Algorithm Analysis Part 2

261217 Data Structures for Computer Engineers

Patiwet Wuttisarnwattana, Ph.D.

patiwet@eng.cmu.ac.th

Computer Engineering, Chiang Mai University

Algorithms

- A computer program should be totally correct, but it should also
  - execute as quickly as possible (time-efficiency)
  - use memory wisely (storage-efficiency)
- How do we compare programs (or algorithms in general) with respect to execution time?
  - various computer run at different speeds due to different processors
  - compilers optimize code before execution
  - the same algorithm can be written differently depending on the programming paradigm
  - Big-O (or order of the algorithm) is a good way to compare a program.

Analyzing Algorithms

- Worst Case
  - Case with maximum number of operations (slowest)
- Best Case
  - Case with minimum number of operations (fastest)
- Average Case
  - Average number of operations over all cases

Number of operations?

Search an array for a value

```
public static int search(int []x, int target){
    for 1(int i=0; 2 i< x.length; 3 i++){
        if 4 (x[i] == target)
            5 return i;
    }
    6 return -1;
}
```

Counting Operations

- Let the x.length = n
- How many times is operation 1 executed?
- How many times is operation 5 and 6 executed in total?
- Total so far

Counting Operations

- Let the x.length = n
- How many times is operation 1 executed? **1**
- How many times is operation 5 and 6 executed in total? **1**
- Total so far **2**

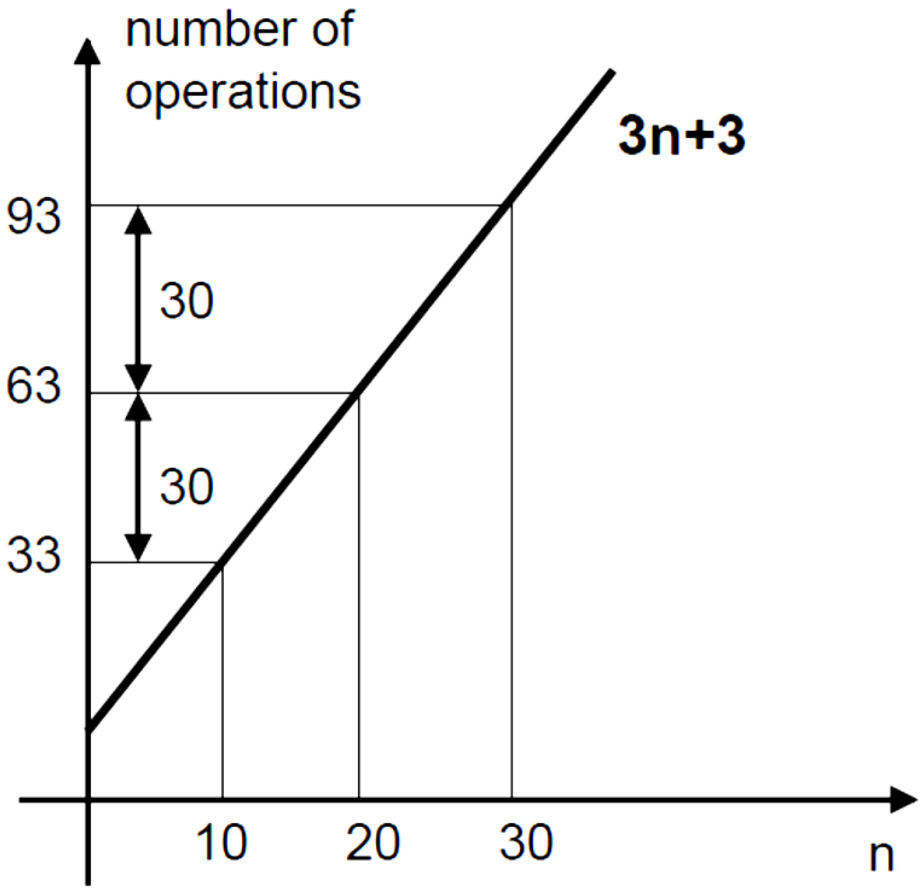
Worse Case

- Worst Case: The target is not in the array
- How many times is operation 2 executed?
- How many times is operation 3 executed?
- How many times is operation 4 executed?
- Total number of operations:

Worse Case

- Worst Case: The target is not in the array
- How many times is operation 2 executed? **n+1**
- How many times is operation 3 executed? **n**
- How many times is operation 4 executed? **n**
- Total number of operations: **3n+3**

Worst Case



n (size of array)	number of operations
10	33
20	63
30	93

Another look

What if we count just the comparison

```
public static int search(int []x, int target){
    for (int i=0; i< x.length; i++){
        if (x[i] == target)
            return i;
    }
    return -1;
}
```

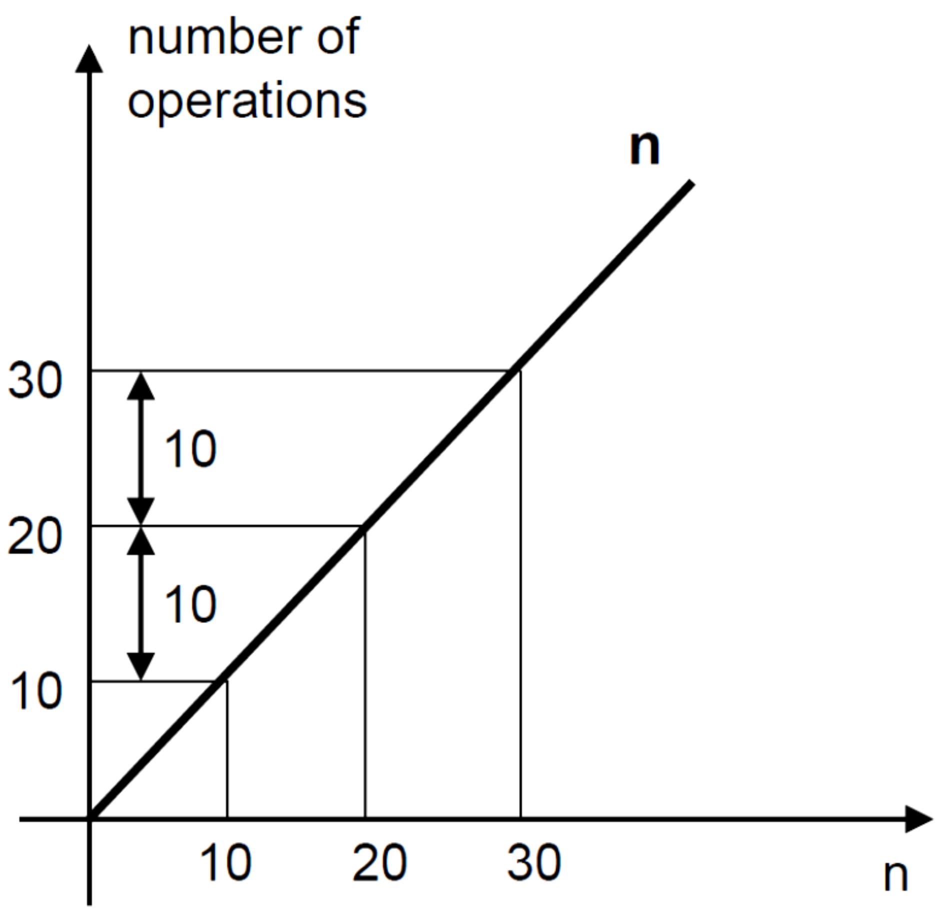
Worst Case

- Worst Case: the target is not in the array
- How many times is the comparison executed?

Worst Case

- Worst Case: the target is not in the array
- How many times is the comparison executed? **n**

Number of operations as a function of ***n***



n (size of array)	number of operations
10	10
20	20
30	30

Linear Algorithm

- In both cases, the amount of work we do is linearly proportional to the number of data values in the array
- If we have n data values in the array, and we double the size of the array, how much work will we do searching the new array in the worst case?

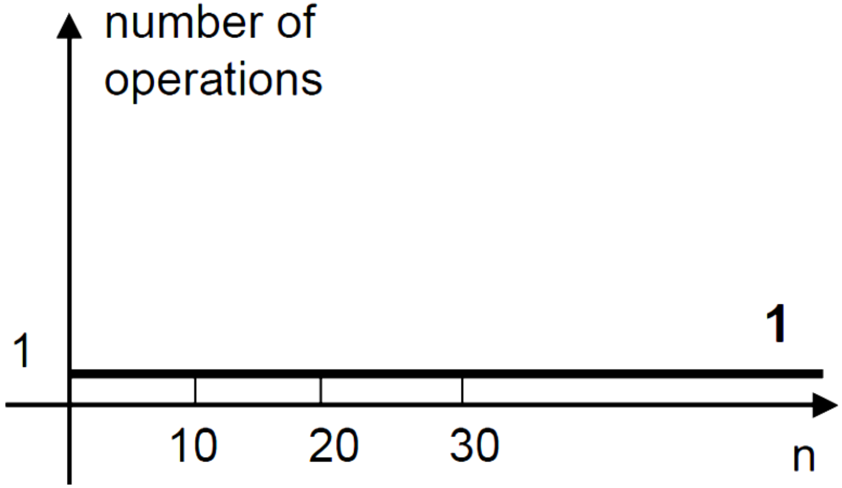
Best Case

- How many comparisons are necessary in the best case for an array of n values?

```java
public static int search(int []x, int target){
    for (int i=0; i< x.length; i++){
        if (x[i] == target)
            return i;
    }
    return -1;
}
```

Best Case

- How many comparisons are necessary in the best case for an array of n values?



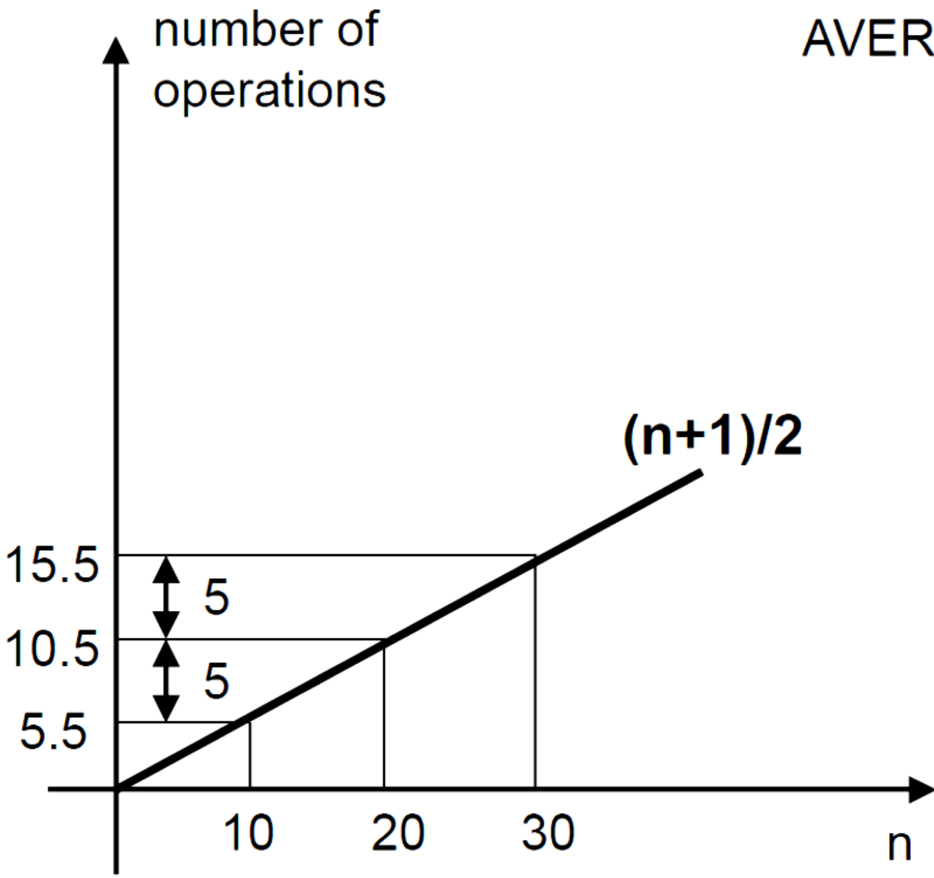
n (size of array)	number of operations
10	1
20	1
30	1

Average Case

- How many comparisons are necessary in the average case for an array of n values (assuming the target is in the array)?

$$\frac{1 + 2 + 3 + \cdots + (n-1) + n}{n} = \frac{(n+1)}{2}$$

Average Case



AVERAGE CASE IS LINEAR

n (size of array)	number of operations
10	5.5
20	10.5
30	15.5

Example

- Consider the problem of summing

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + ... + n$$

Algorithm A	Algorithm B	Algorithm C

`sum = 0`
`for i = 1 to n`
`    sum = sum + i`
	`sum = 0`
	`for i = 1 to n`
	`{`
	`    for j = 1 to i`
	`        sum = sum + 1`
	`}`
		`sum = n * (n + 1) / 2`

Three algorithms for computing the sum
1 + 2 + . . . + n for an integer n > 0

Example

```
// Computing the sum of the consecutive integers from 1 to n:
long n = 10000; // Ten thousand
```

Java code for the three algorithms

Example

```java
// Computing the sum of the consecutive integers from 1 to n:
long n = 10000; // Ten thousand
// Algorithm A
long sum = 0;
for (long i = 1; i <= n; i++)
   sum = sum + i;
System.out.println(sum);
// Algorithm B
sum = 0;
for (long i = 1; i <= n; i++)
{
   for (long j = 1; j <= i; j++)
       sum = sum + 1;
} // end for
System.out.println(sum);
// Algorithm C
sum = n * (n + 1) / 2;
System.out.println(sum);
```

Java code for the three algorithms

- An algorithm has both time and space constraints – that is complexity
  - Time complexity
  - Space complexity
- This study is called analysis of algorithms

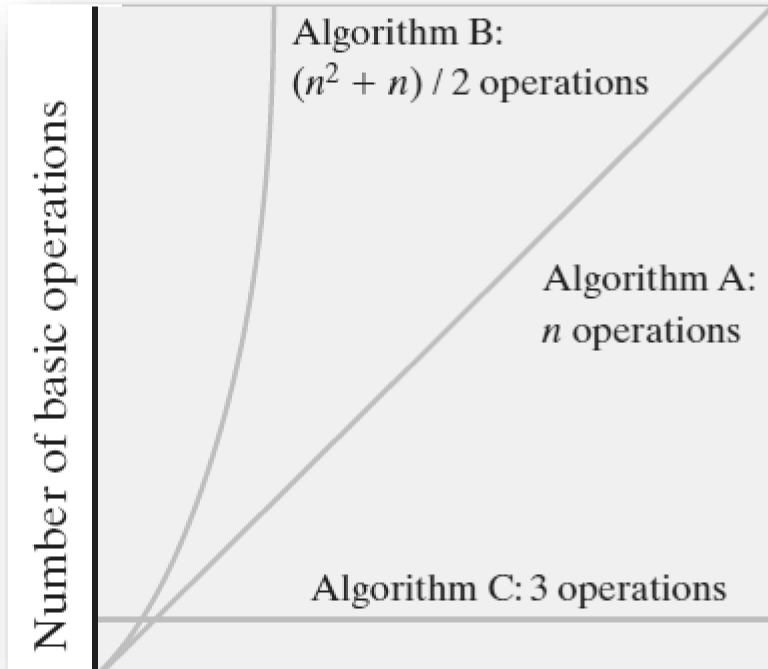
Counting Basic Operations

- A basic operation of an algorithm
  - The most significant contributor to its total time requirement

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n+1)/2$	1
Multiplications			1
Divisions			1
Total basic operations	n	$(n^2+n)/2$	**3**

The number of basic operations required by the algorithms

Counting Basic Operations

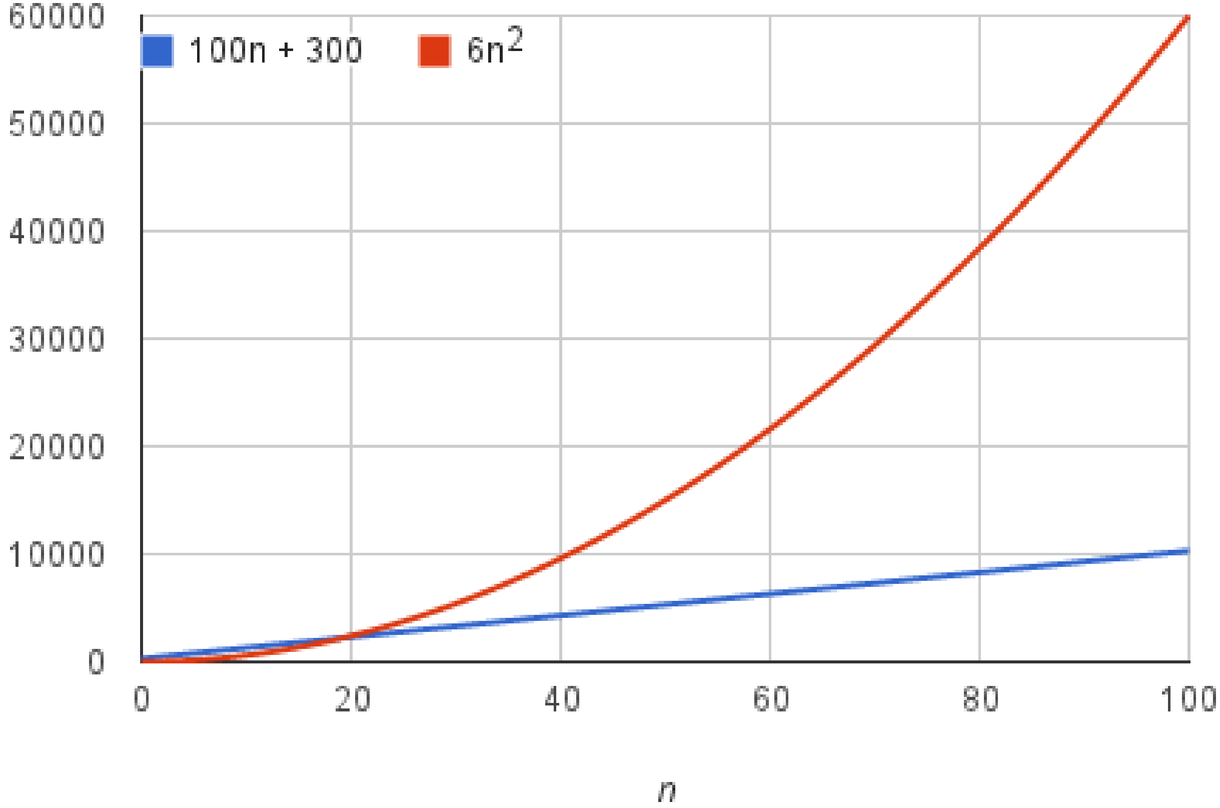


The number of basic operations required by the algorithms

- When n is small, which algorithm is the fastest?
- When n becomes larger and larger, which algorithm is the fastest?

Asymptotic Notation

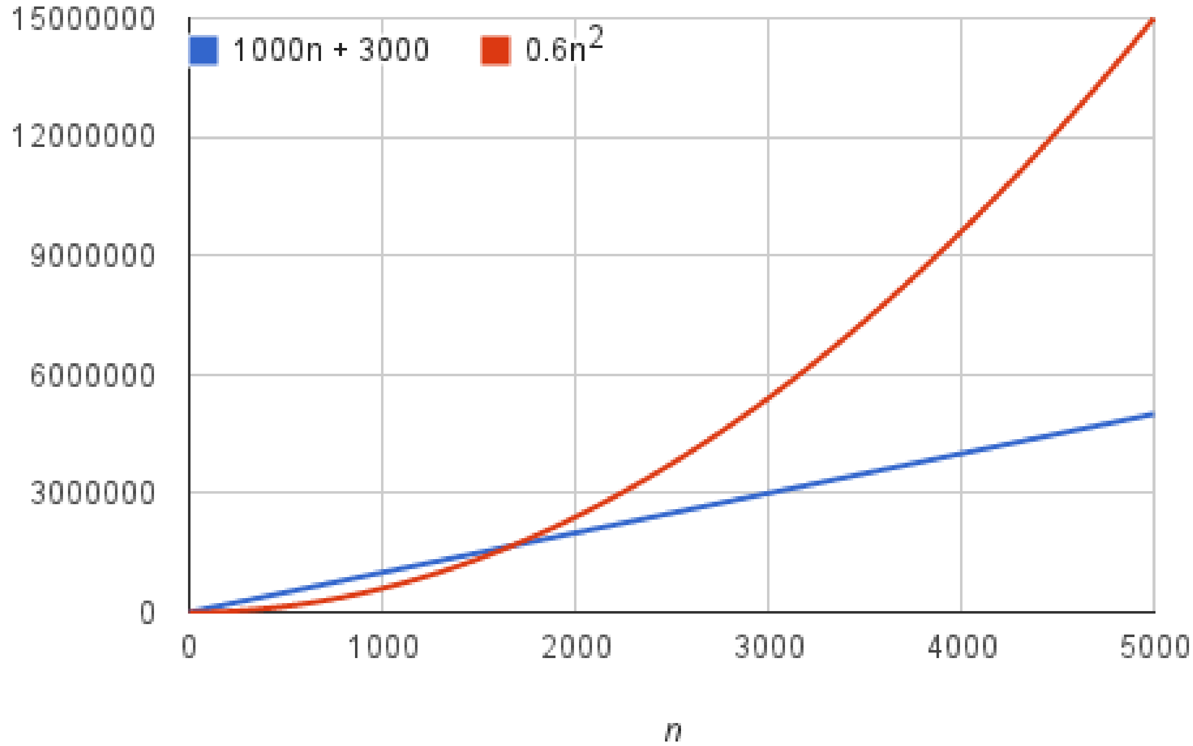
- Suppose there are two algorithms that running time depends on input size **n**.
- Which one is of the following algorithms is preferable?
- Algorithm 1: **100n + 300**
- Algorithm 2: **$6n^2$**



Rate of Growth from different order!

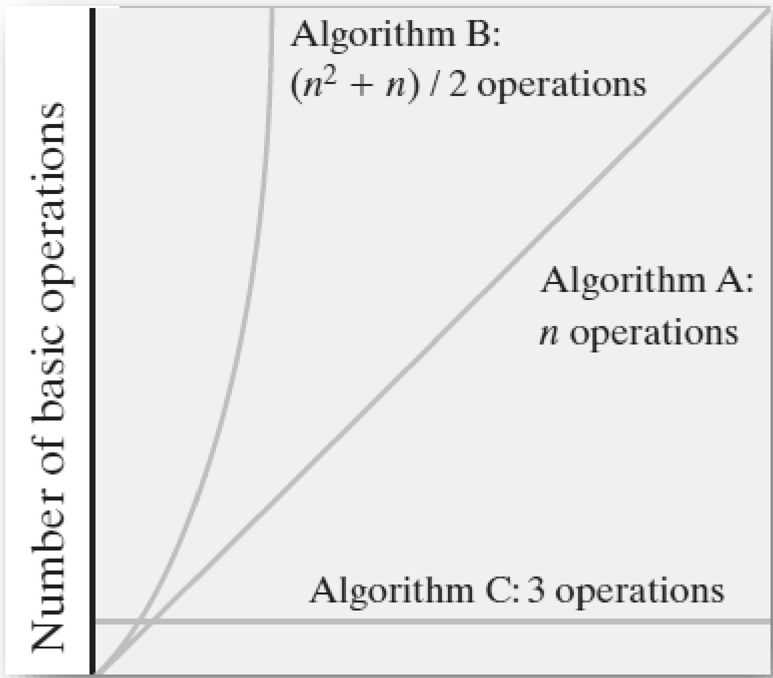
Asymptotic Notation

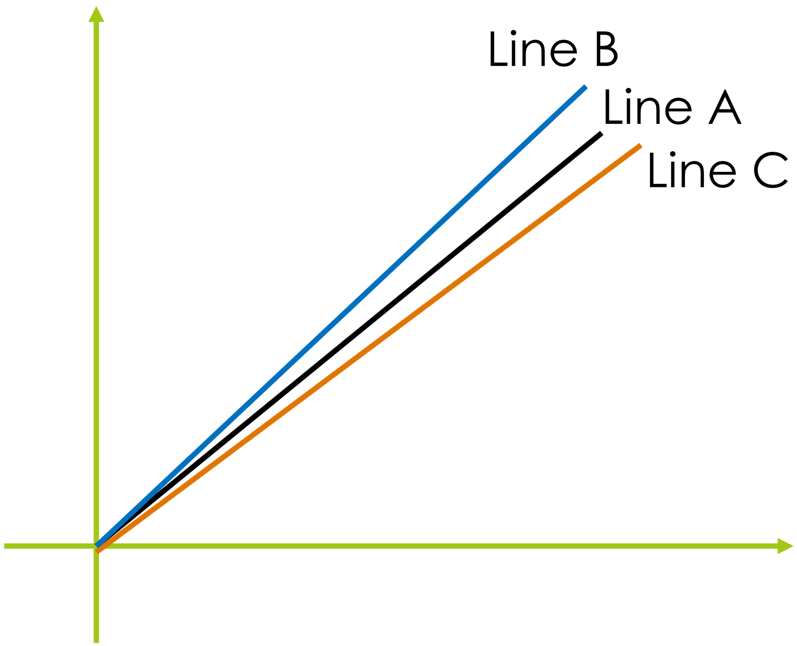
- Suppose there are two algorithms that running time depends on input size **n**.
- Which one is of the following algorithms is preferable?
- Algorithm 1: **1000n + 3000**
- Algorithm 2: **$0.6n^2$**



Rate of Growth from different order!

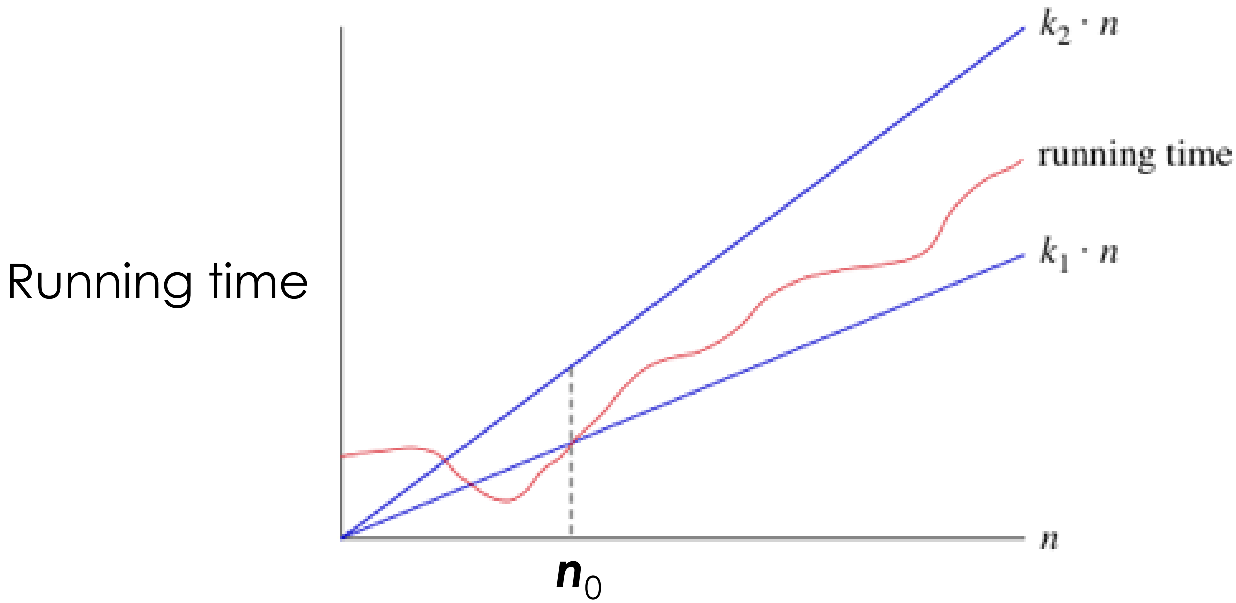
Asymptotic Notation





Line B is the **upper bound** of the Line A and Line C
Line C is the **lower bound** of the Line A and Line B

Running time or Order of the Algorithm



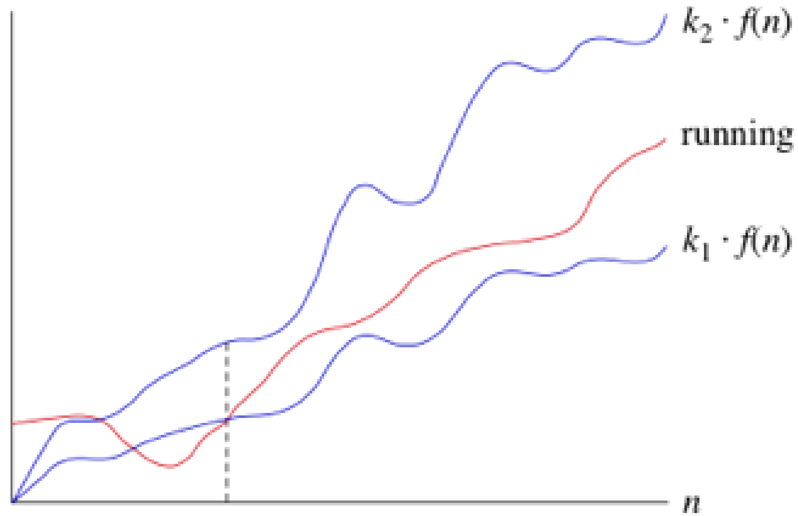
$$g(n) = c_1 n + c_2$$

For $n > n_0$, if there exists k_1 and k_2 such that $k_1 n \leq g(n) \leq k_2 n$

We're saying that once n gets large enough, the running time is at least $k_1 n$ and at most $k_2 n$ OR
We're saying the running time is Θ(n) "Big Theta of **n**"
We're saying the algorithm has "an order of **n**"

Big Theta Θ Notation

Running time



$\boldsymbol{g(n)}$

$\boldsymbol{n}_0$

For $\boldsymbol{n} > \boldsymbol{n}_0$, if there exists $\boldsymbol{k}_1$ and $\boldsymbol{k}_2$ such that $\boldsymbol{k}_1\mathbf{f}(\boldsymbol{n}) \leq \boldsymbol{g}(\boldsymbol{n}) \leq \boldsymbol{k}_2 f(\boldsymbol{n})$

We're saying that once **n** gets large enough, the running time is at least $\boldsymbol{k}_1\mathbf{f}(\boldsymbol{n})$ and at most $\boldsymbol{k}_2 f(\boldsymbol{n})$ OR
We're saying the running time is Θ(f(n)) or "Big Theta of f(n)" or "order of f(n)"

Examples

- What is the Big Theta notation of “3n + 100”
  - $\Theta(n)$
  - There exist k_1 and k_2 such that $k_1 n \le f(n) \le k_2 n$ Give examples?
  - $k_1 = 2$, $k_2 = 4$ such that $2n \le 3n + 100 \le 4n$ for n is a large number ($n \ge 100$)
  - There are infinitely many k_1 and k_2 that satisfy the condition (e.g. $k_2 > 4$)
- What is the Big Theta notation of “$6n^2 + 7n + 30$”
  - $\Theta(n^2)$
  - There exist k_1 and k_2 such that $k_1 n^2 \le f(n) \le k_2 n^2$ Give examples?
  - $k_1 = 5$, $k_2 = 7$ such that $5n^2 \le 6n^2 + 7n + 30 \le 7n^2$ for n is a large number ($n \ge 10$)

Functions in Asymptotic Notation

- What is the Big Theta Notation of an algorithm that has n inputs but always runs in a constant time.
  - For example, the algorithm requires "1000" seconds (or 1000 Bytes, 1000 machine instructions) regardless of the input size
  - Ans: $\Theta(1)$
  - For example, $k_1 = 999$ and $k_2 = 1001$ (any n)
- What is the Big Theta Notation of an algorithm that runs in a logarithmic time.?
  - For example, the algorithm requires "$3 \times \log_2(n) + 4$" seconds
  - Ans: $\Theta(\log_2(n))$ or $\Theta(\log(n))$
  - For example, $k_1 = 2$ and $k_2 = 4$ ($n \geq 2^4$)

Functions in Asymptotic Notation

- The algorithm takes "$3 \times \log_{10}(n) + 4$" seconds. What is the Big Theta Notation?
  - Ans: $\Theta(\log_{10}(n))$ or $\Theta(\log(n))$
  - $\log_a n = \log_b n \times \frac{1}{\log_b a}$
  - For example, $k_1 = 2$ and $k_2 = 4$ ($n \geq 10^4$)
- The algorithm takes "$3n\sqrt{n} + \sqrt{n} + 2n^2\sqrt{n} + n + 10$" seconds. What is the Big Theta Notation?
  - Ans: $\Theta(n^{5/2})$
  - For example, $k_1 = 1$ and $k_2 = 3$ ($n \geq 9$?)

Functions in Asymptotic Notation

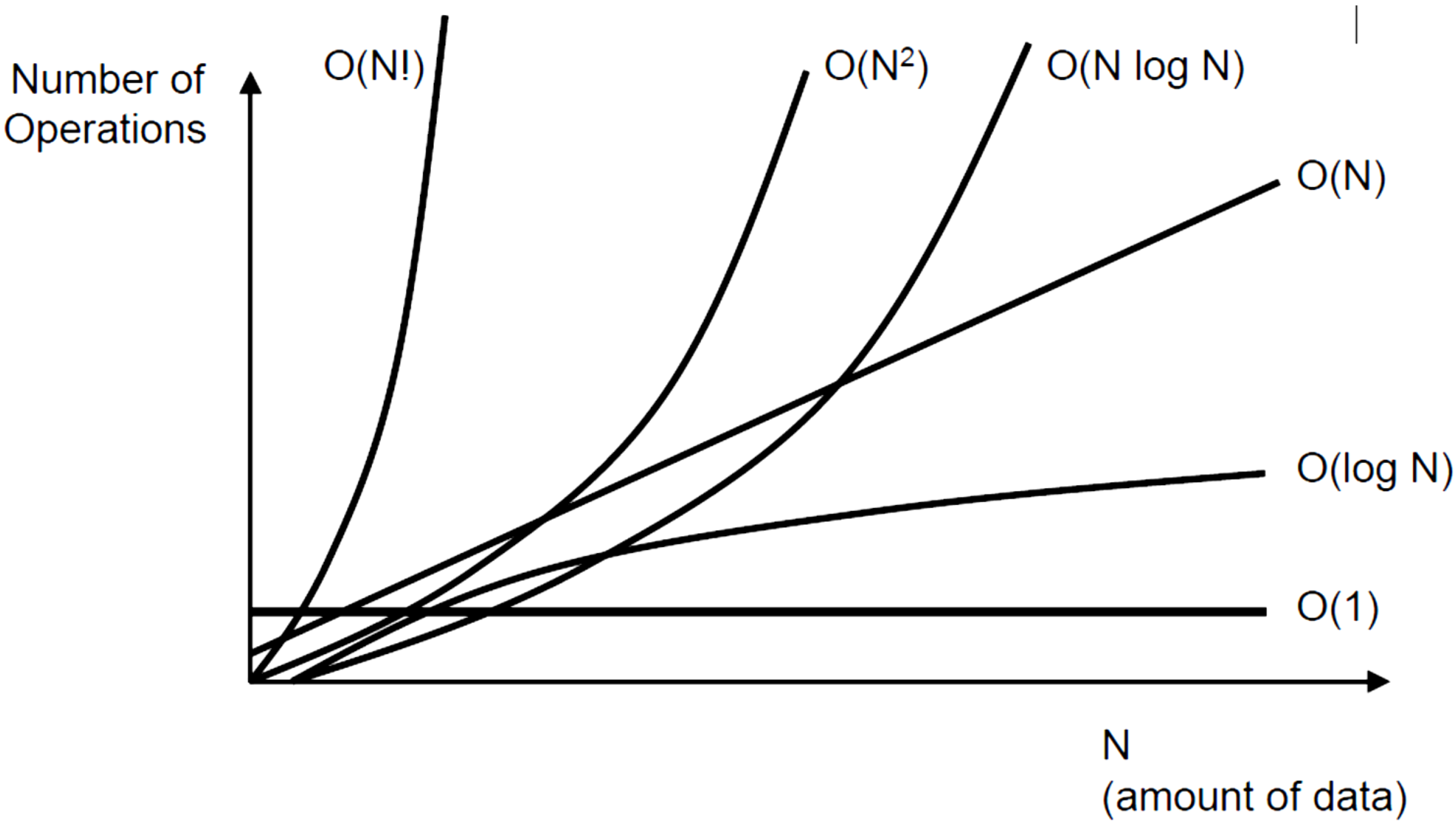
- Commonly used functions in asymptotic notation

Slowest growth
Fastest algorithm

Order/Complexity	Time Adjective
$\Theta(1)$	Constant
$\Theta(\log n)$	Logarithmic
$\Theta(\log^2 n)$	Log-squared
$\Theta(n)$	Linear
$\Theta(n \log n)$	Log-linear
$\Theta(n^2)$	Quadratic
$\Theta(n^2 \log n)$	Log-quadratic
$\Theta(n^3)$	Cubic
$\Theta(a^n)$, $a>1$	Exponential
$\Theta(n!)$	Factorial

Fastest growth
Slowest algorithm

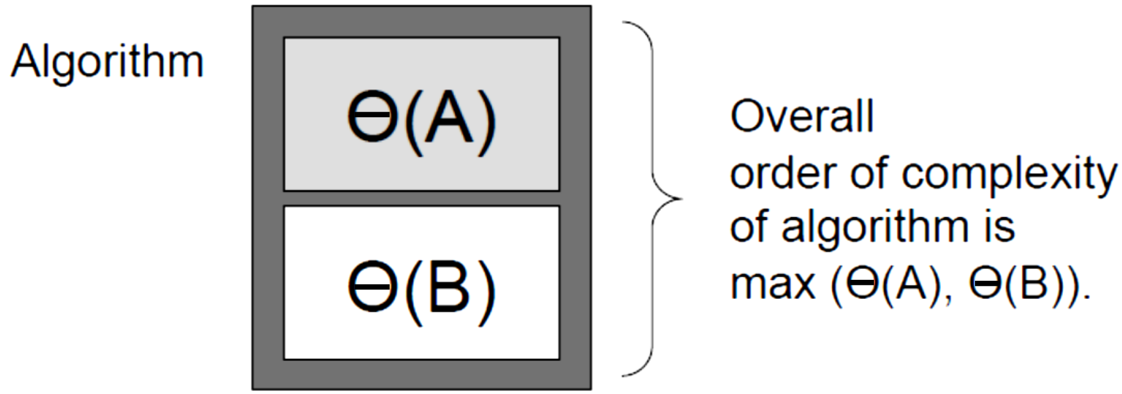
Comparing Big Theta Functions



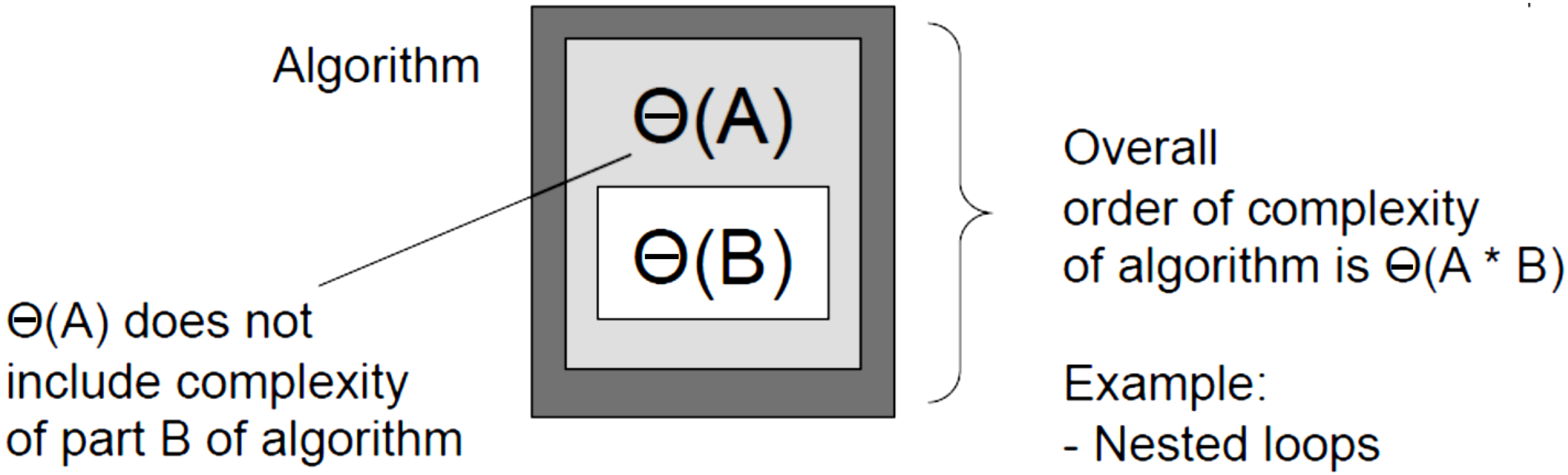
Algorithmic Time

	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n!)$
n = 10	**1 msec**	**1 msec**	**1 msec**
n = 100	10 msec	100 msec	$\frac{100!}{10!}$ msec
n = 1,000	100 msec	10 sec	
n = 10,000	1 sec	16 min 40 sec	
n = 100,000	10 sec	27.7 hr	
n = 1,000,000	1 min 40 sec	115.74 days	

Order of Complexity



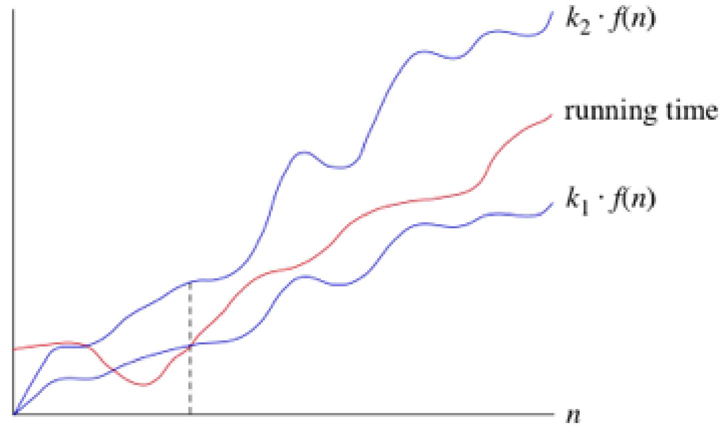
- Examples:
  - $\Theta(\log N) + \Theta(N) = \Theta(N)$
  - $\Theta(N \log N) + \Theta(N) = \Theta(N \log N)$
  - $\Theta(N \log N) + \Theta(N^2) = \Theta(N^2)$
  - $\Theta(2^N) + \Theta(N^2) = \Theta(2^N)$



- Examples:
  - $\Theta(\log N) * \Theta(N) = \Theta(N \log N)$
  - $\Theta(N \log N) * \Theta(N) = \Theta(N^2 \log N)$
  - $\Theta(N) * \Theta(1) = \Theta(N)$

Asymptotically tight bound: Big-Theta Notation

Running time

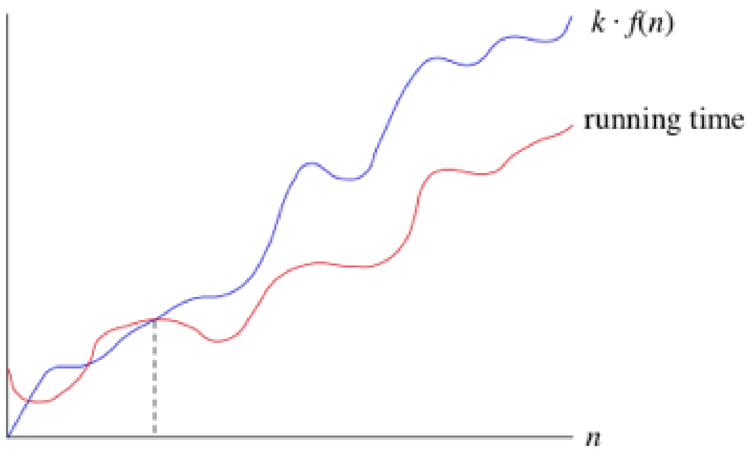


$\boldsymbol{n}_0$ $\boldsymbol{n} > \boldsymbol{n}_0$

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

- If we say that the running time of the algorithm is $\Theta(f(n))$, we imply that the running time is at least $k_1 f(n)$ and at most $k_2 f(n)$ for larger n's.
- Mathematically, this means the running time is tightly bound by the function f(n)

Asymptotic upper bounds: Big-O notation



Running time of the algorithm is "Big-O of $\boldsymbol{f}(\boldsymbol{n})$"
If there is a k constant such that:
running time ≤ *k* x ***f*(n)** for a larger n

Running time is at most $\boldsymbol{k}$ x $\boldsymbol{f}(\boldsymbol{n})$
Running time is **O**($\boldsymbol{f}(\boldsymbol{n})$)

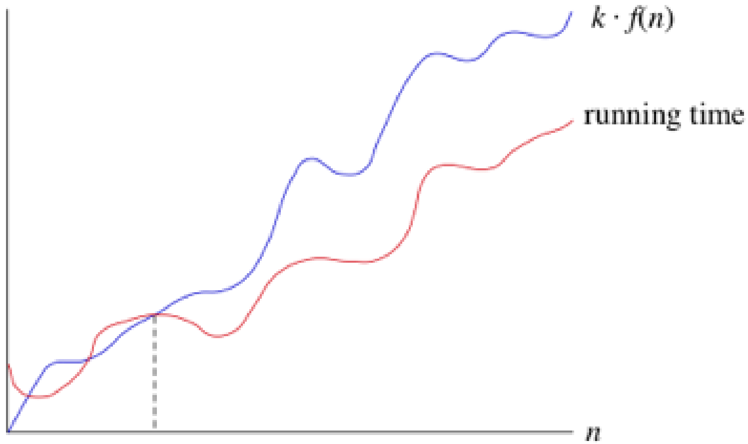
This notation is the most commonly used
in the computer field.
Many times, people mistakenly use Big-**O** for Big-**θ**
These two notations are different.

Example:

- $T(n) = 2n^2 + 3n^3+100$
- $T(n) = O(n^3)$

- $T(n) = (n+1)(1 + n \log n) + \log n^{20}$
- $T(n) = O(n^2 \log n)$

Asymptotic upper bounds: Big-O notation



Running time of the algorithm is "Big-O of ***f***(***n***)"
If there is a k constant such that:
running time ≤ *k* x ***f***(**n**) for a larger n

Running time is at most ***k*** x ***f***(***n***)
Running time is **O**(***f***(***n***))

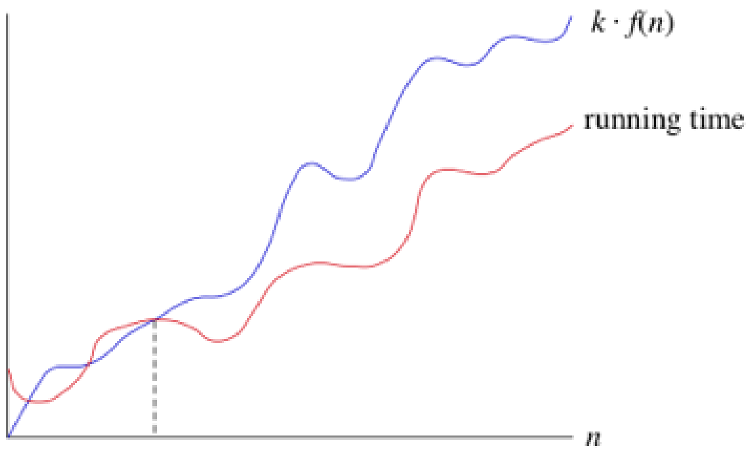
This notation is the most commonly used
in the computer field.
Many times, people mistakenly use Big-**O** for Big-**θ**
These two notations are different.

Examples:

- $T(n) = 2n^2 + 3n^3+100$
- $T(n) = O(n^3)$ -> k=4 (n ≥ 6 ?)

- $T(n) = (n+1)(1 + n \log n) + \log n^2$
- $T(n) = O(n^2 \log n)$ -> k=2 (n ≥ 6 ?)

Asymptotic upper bounds: Big-O notation

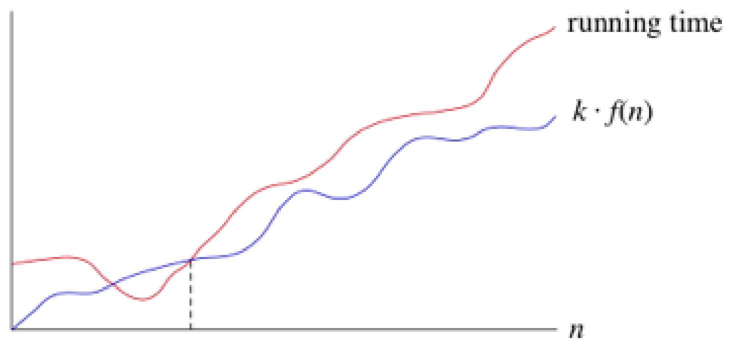


Order	Time Adjective
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(\log^2 n)$	Log-squared
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^2 \log n)$	Log-quadratic
$O(n^3)$	Cubic
$O(a^n)$, $a>1$	Exponential
$O(n!)$	Factorial

Higher order is always an upper bound for the lower order

- $50000 = O(n)$
- $n^2 = O(n^3)$
- $\log n = O(n)$
- $3^n = O(5^n)$

Asymptotic lower bounds: Big-Ω notation



Sometimes, we want to say that an algorithm takes at least a certain amount of time, without providing an upper bound.

Running time of the algorithm is "Big-Ω of ***f***(***n***)"
If there is a k constant such that:
running time ≥ *k* x ***f*(n)** for a larger n

Lower order is always a lower bound for the higher order

- $n = \Omega(50000)$
- $n^3 = \Omega(n^2)$
- $n = \Omega(\log n)$
- $5^n = \Omega(3^n)$