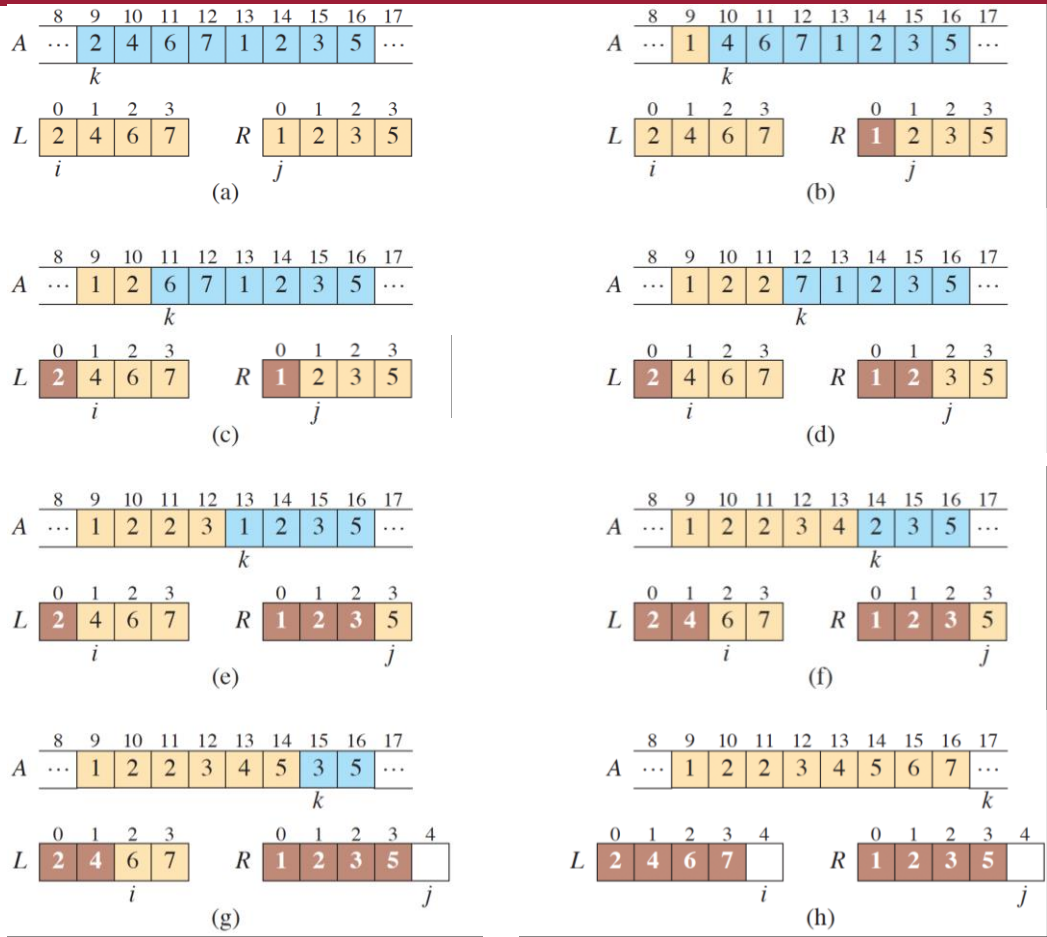


Italic numbers indicate the order where Merge-Sort and Merge are called (pay attention to the order)

Merge two sorted arrays

- *A[p:q]* and *A[q:r]* are sorted
  - Use *L* and *R* as temporary storage
- Merge them to be sorted *A[p,r]*
- Scan *L* and *R* from small to large values to merge

- *A[p:r]* has *n* elements, what is the time complexity of this Merge function?
  - Make the informal right guess
  - Linear to *n*



Merge two sorted arrays

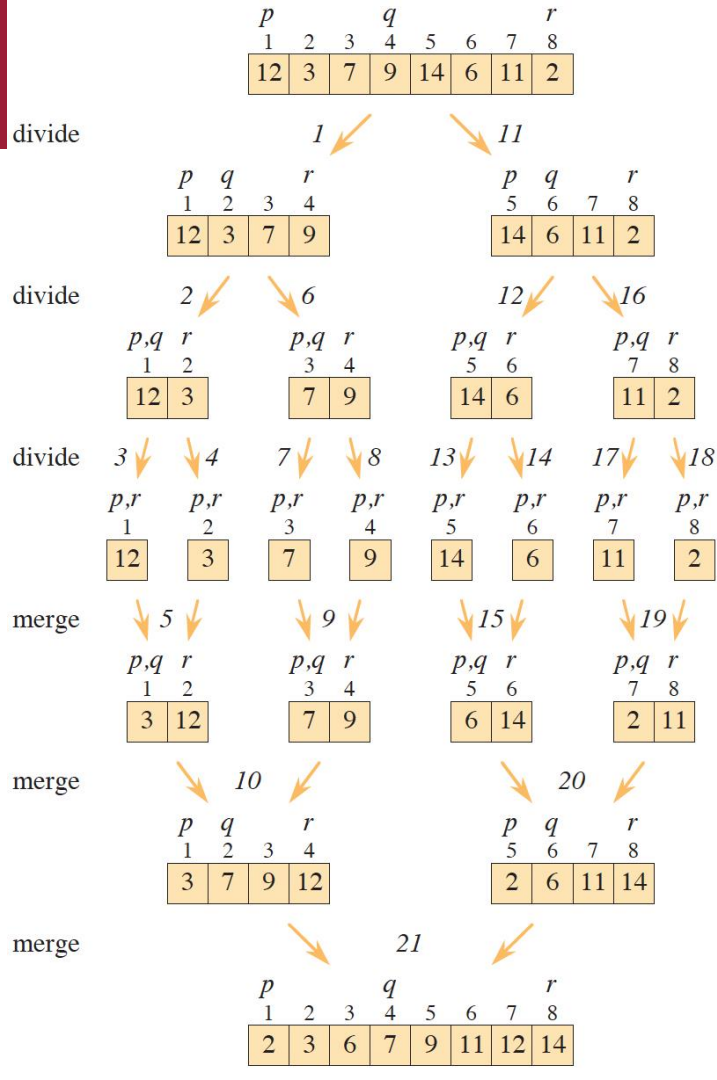
```
MERGE(A, p, q, r)
 1  n_L = q − p + 1          // length of A[p : q]
 2  n_R = r − q              // length of A[q + 1 : r]
 3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
 4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                    // i indexes the smallest remaining element in L
 9  j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //      copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_R
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //      remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Function Calls

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

Stack
(First in last out)

5	Merge(*A, 1, 1, 2*)	Called at Line 7 of Merge-Sort(*A, 1, 2*)
4	Merge-Sort(*A, 2, 2*)	Called at Line 5 of Merge-Sort(*A, 1, 2*)
3	Merge-Sort(*A, 1, 1*)	Called at Line 4 of Merge-Sort(*A, 1, 2*)
2	Merge-Sort(*A, 1, 2*)	Called at Line 4 of Merge-Sort(*A, 1, 4*)
1	Merge-Sort(*A, 1, 4*)	Called at Line 4 of Merge-Sort(*A, 1, 8*)
0	Merge-Sort(*A, 1, 8*)	

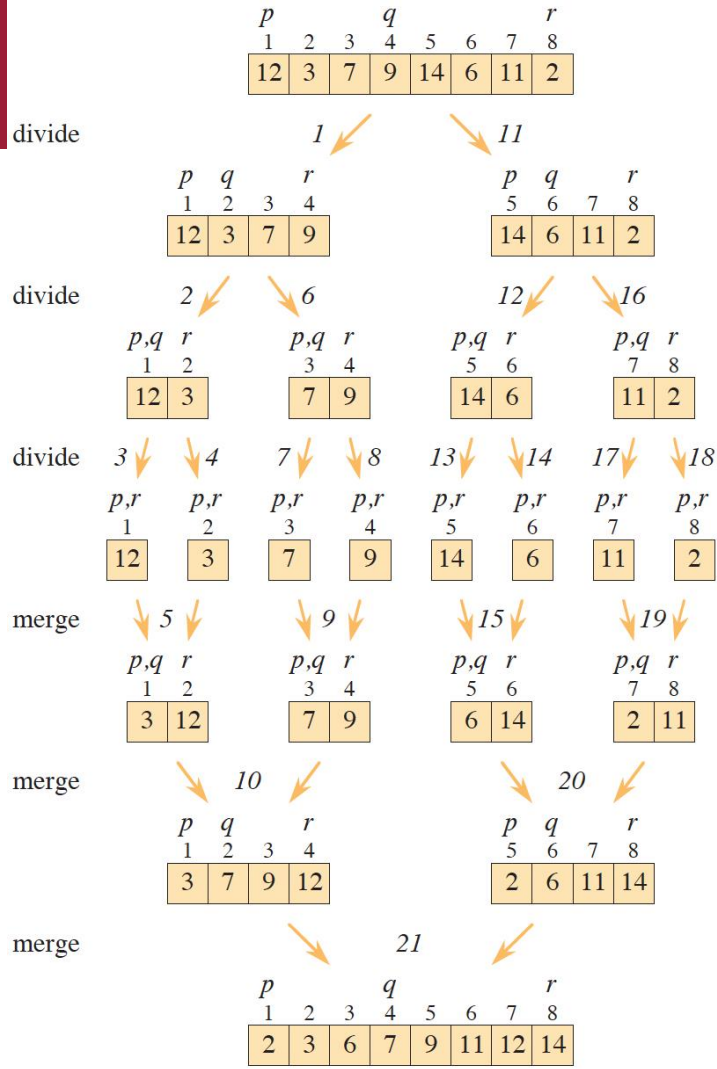


Function Calls

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

Stack
(First in last out)

10	Merge(*A, 1, 2, 4*)	Called at Line 7 of Merge-Sort(*A, 1, 4*)
9	Merge(*A, 3, 3, 4*)	Called at Line 7 of Merge-Sort(*A, 3, 4*)
8	Merge-Sort(*A, 4, 4*)	Called at Line 5 of Merge-Sort(*A, 3, 4*)
7	Merge-Sort(*A, 3, 3*)	Called at Line 4 of Merge-Sort(*A, 3, 4*)
6	Merge-Sort(*A, 3, 4*)	Called at Line 5 of Merge-Sort(*A, 1, 4*)
1	Merge-Sort(*A, 1, 4*)	Called at Line 4 of Merge-Sort(*A, 1, 8*)
0	Merge-Sort(*A, 1, 8*)	



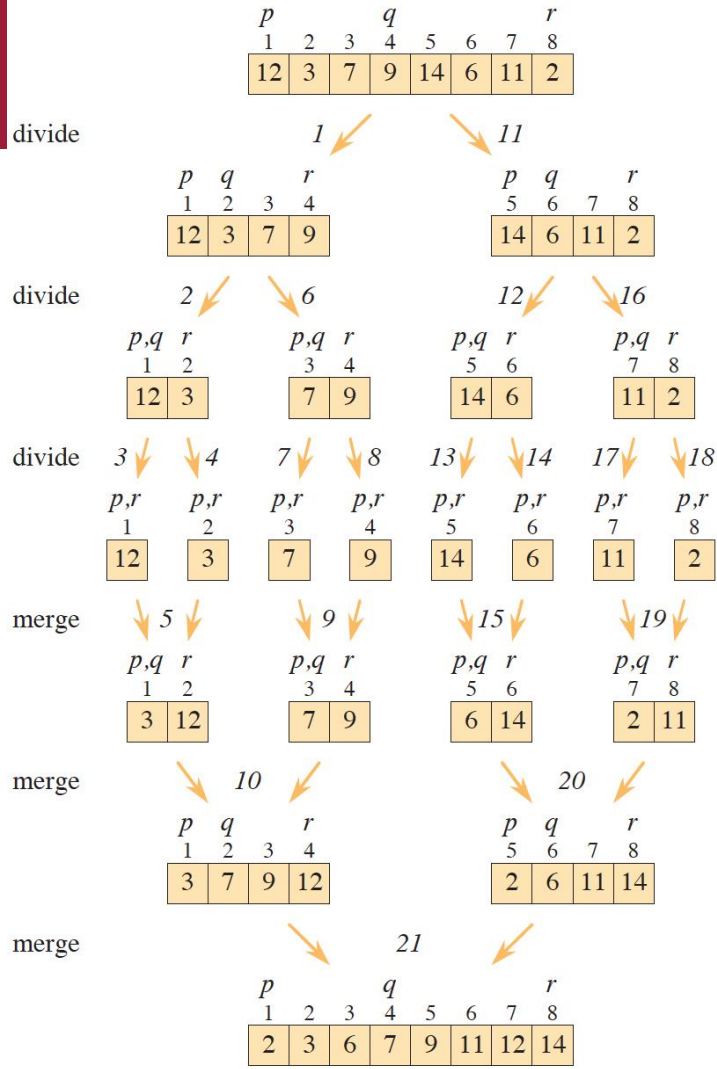
Function Calls

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

Stack
(First in last out)

.
.
.

11	Merge-Sort(*A, 5, 8*)	Called at Line 5 of Merge-Sort(*A, 1, 8*)
0	Merge-Sort(*A, 1, 8*)	



Overview

- Divide and conquer
  - Merge Sort
- **Recursive algorithm complexity**
  - **Recursion tree**
- Introduction on Asymptotic Bounds O, Ω, Θ
- Definitions on Asymptotic Bounds O, Ω, Θ
  - Bounds o, ω
- Properties of Asymptotic Bounds
- Some Notations and Common Functions
-

Complexity of divide and conquer (recursive) algorithms

- Recurrence equation
  - Running time of a problem of size *n*, in terms of the running time of the same algorithm on *smaller* inputs
  - $T(n)$: the worst-case running time on a problem of size *n*
  - Constant time $\Theta(1)$ if the problem size is small enough, $n < n_0$ for some constant $n_0 > 0$
  - Dividing a problem into *a* subproblems, each with size *n/b*.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0, \\ D(n) + aT\left(\frac{n}{b}\right) + C(n) & \text{otherwise.} \end{cases}$$

If n/b is not integer, just use ceiling and floor operations to adjust

Analysis of merge sort

- Divide:
  - Compute the middle position of the subarray
  - $D(n) = \Theta(1)$
- Conquer
  - Recursively solve two subproblems
  - a is 2, b is 2
- Combine
  - Scan every elements in the two sorted subarrays to merge
  - $C(n) = \Theta(n)$

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

For simplicity, just $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

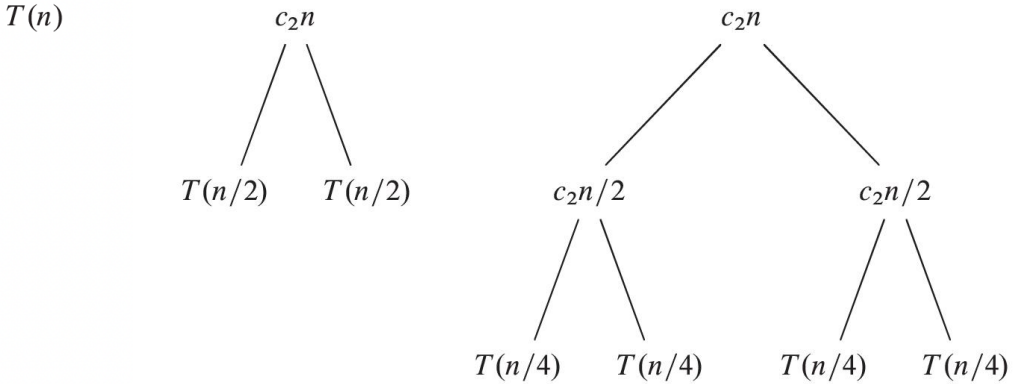
But this is still recursive.
How to solve it?

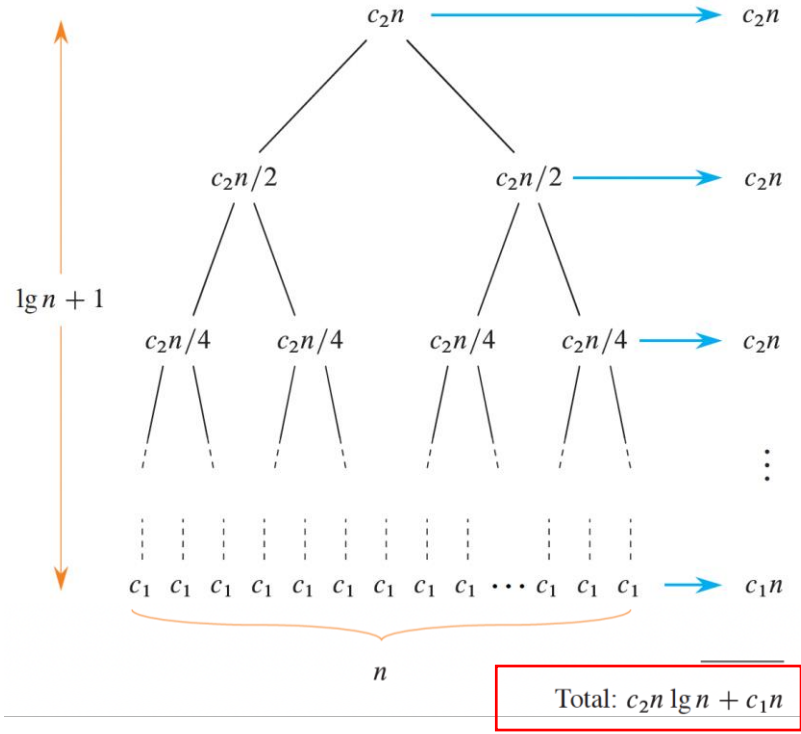
$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0, \\ D(n) + aT\left(\frac{n}{b}\right) + C(n) & \text{otherwise.} \end{cases}$$

Recursion tree of merge sort

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

$$T(n) = \begin{cases} c_1 & \text{if } n = 1, \\ 2T\left(\frac{n}{2}\right) + c_2 n & n > 1. \end{cases}$$





Overview

- Divide and conquer
  - Merge Sort
- Recursive algorithm complexity
  - Recursion tree
- **Introduction on Asymptotic Bounds** O, Ω, Θ
- Definitions on Asymptotic Bounds O, Ω, Θ
  - Bounds o, ω
- Properties of Asymptotic Bounds
- Some Notations and Common Functions
-

Running Time

- Asymptotic efficiency of an algorithm
  - How the running time increases with the input in the limit, as the input size increases without bound (i.e., large input)
  - The multiplicative constants and lower-order terms of an exact running time are dominated by the leading term in a formula

- $T(n) = \frac{c_5+c_6+c_7}{2}\boldsymbol{n^2} + \left(c_1 + c_2 + c_4 + \frac{c_5-c_6-c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$

Important notations about running time

- O
- Ω
- Θ

O-notation

- *Upper bound* on the asymptotic behavior of a function
- $7n^3 + 89n^2 - 10n + 6$
  - This function grows *no faster* than rate n^3, when n is large
  - $O(n^3)$
  - $O(n^4)$?
  - $O(n^5)$?
  - $O(n^c)$ for any $c \geq 3$

Ω-notation

- *Lower bound* on the asymptotic behavior of a function
- $7n^3 + 89n^2 - 10n + 6$
  - This function grows *at least as fast* as rate n^3 when *n* is large
  - $\Omega(n^3)$
  - $\Omega(n^2)$?
  - $\Omega(n)$?
  - $\Omega(n^c)$ for any $c \leq 3$

Θ-notation

- *Tight bound* on the asymptotic behavior of a function
  - A function grows precisely at a certain rate based on the highest-order term
- $7n^3 + 89n^2 - 10n + 6$
  - This function grows *no faster* than rate $n^3, O(n^3)$
  - This function grows *at least as fast* than rate $n^3, \Omega(n^3)$
  - Thus, it is $\Theta(n^3)$
- If a function is both $O(g(n))$ and $\Omega\big(g(n)\big)$, it is $\Theta(g(n))$

Rate/Order of growth

Function	n=16	n=1024	n=1000000
$\lg n_{(\text{base } 2)}$	4	10	~20
$\sqrt{n}$	4	32	1000
n	16	1024	1000000
$n \lg n$	64	10240	~20000000
n^2	256	1048576	10^{12}
n^3	4096	$\sim 10^9$	10^{18}
2^n	65536	… too big to show	…
$n!$	$\sim 2 * 10^{13}$	…	…

O, Ω, Θ

- If a function is $O(n^5)$ and $\Omega(\log n)$, can we get its Θ?
- If a function is $O(n^4)$ and $\Omega(n)$, can we get its Θ?
- If a function is $O(n^3)$ and $\Omega(n \log n)$, can we get its Θ?
- If a function is $O(n^2)$ and $\Omega(n^2)$, can we get its Θ?

O, Ω, Θ

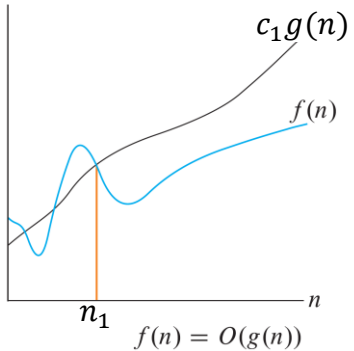
- For an algorithm, we may be interested in its
  - Worst-case O, Ω, Θ
  - Average-case O, Ω, Θ
  - Best-case O, Ω, Θ

Overview

- Divide and conquer
  - Merge Sort
- Recursive algorithm complexity
  - Recursion tree
- Introduction on Asymptotic Bounds O, Ω, Θ
- **Definitions on Asymptotic Bounds O, Ω, Θ**
  - Bounds o, ω
- Properties of Asymptotic Bounds
- Some Notations and Common Functions
-

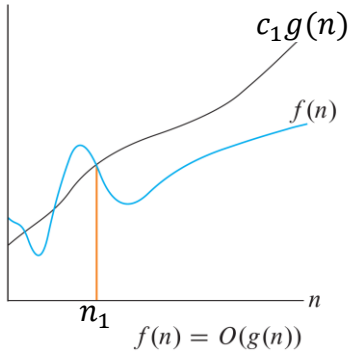
Asymptotic upper/lower/tight bounds O, Ω, Θ

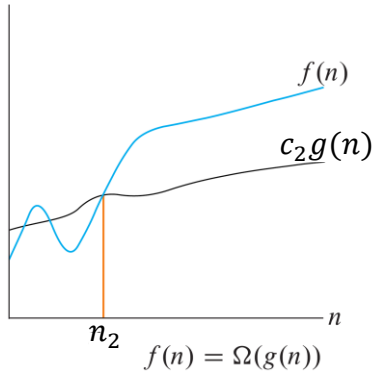
- $O(g(n)) = \{f(n)$: there exist positive constants c_1 and n_1, such that
$$0 \leq f(n) \leq c_1 g(n) \text{ for all } n \geq n_1\}$$
  - $O(g(n))$ is a set of functions with $g(n)$ as an upper bound for sufficiently large $n \geq n_1$
  - There may be many choices of n_1 and c_1



Asymptotic upper/lower/tight bounds O, Ω, Θ

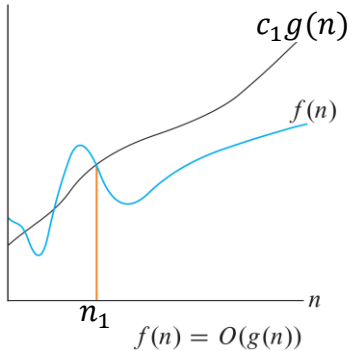
- $O(g(n)) = \{f(n)$: there exist positive constants c_1 and n_1, such that
$$0 \leq f(n) \leq c_1 g(n) \text{ for all } n \geq n_1\}$$
- $\Omega(g(n)) = \{f(n)$: there exist positive constants c_2 and n_2, such that
$$0 \leq c_2 g(n) \leq f(n) \text{ for all } n \geq n_2\}$$
  - $\Omega(g(n))$ is a set of functions with $g(n)$ as a lower bound for sufficiently large $n \geq n_2$
  - There may be many choices of n_2 and c_2

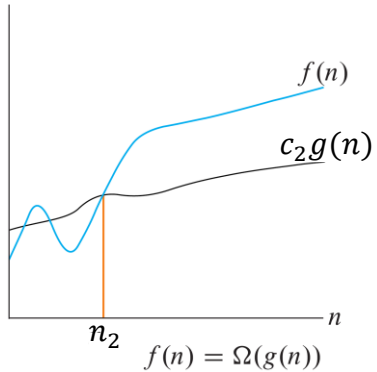


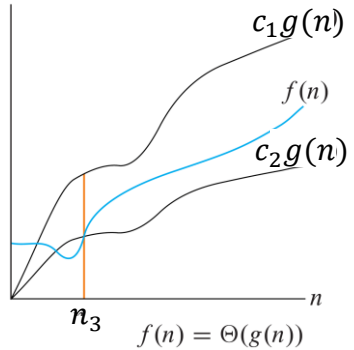


Asymptotic upper/lower/tight bounds O, Ω, Θ

- $O(g(n)) = \{f(n)$: there exist positive constants c_1 and n_1, such that
$$0 \leq f(n) \leq c_1 g(n) \text{ for all } n \geq n_1\}$$
- $\Omega(g(n)) = \{f(n)$: there exist positive constants c_2 and n_2, such that
$$0 \leq c_2 g(n) \leq f(n) \text{ for all } n \geq n_2\}$$
- $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1 *and* c_2 and n_3, such that
$$0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n) \text{ for all } n \geq n_3\}$$







Asymptotic upper bound O

- $O(g(n)) = \{f(n)$: there exist positive constants c_1 and n_1, such that $0 \leq f(n) \leq c_1 g(n)$ for all $n \geq n_1\}$

- Show that $f(n) = 4n^2 + 1000n + 240$ is $O(n^2)$
  - Find proper c_1 and n_1 such that $4n^2 + 1000n + 240 \leq c_1 n^2$ holds for all $n \geq n_1$
  - $4 + 1000/n + 240/n^2 \leq c_1$
  - If $n_1 = 10$, then any $c_1 \geq 106.4$ works to let the inequality hold, e.g., $c_1 = 107$
  - If $n_1 = 25$, then any $c_1 \geq 44.384$ works to let the inequality hold, e.g., $c_1 = 35$
  - There are many choices, and you only need to find one.

  - Another way:
  - $4n^2 + 1000n + 240 \leq 4n^2 + 1000n^2 + 240n^2 \leq 1244n^2$ *(true for all* $n \geq 1$*)*
  - Thus we can choose $c_1 = 1244$ and $n_1 = 1$

Asymptotic upper bound O

- $O(g(n)) = \{f(n)$: there exist positive constants c_1 and n_1, such that
$$0 \leq f(n) \leq c_1 g(n) \text{ for all } n \geq n_1\}$$

- Show that $f(n) = 4n^3 + 1000n + 240$ is not $O(n^2)$
  - Proof by Contradiction:
  - If there exists $c_1 > 0$ and $n_1 > 0$ such that $4n^3 + 1000n + 240 \leq c_1 n^2$ for **all $n \geq n_1$**
  - $n \leq (c_1 - \frac{1000}{n} - 240/n^2)/4$
  - n is bounded by a constant c_1, but n should not be upper bounded. Contradiction occurs

Asymptotic lower bound Ω

- $\Omega(g(n)) = \{f(n)$: there exist positive constants c_2 and n_2, such that
$$0 \leq c_2 g(n) \leq f(n) \text{ for all } n \geq n_2\}$$

- Show that $f(n) = 4n^2 + 1000n + 240$ is $\Omega(n^2)$
  - Find proper c_2 and n_2 such that $4n^2 + 1000n + 240 \geq c_2 n^2$ for all $n \geq n_2$
  - $4 + \frac{1000}{n} + \frac{240}{n^2} \geq c_2$
  - This holds if $c_2 = 4$ and n_2 is any positive integer

Asymptotic tight bound Θ

- $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1 *and* c_2 and n_3, such that
$$0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n) \text{ for all } n \geq n_3\}$$
- Show that $f(n) = 4n^2 + 1000n + 240$ is $\Theta(n^2)$
  - Find proper c_1 and n_1 such that $4n^2 + 1000n + 240 \leq c_1 n^2$ for all $n \geq n_1$
    - $4 + 1000/n + 240/n^2 \leq c_1$
    - If $n_1 = 10$, then any $c_1 \geq 106.4$ works, e.g., $c_1 = 107$
    - $O(n^2)$
  - Find proper c_2 and n_2 such that $4n^2 + 1000n + 240 \geq c_2 n^2$ for all $n \geq n_2$
    - $4 + \frac{1000}{n} + \frac{240}{n^2} \geq c_2$
    - This holds if $c_2 = 4$ and n_2 is any positive integer
    - $\Omega(n^2)$
  - Thus, we find $c_1 = 107$, $c_2 = 4, n_3 = 10$ such that $0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n)$ holds, and thus $f(n)$ is $\Theta(n^2)$

Asymptotic tight bound Θ

- $f(n) = \Theta\big(g(n)\big)$ if and only if $f(n) = \Omega\big(g(n)\big)$ and $f(n) = \mathrm{O}\big(g(n)\big)$

Insertion sort complexity

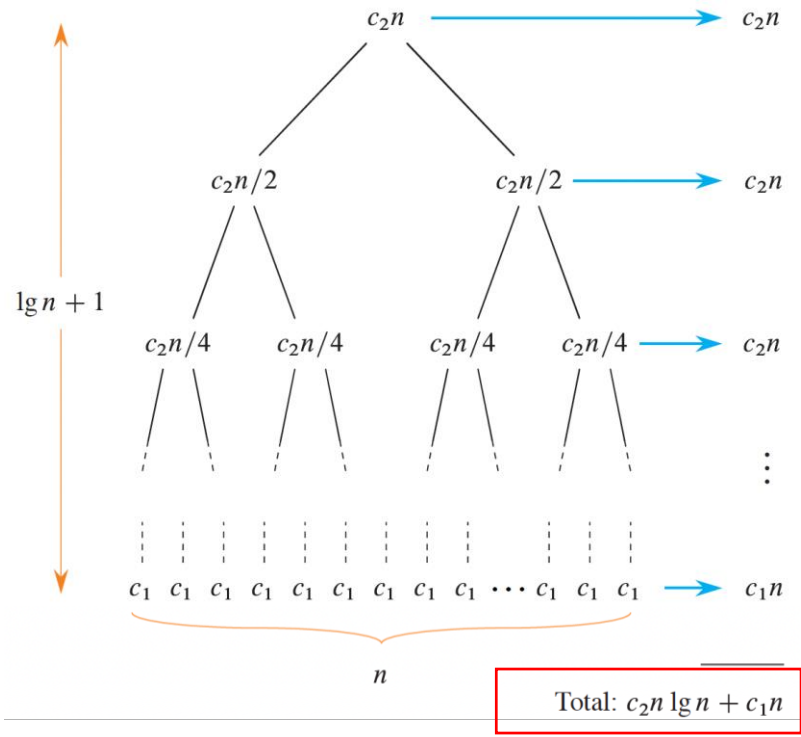
- Insertion sort has *worst-case* running time $\Omega(n^2)$, $O(n^2)$, $\Theta(n^2)$
- Insertion sort has *best-case* running time $\Omega(n)$, $O(n)$, $\Theta(n)$
- Which of the following statements are correct?
  - Insertion sort has running time $O(n^2)$?
  - Insertion sort has running time $\Omega(n)$?
  - Insertion sort has running time $\Omega(n^2)$?
  - Insertion sort has running time $\Theta(n^2)$?
- You need to consider if the statement covers all cases or partial cases and if it is overstating or not.
  - Be precise

As accurate as possible for running time analysis

- Insertion sort has running time $O(n^2)$
- Insertion sort has running time $O(n^3)$
  - This is correct but loose
- In the analysis, try to make O(…) as small as possible

Merge sort complexity

- When analyzing the complexity of merge sort, do we consider if input instances is worst-case or not?
  - No
  - For all cases, merge sort has complexity $\Theta(n \lg n)$
- Which is correct?
  - Merge sort has running time $\Theta(n \lg n)$?
  - Insertion sort has running time $\Theta(n^2)$?



Simplification Rules of O-Notation

These rules provide shortcuts for showing that $f(n)$ is $O(g(n))$

- Ignore constant factors
  - E.g., 3 is $O(1)$
  - E.g., $3n$ is $O(n)$
- Combine fixed number of terms with same complexity
  - E.g., $O(n) + O(n)$ is $O(n)$
  - E.g., $O(n) + O(n) + O(n)$ is $O(n)$
- Polynomial
  - Just use the highest-degree term. Why?
  - E.g., $5n^2 + 3n$ is $O(n^2)$
  - E.g., $2n^3 + 5n^2 + 3n$ is $O(n^3)$

Simplification Rules of O-Notation

- Can we combine a variable number of terms with the same complexity?

- How do you simplify

$$\underbrace{O(n) + O(n) + \ldots\ldots + O(n)}_{n \text{ terms}}$$

  - Is it $O(n)$ or $O(n^2)$?

Overview

- Divide and conquer
  - Merge Sort
- Recursive algorithm complexity
  - Recursion tree
- Introduction on Asymptotic Bounds O, Ω, Θ
- Definitions on Asymptotic Bounds O,Ω, Θ
  - **Bounds o, ω**
- Properties of Asymptotic Bounds
- Some Notations and Common Functions
-

O v.s. o

- $O(g(n)) = \{f(n)$: there *exist* positive constants c_1 and n_1, such that
$$0 \leq f(n) \leq c_1 g(n) \text{ for all } n \geq n_1\}$$
- $o(g(n)) = \{f(n)$: for *any* positive constant c_1, there exists $n_1 > 0$, such that
$$0 \leq f(n) < c_1 g(n) \text{ for all } n \geq n_1\}$$
  - For all constants $c>0$
  - Asymptotically not tight upper bound
- $2n = o(n^2)$
- $2n^2 \neq o(n^2)$
- Informally
  - O(b) is like $a \leq b$
  - o(b) is like $a < b$

Ω v.s. ω

- $\Omega(g(n)) = \{f(n)$: there exist positive constants c_2 and n_2, such that
$$0 \leq c_2 g(n) \leq f(n) \text{ for all } n \geq n_2\}$$
- $\omega(g(n)) = \{f(n)$: for any positive constant c_2, there exists $n_2 > 0$, such that
$$0 \leq c_2 g(n) < f(n) \text{ for all } n \geq n_2\}$$
  - For all constants $c>0$
  - Asymptotically not tight lower bound
- $2n^2 = \omega(n)$
- $2n^2 \neq \omega(n^2)$
- Informally
  - Ω(b) is like $a \geq b$
  - ω(b) is like $a > b$

Overview

- Divide and conquer
  - Merge Sort
- Recursive algorithm complexity
  - Recursion tree
- Introduction on Asymptotic Bounds O, Ω, Θ
- Definitions on Asymptotic Bounds O,Ω, Θ
  - Bounds o, ω
- **Properties of Asymptotic Bounds**
- Some Notations and Common Functions
-

Transitivity

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \rightarrow f(n) = \Omega(h(n))$$
$$f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \rightarrow f(n) = o(h(n))$$
$$f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \rightarrow f(n) = \omega(h(n))$$

$n^{1.5} = O(n^2)$ $n^2 = O(n^3)$ $n^{1.5} = O(n^3)$

Reflexivity, Symmetry, Transpose Symmetry

- Reflexivity

$$f(n) = \Theta(f(n))$$
$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n)).$$

- Symmetry

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))$$

- Transpose Symmetry

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)),$$
$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)).$$

Overview

- Divide and conquer
  - Merge Sort
- Recursive algorithm complexity
  - Recursion tree
- Introduction on Asymptotic Bounds O, Ω, Θ
- Definitions on Asymptotic Bounds O,Ω, Θ
  - Bounds o, ω
- Properties of Asymptotic Bounds
- **Some Notations and Common Functions**
-

Some notations and common functions

- Monotonicity
  - $f(n)$ is monotonically increasing if $m \leq n$ implies $f(m) \leq f(n)$
  - $f(n)$ is strictly increasing if $m < n$ implies $f(m) < f(n)$

  - $f(n)$ is monotonically decreasing if $m \leq n$ implies $f(m) \geq f(n)$
  - $f(n)$ is strictly decreasing if $m < n$ implies $f(m) > f(n)$

Some notations and common functions

- Floor and ceiling
  - Real number x, e.g., 3.4
  - Floor $\lfloor x \rfloor$: the greatest integer $\leq x$
    - $\lfloor 3.4 \rfloor = 3$
    - $\lfloor -3.4 \rfloor = -4$
  - Ceiling $\lceil x \rceil$: the smallest integer $\geq x$
    - $\lceil 3.4 \rceil = 4$
    - $\lceil -3.4 \rceil = -3$

Some notations and common functions

- Modular arithmetic: For integer a and positive integer n
  - $a \bmod n = a - n\lfloor a/n \rfloor$
  - the value a mod n is the remainder (or residue) of the quotient a/n, and the remainder is nonnegative
    - $0 \leq a \bmod n < n$

  - 8 mod 3 = 2
  - -8 mod 3 = 1

Some notations and common functions

- Logarithms

$$\begin{aligned}
\lg n &= \log_2 n && \text{(binary logarithm)} \\
\ln n &= \log_e n && \text{(natural logarithm)} \\
\lg^k n &= (\lg n)^k && \text{(exponentiation)} \\
\lg \lg n &= \lg(\lg n) && \text{(composition)}
\end{aligned}$$

.
.
.

Summary

- Divide and conquer
  - Merge Sort
- Recursive algorithm complexity
  - Recursion tree
- Introduction on Asymptotic Complexity
- Definitions on Asymptotic Bounds O,Ω, Θ
  - Bounds o, ω
- Properties of Asymptotic Bounds
- Some Notations and Common Functions
-