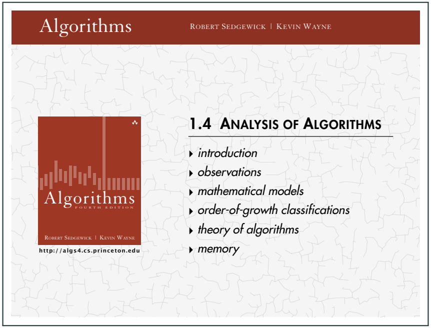
# CS2010: ALGORITHMS AND DATA STRUCTURES

## Lecture 3.1: Examples using Cost Models

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin

# LAST WEEK



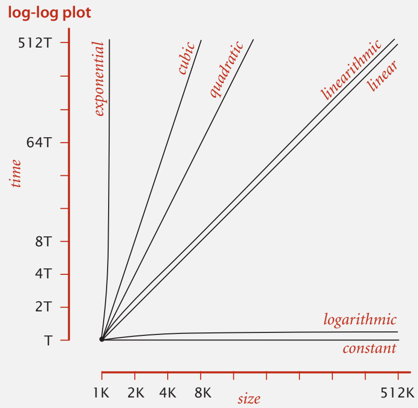

- → Estimate the performance of algorithms by
  - → Experiments & Observations
  - → Precise Mathematical Calculations
  - → Approximate Mathematical Calculations using Cost Models
    - → Every basic operation costs 1 time unit
    - → Keep only the higher-order terms
    - → Count only some operations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

# Common order-of-growth classifications

Good news. The set of functions

$$1, \ \log N, \ N, \ N \log N, \ N^2, \ N^3, \text{ and } 2^N$$

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)`<br>`{ N = N / 2; ... }` | divide in half | binary search | $\sim 1$ |
| $N$ | **linear** | `for (int i = 0; i < N; i++)`<br>`{ ... }` | loop | find the maximum | $2$ |
| $N \log N$ | **linearithmic** | [see mergesort lecture] | divide and conquer | mergesort | $\sim 2$ |
| $N^2$ | **quadratic** | `for (int i = 0; i < N; i++)`<br>`for (int j = 0; j < N; j++)`<br>`{ ... }` | double loop | check all pairs | $4$ |
| $N^3$ | **cubic** | `for (int i = 0; i < N; i++)`<br>`for (int j = 0; j < N; j++)`<br>`for (int k = 0; k < N; k++)`<br>`{ ... }` | 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: $N^3$

# TODAY

- Examples:
  - Binary Search
  - Insertion Sort

# 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. $T(N) \le T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.

left or right half (floored division)

possible to implement with one 2-way compare (instead of 3-way)

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

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

# 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.


http://algs4.cs.princeton.edu

# 1.4 Analysis of Algorithms

# 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.

this course

**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

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

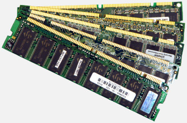
Byte. 8 bits.

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

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

NIST

most computer scientists



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

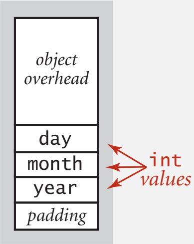
Object overhead. 16 bytes.

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

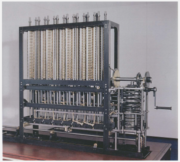
A. $8N + 88 \sim 8N$ bytes.

# 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.