CS2010: ALGORITHMS AND DATA STRUCTURES

Lecture 4.1: More Asymptotic Notation

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin

Running Time Performance Analysis

3 Techniques:

- **Experimental** Running Time

- **Approximate** (using **cost models**) Running Time
  - Count some operations, basic op cost =1tu, tilde notation
  - Used in the book & lectures

- **Asymptotic** Running Time
  - Used in lectures and exams
  - We saw big-Theta (Θ) notation
  - Expresses the order of growth
  - Easier calculations in many common code patterns

Running Time Performance Analysis

3 Techniques:

- **Experimental** Running Time

- **Approximate** (using **cost models**) Running Time
  - Count some operations, basic op cost =1tu, tilde notation
  - Used in the book & lectures

- **Asymptotic** Running Time
  - Used in lectures and exams
  - We saw big-Theta (Θ) notation
  - Expresses the order of growth
  - Easier calculations in many common

Can calculate the running time for **3 possible kinds of input**:

1. Worst Case
2. Best Case
3. Average Case

InsertionSort – **asymptotic worst-case running time**

	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`	$\Theta(1)$	$\Theta(1)$

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

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

$= \Theta(N^2)$

InsertionSort – **asymptotic best-case running time?**

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 <u>best</u>-case running time**

Code	cost	No of times
`for j = 1 to A.length {`	Θ(1)	
` //shift A[j] into the sorted A[0..j-1]`		
` i=j-1`	Θ(1)	
` 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 best-case running time**

	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)	Θ(1)

InsertionSort – **asymptotic best-case running time**

Code	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)
`swap A[i], A[i+1]`	Θ(1)	
`i=i-1`	Θ(1)	
`}}`		
`return A`	Θ(1)	Θ(1)

InsertionSort – **asymptotic best-case running time**

	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)
`  swap A[i], A[i+1]`	Θ(1)	**0**
`  i=i-1`	Θ(1)	**0**
`}}`		
`return A`	Θ(1)	Θ(1)

InsertionSort – **asymptotic best-case running time**

	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)
`  swap A[i], A[i+1]`	Θ(1)	**0**
`  i=i-1`	Θ(1)	**0**
`}}`		
`return A`	Θ(1)	Θ(1)

T(n) = Θ(1) x Θ(N) + Θ(1) x Θ(N) + Θ(1) x Θ(1)

InsertionSort – **asymptotic best-case running time**

	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)
`  swap A[i], A[i+1]`	Θ(1)	**0**
`  i=i-1`	Θ(1)	**0**
`}}`		
`return A`	Θ(1)	Θ(1)

$$
\begin{aligned}
T(n) &= \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(1) \\
&= \Theta(N) + \Theta(N) + \Theta(1) \\
&= \Theta(N^2)
\end{aligned}
$$

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	$T(N)$

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
```

1	2	3	9	11	13	17	25	57	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

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) = Θ(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:

Code	Cost	No of times
`boolean isContained1(int[] A, int[] B) {`		
`  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

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.

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

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

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 $\Omega(N)$.

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.

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10\,N^2$	$10\,N^2$ $10\,N^2 + 22\,N \log N$ $10\,N^2 + 2\,N + 37$	provide approximate model
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2}\,N^2$ $10\,N^2$ $5\,N^2 + 22\,N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10\,N^2$ $100\,N$ $22\,N \log N + 3\,N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2}\,N^2$ N^5 $N^3 + 22\,N \log N + 3\,N$	develop lower bounds

Common mistake. Interpreting big-Oh as an approximate model.

This course. Focus on approximate models: use Tilde-notation

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

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

Basics

Bit. 0 or 1.

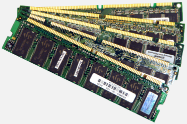
Byte. 8 bits.

NIST

most computer scientists

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

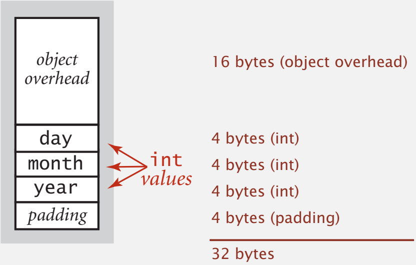
Object overhead. 16 bytes.

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;
...
}
```



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.