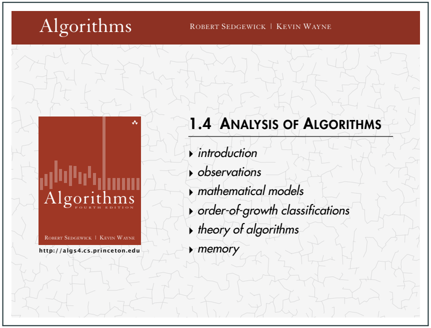
CSU22010: ALGORITHMS AND DATA STRUCTURES

Lecture 4: Order of Growth, Asymptotic Notation, Memory Performance

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin

LAST WEEK



- → Estimate the performance of algorithms by
  - → Experiments & Observations
  - → Precise Mathematical Calculations
  - → Approximate Mathematical Calculations using Cost Models
    - → Every basic operation costs 1 time unit
    - → Keep only the higher-order terms
    - → Count only some operations
- → This Lecture: Classification according to running time order of growth

Doubling hypothesis

Doubling hypothesis. Quick way to estimate b in a power-law relationship.

Run program, doubling the size of the input.

N	time (seconds) †	ratio	lg ratio
250	0.0		–
500	0.0	4.8	2.3
1,000	0.1	6.9	2.8
2,000	0.8	7.7	2.9
4,000	6.4	8.0	3.0
8,000	51.1	8.0	3.0

lg (6.4 / 0.8) = 3.0

seems to converge to a constant b ≈ 3

$$\frac{T(2N)}{T(N)} = \frac{a(2N)^b}{aN^b} = 2^b$$

Hypothesis. Running time is about $a N^b$ with $b = \lg$ ratio.

Caveat. Cannot identify logarithmic factors with doubling hypothesis.

http://algs4.cs.princeton.edu

1.4 ANALYSIS OF ALGORITHMS

- introduction
- observations
- mathematical models
- *order-of-growth classifications*
- theory of algorithms
- memory

Common order-of-growth classifications

Definition. If $f(N) \sim c\, g(N)$ for some constant $c > 0$, then the order of growth of $f(N)$ is $g(N)$.

- Ignores leading coefficient.
- Ignores lower-order terms.

Ex. The order of growth of the running time of this code is N^3.

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] + a[k] == 0)
            count++;
```

Typical usage. With running times.

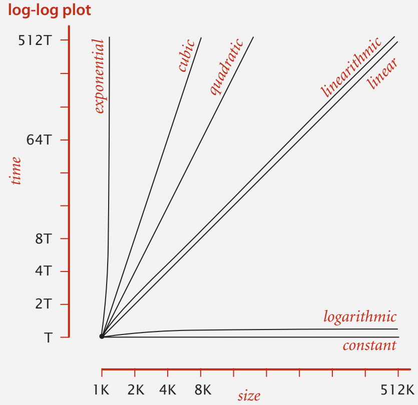
where leading coefficient
depends on machine, compiler, JVM, ...

Common order-of-growth classifications

Good news. The set of functions

$1, \ \log N, \ N, \ N \log N, \ N^2, \ N^3$, and 2^N

suffices to describe the order of growth of most common algorithms.



Typical orders of growth

Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	**constant**	`a = b + c;`	statement	add two numbers	1
$\log N$	**logarithmic**	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	**linear**	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
$N \log N$	**linearithmic**	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	**quadratic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	**cubic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	**exponential**	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Example: 3-SUM

Q. Approximately how many array accesses as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] + a[k] == 0)
            count++;
```

"inner loop"

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \sim \frac{1}{6}N^3$$

A. $\sim \frac{1}{2} N^3$ array accesses.

- → Count only array accesses
- → Cost of each array access: 1 time unit
- → use tilde notation

Order of Growth: N^3

Example: Binary Search

Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo														↑ hi

Binary search: Java implementation

Trivial to implement?

- First binary search published in 1946.
- First bug-free one in 1962.
- Bug in Java's `Arrays.binarySearch()` discovered in 2006.

```
public static int binarySearch(int[] a, int key)
{
   int lo = 0, hi = a.length-1;
   while (lo <= hi)
   {
       int mid = lo + (hi - lo) / 2;
       if      (key < a[mid]) hi = mid - 1;
       else if (key > a[mid]) lo = mid + 1;
       else return mid;
   }
   return -1;
}
```

one "3-way compare"

Invariant. If `key` appears in the array `a[]`, then `a[lo]` ≤ `key` ≤ `a[hi]`.

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N.

Def. $T(N)$ = # key compares to binary search a sorted subarray of size ≤ N.

Binary search recurrence. $T(N) \le \underbrace{T(N/2)}_{\text{left or right half (floored division)}} + \underbrace{1}_{\text{possible to implement with one 2-way compare (instead of 3-way)}}$ for $N > 1$, with $T(1) = 1$.

Pf sketch. [assume N is a power of 2]

$$
\begin{aligned}
T(N) &\le T(N/2) + 1 && \text{[given]} \\
&\le T(N/4) + 1 + 1 && \text{[apply recurrence to first term]} \\
&\le T(N/8) + 1 + 1 + 1 && \text{[apply recurrence to first term]} \\
&\;\;\vdots \\
&\le T(N/N) + 1 + 1 + \ldots + 1 && \text{[stop applying, } T(1) = 1 \text{]} \\
&= 1 + \lg N
\end{aligned}
$$

- The base of the logarithm contributes only a constant factor to the running time.

$$log_a N = \frac{log_b N}{log_b a} = c \cdot log_b N$$

  where $c = 1/log_b a$ is a constant (does not depend on N).

- **EX.** BINS (Binary search) runs in $T_{\text{BINS}}(N) \sim lgN$ [1] time.

  Suppose SUPERBINS, a faster algorithm for binary search, that runs in $T_{suberbin}(N) \sim log_{16} N$ time.

  Then we would have $T_{\text{SUPERBINS}}(N) \sim \frac{1}{log_2 16} lgN = \frac{1}{4} lgN \sim \frac{1}{4} T_{\text{BINS}}(N)$.

  Although the faster algorithm runs in 1/4 of the time of binary search, it still has the same order of growth:

  - When the input size **doubles**, the running time increases by the same amount:

$$\frac{T_{\text{SUPERBINS}}(2N)}{T_{\text{SUPERBINS}}(N)} = \frac{\frac{1}{4} T_{\text{BINS}}(2N)}{\frac{1}{4} T_{\text{BINS}}(N)} = \frac{T_{\text{BINS}}(2N)}{T_{\text{BINS}}(N)}$$

[1] lgN is notation for $log_2 N$

Example: 3-SUM

Q. Approximately how many array accesses as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] + a[k] == 0)
            count++;
```

"inner loop"

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \sim \frac{1}{6}N^3$$

A. $\sim \frac{1}{2} N^3$ array accesses.

Can we do better?

An $N^2 \log N$ algorithm for 3-SUM

Algorithm.

- Step 1: Sort the N (distinct) numbers.
- Step 2: For each pair of numbers `a[i]` and `a[j]`, binary search for `-(a[i] + a[j])`.

What is the order of growth?

input

`30 -40 -20 -10 40 0 10 5`

sort

`-40 -20 -10 0 5 10 30 40`

binary search

(-40, -20)	60
(-40, -10)	50
(-40, 0)	40
(-40, 5)	35
(-40, 10)	30
⋮	⋮
(-20, -10)	30
⋮	⋮
(-10, 0)	10
⋮	⋮
(10, 30)	~~-40~~
(10, 40)	-50
(30, 40)	-70

only count if a[i] < a[j] < a[k] to avoid double counting

An $N^2 \log N$ algorithm for 3-SUM

Algorithm.

- Step 1: Sort the N (distinct) numbers.
- Step 2: For each pair of numbers `a[i]` and `a[j]`, binary search for `-(a[i] + a[j])`.

Analysis. Order of growth is $N^2 \log N$.

- Step 1: N^2 with insertion sort.
- Step 2: $N^2 \log N$ with binary search.

Remark. Can achieve N^2 by modifying binary search step.

input

```
30 -40 -20 -10 40  0 10  5
```

sort

```
-40 -20 -10  0  5 10 30 40
```

binary search

```
(-40, -20)    60
(-40, -10)    50
(-40,   0)    40
(-40,   5)    35
(-40,  10)    30
    ⋮          ⋮
(-20, -10)    30
    ⋮          ⋮
(-10,   0)    10
    ⋮          ⋮
( 10,  30)   -40
( 10,  40)   -50
( 30,  40)   -70
```

only count if a[i] < a[j] < a[k] to avoid double counting

Comparing programs

Hypothesis. The sorting-based $N^2 \log N$ algorithm for 3-SUM is significantly faster in practice than the brute-force N^3 algorithm.

N	time (seconds)
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

ThreeSum.java

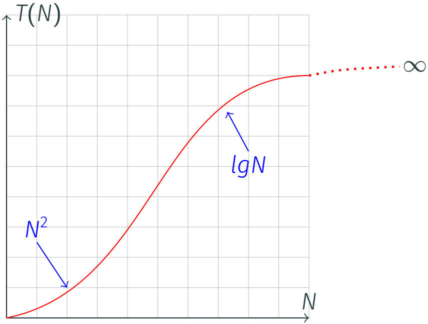
N	time (seconds)
1,000	0.14
2,000	0.18
4,000	0.34
8,000	0.96
16,000	3.67
32,000	14.88
64,000	59.16

ThreeSumDeluxe.java

Guiding principle. Typically, better order of growth $\Rightarrow$ faster in practice.

Asymptotic Notation

- In the Theory of Algorithms we are interested in the order of growth of runtime $T(N)$, expressed as a function of the input size N.
- $T(N)$ may not be a simple function and can have **different orders of growth** for different values of N.



- Rationale of asymptotic notation:
  - larger N's are more important than smaller ones
  - consider the order of growth when $N \to \infty$ (asymptotic order of growth)

→ $\Theta(g(N))$: the **set** of functions with **asymptotic order of growth** $g(N)$.

EX. The (worst case) running time $T(N)$ of Insertion Sort is in $\Theta(N^2)$.

We write $T(N) = \Theta(N^2)$ †

† Abuse of notation, means $T(N) \in \Theta(N^2)$.

→ $\Theta(g(N))$: the **set** of functions with **asymptotic order of growth** $g(N)$.

EX. The (worst case) running time $T(N)$ of Insertion Sort is in $\Theta(N^2)$.

We write $T(N) = \Theta(N^2)$ †

→ $O(g(N))$: the **set** of functions with **asymptotic order of growth** $\leq g(N)$.

EX. The (worst case) running time $T(N)$ of Insertion Sort is in $O(N^3)$.

We write $T(N) = O(N^3)$ †

† Abuse of notation, means $T(N) \in \Theta(N^2)$.

- $\Theta(g(N))$: the **set** of functions with **asymptotic order of growth** $g(N)$.

  EX. The (worst case) running time $T(N)$ of Insertion Sort is in $\Theta(N^2)$.

  We write $T(N) = \Theta(N^2)$ †

- $O(g(N))$: the **set** of functions with **asymptotic order of growth** $\leq g(N)$.

  EX. The (worst case) running time $T(N)$ of Insertion Sort is in $O(N^3)$.

  We write $T(N) = O(N^3)$ †

- $\Omega(g(N))$: the **set** of functions with **asymptotic order of growth** $\geq g(N)$.

  EX. The (worst case) running time $T(N)$ of Insertion Sort is in $\Omega(N)$.

  We write $T(N) = \Omega(N)$ †

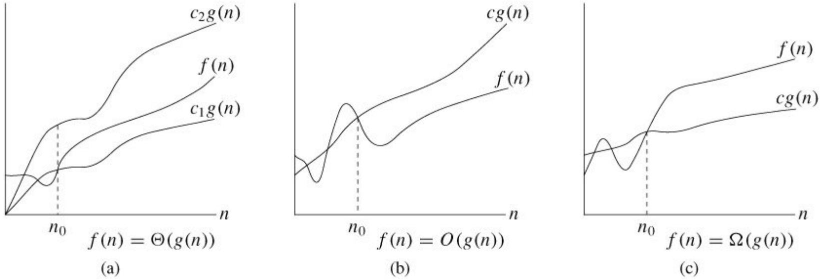
---

†Abuse of notation, means $T(N) \in \Theta(N^2)$.

Commonly-used notations in the theory of algorithms

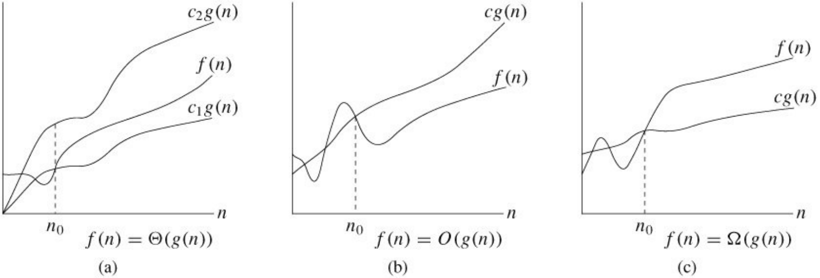
notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ $\vdots$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ $\vdots$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ $\vdots$	develop lower bounds

(image CLRS book)



→ $\Theta(g(N)) = \Big\{ f(N) \;:\;$ there exist positive constants c_1, c_2, N_0 such that $0 \leq c_1 g(N) \leq f(N) \leq c_2 g(N)$ for all $N \geq N_0 \Big\}$

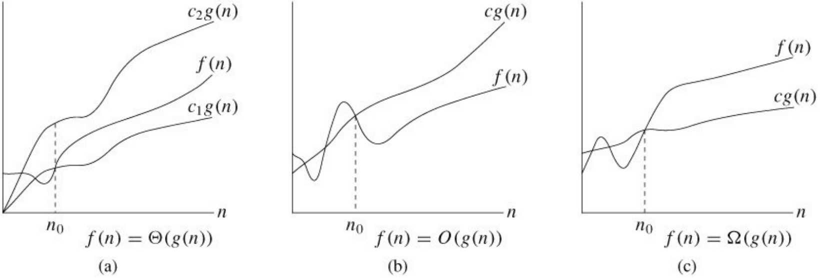
(image CLRS book)



→ $\Theta(g(N)) = \Big\{ f(N) \;:\;$ there exist positive constants c_1, c_2, N_0 such that $0 \leq c_1 g(N) \leq f(N) \leq c_2 g(N)$ for all $N \geq N_0 \Big\}$

→ $O(g(N)) = \Big\{ f(N) \;:\;$ there exist positive constants c, N_0 such that $0 \leq f(N) \leq c\, g(N)$ for all $N \geq N_0 \Big\}$

(image CLRS book)



→ $\Theta(g(N)) = \Big\{ f(N) \; : \;$ there exist positive constants c_1, c_2, N_0 such that $0 \le c_1 g(N) \le f(N) \le c_2 g(N)$ for all $N \ge N_0 \Big\}$

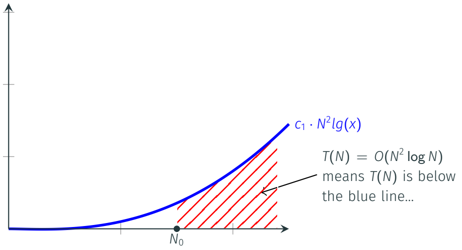
→ $O(g(N)) = \Big\{ f(N) \; : \;$ there exist positive constants c, N_0 such that $0 \le f(N) \le c\, g(N)$ for all $N \ge N_0 \Big\}$

→ $\Omega(g(N)) = \Big\{ f(N) \; : \;$ there exist positive constants c, N_0 such that $0 \le c_1 g(N) \le f(N)$ for all $N \ge N_0 \Big\}$

Suppose `myAlgorith` has an asymptotic running time $T(N) = O(N^2 \log N)$

Does that necessarily mean

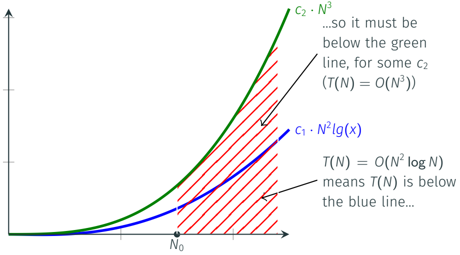
- $T(N) = O(N^3)$?
- $T(N) = O(N^2)$?
- $T(N) = \Omega(N)$?
- $T(N) = \Omega(N^3)$?
- $T(N) = \Omega(N^2 \log N)$?
- $T(N) = \Theta(N^2 \log N)$?



Suppose `myAlgorith` has an asymptotic running time $T(N) = O(N^2 \log N)$

Does that necessarily mean

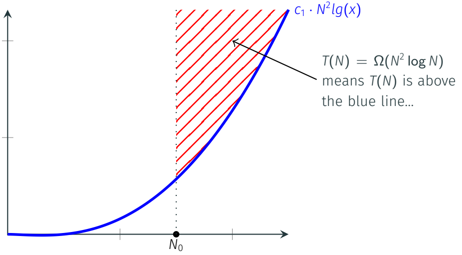
- $\rightarrow$ $T(N) = O(N^3)$? YES
- $\rightarrow$ $T(N) = O(N^2)$? NO
- $\rightarrow$ $T(N) = \Omega(N)$? NO
- $\rightarrow$ $T(N) = \Omega(N^3)$? NO
- $\rightarrow$ $T(N) = \Omega(N^2 \log N)$? NO
- $\rightarrow$ $T(N) = \Theta(N^2 \log N)$? NO



Suppose `myAlgorith` has an asymptotic running time $T(N) = \Omega(N^2 \log N)$

Does that necessarily mean

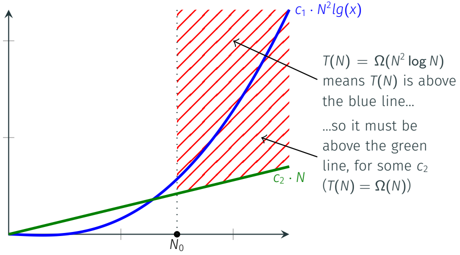
- $T(N) = O(N^3)$?
- $T(N) = O(N^2)$?
- $T(N) = \Omega(N)$?
- $T(N) = \Omega(N^3)$?
- $T(N) = O(N^2 \log N)$?
- $T(N) = \Theta(N^2 \log N)$?



Suppose `myAlgorith` has an asymptotic running time $T(N) = \Omega(N^2 \log N)$

Does that necessarily mean

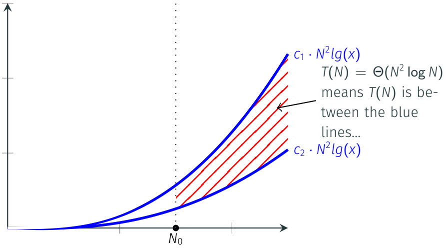
- → $T(N) = O(N^3)$? NO
- → $T(N) = O(N^2)$? NO
- → $T(N) = \Omega(N)$? YES
- → $T(N) = \Omega(N^3)$? NO
- → $T(N) = O(N^2 \log N)$? NO
- → $T(N) = \Theta(N^2 \log N)$? NO



Suppose `myAlgorith` has an asymptotic running time $T(N) = \Theta(N^2 \log N)$

Does that necessarily mean

- $T(N) = O(N^3)$?
- $T(N) = O(N^2)$?
- $T(N) = \Omega(N)$?
- $T(N) = \Omega(N^3)$?
- $T(N) = \Omega(N^2 \log N)$?
- $T(N) = O(N^2 \log N)$?

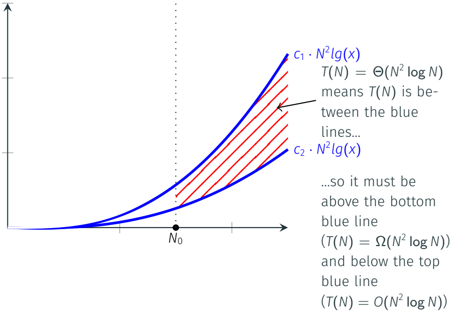


EXERCISE: NECESSARY CONCLUSIONS

Suppose `myAlgorith` has an asymptotic running time $T(N) = \Theta(N^2 \log N)$

Does that necessarily mean

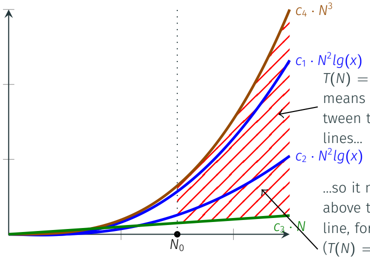
- $T(N) = O(N^3)$?
- $T(N) = O(N^2)$?
- $T(N) = \Omega(N)$?
- $T(N) = \Omega(N^3)$?
- $T(N) = \Omega(N^2 \log N)$? YES
- $T(N) = O(N^2 \log N)$? YES



Suppose `myAlgorith` has an asymptotic running time $T(N) = \Theta(N^2 \log N)$

Does that necessarily mean

- $T(N) = O(N^3)$? YES
- $T(N) = O(N^2)$? NO
- $T(N) = \Omega(N)$? YES
- $T(N) = \Omega(N^3)$? NO
- $T(N) = \Omega(N^2 \log N)$? YES
- $T(N) = O(N^2 \log N)$? YES



$T(N) = \Theta(N^2 \log N)$ means $T(N)$ is between the blue lines…

…so it must be above the green line, for some c_3 ($T(N) = \Omega(N)$) and below the brown line, for some c_4 ($T(N) = O(N^3)$)

- → **Tilde notation**: an **approximate model**
- → **Asymptotic notation** *O*/*Theta*/*Omega*: order of growth when $N \to \infty$

- **Tilde notation**: an **approximate model**
- **Asymptotic notation** *O/Theta/Omega*: order of growth when $N \to \infty$
- **Common mistake 1**: interpret *O/Theta/Omega* as worst/average/best case running times
- **Common mistake 2**: interpret asymptotic notation as an approximate model

- **Tilde notation**: an **approximate model**
- **Asymptotic notation** *O/Theta/Omega*: order of growth when $N \to \infty$
- **Common mistake 1**: interpret *O/Theta/Omega* as worst/average/best case running times
- **Common mistake 2**: interpret asymptotic notation as an approximate model
- **Book S&W**: uses the Tilde-notation.
- **This module**: use asymptotic notation.
  - **Easy calculations** because of the properties of asymptotic notation

Simplify: Constant coefficients are equivalent to 1;
Keep only highest order term

Ex:

- $\Theta(1) = \Theta(2) = \Theta(300)$
- $\Theta(N^2) = \Theta(10N^2) = \Theta(100N^2)$
- $\Theta(lgN) = \Theta(\log_{10} N) = \Theta(\log_{20} N)$
- $\Theta(N + \log N) = \Theta(N)$
- $\Theta(10N^2 + 5 \log N + 30) = \Theta(N^2)$

And the same for O and Ω

Always use the simplest forms!

Addition: keep only the highest order terms

Theorem
$\Theta(f(N)) + \Theta(g(N)) = \Theta(f(N)) \quad \textit{when } f(N) \geq_\infty g(N)$

Ex:

- $\Theta(1) + \Theta(1) = \Theta(1)$
- $\Theta(N^2) + \Theta(N \log N) = \Theta(N^2)$
- $\Theta(\log N) + \Theta(N \log N) = \Theta(N \log N)$

And the same for O and $\boldsymbol{\Omega}$

Multiplication: multiply inner functions

Theorem
$\Theta(f(N)) \times \Theta(g(N)) = \Theta(f(N) \times g(N))$

Ex:

- $\Theta(1) \times \Theta(1) = \Theta(1)$
- $\Theta(N^2) \times \Theta(N \log N) = \Theta(N^3 \log N)$
- $\Theta(\log N) \times \Theta(2^N) = \Theta(2^N \log N)$

And the same for O and Ω

```java
public static int binarySearch(int[] a, int key) {
  int lo = 0, hi = a.length-1;
  while (lo <= hi) {
    int mid = lo + (hi - lo)/2;
    if      (key < a[mid]) hi = mid - 1;
    else if (key > a[mid]) lo = mid + 1;
    else return mid;
  }
  return -1;
}
```

Asymptotic (worst case) analysis:

EXAMPLE: BINARY SEARCH

```
1  public static int binarySearch(int[] a, int key) {
2    int lo = 0, hi = a.length-1;
3    while (lo <= hi) {
4      int mid = lo + (hi - lo)/2;
5      if      (key < a[mid]) hi = mid - 1;
6      else if (key > a[mid]) lo = mid + 1;
7      else return mid;
8    }
9    return -1;
10 }
```

Asymptotic (worst case) analysis:

Line 2: executed $\Theta(1)$ times, execution takes $\Theta(1)$ time $\Rightarrow$ $T_2 = \Theta(1)$

Lines 3 – 8: executed $\Theta(\log N)$ times, each execution takes $\Theta(1)$ time $\Rightarrow$ $T_{3-8} = \Theta(\log N) \times \Theta(1) = \Theta(\log N)$

Lines 9 – 10: executed $\Theta(1)$ times, execution takes $\Theta(1)$ time $\Rightarrow$ $T_{9-10} = \Theta(1)$

EXAMPLE: BINARY SEARCH

```java
1   public static int binarySearch(int[] a, int key) {
2     int lo = 0, hi = a.length-1;
3     while (lo <= hi) {
4       int mid = lo + (hi - lo)/2;
5       if      (key < a[mid]) hi = mid - 1;
6       else if (key > a[mid]) lo = mid + 1;
7       else return mid;
8     }
9     return -1;
10  }
```

Asymptotic (worst case) analysis:

Line 2: executed $\Theta(1)$ times, execution takes $\Theta(1)$ time $\Rightarrow$ $T_2 = \Theta(1)$

Lines 3 – 8: executed $\Theta(\log N)$ times, each execution takes $\Theta(1)$ time $\Rightarrow$ $T_{3-8} = \Theta(\log N) \times \Theta(1) = \Theta(\log N)$

Lines 9 – 10: executed $\Theta(1)$ times, execution takes $\Theta(1)$ time $\Rightarrow$ $T_{9-10} = \Theta(1)$

Total Running time: $T(N) = T_2 + T_{3-8} + T_{9-10} = \Theta(1) + \Theta(\log N) + \Theta(1) = \Theta(\log N)$

```java
public void insertion_sort(int[] a) {
  for (int j = 1; j < a.length; j++) {
    int i = j - 1;
    while(i >= 0 && a[i] > a[i+1]) {
      int temp = a[i];
      a[i] = a[i+1];
      a[i+1] = temp;
      i--;
    }
  }
}
```

Asymptotic (worst case) analysis:

EXAMPLE: INSERTION SORT

```
1   public void insertion_sort(int[] a) {
2     for (int j = 1; j < a.length; j++) {
3       int i = j - 1;
4       while(i >= 0 && a[i] > a[i+1]) {
5         int temp = a[i];
6         a[i] = a[i+1];
7         a[i+1] = temp;
8         i--;
9       }
10    }
11  }
```

Asymptotic (worst case) analysis:

Lines 2,3,10: executed $\Theta(N)$ times, execution takes $\Theta(1)$ time $\Rightarrow$ $T_{2,3,10} = \Theta(N)$

EXAMPLE: INSERTION SORT

```
1  public void insertion_sort(int[] a) {
2    for (int j = 1; j < a.length; j++) {
3      int i = j - 1;
4      while(i >= 0 && a[i] > a[i+1]) {
5        int temp = a[i];
6        a[i] = a[i+1];
7        a[i+1] = temp;
8        i--;
9      }
10   }
11 }
```

Asymptotic (worst case) analysis:

Lines 2,3,10: executed $\Theta(N)$ times, execution takes $\Theta(1)$ time $\quad\Rightarrow\quad T_{2,3,10} = \Theta(N)$

Lines 4 – 9: executed $\Theta(\log N^2)$ times, each execution takes $\Theta(1)$ time $\quad\Rightarrow$
$T_{4-9} = \Theta(N^2) \times \Theta(1) = \Theta(N^2)$

Line 11: executed $\Theta(1)$ times, execution takes $\Theta(1)$ time $\quad\Rightarrow\quad T_{11} = \Theta(1)$

Total Running time: $T(N) = T_{2,3,10} + T_{4-9} + T_{11} = \Theta(N) + \Theta(N^2) + \Theta(1) = \Theta(N^2)$

EXERCISE: ASYMPTOTIC ANALYSIS

A software engineer was asked to design an algorithm which will input two **unsorted** arrays of integers, A (of size *N*) and B (also of size *N*), and will output `true` when all integers in A are present in B. The engineer came up with **two** alternatives. Which one is better?

```
boolean isContained1(int[] A, int[] B) {
  boolean AInB = true;
  for (int i = 0; i < A.length; i++) {
    boolean iInB = linearSearch(B, A[i]);
    AInB = AInB && iInB;
  }
  return AinB;
}
```

```
boolean isContained2(int[] A, int[] B) {
  int[] C = new int[B.length];
  for (int i = 0; i < B.length; i++) { C[i] = B[i] }
  sort(C); // heapsort
  boolean AInC = true;
  for (int i = 0; i < A.length; i++) {
    boolean iInC = binarySearch(C, A[i]);
    AInC = AInC && iInC;
  }
  return AinC;
}
```

Write the following asymptotic order of growths in asceding order, from the most to the least efficient, using $<$ or $=$ to show the equivalences and inequivalences between them.

$\Theta(N \log N)$

$\Theta(N)$

$\Theta(N^2 + 3N + 1)$

$\Theta(1)$

$\Theta(5N)$

$\Theta(N^3 + \log N)$

$\Theta(N^2)$

$\Theta(10)$

$\Theta(10N^3 + 2\lg(N))$

$\Theta(10\lg(N))$

$\Theta(2^N)$

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

- introduction
- observations
- mathematical models
- order-of-growth classifications
- **theory of algorithms**
- memory

Types of analyses

~~**Best case.** Lower bound on cost.~~
- ~~Determined by "easiest" input.~~
- ~~Provides a goal for all inputs.~~

Worst case. Upper bound on cost.
- Determined by "most difficult" input.
- Provides a guarantee for all inputs.

Average case. Expected cost for random input.
- Need a model for "random" input.
- Provides a way to predict performance.

this course

Ex 1. Array accesses for brute-force 3-SUM.

~~Best: $\sim \frac{1}{2} N^3$~~

Average: $\sim \frac{1}{2} N^3$

Worst: $\sim \frac{1}{2} N^3$

Ex 2. Compares for binary search.

~~Best: ~ 1~~

Average: $\sim \lg N$

Worst: $\sim \lg N$

Theory of algorithms

Goals.

- Establish "difficulty" of a problem.
- Develop "optimal" algorithms.

Approach.

- Suppress details in analysis: analyze "to within a constant factor."
- Eliminate variability in input model: focus on the worst case.

Upper bound. Performance guarantee of algorithm for any input.

Lower bound. Proof that no algorithm can do better. (for worst case inputs)

Optimal algorithm. Lower bound = upper bound (to within a constant factor).

Theory of algorithms: example 1

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 1-SUM = "*Is there a* 0 *in the array?*"

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 1-SUM: Look at every array entry.
- Running time of the optimal algorithm for 1-SUM is $O(N)$.

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all N entries (any unexamined one might be 0).
- Running time of the optimal algorithm for 1-SUM is $\Omega(N)$. (for worst case inputs)

Optimal algorithm.

- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is optimal: its running time is $\Theta(N)$.

Theory of algorithms: example 2

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^3)$.

Theory of algorithms: example 2

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.

- Ex. Improved algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^2 \log N)$.

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all N entries to solve 3-SUM.
- Running time of the optimal algorithm for solving 3-SUM is $\Omega(N)$.

Open problems.

- Optimal algorithm for 3-SUM?
- Subquadratic algorithm for 3-SUM?
- Quadratic lower bound for 3-SUM?

Algorithm design approach

Start.

- Develop an algorithm.
- Prove a lower bound.

Gap?

- Lower the upper bound (discover a new algorithm).
- Raise the lower bound (more difficult).

Golden Age of Algorithm Design.

- 1970s–.
- Steadily decreasing upper bounds for many important problems.
- Many known optimal algorithms.

Caveats.

- Overly pessimistic to focus on worst case?
- Need better than “to within a constant factor” to predict performance.

http://algs4.cs.princeton.edu

1.4 ANALYSIS OF ALGORITHMS

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth classifications*
- *theory of algorithms*
- *memory*

Basics

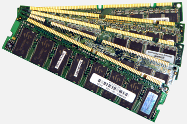
NIST most computer scientists

Bit. 0 or 1.

Byte. 8 bits.

Megabyte (MB). 1 million or 2^{20} bytes.

Gigabyte (GB). 1 billion or 2^{30} bytes.



64-bit machine. We assume a 64-bit machine with 8-byte pointers.

- Can address more memory.
- Pointers use more space.

some JVMs "compress" ordinary object pointers to 4 bytes to avoid this cost

Typical memory usage for primitive types and arrays

type	bytes
`boolean`	1
`byte`	1
`char`	2
`int`	4
`float`	4
`long`	8
`double`	8

primitive types

type	bytes
`char[]`	$2N + 24$
`int[]`	$4N + 24$
`double[]`	$8N + 24$

one-dimensional arrays

type	bytes
`char[][]`	$\sim 2\,M\,N$
`int[][]`	$\sim 4\,M\,N$
`double[][]`	$\sim 8\,M\,N$

two-dimensional arrays

Typical memory usage for objects in Java

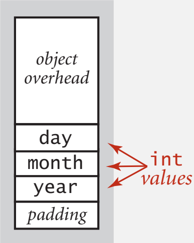
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Each object uses a multiple of 8 bytes.

Ex 1. A `Date` object uses 32 bytes of memory.

```
public class Date
{
   private int day;
   private int month;
   private int year;
...
}
```



16 bytes (object overhead)
4 bytes (int)
4 bytes (int)
4 bytes (int)
4 bytes (padding)

32 bytes

Typical memory usage summary

Total memory usage for a data type value:

- Primitive type: 4 bytes for `int`, 8 bytes for `double`, ...
- Object reference: 8 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable.
- Padding: round up to multiple of 8 bytes.

+ 8 extra bytes per inner class object
(for reference to enclosing class)

Shallow memory usage: Don't count referenced objects.

Deep memory usage: If array entry or instance variable is a reference, count memory (recursively) for referenced object.

Example

Q. How much memory does `WeightedQuickUnionUF` use as a function of N? Use tilde notation to simplify your answer.

```
public class WeightedQuickUnionUF
{
   private int[] id;
   private int[] sz;
   private int count;

   public WeightedQuickUnionUF(int N)
   {
      id = new int[N];
      sz = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
      for (int i = 0; i < N; i++) sz[i] = 1;
   }
   ...
}
```

16 bytes (object overhead)

8 + (4N + 24) bytes each (reference + `int[]` array)

4 bytes (`int`)

4 bytes (padding)

8N + 88 bytes

A. $8N + 88 \sim 8N$ bytes.

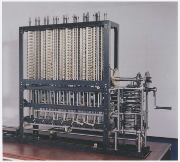
Turning the crank: summary

Empirical analysis.

- Execute program to perform experiments.
- Assume power law and formulate a hypothesis for running time.
- Model enables us to make predictions.

Mathematical analysis.

- Analyze algorithm to count frequency of operations.
- Use tilde notation to simplify analysis.
- Model enables us to explain behavior.



Scientific method.

- Mathematical model is independent of a particular system; applies to machines not yet built.
- Empirical analysis is necessary to validate mathematical models and to make predictions.