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 code patterns

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

1. for j = 1 to A.length {
2.   //shift A[j] into the sorted A[0..j-1]
3.   i=j-1
4.   while i>=0 and A[i]>A[i+1] {
5.     swap A[i], A[i+1]
6.     i=i-1
7.   }
8. return A

\[ T(n) = \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(N^2) + \Theta(1) \times \Theta(N^2) + \Theta(1) \times \Theta(N^2) + \Theta(1) \times \Theta(1) \]
\[ = \Theta(N) + \Theta(N) + \Theta(N^2) + \Theta(N^2) + \Theta(N^2) + \Theta(1) \]
\[ = \Theta(N^2) \]
InsertionSort – asymptotic best-case running time?

1. for j = 1 to A.length {
2.   //shift A[j] into the sorted A[0..j-1]
3.   i=j-1
4.   while i>=0 and A[i]>A[i+1] {
5.     swap A[i], A[i+1]
6.     i=i-1
7.   }
8.   return A
InsertionSort – asymptotic best-case running time

1. for j = 1 to A.length {
   2. //shift A[j] into the sorted A[0..j-1]
   3. i=j-1
   4. while i>=0 and A[i]>A[i+1] {
      5. swap A[i], A[i+1]
      6. i=i-1
   7. }
   8. return A
InsertionSort – asymptotic best-case running time

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>No of times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. for j = 1 to A.length {</td>
<td>Θ(1)</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>2. //shift A[j] into the sorted A[0..j-1]</td>
<td>0(1)</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>3. i = j - 1</td>
<td>Θ(1)</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>4. while i &gt;= 0 and A[i] &gt; A[i + 1] {</td>
<td>0(1)</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>5. swap A[i], A[i + 1]</td>
<td>0(1)</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>6. i = i - 1</td>
<td>0(1)</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>7. }</td>
<td>0(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>8. return A</td>
<td>0(1)</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>
**InsertionSort – asymptotic best-case running time**

1. for j = 1 to A.length {
2. //shift A[j] into the sorted A[0..j-1]
3. i = j - 1
4. while i >= 0 and A[i] > A[i+1] {
5. swap A[i], A[i+1]
6. i = i - 1
7. }
8. return A
InsertionSort – asymptotic best-case running time

1. for j = 1 to A.length {
2. //shift A[j] into the sorted A[0..j-1]
3. i=j-1
4. while i>=0 and A[i]>A[i+1] {
5. swap A[i], A[i+1]
6. i=i-1
7. }
8. return A
InsertionSort – asymptotic best-case running time

1. for j = 1 to A.length {
2.     //shift A[j] into the sorted A[0..j-1]
3.     i=j-1
4.     while i>=0 and A[i]>A[i+1] {
5.         swap A[i], A[i+1]
6.         i=i-1
7.     }
8. return A

\[ T(n) = \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(1) \]
InsertionSort – asymptotic best-case running time

1. for j = 1 to A.length {
2. //shift A[j] into the sorted A[0..j-1]
3. i=j-1
4. while i>=0 and A[i]>A[i+1] {
5. swap A[i], A[i+1]
6. i=i-1
7. }
8. return A

\[ T(n) = \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(N) + \Theta(1) \times \Theta(1) \]
\[ = \Theta(N) + \Theta(N) + \Theta(1) \]
\[ = \Theta(N^2) \]
## Common order-of-growth classifications

<table>
<thead>
<tr>
<th>order of growth</th>
<th>name</th>
<th>typical code framework</th>
<th>description</th>
<th>example</th>
<th>$T(2N) / T(N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>$a = b + c$;</td>
<td>statement</td>
<td>add two numbers</td>
<td>1</td>
</tr>
<tr>
<td>$\log N$</td>
<td>logarithmic</td>
<td>while $(N &gt; 1)$</td>
<td>divide in half</td>
<td>binary search</td>
<td>$\sim 1$</td>
</tr>
<tr>
<td>$N$</td>
<td>linear</td>
<td>for (int $i = 0; i &lt; N; i++$)</td>
<td>loop</td>
<td>find the maximum</td>
<td>2</td>
</tr>
<tr>
<td>$N \log N$</td>
<td>linearithmic</td>
<td>[see mergesort lecture]</td>
<td>divide and conquer</td>
<td>mergesort</td>
<td>$\sim 2$</td>
</tr>
<tr>
<td>$N^2$</td>
<td>quadratic</td>
<td>for (int $i = 0; i &lt; N; i++$)</td>
<td>double loop</td>
<td>check all pairs</td>
<td>4</td>
</tr>
<tr>
<td>$N^3$</td>
<td>cubic</td>
<td>for (int $i = 0; i &lt; N; i++$)</td>
<td>triple loop</td>
<td>check all triples</td>
<td>8</td>
</tr>
<tr>
<td>$2^N$</td>
<td>exponential</td>
<td>[see combinatorial search lecture]</td>
<td>exhaustive search</td>
<td>check all subsets</td>
<td>$T(N)$</td>
</tr>
</tbody>
</table>
One more example: Binary Search

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

1. \( \text{lo} = 0, \text{hi} = a.\text{length}-1 \)
2. while (\( \text{lo} \leq \text{hi} \)) {
3.     \( \text{int} \ \text{mid} = \text{lo} + (\text{hi} - \text{lo}) / 2 \)
4.     if (key < a[mid]) then hi = mid - 1
5.     else if (key > a[mid]) then lo = mid + 1
6.     else return mid
7. }
8. return -1
BinarySearch – worst case asymptotic running time

1. \( lo = 0, hi = a.length - 1 \)
2. while \( (lo <= hi) \) {
3. \( \text{int mid} = lo + (hi - lo) / 2 \)
4. if \( (key < a[mid]) \) then \( hi = mid - 1 \)
5. else if \( (key > a[mid]) \) then \( lo = mid + 1 \)
6. else return mid
7. }
8. return -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:

```java
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;
}
```

```java
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:

```java
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;
}
```

```
<table>
<thead>
<tr>
<th>Cost</th>
<th>No of times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>Θ(N)</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>Θ(N)</td>
</tr>
</tbody>
</table>
```

```java
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)

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

---

**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: $\sim 1$
- 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

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

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

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

**This course.** Focus on approximate models: use Tilde-notation
1.4 Analysis of Algorithms

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>0 or 1.</td>
</tr>
<tr>
<td>Byte</td>
<td>8 bits.</td>
</tr>
<tr>
<td>Megabyte (MB)</td>
<td>1 million or $2^{20}$ bytes.</td>
</tr>
<tr>
<td>Gigabyte (GB)</td>
<td>1 billion or $2^{30}$ bytes.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
</tbody>
</table>

#### Primitive types

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[]</td>
<td>$2N+24$</td>
</tr>
<tr>
<td>int[]</td>
<td>$4N+24$</td>
</tr>
<tr>
<td>double[]</td>
<td>$8N+24$</td>
</tr>
</tbody>
</table>

#### One-dimensional arrays

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[][]</td>
<td>$\sim 2MN$</td>
</tr>
<tr>
<td>int[][]</td>
<td>$\sim 4MN$</td>
</tr>
<tr>
<td>double[][]</td>
<td>$\sim 8MN$</td>
</tr>
</tbody>
</table>

#### 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.

```java
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.

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;
    }
    ...
}
```

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