CE100 Algorithms and Programming II

Week-1 (Introduction to Analysis of Algorithms)

Spring Semester, 2021-2022

Download DOC-PDF, DOC-DOCX, SLIDE, PPTX

Brief Description of Course and Rules

We will first talk about,

1. Course Plan and Communication
2. Grading System, Homeworks, and Exams

please read the syllabus carefully.

Outline (1)

- Introduction to Analysis of Algorithms
  - Algorithm Basics
  - Flowgorithm
  - Pseudocode

Outline (2)

- RAM (Random Access Machine Model)
- Sorting Problem
- Insertion Sort Analysis
- Algorithm Cost Calculation for Time Complexity
- Worst, Average, and Best Case Summary
- Merge Sort Analysis

Outline (3)

- Asymptotic Notation
  - Big O Notation
  - Big Teta Notation
  - Big Omega Notation
  - Small o Notation
  - Small omega Notation

We Need Mathematical Proofs (1)

- Direct proof
- Proof by mathematical induction
- Proof by contraposition
- Proof by contradiction
- Proof by construction
- Proof by exhaustion

We Need Mathematical Proofs (2)

- Probabilistic proof
- Combinatorial proof
- Nonconstructive proof
  - Statistical proofs in pure mathematics
  - Computer-assisted proofs

Mathematical proof - Wikipedia

Introduction to Analysis of Algorithms

- Study two sorting algorithms as examples
  - Insertion sort: Incremental algorithm
  - Merge sort: Divide-and-conquer
- Introduction to runtime analysis
  - Best vs. worst vs. average case
  - Asymptotic analysis

What is Algorithm

Algorithm: A sequence of computational steps that transform the input to the desired output

Procedure vs. algorithm

An algorithm must halt within finite time with the right output

We Need to Measure Performance Metrics

- Processing Time
- Allocated Memory
- Network Congestion
- Power Usage etc.

Example Sorting Algorithms

Input: a sequence of n numbers

$$\langle a_1, a_2, \ldots, a_n \rangle$$

Algorithm: Sorting / Permutation

$$\prod = \langle \prod_{(1)}, \prod_{(2)}, \ldots, \prod_{(n)} \rangle$$

Output: sorted permutation of the input sequence

$$\langle a_{\prod_{(1)}} \leqslant a_{\prod_{(2)}} \leqslant, \ldots, a_{\prod_{(n)}} \rangle$$

Pseudo-code notation (1)

- Objective: Express algorithms to humans in a clear and concise way
- Liberal use of English
- Indentation for block structures
- Omission of error handling and other details (needed in real programs)

You can use Flowgorithm application to understand concept easily.

Pseudo-code notation (2)

Links and Examples

Wikipedia

CS50

University of North Florida

GeeksforGeeks

Correctness (1)

We often use a **loop invariant** to help us to understand why an algorithm gives the correct answer.

Example: (Insertion Sort) at the start of each iteration of the "outer" for loop - the loop indexed by j - the subarray $A[1 \ldots j-1]$ consist of the elements originally in $A[1 \ldots j-1]$ but in sorted order.

Correctness (2)

To use a loop invariant to prove correctness, we must show 3 things about it.

- **Initialization:** It is true to the first iteration of the loop.
- **Maintaince:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant - usually along with the reason that the loop terminated - gives us a usefull property that helps show that the algorithm is correct.

RAM (Random Access Machine Model) $\Longrightarrow \Theta(1)$ (1)

- Operations
  - Single Step
  - Sequential
  - No Concurrent
  - Arithmetic
    - add, subtract, multiply, divide, remainder, floor, ceiling,
    - shift left/shift right (good by multiply/dividing 2^k)

RAM (Random Access Machine Model) $\Longrightarrow \Theta(1)$ (2)

- Data Movement
  - load, store, copy
- Control
  - conditional / unconditional branch
  - subroutine calls
  - returns

RAM (Random Access Machine Model) $\Longrightarrow \Theta(1)$ (3)

- Each instruction take a constant amount of time
- Integer will be represented by $clogn\ c \geq 1$
- $T(n)$ the running time of the algorithm:

$$\sum_{\text{all statement}} (\text{cost of statement}) * (\text{number of times statement is executed}) = T(n)$$

What is the processing time ?

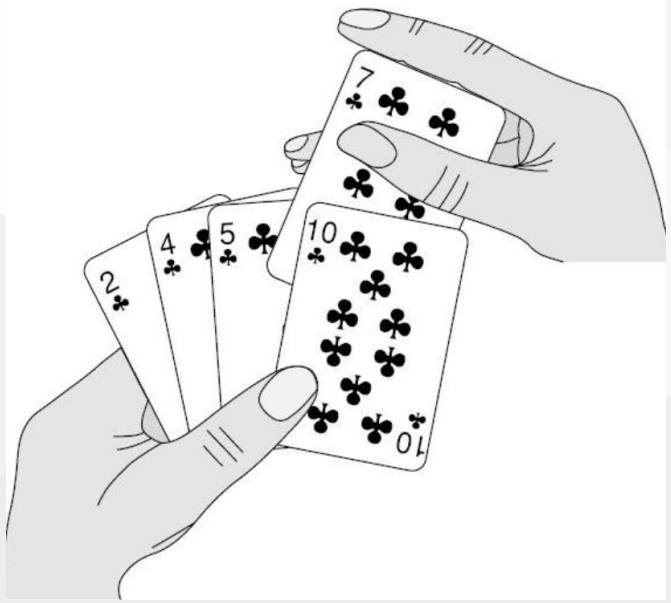
	1 Second	1 Minute	1 Hour	1 Day	1 Month	1 Year	1 Century
lgn							
$\sqrt{n}$							
n							
$nlgn$							
n^2							
n^3							
2^n							
$n!$							

Insertion Sort Algorithm (1)

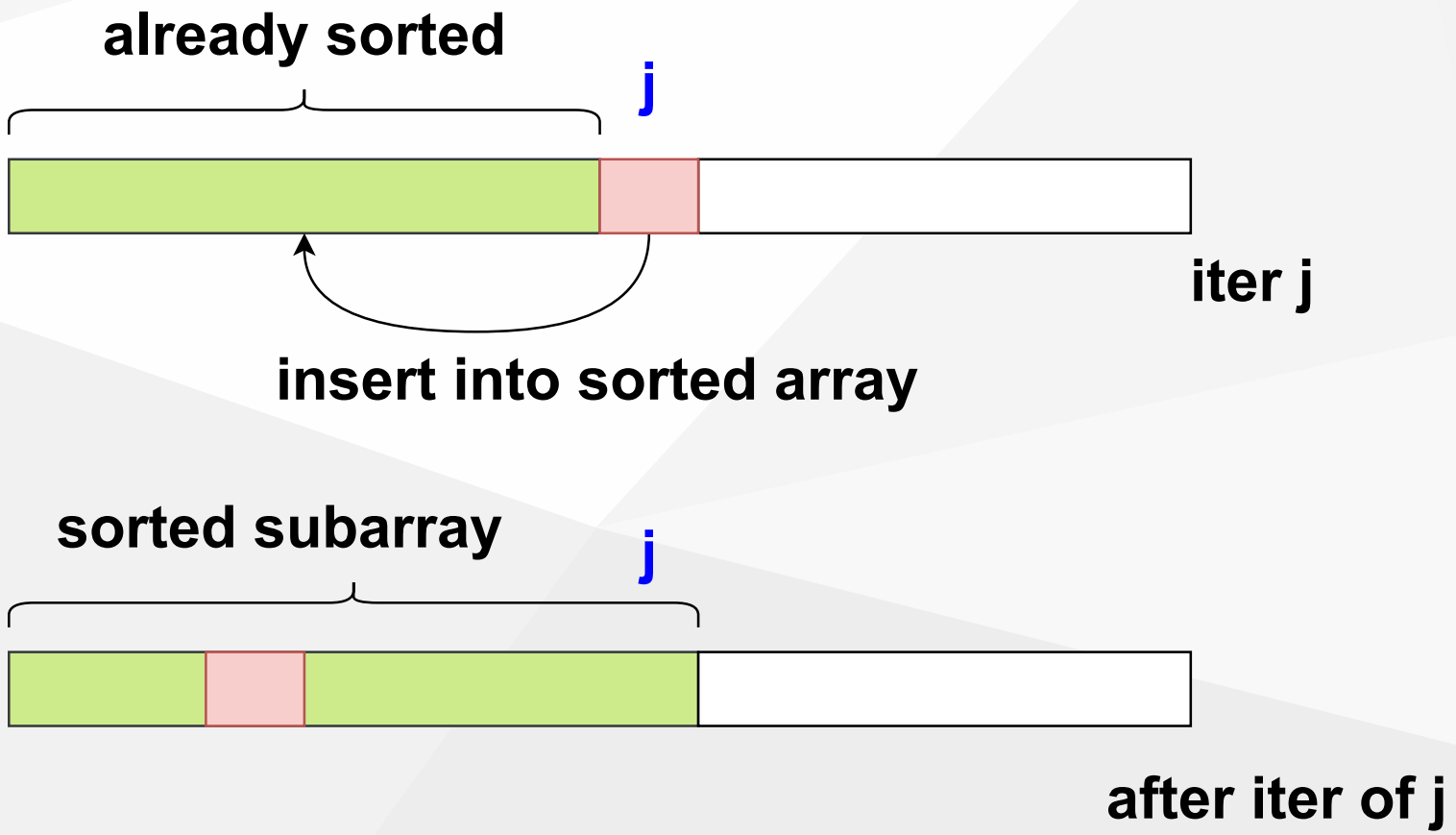
Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands

The array is virtually split into a sorted and an unsorted part

Values from the unsorted part are picked and placed at the correct position in the sorted part.



- Assume input array : $A[1..n]$
- Iterate j from 2 to n



Insertion Sort Algorithm (2)

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.             A[i+1]=A[i];
6.             i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```

Insertion Sort Algorithm (Pseudo-Code) (3)

```
Insertion-Sort(A)
1. for j=2 to A.length
2.     key = A[j]
3.     //insert A[j] into the sorted sequence 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
```

Insertion Sort Step-By-Step Description (1)

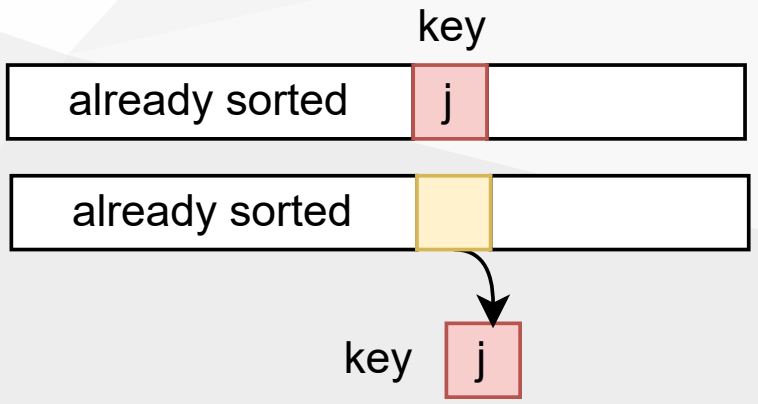
```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```

Iterate over array

Loop invariant:

The subarray $A[1..j-1]$

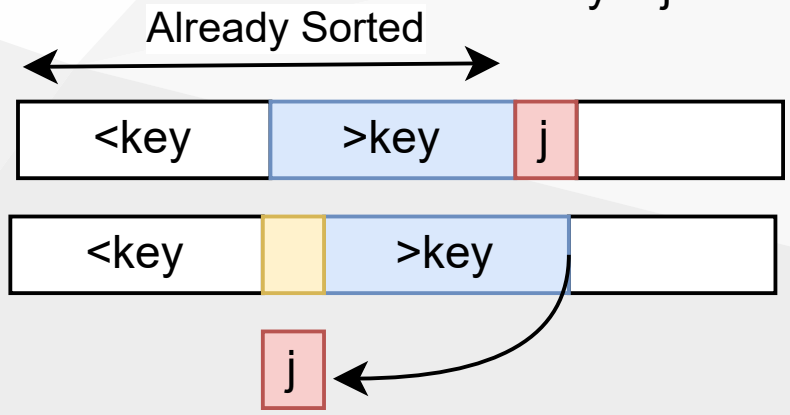
is always sorted



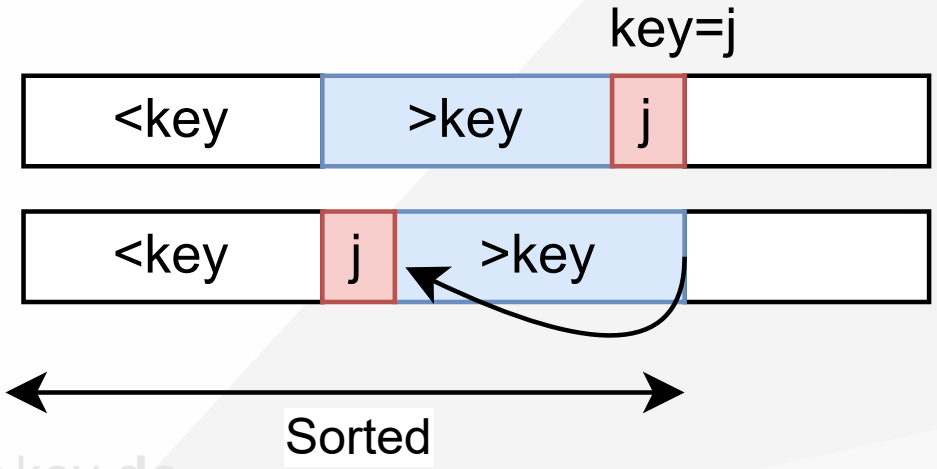
Insertion Sort Step-By-Step Description (2)

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```

Shift right the entries in $A[1..j-1]$ that are bigger than key = j



Insertion Sort Step-By-Step Description (3)




```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```

Insert key to the correct location

End of iteration $j : A[1..j]$ is sorted

Insertion Sort Example

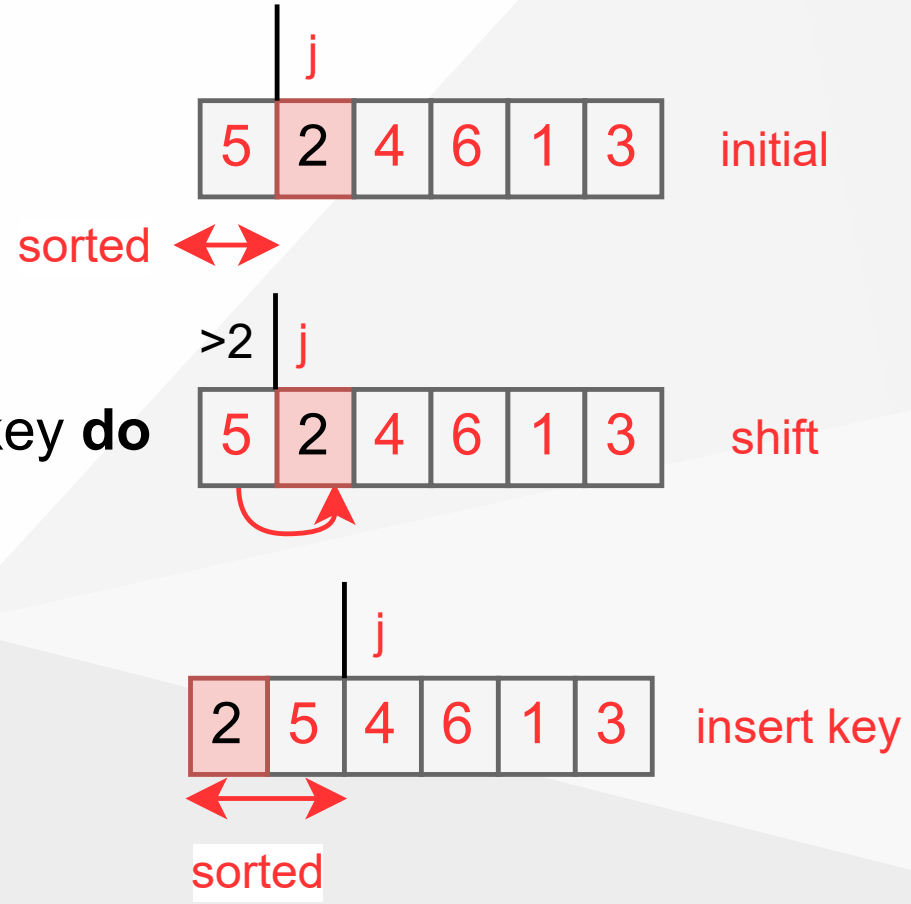
Insertion Sort Step-1 (initial)

5	2	4	6	1	3

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```

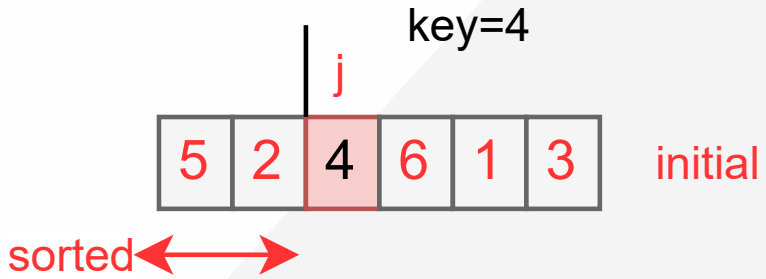
Insertion Sort Step-2 (j=2)

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```

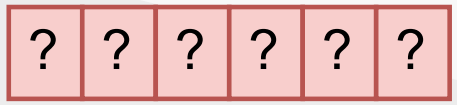


Insertion Sort Step-3 (j=3)

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```

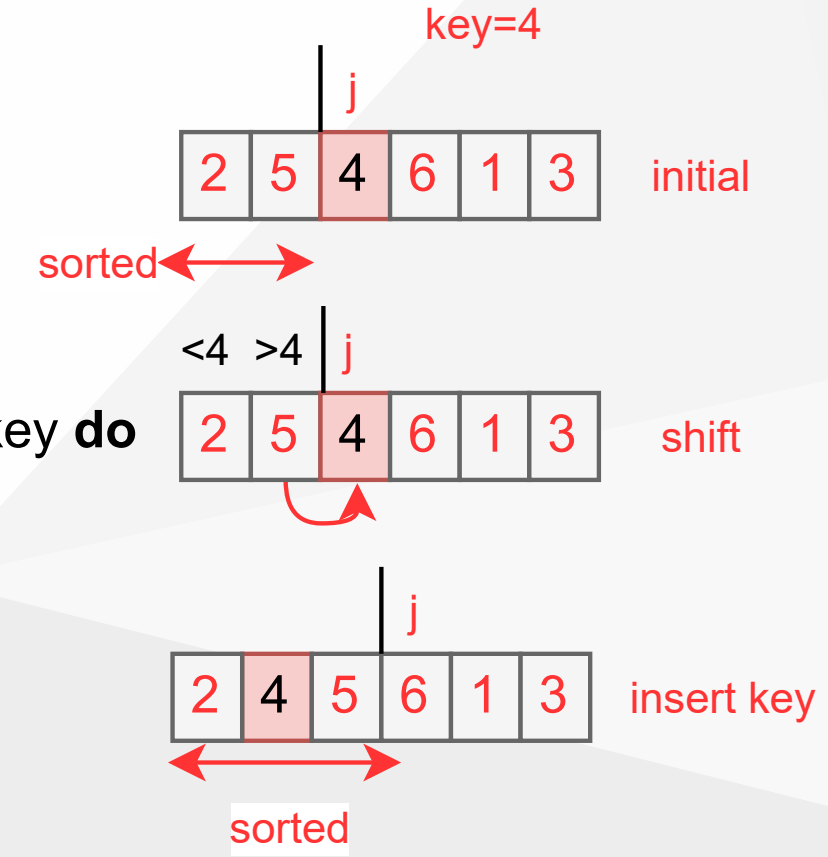


What are the entries at the end of iteration j=3?



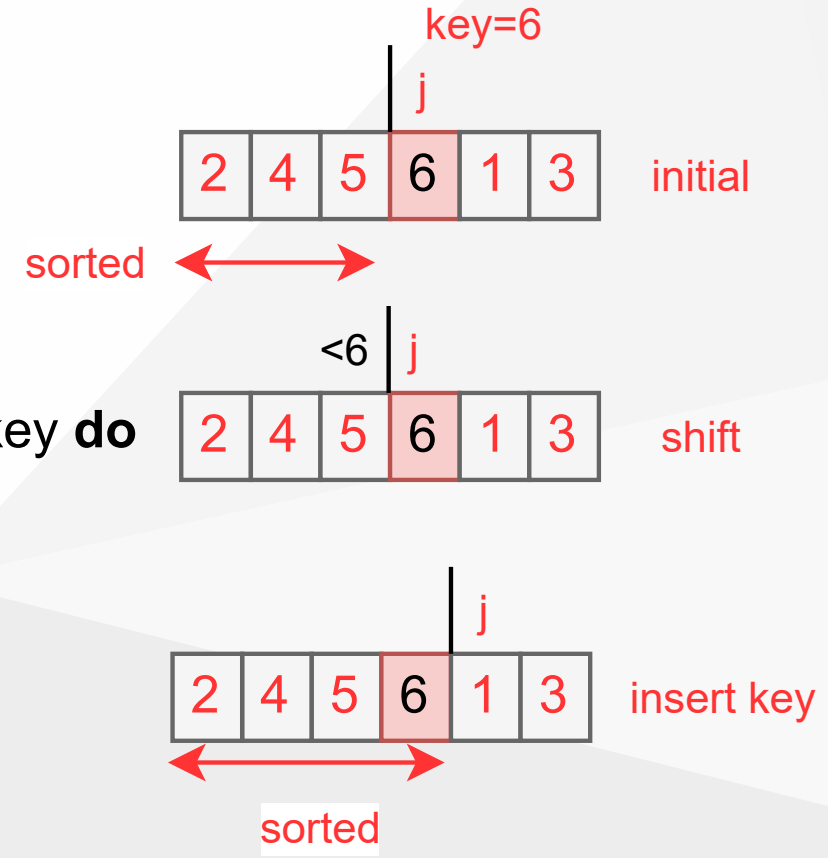
Insertion Sort Step-4 (j=3)

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```



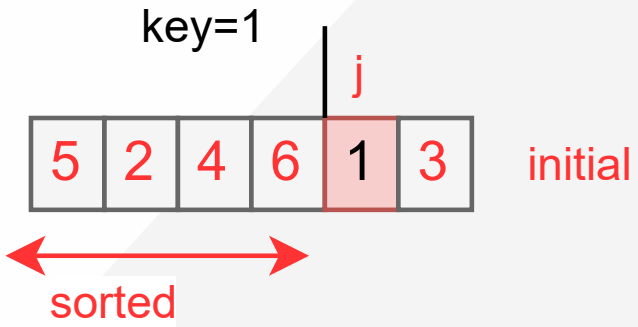
Insertion Sort Step-5 (j=4)

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```

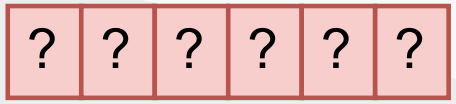


Insertion Sort Step-6 (j=5)

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```

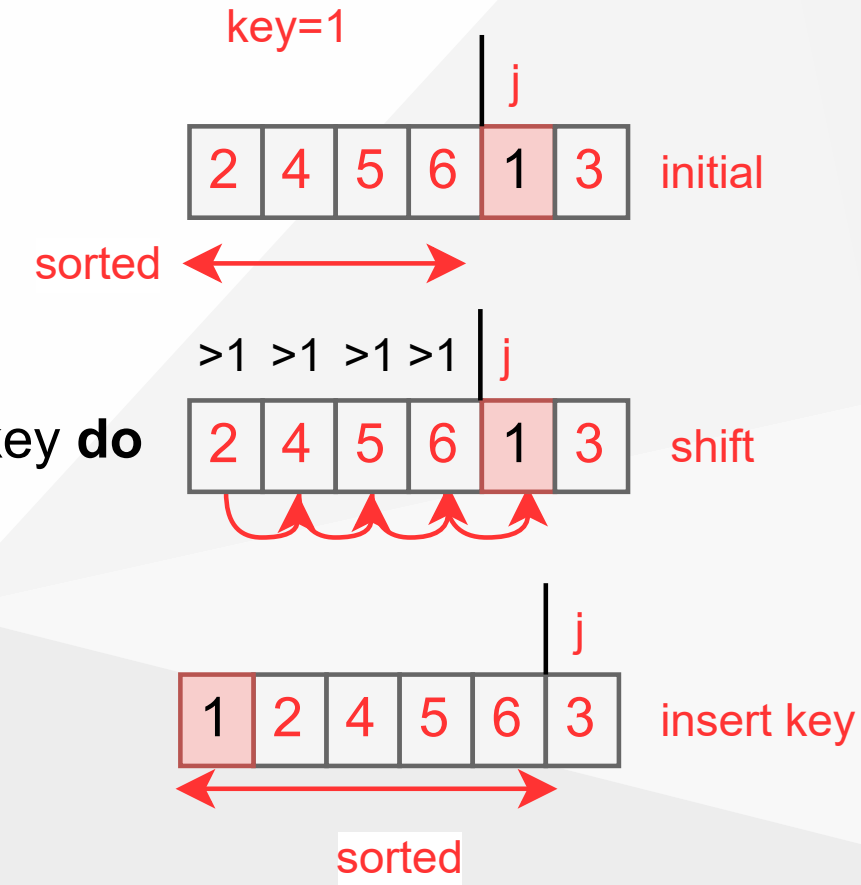


What are the entries at the end of iteration j=5?



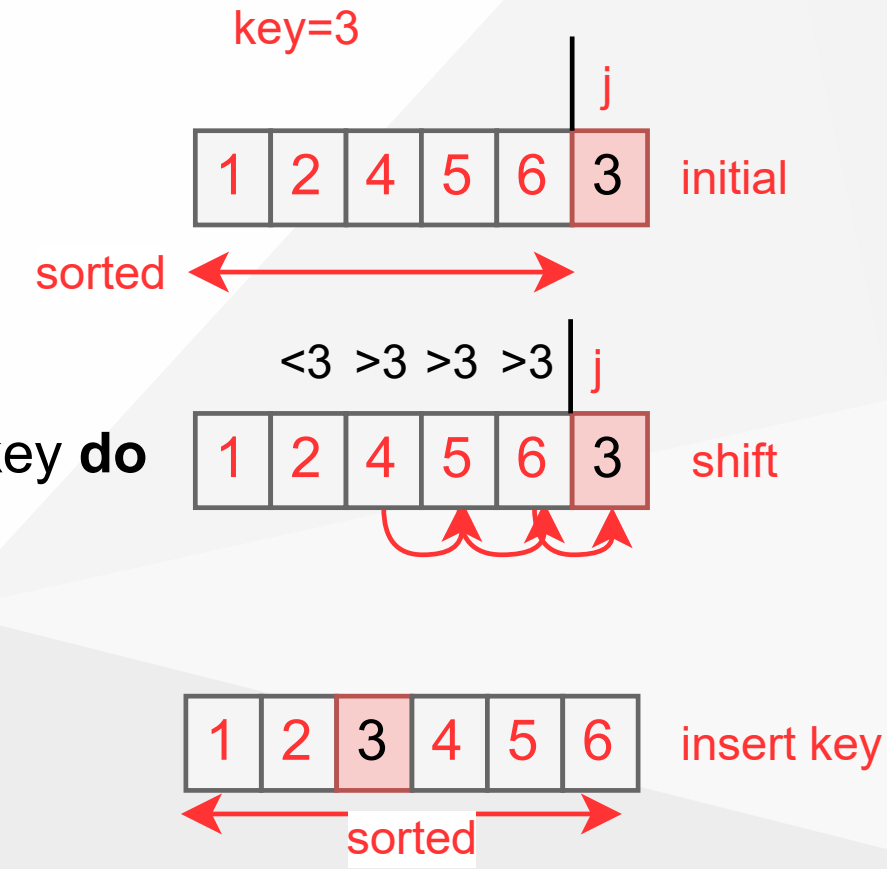
Insertion Sort Step-7 (j=5)

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```



Insertion Sort Step-8 (j=6)

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.           A[i+1]=A[i];
6.           i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```



Insertion Sort Review (1)

- Items sorted in-place
  - Elements are rearranged within the array.
  - At a most constant number of items stored outside the array at any time (e.,g. the variable key)
  - Input array A contains a sorted output sequence when the algorithm ends

Insertion Sort Review (2)

- Incremental approach
  - Having sorted $A[1..j-1]$, place $A[j]$ correctly so that $A[1..j]$ is sorted
- Running Time
  - It depends on Input Size (5 elements or 5 billion elements) and Input Itself (partially sorted)
- Algorithm approach to *upper bound* of overall performance analysis

Visualization of Insertion Sort

Sorting (Bubble, Selection, Insertion, Merge, Quick, Counting, Radix) - VisuAlgo

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

https://algorithm-visualizer.org/

HMvHTs - Online C++ Compiler & Debugging Tool - Ideone.com

Kinds of Running Time Analysis (Time Complexity)

- **Worst Case (Big-O Notation)**
  - $T(n)$ = maximum processing time of any input n
  - Presentation of Big-O : $O(n)$
- **Average Case (Teta Notation)**
  - $T(n)$ = average time over all inputs of size n, inputs can have a uniform distribution
  - Presentation of Big-Theta : $\Theta(n)$
- **Best Case (Omega Notation)**
  - $T(n)$ = min time on any input of size n, for example sorted array
  - Presentation of Big-Omega : $\Omega(n)$

Array Sorting Algorithms Time and Space Complexity

very-fast
fast
medium
slow
very-slow

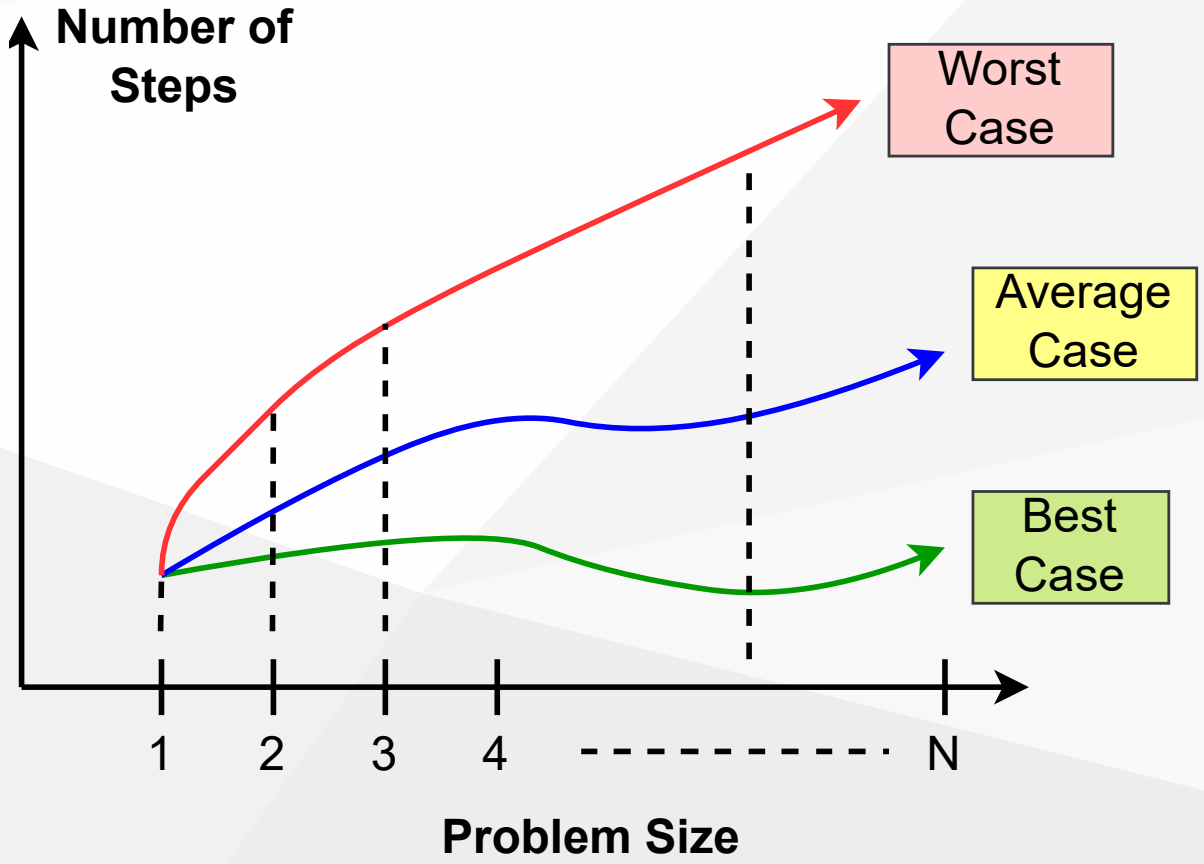
Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quick Sort	$\Omega(nlgn)$	$\Theta(nlgn)$	$O(n^2)$	$O(lgn)$
Merge Sort	$\Omega(nlgn)$	$\Theta(nlgn)$	$O(nlgn)$	$O(n)$
Tim Sort	$\Omega(n)$	$\Theta(nlgn)$	$O(nlgn)$	$O(n)$
Heap Sort	$\Omega(nlgn)$	$\Theta(nlgn)$	$O(nlgn)$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(nlgn)$	$\Theta(nlgn)$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(nlgn)$	$\Theta(n(lgn)^2)$	$O(n(lgn)^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Cube Sort	$\Omega(n)$	$\Theta(nlgn)$	$O(nlgn)$	$O(n)$

Comparison of Time Analysis Cases

For insertion sort, worst-case time depends on the speed of primitive operations such as

- **Relative Speed** (on the same machine)
- **Absolute Speed** (on different machines)
- Asymptotic Analysis
  - Ignore machine-dependent constants
  - Look at the growth of $T(n)|n \to \infty$

Asymptotic Analysis (1)



Asymptotic Analysis (2)

Theta-Notation (Average-Case)

- Drop low order terms
- Ignore leading constants

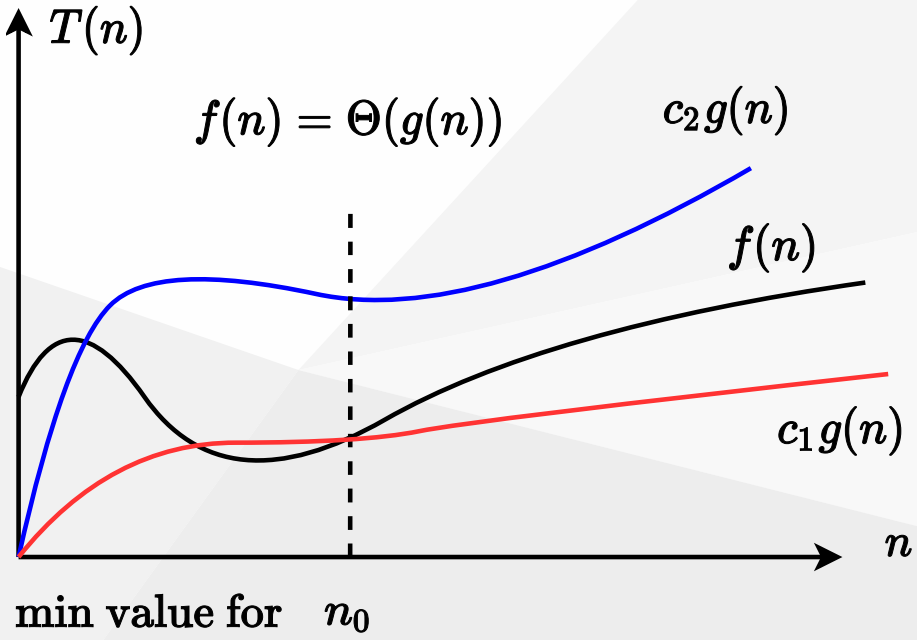
e.g

$$2n^2 + 5n + 3 = \Theta(n^2)$$
$$3n^3 + 90n^2 - 2n + 5 = \Theta(n^3)$$

- As n gets large, a $\Theta(n^2)$ algorithm runs faster than a $\Theta(n^3)$ algorithm

Asymptotic Analysis (3)

For both algorithms, we can see a minimum item size in the following chart. After this point, we can see performance differences. Some algorithms for small item size can be run faster than others but if you increase item size you will see a reference point that notation proof performance metrics.



Insertion Sort - Runtime Analysis (1)

```
Cost   Times   Insertion-Sort(A)
----   -----   --------------------
c1     n       1. for j=2 to A.length
c2     n-1     2.     key = A[j]
c3     n-1     3.     //insert A[j] into the sorted sequence A[1...j-1]
c4     n-1     4.     i = j - 1
c5     k5      5.     while i>0 and A[i]>key do
c6     k6      6.         A[i+1] = A[i]
c7     k6      7.         i = i - 1
c8     n-1     8.     A[i+1] = key
```

we have two loops here, if we sum up costs as follow we can see big-O worst case notation.

$$k_5 = \sum_{j=2}^{n} t_j \text{ and } k_6 = \sum_{j=2}^{n} t_i - 1$$

for operation counts

Insertion Sort - Runtime Analysis (2)

cost function can be evaluated as follow;

$$T(n) = c_1 n + c_2(n-1) + 0(n-1) + c_4(n-1)$$
$$+ c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} t_i - 1$$
$$+ c_7 \sum_{j=2}^{n} t_i - 1 + c_8(n-1)$$

Insertion Sort - Runtime Analysis (3)

$$\sum_{j=2}^{n} j = (n(n+1)/2) - 1$$

and

$$\sum_{j=2}^{n} j - 1 = n(n-1)/2$$

Insertion Sort - Runtime Analysis (4)

$$
\begin{aligned}
T(n) = &(c_5/2 + c_6/2 + c_7/2)n^2 \\
&+ (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n \\
&- (c_2 + c_4 + c_5 + c_6)
\end{aligned}
$$

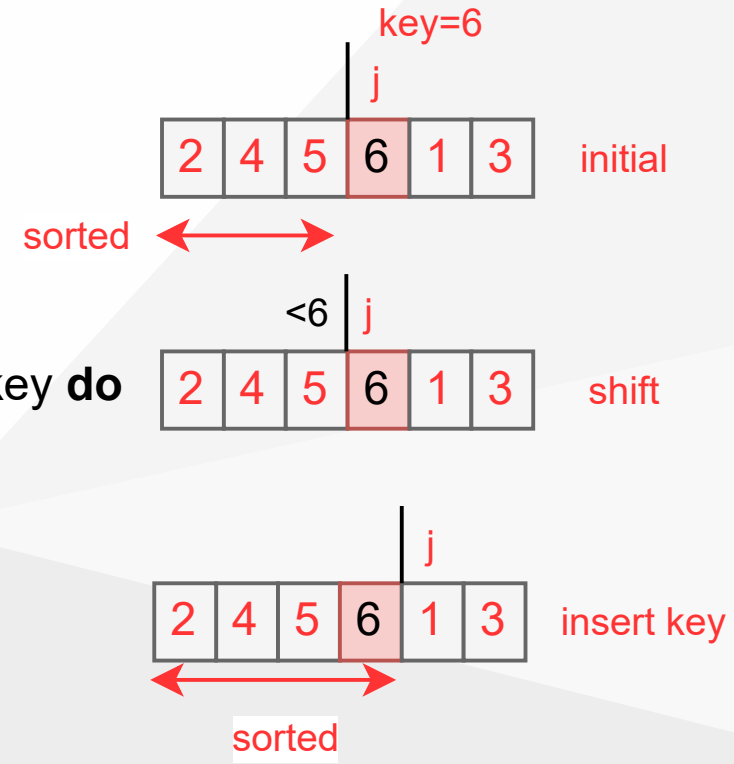
Insertion Sort - Runtime Analysis (5)

$$T(n) = an^2 + bn + c$$
$$= O(n^2)$$

Best-Case Scenario (Sorted Array) (1)

Problem-1, If $A[1...j]$ is already sorted, what will be $t_j = ?$

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```



$t_j = 1$

Best-Case Scenario (Sorted Array) (2)

Parameters are taken from image

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

$t_j = 1$ for all j

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

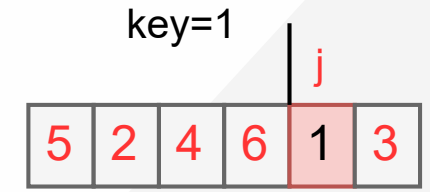
$$T(n) = an - b$$

$$= \Omega(n)$$

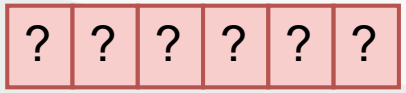
Worst-Case Scenario (Reversed Array) (1)

Problem-2 If $A[j]$ is smaller than every entry in $A[1...j-1]$, what will be $t_j = ?$

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```



What are the entries at the end of iteration j=5?



$t_j = ?$

Worst-Case Scenario (Reversed Array) (2)

The input array is reverse sorted $t_j = j$ for all j after calculation worst case runtime will be

$$
\begin{aligned}
T(n) &= 1/2(c_4 + c_5 + c_6)n^2 \\
&\quad + (c_1 + c_2 + c_3 + 1/2(c_4 - c_5 - c_6) + c_7)n - (c_2 + c_3 + c_4 + c_7) \\
T(n) &= 1/2an^2 + bn - c \\
&= O(n^2)
\end{aligned}
$$

Asymptotic Runtime Analysis of Insertion-Sort

```
Insertion-Sort(A)
1. for j = 2 to n do
2.     key = A[j];
3.     i = j-1;
4.     while i>0 and A[i]>key do
5.         A[i+1]=A[i];
6.         i = i-1;
       endwhile
7.     A[i+1]=key;
   endfor
```

$\Theta(1)$

$\Theta(1)$

$\Theta(1)$

Insertion-Sort Worst-case (input reverse sorted)

Inner Loop is $\Theta(j)$

$$T(n) = \sum_{j=2}^{n} \Theta(j)$$

$$= \Theta(\sum_{j=2}^{n} j)$$

$$= \Theta(n^2)$$

Insertion-Sort Average-case (all permutations uniformly distributed)

Inner Loop is $\Theta(j/2)$

$$
\begin{aligned}
T(n) &= \sum_{j=2}^{n} \Theta(j/2) \\
&= \sum_{j=2}^{n} \Theta(j) \\
&= \Theta(n^2)
\end{aligned}
$$

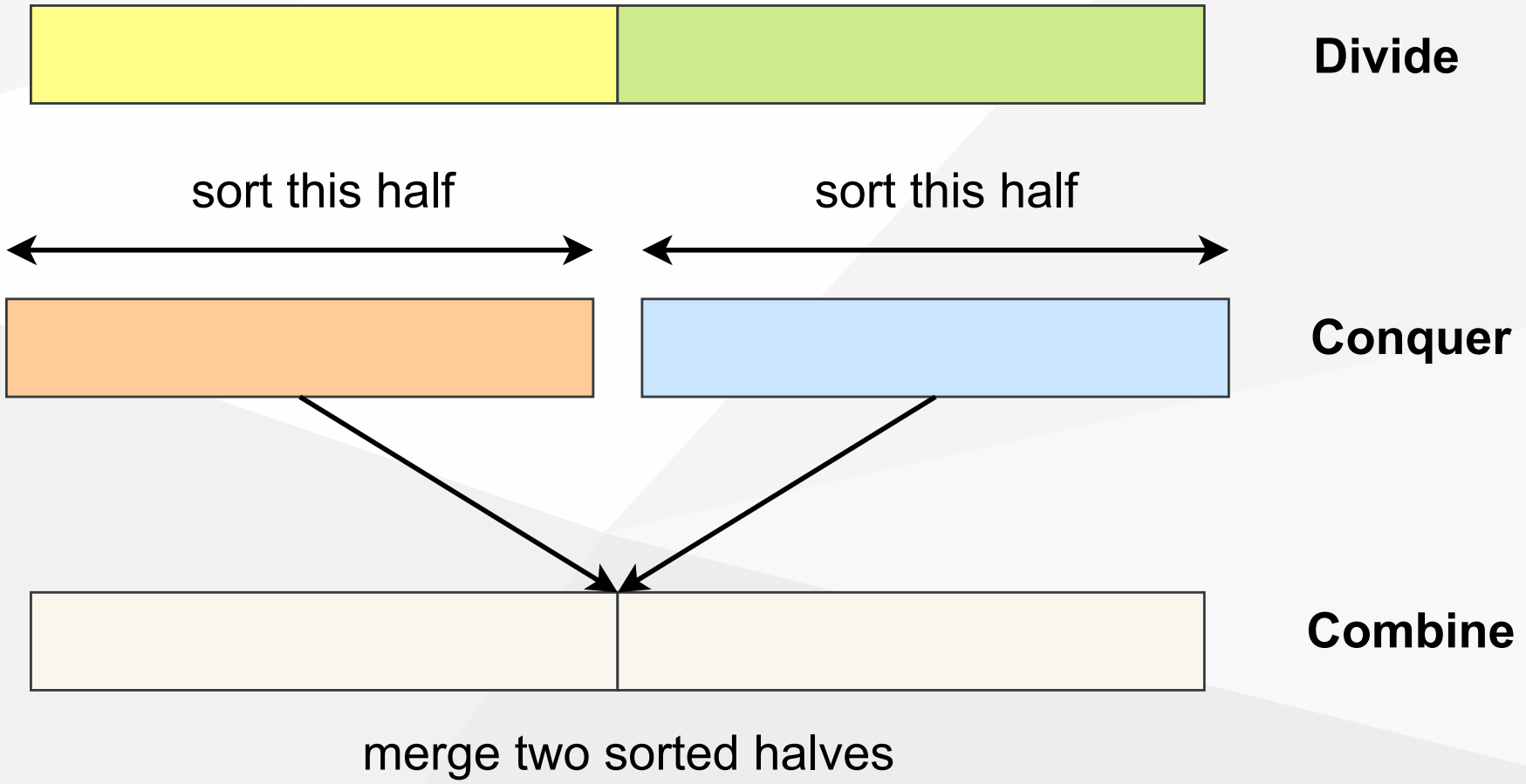
Array Sorting Algorithms Time/Space Complexities

To compare this sorting algorithm please check the following map again.

very-fast
fast
medium
slow
very-slow

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quick Sort	$\Omega(nlgn)$	$\Theta(nlgn)$	$O(n^2)$	$O(lgn)$
Merge Sort	$\Omega(nlgn)$	$\Theta(nlgn)$	$O(nlgn)$	$O(n)$
Tim Sort	$\Omega(n)$	$\Theta(nlgn)$	$O(nlgn)$	$O(n)$
Heap Sort	$\Omega(nlgn)$	$\Theta(nlgn)$	$O(nlgn)$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(nlgn)$	$\Theta(nlgn)$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(nlgn)$	$\Theta(n(lgn)^2)$	$O(n(lgn)^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Cube Sort	$\Omega(n)$	$\Theta(nlgn)$	$O(nlgn)$	$O(n)$

Merge Sort : Divide / Conquer / Combine (1)



Merge Sort : Divide / Conquer / Combine (2)

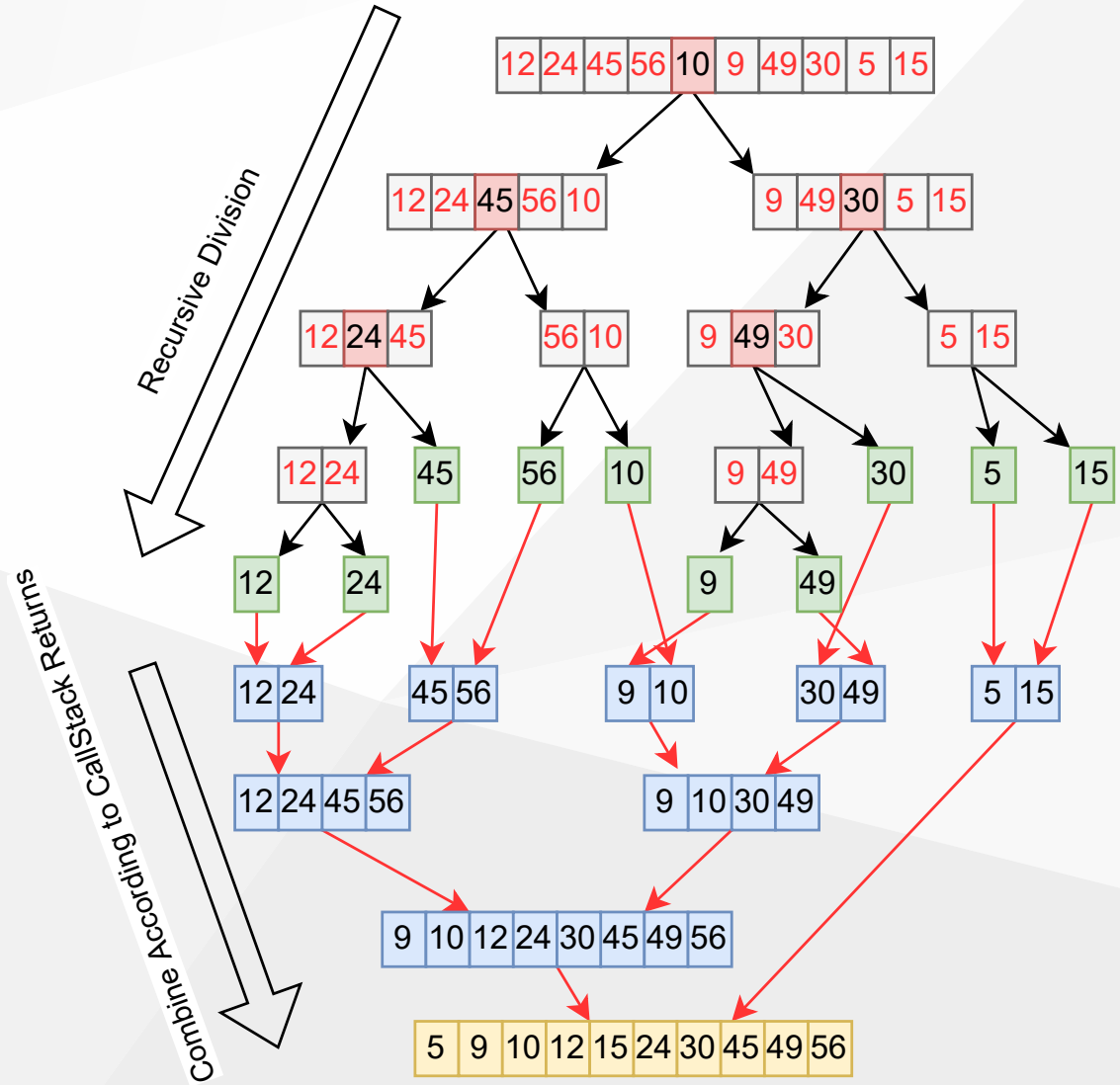
Divide: we divide the problem into a number of subproblems

Conquer: We solve the subproblems recursively

Base-Case: Solve by Brute-Force

Combine: Subproblem solutions to the original problem

Merge Sort Example



Merge Sort Algorithm (initial setup)

Merge Sort is a recursive sorting algorithm, for initial case we need to call `Merge-Sort(A,1,n)` for sorting $A[1..n]$

initial case

```
A : Array
p : 1 (offset)
r : n (length)
Merge-Sort(A,1,n)
```

Merge Sort Algorithm (internal iterations)

internal iterations

```
A : Array
p : offset
r : length
Merge-Sort(A,p,r)
    if p=r then                    (CHECK FOR BASE-CASE)
        return
    else
        q = floor((p+r)/2)     (DIVIDE)
        Merge-Sort(A,p,q)      (CONQUER)
        Merge-Sort(A,q+1,r)    (CONQUER)
        Merge(A,p,q,r)         (COMBINE)
    endif
```

Merge Sort Algorithm (Combine-1)

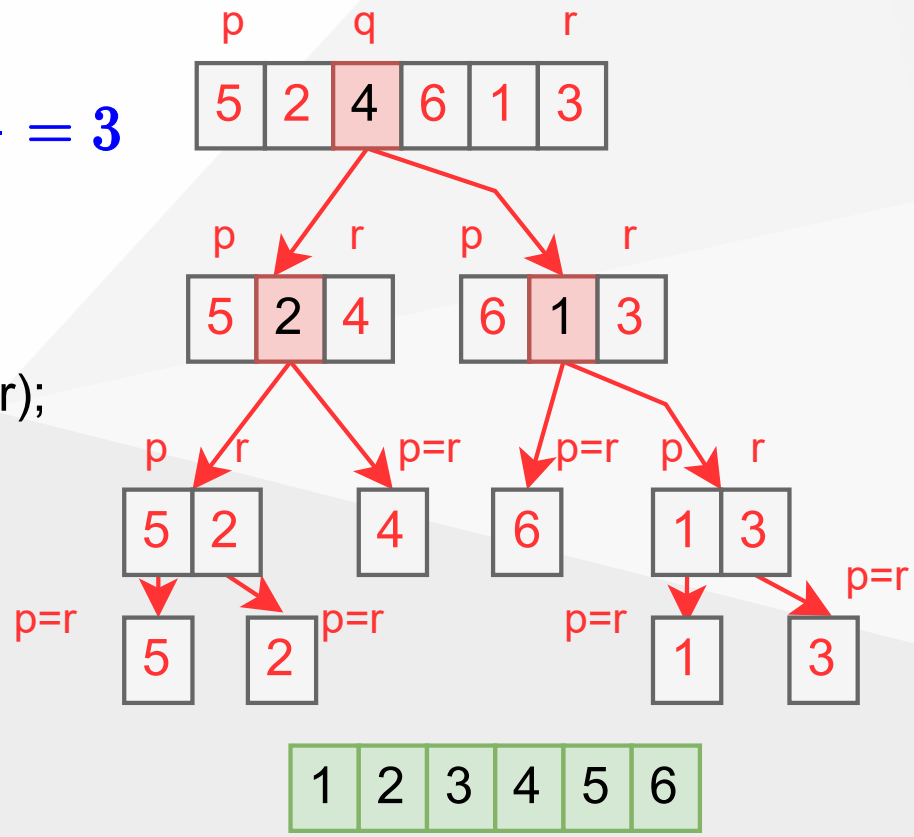
$p = start - point$

$q = mid - point$

$r = end - point$

```
Merge-Sort(A,p,r)
1. if p==r then
2.     return;
3. else
4.     q = floor((p+r)/2)
5.     Merge-Sort(A,p,q);
6.     Merge-Sort(A,q+1,r);
7.     Merge(A,p,q,r)
   endif
```

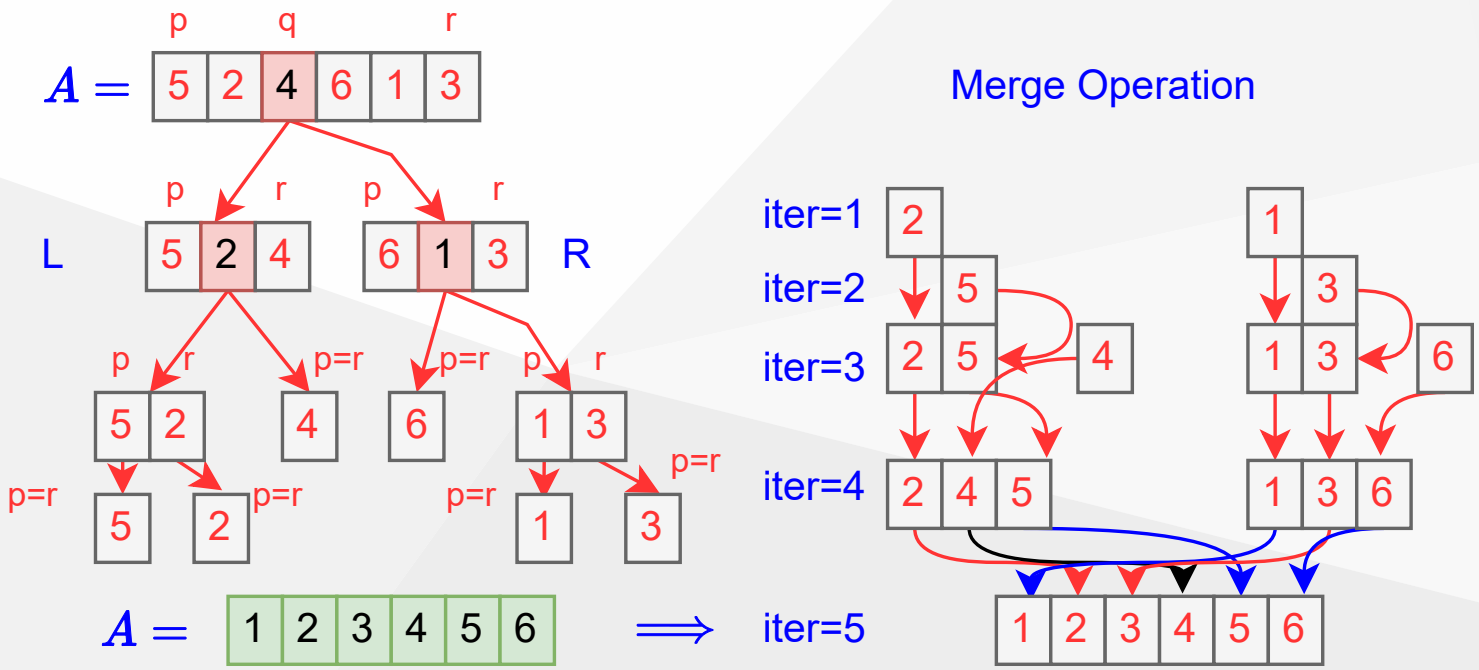
$$p = \frac{6}{2} = 3$$



Merge Sort Algorithm (Combine-2)

brute-force task, merging two sorted subarrays

The pseudo-code in the textbook (Sec. 2.3.1)



Merge Sort Combine Algorithm (1)

```
Merge(A,p,q,r)
    n1 = q-p+1
    n2 = r-q

    //allocate left and right arrays
    //increment will be from left to right
    //left part will be bigger than right part

    L[1...n1+1] //left array
    R[1...n2+1] //right array

    //copy left part of array
    for i=1 to n1
        L[i]=A[p+i-1]

    //copy right part of array
    for j=1 to n2
        R[j]=A[q+j]

    //put end items maximum values for termination
    L[n1+1]=inf
    R[n2+1]=inf

    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
```

What is the complexity of merge operation?

You can find by counting loops will provide you base constant nested level will provide you exponent of this constant, if you drop constants you will have complexity

we have 3 for loops

it will look like $3n$ and $\Theta(n)$ will be merge complexity

Merge Sort Correctness

- **Base case**
  - $p = r$ (Trivially correct)
- **Inductive hypothesis**
  - MERGE-SORT is correct for any subarray that is a strict (smaller) subset of $A[p, q]$.
- **General Case**
  - MERGE-SORT is correct for $A[p, q]$. From inductive hypothesis and correctness of Merge.

Merge Sort Algorithm (Pseudo-Code)

```
A : Array
p : offset
r : length
Merge-Sort(A,p,r)
    if p=r then                    (CHECK FOR BASE-CASE)
        return
    else
        q = floor((p+r)/2)     (DIVIDE)
        Merge-Sort(A,p,q)      (CONQUER)
        Merge-Sort(A,q+1,r)    (CONQUER)
        Merge(A,p,q,r)         (COMBINE)
    endif
```

Merge Sort Algorithm Complexity

```
A : Array
p : offset
r : length
Merge-Sort(A,p,r)-------------> T(n)
    if p=r then--------------->Theta(1)
        return
    else
        q = floor((p+r)/2)---->Theta(1)
        Merge-Sort(A,p,q)-----> T(n/2)
        Merge-Sort(A,q+1,r)---> T(n/2)
        Merge(A,p,q,r)-------->Theta(n)
    endif
```

Merge Sort Algorithm Recurrence

We can describe a function recursively in terms of itself, to analyze the performance of recursive algorithms

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

How To Solve Recurrence (1)

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

How To Solve Recurrence (2)

We will assume $T(n) = \Theta(1)$ for sufficiently small n to rewrite equation as

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

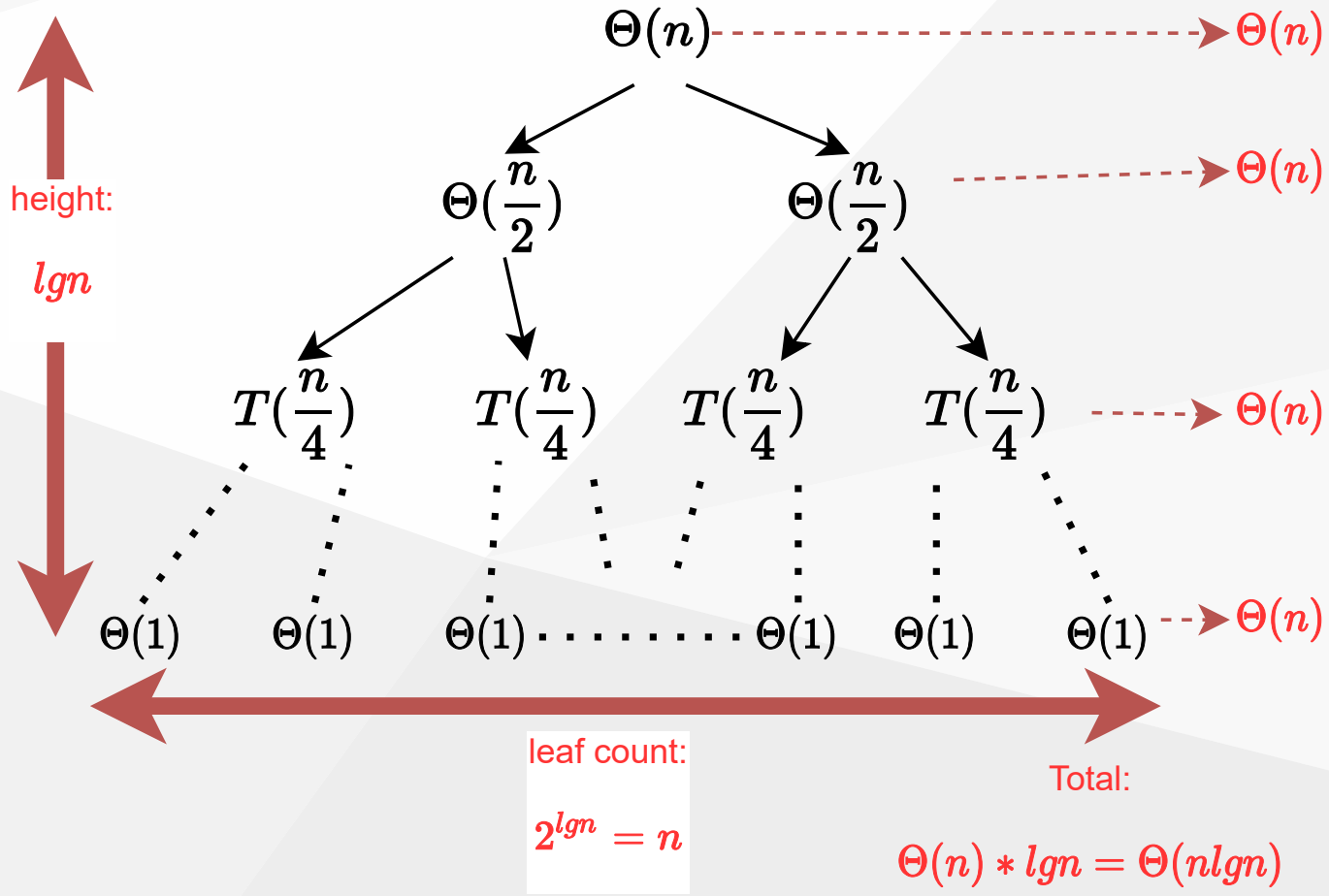
Solution for this equation will be $\Theta(nlgn)$ with following recursion tree.

How To Solve Recurrence (3)

Multiply by height $\Theta(lgn)$ with each level cost $\Theta(n)$ we can found $\Theta(nlgn)$

How To Solve Recurrence (4)

This tree is binary-tree and binary-tree height is related with item size.



How Height of a Binary Tree is Equal to $logn$? (1)

Merge-Sort recursion tree is a perfect binary tree, a binary tree is a tree which every node has at most two children, A perfect binary tree is binary tree in which all internal nodes have exactly two children and all leaves are at the same level.

How Height of a Binary Tree is Equal to $logn$? (2)

Let n be the number of nodes in the tree and let l_k denote the number of nodes on level k. According to this;

- $l_k = 2l_{k-1}$ i.e. each level has exactly twice as many nodes as the previous level
- $l_0 = 1$, i.e. on the first level we have only one node (the root node)
- The leaves are at the last level, l_h where h is the height of the tree.

How Height of a Binary Tree is Equal to $logn$? (3)

The total number of nodes in the tree is equal to the sum of the nodes on all the levels: nodes n

$$1 + 2^1 + 2^2 + 2^3 + ... + 2^h = n$$
$$1 + 2^1 + 2^2 + 2^3 + ... + 2^h = 2^{h+1} - 1$$
$$2^{h+1} - 1 = n$$
$$2^{h+1} = n + 1$$
$$log_2 2^{h+1} = log_2(n+1)$$
$$h + 1 = log_2(n+1)$$
$$h = log_2(n+1) - 1$$

How Height of a Binary Tree is Equal to $logn$? (3)

If we write it as asymptotic approach, we will have the following result

$$\text{height of tree is } h = log_2(n+1) - 1 = O(logn)$$

also

$$\text{number of leaves is } l_h = (n+1)/2$$

nearly half of the nodes are at the leaves

Review

$\Theta(nlgn)$ grows more slowly than $\Theta(n^2)$

Therefore Merge-Sort beats Insertion-Sort in the worst case

In practice Merge-Sort beats Insertion-Sort for $n > 30$ or so

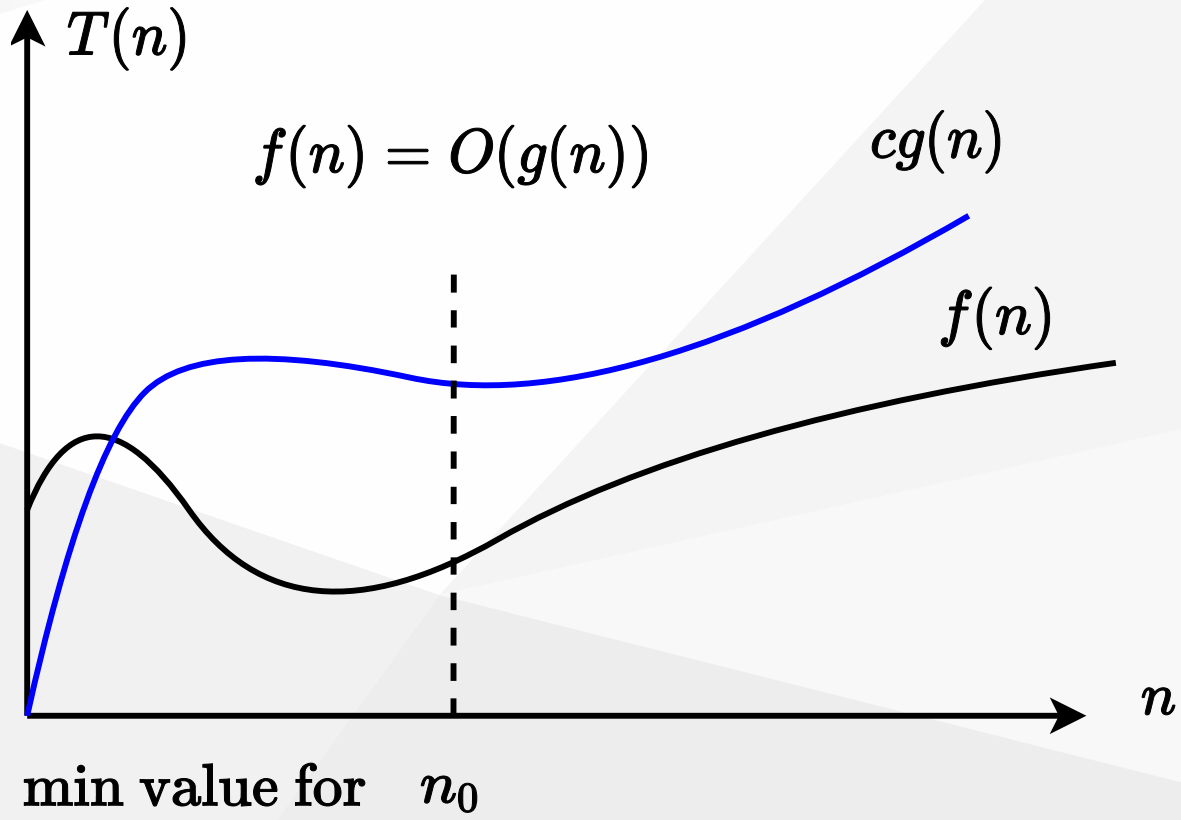
Asymptotic Notations

Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (1)

$f(n) = O(g(n))$ if $\exists$ positive constants c, n_0 such that

$$0 \leq f(n) \leq cg(n), \forall n \geq n_0$$

Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (2)



Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (3)

Asymptotic running times of algorithms are usually defined by functions whose domain are $N = 0, 1, 2, \ldots$ (natural numbers)

Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (4)

Example-1

Show that $2n^2 = O(n^3)$

we need to find two positive constant c and n_0 such that:

$$0 \leq 2n^2 \leq cn^3 \text{ for all } n \geq n_0$$

Choose $c = 2$ and $n_0 = 1$

$$2n^2 \leq 2n^3 \text{ for all } n \geq 1$$

Or, choose $c = 1$ and $n_0 = 2$

$$2n^2 \leq n^3 \text{ for all } n \geq 2$$

Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (5)

Example-2

Show that $2n^2 + n = O(n^2)$

We need to find two positive constant c and n_0 such that:

$$0 \leq 2n^2 + n \leq cn^2 \text{ for all } n \geq n_0$$

$$2 + (1/n) \leq c \text{ for all } n \geq n_0$$

Choose $c = 3$ and $n_0 = 1$

$$2n^2 + n \leq 3n^2 \text{ for all } n \geq 1$$

Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (6)

We can say the followings about $f(n) = O(g(n))$ equation

The notation is a little sloppy

One-way equation, e.q. $n^2 = O(n^3)$ but we cannot say $O(n^3) = n^2$

Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (7)

$O(g(n))$ is in fact a set of functions as follow

$$O(g(n)) = \{f(n) : \exists \text{ positive constant } c, n_0 \text{ such that } 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}$$

Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (8)

In other words $O(g(n))$ is in fact, the set of functions that have asymptotic upper bound $g(n)$

e.q $2n^2 = O(n^3)$ means $2n^2 \in O(n^3)$

Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (9)

Example-1

$10^9 n^2 = O(n^2)$

$0 \leq 10^9 n^2 \leq cn^2 \text{ for } n \geq n_0$

choose $c = 10^9$ and $n_0 = 1$

$0 \leq 10^9 n^2 \leq 10^9 n^2 \text{ for } n \geq 1$

CORRECT

Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (10)

Example-2

$100n^{1.9999} = O(n^2)$

$0 \leq 100n^{1.9999} \leq cn^2 \text{ for } n \geq n_0$

choose $c = 100$ and $n_0 = 1$

$0 \leq 100n^{1.9999} \leq 100n^2 \text{ for } n \geq 1$

CORRECT

Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (11)

Example-3

$10^{-9}n^{2.0001} = O(n^2)$

$0 \leq 10^{-9}n^{2.0001} \leq cn^2 \text{ for } n \geq n_0$

$10^{-9}n^{0.0001} \leq c \text{ for } n \geq n_0$

INCORRECT (Contradiction)

Big-O / O- Notation : Asymptotic Upper Bound (Worst-Case) (12)

If we analysis $O(n^2)$ case, O-notation is an upper bound notation and the runtime $T(n)$ of algorithm A is **at least** $O(n^2)$.

$O(n^2)$: The set of functions with asymptotic **upper bound** n^2

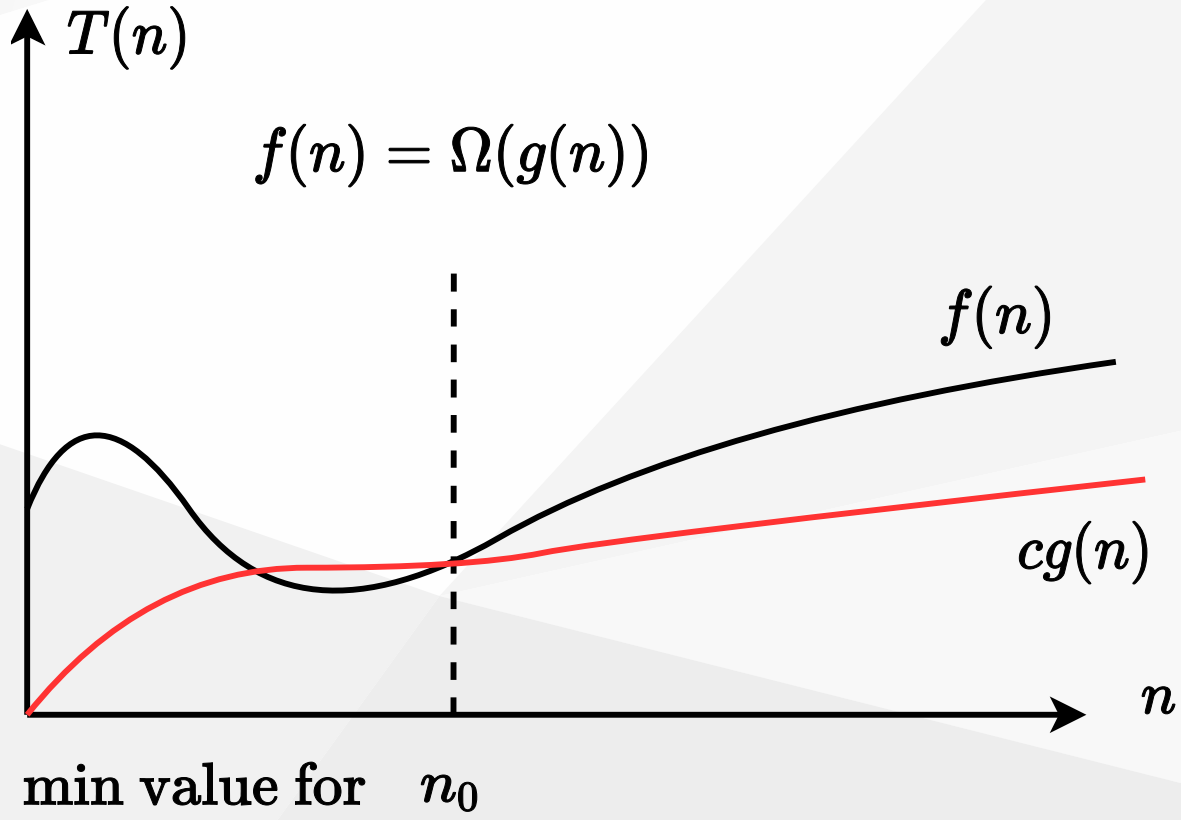
$T(n) \geq O(n^2)$ means $T(n) \geq h(n)$ for some $h(n) \in O(n^2)$

$h(n) = 0$ function is also in $O(n^2)$. Hence : $T(n) \geq 0$, runtime must be nonnegative.

Big-Omega / Ω-Notation : Asymptotic Lower Bound (Best-Case) (1)

$f(n) = \Omega(g(n))$ if $\exists$ positive constants c, n_0 such that $0 \leq cg(n) \leq f(n), \forall n \geq n_0$

Big-Omega / Ω-Notation : Asymptotic Lower Bound (Best-Case) (2)



Big-Omega / Ω-Notation : Asymptotic Lower Bound (Best-Case) (3)

Example-1

Show that $2n^3 = \Omega(n^2)$

We need to find two positive constants c and n_0 such that:

$$0 \leq cn^2 \leq 2n^3 \text{ for all } n \geq n_0$$

Choose $c = 1$ and $n_0 = 1$

$$n^2 \leq 2n^3 \text{ for all } n \geq 1$$

Big-Omega / Ω-Notation : Asymptotic Lower Bound (Best-Case) (4)

Example-4

Show that $\sqrt{n} = \Omega(lgn)$

We need to find two positive constants c and n_0 such that:

$$clgn \leq \sqrt{n} \text{ for all } n \geq n_0$$

Choose $c = 1$ and $n_0 = 16$

$$lgn \leq \sqrt{n} \text{ for all } n \geq 16$$

Big-Omega / Ω-Notation : Asymptotic Lower Bound (Best-Case) (5)

$\Omega(g(n))$ is the set of functions that have asymptotic lower bound $g(n)$

$$\Omega(g(n)) = \{f(n) : \exists \text{ positive constants } c, n_0 \text{ such that } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$$

Big-Omega / Ω-Notation : Asymptotic Lower Bound (Best-Case) (6)

Example-1

$10^9 n^2 = \Omega(n^2)$

$0 \leq cn^2 \leq 10^9 n^2 \text{ for } n \geq n_0$

Choose $c = 10^9$ and $n_0 = 1$

$0 \leq 10^9 n^2 \leq 10^9 n^2 \text{ for } n \geq 1$

CORRECT

Big-Omega / Ω-Notation : Asymptotic Lower Bound (Best-Case) (7)

Example-2

$100n^{1.9999} = \Omega(n^2)$

$0 \leq cn^2 \leq 100n^{1.9999}$ for $n \geq n_0$

$n^{0.0001} \leq (100/c)$ for $n \geq n_0$

INCORRECT(Contradiction)

Big-Omega / Ω-Notation : Asymptotic Lower Bound (Best-Case) (8)

Example-3

$10^{-9}n^{2.0001} = \Omega(n^2)$

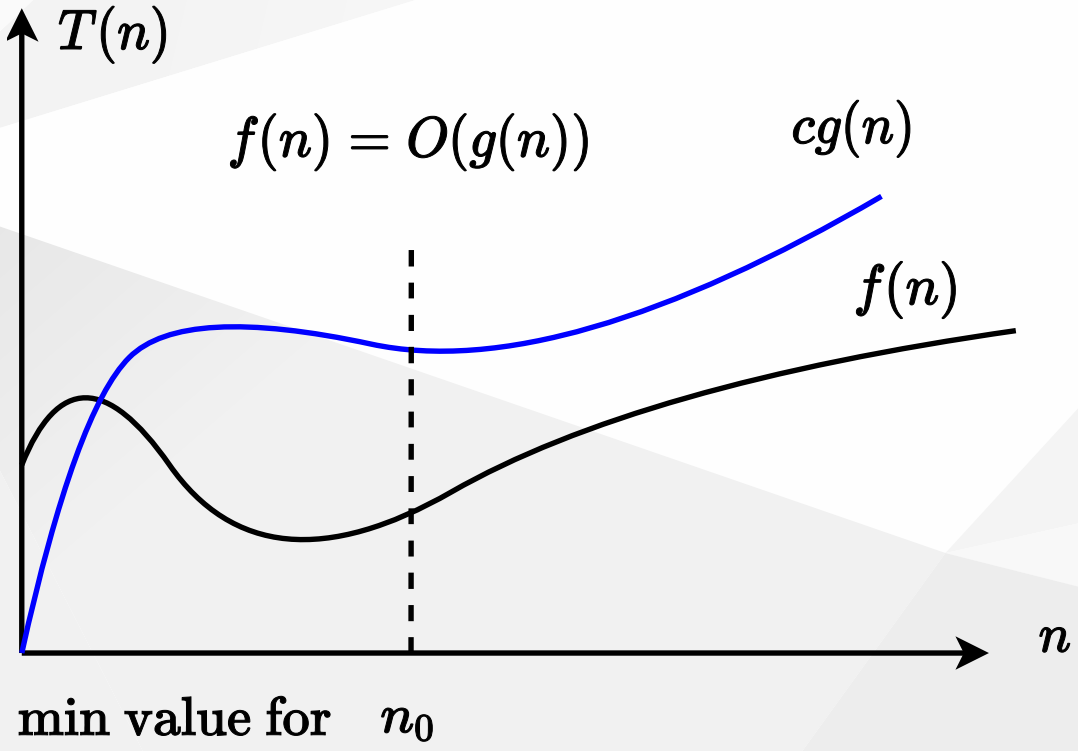
$0 \leq cn^2 \leq 10^{-9}n^{2.0001}$ for $n \geq n_0$

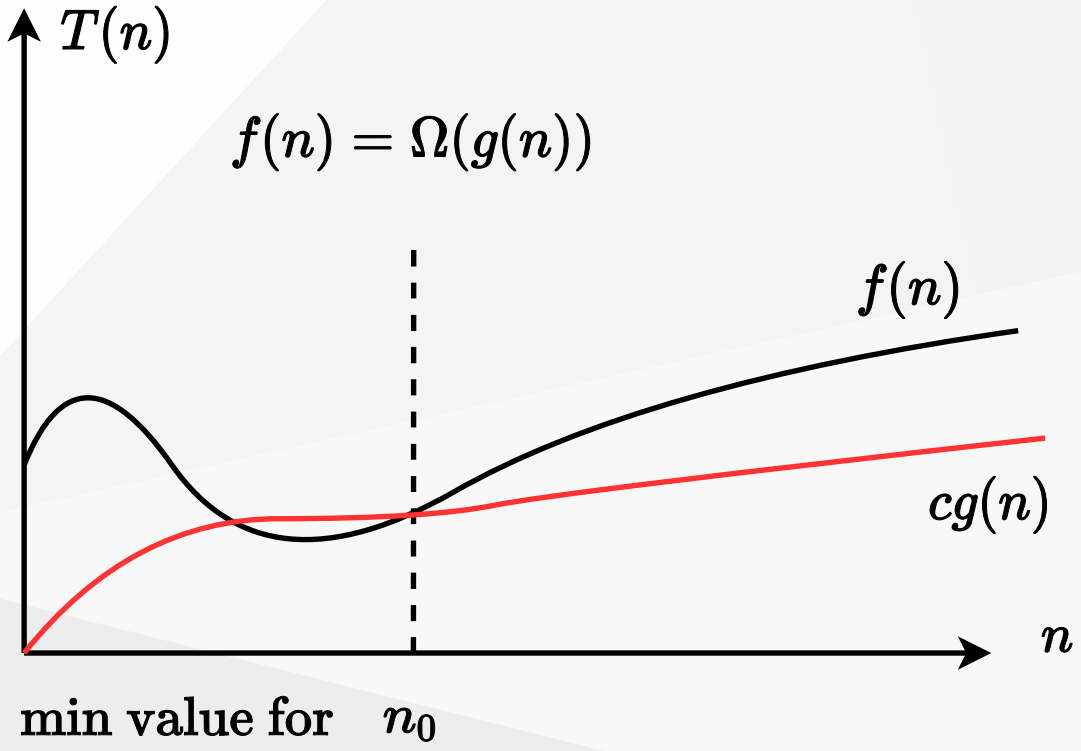
Choose $c = 10^{-9}$ and $n_0 = 1$

$0 \leq 10^{-9}n^2 \leq 10^{-9}n^{2.0001}$ for $n \geq 1$

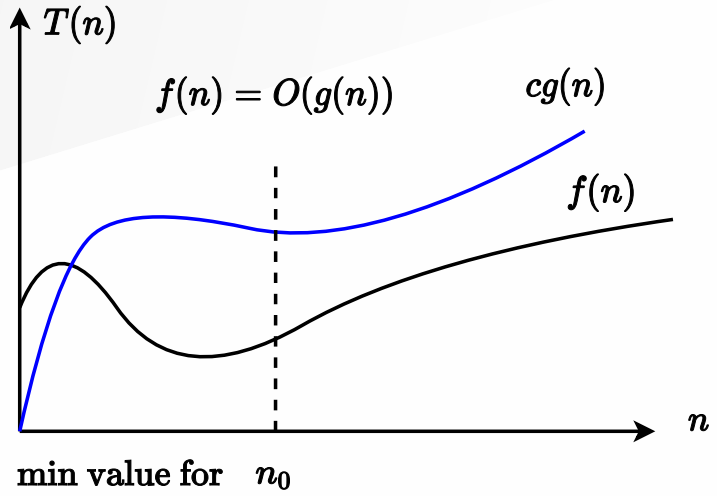
CORRECT

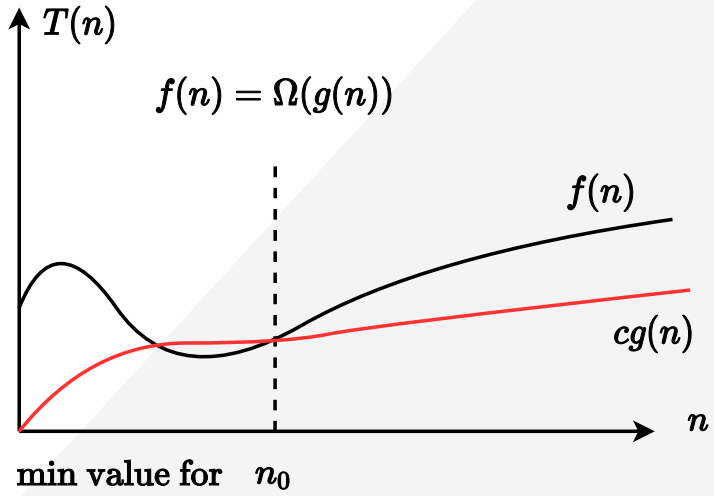
Comparison of Notations (1)

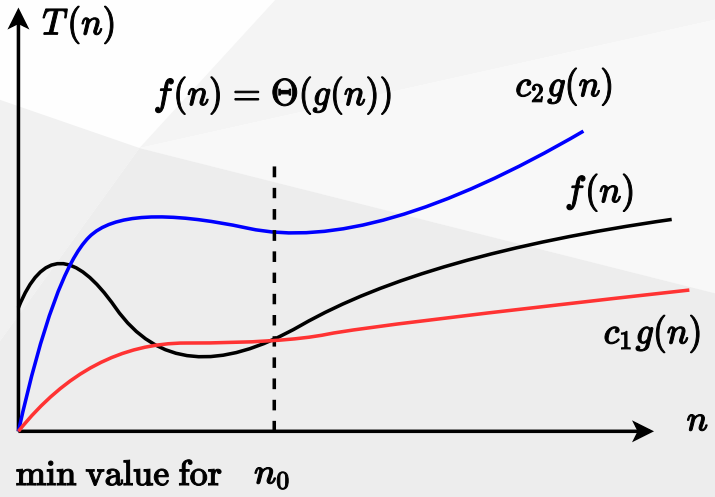




Comparison of Notations (2)





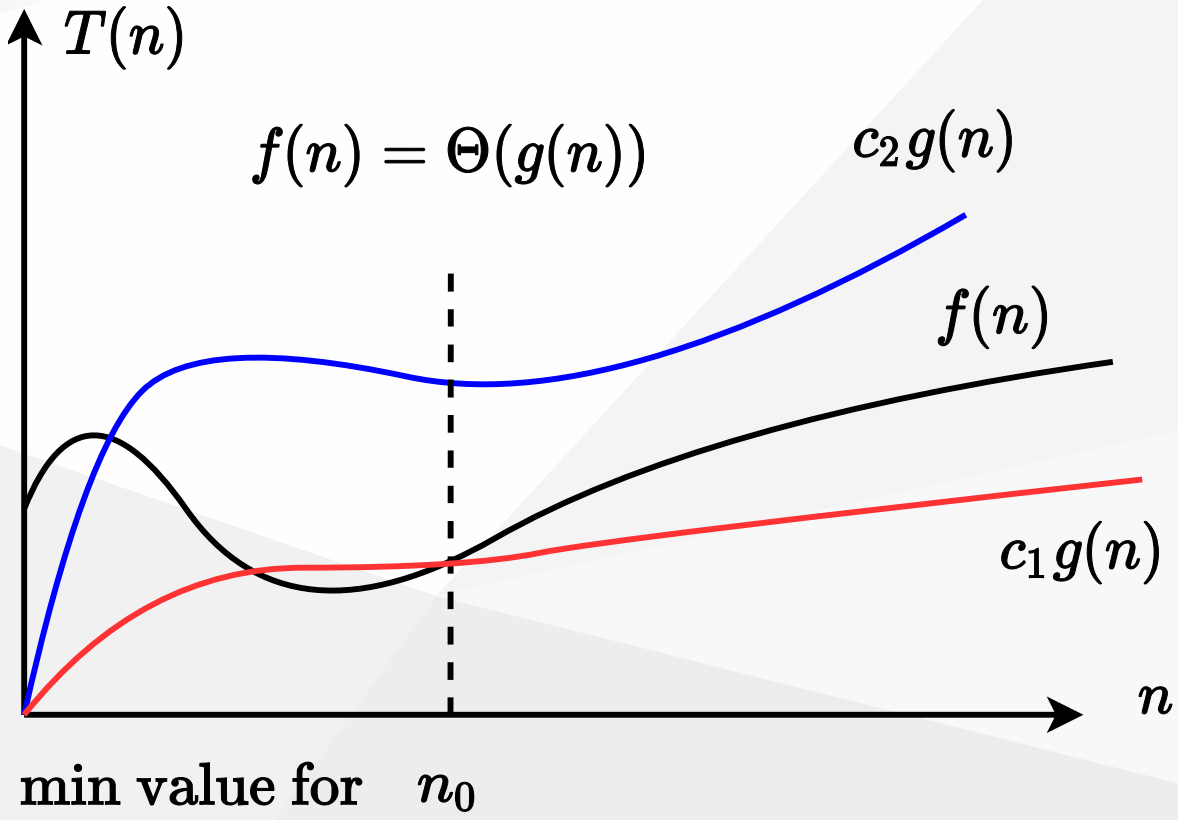


Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (1)

$$f(n) = \Theta(g(n)) \; if \; \exists \text{ positive constants } c_1, c_2, n_0 \text{such that}$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0$$

Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (2)



Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (3)

Example-1

Show that $2n^2 + n = \Theta(n^2)$

We need to find 3 positive constants c_1, c_2 and n_0 such that:

$0 \leq c_1 n^2 \leq 2n^2 + n \leq c_2 n^2$ for all $n \geq n_0$

$c_1 \leq 2 + (1/n) \leq c_2$ for all $n \geq n_0$

Choose $c_1 = 2, c_2 = 3$ and $n_0 = 1$

$2n^2 \leq 2n^2 + n \leq 3n^2$ for all $n \geq 1$

Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (4)

Example-2.1

Show that $1/2n^2 - 2n = \Theta(n^2)$

We need to find 3 positive constants c_1, c_2 and n_0 such that:

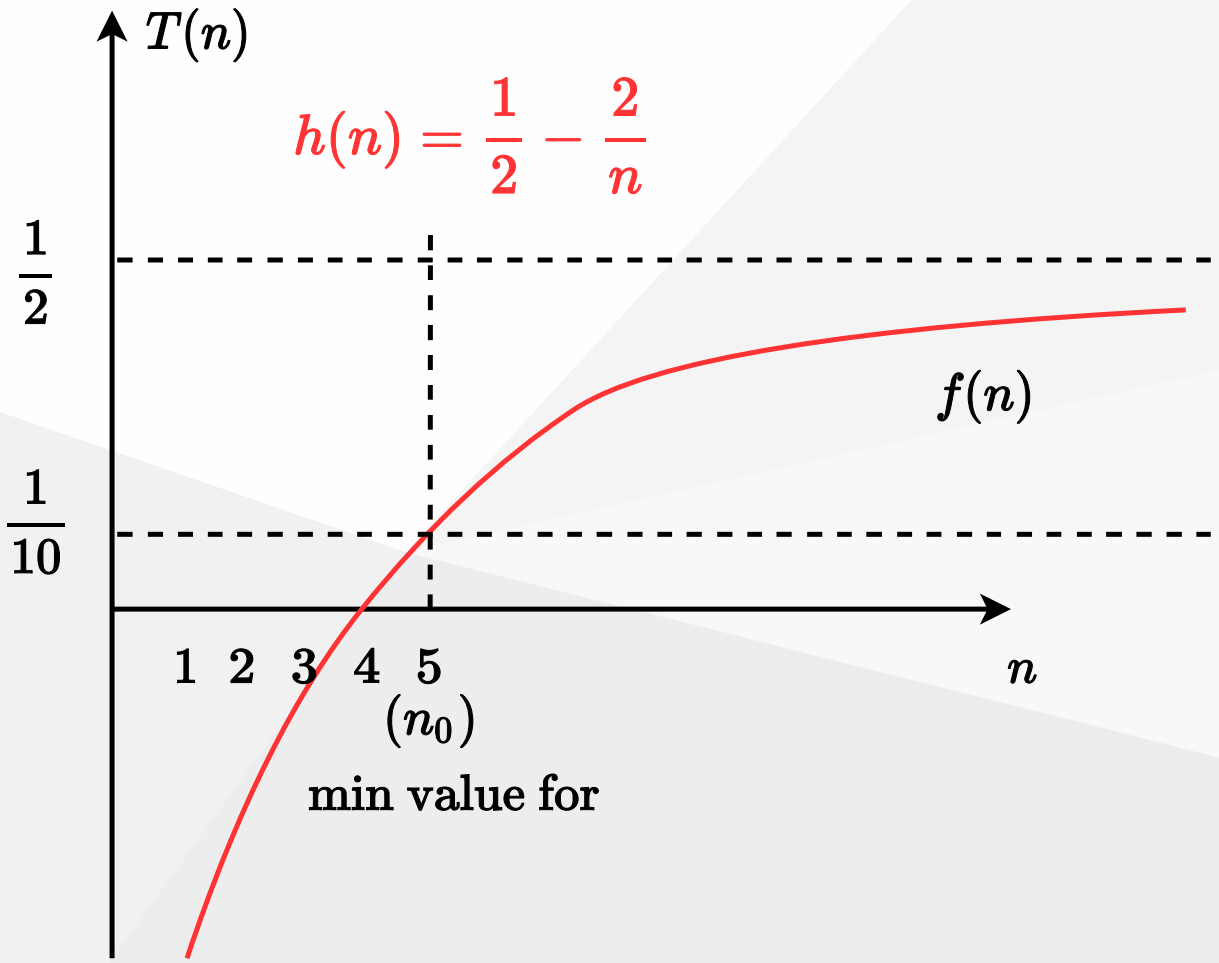
$0 \leq c_1 n^2 \leq 1/2n^2 - 2n \leq c_2 n^2$ for all $n \geq n_0$

$c_1 \leq 1/2 - 2/n \leq c_2$ for all $n \geq n_0$

Choose 3 positive constants c_1, c_2, n_0 that satisfy $c_1 \leq 1/2 - 2/n \leq c_2$ for all $n \geq n_0$

Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (5)

Example-2.2



Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (6)

Example-2.3

$$1/10 \leq 1/2 - 2/n \text{ for } n \geq 5$$

$$1/2 - 2/n \leq 1/2 \text{ for } n \geq 0$$

Therefore we can choose $c_1 = 1/10, c_2 = 1/2, n_0 = 5$

Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (7)

Theorem: leading constants & low-order terms don't matter

Justification: can choose the leading constant large enough to make high-order term dominate other terms

Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (8)

Example-1

$10^9 n^2 = \Theta(n^2)$ **CORRECT**

$100 n^{1.9999} = \Theta(n^2)$ **INCORRECT**

$10^9 n^{2.0001} = \Theta(n^2)$ **INCORRECT**

Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (9)

$\Theta(g(n))$ is the set of functions that have asymptotically tight bound $g(n)$

$\Theta(g(n)) = \{f(n) : \exists$
positive constants c_1, c_2, n_0 such that
$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$

Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (10)

Theorem:

$f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Θ is stronger than both O and Ω

$\Theta(g(n)) \subseteq O(g(n))$ and $\Theta(g(n)) \subseteq \Omega(g(n))$

Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (11)

Example-1.1

Prove that $10^{-8}n^2 \neq \Theta(n)$

We can check that $10^{-8}n^2 = \Omega(n)$ and $10^{-8}n^2 \neq O(n)$

Proof by contradiction for $O(n)$ notation

$$O(g(n)) = \{f(n) : \exists \text{ positive constant } c, n_0 \text{ such that } 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}$$

Big-Theta /Θ-Notation : Asymptotically tight bound (Average Case) (12)

Example-1.2

Suppose positive constants c_2 and n_0 exist such that:

$$10^{-8}n^2 \leq c_2 n, \forall n \geq n_0$$

$$10^{-8}n \leq c_2, \forall n \geq n_0$$

Contradiction: c_2 is a constant

Summary of O, Ω and Θ notations (1)

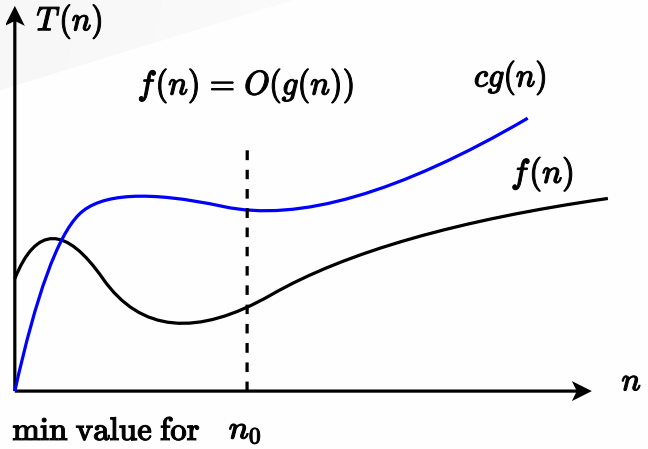
$O(g(n))$: The set of functions with asymptotic upper bound $g(n)$

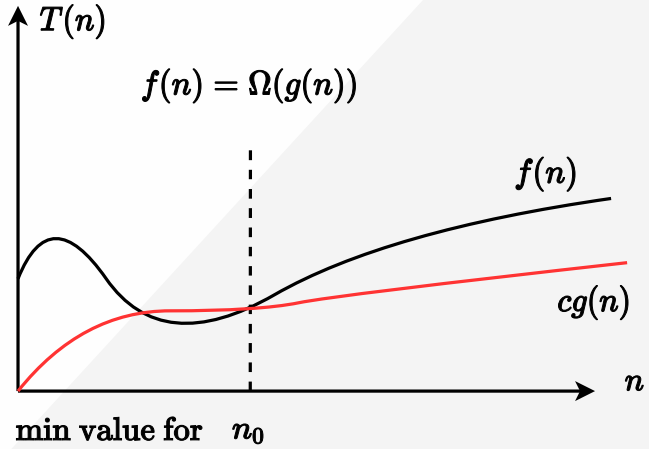
$\Omega(g(n))$: The set of functions with asymptotic lower bound $g(n)$

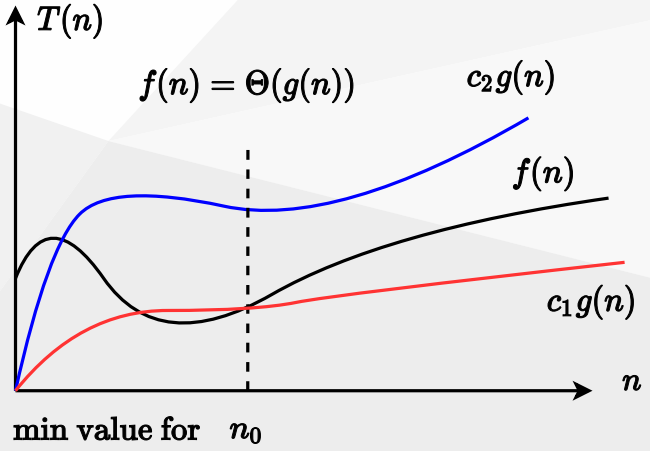
$\Theta(n)$: The set of functions with asymptotically tight bound $g(n)$

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

Summary of O, Ω and Θ notations (2)







Small-o / o-Notation : Asymptotic upper bound that is not tight (1)

Remember, upper bound provided by big- O notation can be tight or not tight

Tight mean values are close the original function

e.g. followings are true

$2n^2 = O(n^2)$ is asymptotically tight

$2n = O(n^2)$ is not asymptotically tight

According to this small-o notation is an upper bound that is not asymptotically tight

Small-o / o-Notation : Asymptotic upper bound that is not tight (2)

Note that in equations equality is removed in small notations

$$o(g(n)) = \{f(n) : \text{ for any constant} c > 0, \exists \text{ a constant } n_0 > 0, \text{ such that } 0 \leq f(n) < cg(n), \forall n \geq n_0\}$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

e.g $2n = o(n^2)$ any positive c satisfies but $2n^2 \neq o(n^2)$ $c = 2$ does not satisfy

Small-omega / ω-Notation: Asymptotic lower bound that is not tight (1)

$$\omega(g(n)) = \{f(n) : \text{ for any constant } c > 0, \exists \text{ a constant } n_0 > 0,$$
$$\text{such that } 0 \leq cg(n) < f(n),$$
$$\forall n \geq n_0$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

e.g. $n^2/2 = \omega(n)$, any positive c satisfies but $n^2/2 \neq \omega(n^2)$, $c = 1/2$ does not satisfy

(Important) Analogy to compare of two real numbers (1)

$$f(n) = O(g(n)) \leftrightarrow a \leq b$$
$$f(n) = \Omega(g(n)) \leftrightarrow a \geq b$$
$$f(n) = \Theta(g(n)) \leftrightarrow a = b$$
$$f(n) = o(g(n)) \leftrightarrow a < b$$
$$f(n) = \omega(g(n)) \leftrightarrow a > b$$

(Important) Analogy to compare of two real numbers (2)

$$O \approx \leq$$

$$\Theta \approx =$$

$$\Omega \approx \geq$$

$$\omega \approx >$$

$$o \approx <$$

(Important) Trichotomy property for real numbers

For any two real numbers a and b, we have either

$a < b$, or $a = b$, or $a > b$

Trichotomy property does not hold for asymptotic notation, for two functions $f(n)$ and $g(n)$, it may be the case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds.

e.g. n and $n^{1+sin(n)}$ cannot be compared asymptotically

Examples

$5n^2 = O(n^2)$	**TRUE**	$n^2lgn = O(n^2)$	**FALSE**
$5n^2 = \Omega(n^2)$	**TRUE**	$n^2lgn = \Omega(n^2)$	TRUE
$5n^2 = \Theta(n^2)$	**TRUE**	$n^2lgn = \Theta(n^2)$	FALSE
$5n^2 = o(n^2)$	FALSE	$n^2lgn = o(n^2)$	FALSE
$5n^2 = \omega(n^2)$	FALSE	$n^2lgn = \omega(n^2)$	TRUE
$2^n = O(3^n)$	**TRUE**		
$2^n = \Omega(3^n)$	FALSE	$2^n = o(3^n)$	TRUE
$2^n = \Theta(3^n)$	FALSE	$2^n = \omega(3^n)$	FALSE

Asymptotic Function Properties

Transitivity: holds for all

e.g. $f(n) = \Theta(g(n)) \& g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

Reflexivity: holds for Θ, O, Ω

e.g. $f(n) = O(f(n))$

Symmetry: hold only for Θ

e.g. $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

Transpose Symmetry: holds for $(O \leftrightarrow \Omega)$ and $(o \leftrightarrow \omega)$

e.g. $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

Using O-Notation to Describe Running Times (1)

Used to bound worst-case running times, Implies an upper bound runtime for arbitrary inputs as well

Example:

Insertion sort has worst-case runtime of $O(n^2)$

Note:

- This $O(n^2)$ upper bound also applies to its running time on every input
  - Abuse to say “running time of insertion sort is $O(n^2)$"
- For a given n, the actual running time depends on the particular input of size n
  - i.e., running time is not only a function of n
- However, **worst-case** running time is only a function of n

Using O-Notation to Describe Running Times (2)

- When we say:
  - Running time of insertion sort is $O(n^2)$
- What we really mean is
  - Worst-case running time of insertion sort is $O(n^2)$
- or equivalently
  - No matter what particular input of size n is chosen, the running time on that set of inputs is $O(n^2)$

Using Ω-Notation to Describe Running Times (1)

Used to bound best-case running times, Implies a lower bound runtime for arbitrary inputs as well

Example:

Insertion sort has best-case runtime of $\Omega(n)$

Note:

- This $\Omega(n)$ lower bound also applies to its running time on every input

Using Ω-Notation to Describe Running Times (2)

- When we say
  - Running time of algorithm A is $\Omega(g(n))$
- What we mean is
  - For any input of size n, the runtime of A is *at least* a constant times $g(n)$ for sufficiently large n
- It's not contradictory to say
  - **worst-case** running time of insertion sort is $\Omega(n^2)$
  - Because there exists an input that causes the algorithm to take $\Omega(n^2)$

Using Θ-Notation to Describe Running Times (1)

Consider 2 cases about the runtime of an algorithm

- **Case 1:** Worst-case and best-case not asymptotically equal
  - Use Θ-notation to bound worst-case and best-case runtimes separately
- **Case 2:** Worst-case and best-case asymptotically equal
  - Use Θ-notation to bound the runtime for any input

Using Θ-Notation to Describe Running Times (2)

- **Case 1:** Worst-case and best-case not asymptotically equal
  - Use Θ-notation to bound the worst-case and best-case runtimes separately
  - We can say:
    - "The worst-case runtime of insertion sort is $\Theta(n^2)$"
    - "The best-case runtime of insertion sort is $\Theta(n)$"
  - But, we can't say:
    - "The runtime of insertion sort is $\Theta(n^2)$ for every input"
  - A Θ-bound on worst/best-case running time does not apply to its running time on arbitrary inputs

Worst-Case and Best-Case Equation for Merge-Sort

e.g. for merge-sort, we have:

$$T(n) = \Theta(nlgn) \begin{cases} T(n) = O(nlgn) \\ T(n) = \Omega(nlgn) \end{cases}$$

Using Asymptotic Notation to Describe Runtimes Summary (1)

- "The worst case runtime of Insertion Sort is $O(n^2)$"
  - Also implies: "The runtime of Insertion Sort is $O(n^2)$"
- "The best-case runtime of Insertion Sort is $\Omega(n)$"
  - Also implies: "The runtime of Insertion Sort is $\Omega(n)$"

Using Asymptotic Notation to Describe Runtimes Summary (2)

- "The worst case runtime of Insertion Sort is $\Theta(n^2)$"
  - But: "The runtime of Insertion Sort is not $\Theta(n^2)$"
- "The best case runtime of Insertion Sort is $\Theta(n)$"
  - But: "The runtime of Insertion Sort is not $\Theta(n)$"

Using Asymptotic Notation to Describe Runtimes Summary (3)

Which one is true?

- **FALSE** "The worst case runtime of Merge Sort is $\Theta(nlgn)$"
- **FALSE** "The best case runtime of Merge Sort is $\Theta(nlgn)$"
- **TRUE** "The runtime of Merge Sort is $\Theta(nlgn)$"
    - This is true, because the best and worst case runtimes have asymptotically the same tight bound $\Theta(nlgn)$

Asymptotic Notation in Equations (RHS)

- Asymptotic notation appears alone on the **RHS** of an equation:
  - implies set membership
    - e.g., $n = O(n^2)$ means $n \in O(n^2)$

Asymptotic notation appears on the **RHS** of an equation
stands for some anonymous function in the set

- e.g., $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means:
- $2n^2 + 3n + 1 = 2n^2 + h(n)$, for some $h(n) \in \Theta(n)$
  - i.e., $h(n) = 3n + 1$

Asymptotic Notation in Equations (LHS)

- Asymptotic notation appears on the **LHS** of an equation:
  - stands for any anonymous function in the set
    - e.g., $2n^2 + \Theta(n) = \Theta(n^2)$ means:
  - for any function $g(n) \in \Theta(n)$
  - $\exists$ some function $h(n) \in \Theta(n^2)$
    - such that $2n^2 + g(n) = h(n)$
- **RHS** provides coarser level of detail than **LHS**

References

- Introduction to Algorithms, Third Edition | The MIT Press
- Bilkent CS473 Course Notes (new)
- Bilkent CS473 Course Notes (old)
- Insertion Sort - GeeksforGeeks
- NIST Dictionary of Algorithms and Data Structures
- NIST - Dictionary of Algorithms and Data Structures
- NIST - big-O notation
- NIST - big-Omega notation

$$-End-Of-Week-1-Course-Module-$$