cs2010: algorithms and data structures

Lecture 4: Asymptotic notation

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin

Running Time Performance analysis

Techniques until now:

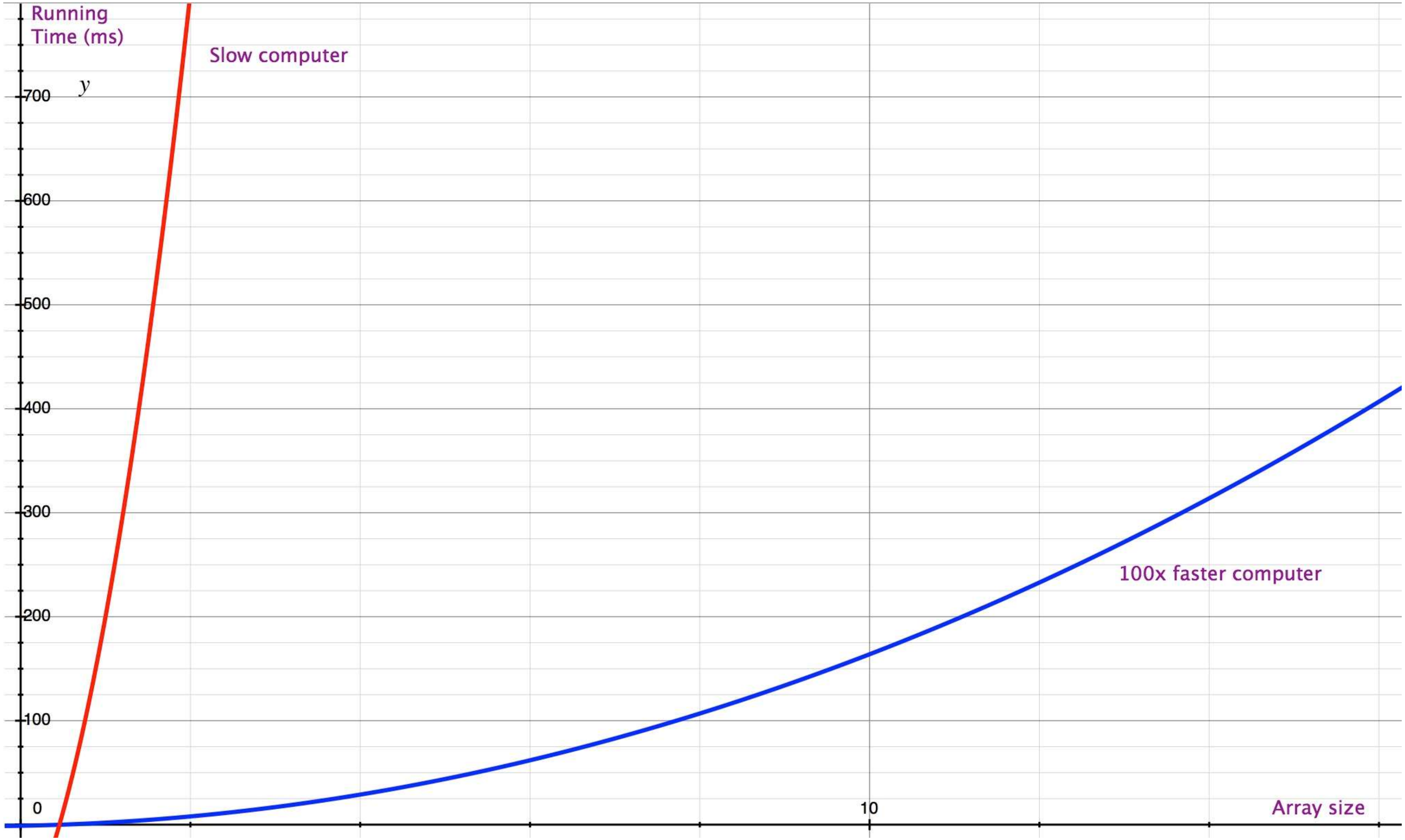
- Experimental
- Approximate running time using cost models
  - counting execution of operations or lines of code.
  - under some assumptions
    - only some operations count
    - cost of each operation = 1 time unit
    - Tilde notation: $T(n) \sim (5/3)n^2$

Today:

- **Asymptotic Running Time: Θ/O/Ω-notation**
- Examples: insertionSort & binarySearch

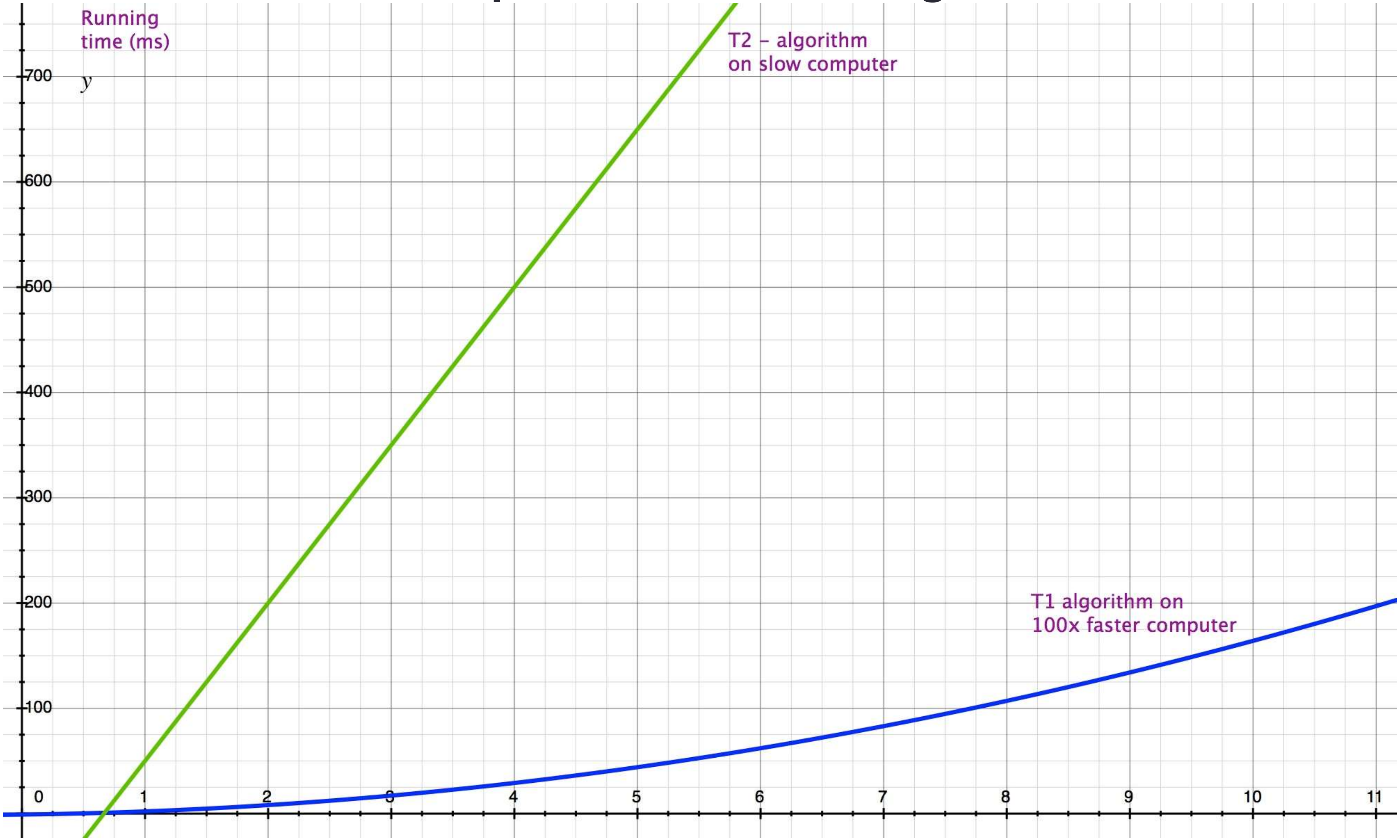
How fast is $T(n) = (3/2)n^2 + (3/2)n - 1$?

Fast computer vs. Slow computer



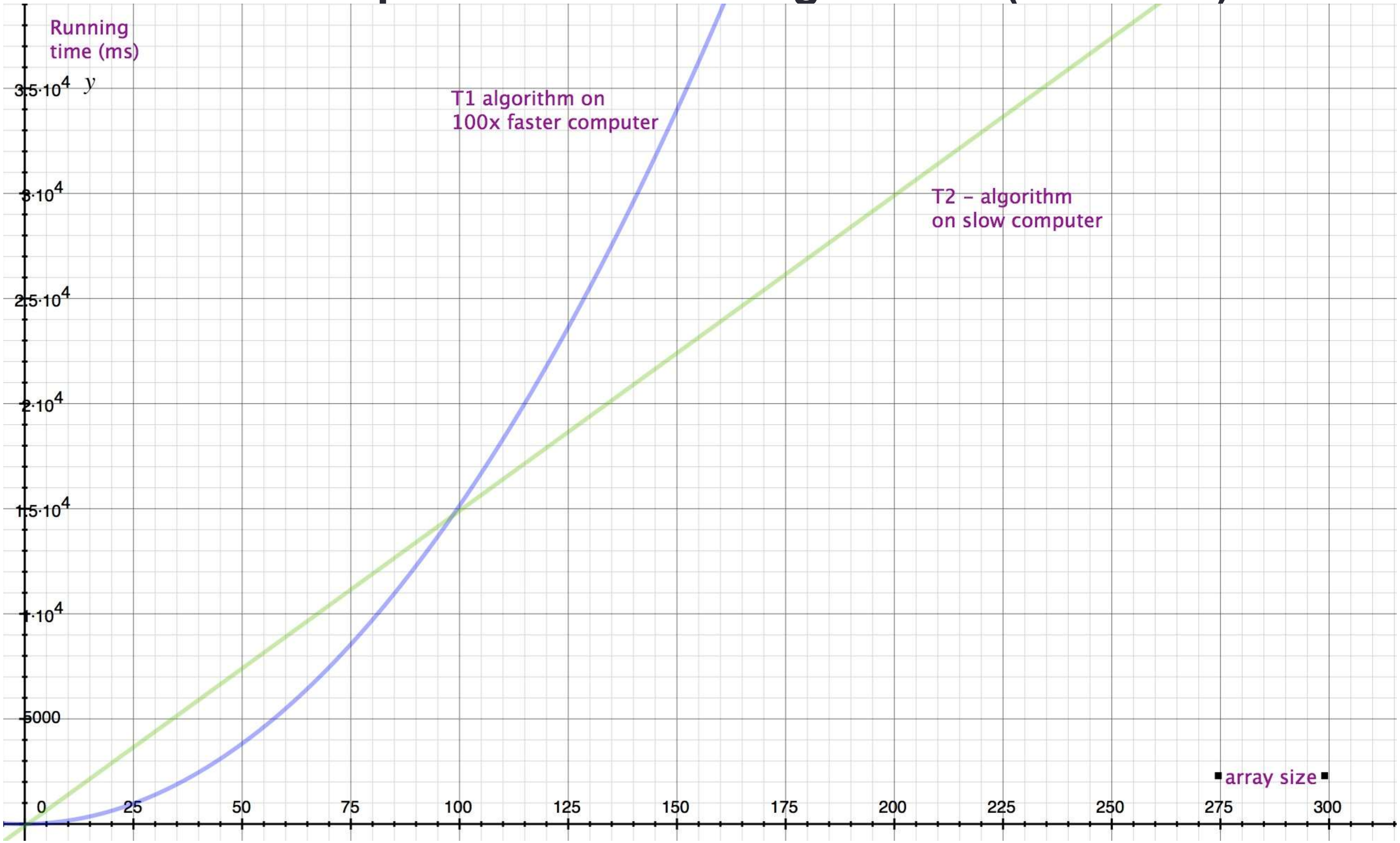
$T1(n) = (3/2)n^2 + (3/2)n - 1$ $T2(n) = (3/2)n - 1$

Fast Computer vs. Smart Programmer



$T1(n) = (3/2)n^2 + (3/2)n - 1$ $T2(n) = (3/2)n - 1$

Fast Computer vs Smart Programmer (rematch!)



A **smart programmer** with a better algorithm always **beats** a **fast computer** with a worst algorithm for <u>sufficiently large inputs.</u>

At large enough input sizes only the rate of growth of an algorithm's running time matters.

- That's why we dropped the lower-order terms in the approximate tilde notation:
  - When $T(n) = (3/2)n^2 + (3/2)n - 1$
  - we write: $\mathbf{T(n) \sim (3/2)n^2}$
  - **However: to calculate $(3/2)n^2$ we need to first calculate $(3/2)n^2 + (3/2)n - 1$**
  - It is not possible to calculate the coefficient 3/2 without the complete polynomials.

Worst Case

	cost	no of times
`for (j = 1; j<A.length; j++) {`		
`//shift A[j] into the sorted A[0..j-1]`		
`i=j-1`		
`while i>=0 and A[i]>A[i+1] {`	1	2+…+n
`swap A[i], A[i+1]`	1	1+…+(n-1)
`i=i-1`		
`}}`		
`return A`		

In the worst case the array is in reverse sorted order.

$$T(n) = \Sigma_{i=2..n}(i) + \Sigma_{i=1..n-1}(i) \quad = \quad \Sigma_{i=1..n}(i) - 1 + \Sigma_{i=1..n-1}(i)$$

$$= (n(n+1)/2 \;-\; 1) + 2n(n-1)/2$$

$$\sim (5/2)n^2$$

Simpler approach

It turns out that **even the coefficient of the highest order term of polynomials is not all that important** for large enough inputs.

This leads us to **Asymptotic running time:**

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

- We calculate **directly** the **growth function**
- Even with such a simplification, we can **compare algorithms** to discover the best ones
- Sometimes constants matter in the real-world performance of algorithms, but in many cases we can ignore them.
- We can write the asymptotic running time of best/worst/average case

Important Growth Functions

From better to worse:

Function f	Name
• **1**	constant
• **log n**	logarithmic
• **n**	linear
• **n·log n**	
• $\mathbf{n^2}$	quadratic
• $\mathbf{n^3}$	cubic
• …	
• $\mathbf{2^n}$	exponential
• …	

Important Growth Functions

From better to worse:

Function f	Name
1	**constant**
$\log n$	**logarithmic**
n	**linear**
$n \cdot \log n$	
n^2	quadratic
n^3	cubic
…	
2^n	exponential
…	

The first 4 are **practically fast** **(most commercial programs run in such Θ-time)**

Anything less than exponential is **theoretically fast** **(P vs NP)**

Important Growth Functions

From better to worse:

Function f	Name	Problem size solved in mins (today)
1	constant	any
log n	logarithmic	any
n	linear	billions
n·log n		hundreds of millions
$\mathbf{n^2}$	quadratic	tens of thousands
$\mathbf{n^3}$	cubic	thousands
…		
$\mathbf{2^n}$	exponential	100
…		

What programs have these running times?

- With experience we can recognise **patterns in code** with known running time

$\Theta(1)$

- Any fixed number of basic operations.
  - assignments
    - int i = 5
  - memory access
    - A[5]
  - fixed number of combinations of the above
    - swap A[i], A[j] (3 array accesses & 3 assignments)
  - any other basic command

- But not:
  - array initialisation
    - int A[] = new int[n] **(Θ(n))**

Θ(n)

- Pattern of code: loops where each iteration **decreases the problem size by a constant factor**

```
for(j=1; j<n; j++){ …<constant cost operations>…  }
```

- Problem: iterate from 1 … n
- Initial problem size: n
- each iteration decreases problem size by 1.

```
k=n;
while(k>0){ …<constant cost operations>… ; k=k-100; }
```

- Problem: iterate from n … 1
- Initial problem size: n
- each iteration decreases problem size by 100.

Θ(log n)

- Pattern of code: loops where each iteration **divides the problem size by a constant**

```
j=n;
while(j>=0){…<constant cost operations>…; j=j/2}
```

- Problem: iterate from n … 0
- Initial problem size: n
- each iteration divides problem size by 2.

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	$T(N)$

Asymptotic Running Time $\Theta(f(n))$

It has useful **operations**:

- **Simplify:** replace multiplicative constants with 1
  - $\Theta(2) = \Theta(1)$
  - $\Theta(10n^2) = \Theta(n^2)$
  - $\Theta(\log_{300} n) = \Theta(\log_2 n)$
- Always use the simplest possible functions

Asymptotic Running Time $\Theta(f(n))$

It has useful **operations**:

- **Addition:** keep only highest factor
  - $\Theta(1) + \Theta(1) = \Theta(1)$
  - $\Theta(n) + \Theta(n^2) = \Theta(n^2)$
  - $\Theta(n^3) + \Theta(n^3) = \Theta(n^3)$
  - $\Theta(n^2 \cdot \log n) + \Theta(n^2) = \Theta(n^2 \cdot \log n)$

Asymptotic Running Time $\Theta(f(n))$

It has useful **operations**:

- **Multiplication:** multiply inner functions
  - $\Theta(n) \times \Theta(n^2) = \Theta(n^3)$
  - $\Theta(n) \times \Theta(\log n) = \Theta(n \cdot \log n)$
  - $\Theta(1) \times \Theta(\log n) = \Theta(\log n)$

Asymptotic Running Time $\Theta(f(n))$

It has useful **operations**:

- **Simplify:** replace multiplicative constants with 1
  - $\Theta(2) = \Theta(1)$
  - $\Theta(10n^2) = \Theta(n^2)$
  - $\Theta(\log_{200}(n)) = \Theta(\log_2(n))$
- **Addition:** keep only highest factor
  - $\Theta(1) + \Theta(1) = \Theta(1)$
  - $\Theta(n) + \Theta(n^2) = \Theta(n^2)$
  - $\Theta(n^3) + \Theta(n^3) = \Theta(n^3)$
  - $\Theta(n^2\log(n)) + \Theta(n^2) = \Theta(n^2\log(n))$
- **Multiplication:** multiply inner functions
  - $\Theta(n) \times \Theta(n^2) = \Theta(n^3)$
  - $\Theta(n) \times \Theta(\log n) = \Theta(n \cdot \log n)$
  - $\Theta(n^2 \cdot \log n) + \Theta(n^2) = \Theta(n^2 \cdot \log n)$

The above are because of the following general theorems:

- $\Theta(f(n)) + \Theta(g(n)) = \Theta(g(n))$ if $\Theta(f(n)) \leq \Theta(g(n))$
- $\Theta(f(n)) \times \Theta(g(n)) = \Theta(f(n) \times g(n))$
- If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ then $f(n) = \Theta(h(n))$

How to calculate Asymptotic running times?

First decide what **case** you want to calculate

- Worst case input ← usually
- Best case input
- Average case input

Then add up costs of each operation multiplied with number of times called

- As we did with the precise/approximate running times
- but use asymptotic notation
- take advantage of the simple addition and multiplication operations

InsertionSort – **asymptotic worst-case analysis**

cost **No of times**

```
for j = 1 to A.length {
 //shift A[j] into the sorted A[0..j-1]
 i=j-1
 while i>=0 and A[i]>A[i+1] {
  swap A[i], A[i+1]
  i=i-1
}}
return A
```

InsertionSort – **asymptotic worst-case analysis**

	cost	No of times
for j = 1 **to** A.length {	$\Theta(1)$	
//shift A[j] into the sorted A[0..j-1]		
i=j-1	$\Theta(1)$	
while i>=0 **and** A[i]>A[i+1] {	$\Theta(1)$	
swap A[i], A[i+1]	$\Theta(1)$	
i=i-1	$\Theta(1)$	
}}		
return A	$\Theta(1)$	

InsertionSort – **asymptotic worst-case analysis**

		cost	No of times
`for j = 1 to A.length {`	Θ(1)	Θ(N)	
`//shift A[j] into the sorted A[0..j-1]`			
`i=j-1`		Θ(1)	Θ(N)
`while i>=0 and A[i]>A[i+1] {`	Θ(1)		
`swap A[i], A[i+1]`	Θ(1)		
`i=i-1`		Θ(1)	
`}}`			
`return A`		Θ(1)	

InsertionSort – **asymptotic worst-case analysis**

		cost	No of times
for j = 1 **to** A.length {	Θ(1)	Θ(N)	
//shift A[j] into the sorted A[0..j-1]			
i=j-1		Θ(1)	Θ(N)
while i>=0 **and** A[i]>A[i+1] {	Θ(1)	Θ(N)xΘ(N)	
swap A[i], A[i+1]	Θ(1)		
i=i-1		Θ(1)	
}}			
return A		Θ(1)	

InsertionSort – **asymptotic worst-case analysis**

		cost	No of times
for j = 1 **to** A.length {	Θ(1)	Θ(N)	
//shift A[j] into the sorted A[0..j-1]			
i=j-1		Θ(1)	Θ(N)
while i>=0 **and** A[i]>A[i+1] {	Θ(1)	Θ(N)xΘ(N)	
swap A[i], A[i+1]	Θ(1)	Θ(N^2)	
i=i-1		Θ(1)	Θ(N^2)
}}			
return A		Θ(1)	Θ(1)

InsertionSort – **asymptotic worst-case analysis**

		cost	No of times
for j = 1 **to** A.length {	Θ(1)	Θ(N)	
//shift A[j] into the sorted A[0..j-1]			
i=j-1		Θ(1)	Θ(N)
while i>=0 **and** A[i]>A[i+1] {	Θ(1)	Θ(N)xΘ(N)	
swap A[i], A[i+1]	Θ(1)	$Θ(N^2)$	
i=i-1		Θ(1)	$Θ(N^2)$
}}			
return A		Θ(1)	Θ(1)

$T(n) = \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(N^2) + \Theta(1) \times \Theta(N^2) + \Theta(1) \times \Theta(N^2) + \Theta(1) \times \Theta(1)$

InsertionSort – **asymptotic worst-case analysis**

		cost	No of times
for j = 1 **to** A.length {	$\Theta(1)$	$\Theta(N)$	
//shift A[j] into the sorted A[0..j-1]			
i=j-1		$\Theta(1)$	$\Theta(N)$
while i>=0 **and** A[i]>A[i+1] {	$\Theta(1)$	$\Theta(N)x\Theta(N)$	
swap A[i], A[i+1]	$\Theta(1)$	$\Theta(N^2)$	
i=i-1		$\Theta(1)$	$\Theta(N^2)$
}}			
return A		$\Theta(1)$	$\Theta(1)$

$$T(n) = \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(N^2) + \Theta(1) \times \Theta(N^2) + \Theta(1) \times \Theta(N^2) + \Theta(1) \times \Theta(1)$$

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

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

One more example: BinarySeach

Specification:

- **Input:** array a[0..n-1], integer key
- **Input property:** a is sorted
- **Output:** integer pos
- **Output property:** if key==a[i] then pos==I

BinarySearch – **worst case** asymptotic running time

Cost **No of times**

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

BinarySearch – **worst case** asymptotic running time

Code	Cost	No of times
`lo = 0, hi = a.length-1`	Θ(1)	Θ(1)
`while (lo <= hi) {`	Θ(1)	Θ(log n)
`  int mid = lo + (hi - lo) / 2`	Θ(1)	Θ(log n)
`  if (key < a[mid]) then hi = mid – 1`	Θ(1)	Θ(log n)
`  else if (key > a[mid]) then lo = mid + 1`	Θ(1)	Θ(log n)
`  else return mid`	Θ(1)	Θ(log n)
`}`		
`return -1`	Θ(1)	Θ(1)

$T(n) = \Theta(\log n)$

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:

```
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;
  }
```

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:

```
boolean isContained1(int[] A, int[] B) {                 Cost     No of times
  boolean AInB = true;                                              Θ(1)     Θ(1)
  for (int i = 0; i < A.length; i++) {                              Θ(1)     Θ(N)
    boolean iInB = linearSearch(B, A[i]);Θ(N)   Θ(N)
    AInB = AInB && iInB;                 Θ(1)   Θ(N)
  }
  return AInB;                                   Θ(1)     Θ(1)
}
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;
  }
```

Examples (comparisons)

- $\Theta(n \log n)$ =?= $\Theta(n)$

Examples (comparisons)

- $\Theta(n \log n) > \Theta(n)$
- $\Theta(n^2 + 3n - 1)$ =?= $\Theta(n^2)$

Examples (comparisons)

- $\Theta(n \log n) > \Theta(n)$
- $\Theta(n^2 + 3n - 1) = \Theta(n^2)$

Examples (comparisons)

- $\Theta(n \log n) > \Theta(n)$
- $\Theta(n^2 + 3n - 1) = \Theta(n^2)$
- $\Theta(1)$ =?= $\Theta(10)$
- $\Theta(5n)$ =?= $\Theta(n^2)$
- $\Theta(n^3 + \log(n))$ =?= $\Theta(100n^3 + \log(n))$
- Write all of the above in order, writing = or < between them

Examples (comparisons)

MyAlgorithm has an asymptotic worst case running time:

$$T(N) = \mathbf{O}(N^2 \lg N)$$

- Does that mean:
  - $T(N) = O(N^3)$?
  - $T(N) = O(N^2)$?
  - $T(N) = \Omega(N)$?
  - $T(N) = \Omega(N^3)$?
  - $T(N) = \Theta(N^2 \lg N)$?

Examples (comparisons)

MyAlgorithm has an asymptotic worst case running time:

$$T(N) = \mathbf{O}(N^2 \lg N)$$

- Does that mean:
  - $T(N) = O(N^3)$? YES
  - $T(N) = O(N^2)$? NO
  - $T(N) = \Omega(N)$? NO
  - $T(N) = \Omega(N^3)$? NO
  - $T(N) = \Theta(N^2 \lg N)$? NO

Examples (comparisons)

MyAlgorithm has an asymptotic worst case running time:

$$T(N) = \mathbf{\Omega}(N^2 \lg N)$$

- Does that mean :
  - $T(N) = O(N^3)$?
  - $T(N) = O(N^2)$?
  - $T(N) = \Omega(N)$?
  - $T(N) = \Omega(N^3)$?
  - $T(N) = \Theta(N^2 \lg N)$?

Examples (comparisons)

MyAlgorithm has an asymptotic worst case running time:

$$T(N) = \boldsymbol{\Omega}(N^2 \lg N)$$

- Does that mean:
  - $T(N) = O(N^3)$? NO
  - $T(N) = O(N^2)$? NO
  - $T(N) = \Omega(N)$? YES
  - $T(N) = \Omega(N^3)$? NO
  - $T(N) = \Theta(N^2 \lg N)$? NO

Examples (comparisons)

MyAlgorithm has an asymptotic worst case running time:

$$T(N) = \Theta(N^2 \lg N)$$

- Does that mean:
  - $T(N) = O(N^3)$?
  - $T(N) = O(N^2)$?
  - $T(N) = \Omega(N)$?
  - $T(N) = \Omega(N^3)$?

Examples (comparisons)

MyAlgorithm has an asymptotic worst case running time:

$$T(N) = \Theta(N^2 \lg N)$$

- Does that mean:
  - $T(N) = O(N^3)$? Yes
  - $T(N) = O(N^2)$? NO
  - $T(N) = \Omega(N)$? YES
  - $T(N) = \Omega(N^3)$? NO

*Because the above means MyAlgorithm is both $\mathbf{O}(N^2 \lg N)$ and $\mathbf{\Omega}(N^2 \lg N)$.