CS2010: ALGORITHMS AND DATA STRUCTURES

Lecture 3: Cost Models of Running Time

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin
→ Estimate the performance of algorithms by
  → **Experiments** & Observations
    + Easy experiments
    - Works only for running times of the form $T(N) = aN^b$
  → **Precise Mathematical Calculations**
    + Works for any running time function
    - Tedious & difficult
Data analysis

**Log-log plot.** Plot running time $T(N)$ vs. input size $N$ using log-log scale.

Regression. Fit straight line through data points: $a N^b$.

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.
Example: 2-SUM

Q. How many instructions as a function of input size \( N \)?

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

\[
T(N) = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E + c_6 F
\]

\[
0 + 1 + 2 + \ldots + (N - 1) = \frac{1}{2} N (N - 1)
\]

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>=A</th>
<th>=B</th>
<th>=C</th>
<th>=D</th>
<th>=E</th>
<th>=F</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>( N + 2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>assignment statement</td>
<td>( N + 2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>less than compare</td>
<td>( \frac{1}{2} (N + 1) (N + 2) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>equal to compare</td>
<td>( \frac{1}{2} N (N - 1) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>array access</td>
<td>( N (N - 1) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>increment</td>
<td>( \frac{1}{2} N (N - 1) ) to ( N (N - 1) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

"best case" vs "worst case" input of size \( N \)
Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.

\[ T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E \]

- \( A \) = array access
- \( B \) = integer add
- \( C \) = integer compare
- \( D \) = increment
- \( E \) = variable assignment

Bottom line. We use approximate models in this course: \( T(N) \sim c N^3 \).
1. **Approximate Calculations** using different **Cost Models**

2. Classifying algorithms based on **order of growth**
Cost Model 1: all constant costs = 1
COST MODEL 1: ALL CONSTANT COSTS = 1

→ New generation computers have smaller constants than previous generation

\[ c_i = 1 \]

\[ T_N = A + B + C + D + E \]

Where

- \( A \) : number of array accesses
- \( B \) : number of integer additions
- \( C \) : number of integer comparisons
- \( D \) : number of increments
- \( E \) : number of assignments
Careful!

There are operations that do not have a constant cost:

→ Naive string concatenation: \( s = \text{str} + "ABCDEFG"; \)

→ Method calls: \( \text{max} = \text{Collections.max(myList)}; \)
Careful!

There are operations that do not have a constant cost:

→ Naive string concatenation: \( s = str + "ABCDEFG"; \)
  → the cost of this operation is linear to the size of \( str \)

→ Method calls: \( max = Collections.max(myList); \)
Careful!

There are operations that do not have a constant cost:

→ Naive string concatenation: \( s = str + "ABCDEFG"; \)
  → the cost of this operation is linear to the size of \( str \)
  → when efficiency is important use \{StringBuilder\}

→ Method calls: \( max = Collections.max(myList); \)
Careful!

There are operations that **do not** have a constant cost:

- Naive string concatenation: \( s = \text{str} + \text{"ABCDEFG"}; \)
  - the cost of this operation is linear to the size of \( \text{str} \)
  - when efficiency is important use `StringBuilder`

- Method calls: \( \text{max} = \text{Collections.max(myList)}; \)
  - the cost of this operation is the cost of running the algorithm in `Collections.max` with an input of size `myList.size()`. 
Example: 2-SUM

Q. How many instructions as a function of input size $N$?

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

Estimate performance by adding up frequencies

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>$N + 2$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>$N + 2$</td>
</tr>
<tr>
<td>less than compare</td>
<td>$\frac{1}{2} (N + 1) (N + 2)$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2} N (N - 1)$</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N - 1)$</td>
</tr>
<tr>
<td>increment</td>
<td>$\frac{1}{2} N (N - 1)$ to $N (N - 1)$</td>
</tr>
</tbody>
</table>

$0 + 1 + 2 + \ldots + (N - 1) = \frac{1}{2} N (N - 1) = \binom{N}{2}$
Cost model 2: only highest order terms count
Simplification 2: tilde notation

- Estimate running time (or memory) as a function of input size $N$.
- Ignore lower order terms.
  - when $N$ is large, terms are negligible
  - when $N$ is small, we don't care

Ex 1. \( \frac{1}{6} N^3 + 20N + 16 \approx \frac{1}{6} N^3 \)
Ex 2. \( \frac{1}{6} N^3 + 100N^{4/3} + 56 \approx \frac{1}{6} N^3 \)
Ex 3. \( \frac{1}{6} N^3 - \frac{1}{2} N^2 + \frac{1}{3} N \approx \frac{1}{6} N^3 \)

discard lower-order terms
(e.g., $N = 1000$: 166.67 million vs. 166.17 million)

Technical definition. $f(N) \sim g(N)$ means $\lim_{N\to\infty} \frac{f(N)}{g(N)} = 1$
Simplification 2: tilde notation

- Estimate running time (or memory) as a function of input size $N$.
- Ignore lower order terms.
  - when $N$ is large, terms are negligible
  - when $N$ is small, we don't care

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>tilde notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>$N + 2$</td>
<td>$\sim N$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>$N + 2$</td>
<td>$\sim N$</td>
</tr>
<tr>
<td>less than compare</td>
<td>$\frac{1}{2} (N + 1) (N + 2)$</td>
<td>$\sim \frac{1}{2} N^2$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2} N (N - 1)$</td>
<td>$\sim \frac{1}{2} N^2$</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N - 1)$</td>
<td>$\sim N^2$</td>
</tr>
<tr>
<td>increment</td>
<td>$\frac{1}{2} N (N - 1)$ to $N (N - 1)$</td>
<td>$\sim \frac{1}{2} N^2$ to $\sim N^2$</td>
</tr>
</tbody>
</table>

Estimate performance by adding up simplified frequencies
Cost model 3: count only SOME operations
Simplifying the calculations

“*It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings.*” — Alan Turing
Simplification 1: cost model

Cost model. Use some basic operation as a proxy for running time.

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

\[ 0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \binom{N}{2} \]

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>( N + 2 )</td>
</tr>
<tr>
<td>assignment statement</td>
<td>( N + 2 )</td>
</tr>
<tr>
<td>less than compare</td>
<td>( \frac{1}{2} (N + 1) (N + 2) )</td>
</tr>
<tr>
<td>equal to compare</td>
<td>( \frac{1}{2} N (N - 1) )</td>
</tr>
<tr>
<td><strong>array access</strong></td>
<td>( N (N - 1) )</td>
</tr>
<tr>
<td>increment</td>
<td>( \frac{1}{2} N (N - 1) ) to ( N (N - 1) )</td>
</tr>
</tbody>
</table>

Performance estimate = \((\text{array accesses}) \times c_{\text{arr access}}\)
Careful!

Make sure that the operations you are not counting add up to a factor lower than the operations you do count.
Combinations of Cost Models
Each cost model makes a simplification in the calculation of running time.

\[ \Rightarrow \text{approximation} \] of running time.

We can even combine the assumptions of different cost models.
Q. Approximately how many array accesses as a function of input size $N$?

A. $\sim N^2$ array accesses.

Performance estimate = simplified number of array accesses

Bottom line. Use cost model and tilde notation to simplify counts.
Example: 3-SUM

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

```java
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++;
```

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

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

Bottom line. Use cost model and tilde notation to simplify counts.
Diversion: estimating a discrete sum

Q. How to estimate a discrete sum?
A1. Take a discrete mathematics course.
A2. Replace the sum with an integral, and use calculus!

Ex 1. $1 + 2 + \ldots + N.$
\[
\sum_{i=1}^{N} i \sim \int_{x=1}^{N} x \, dx \sim \frac{1}{2} N^2
\]

Ex 2. $1^k + 2^k + \ldots + N^k.$
\[
\sum_{i=1}^{N} i^k \sim \int_{x=1}^{N} x^k \, dx \sim \frac{1}{k+1} N^{k+1}
\]

Ex 3. $1 + 1/2 + 1/3 + \ldots + 1/N.$
\[
\sum_{i=1}^{N} \frac{1}{i} \sim \int_{x=1}^{N} \frac{1}{x} \, dx = \ln N
\]

Ex 4. 3-sum triple loop.
\[
\sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=j}^{N} 1 \sim \int_{x=1}^{N} \int_{y=x}^{N} \int_{z=y}^{N} dz \, dy \, dx \sim \frac{1}{6} N^3
\]
Estimating a discrete sum

Q. How to estimate a discrete sum?
A1. Take a discrete mathematics course.
A2. Replace the sum with an integral, and use calculus!

Ex 4. $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$

\[
\sum_{i=0}^{\infty} \left( \frac{1}{2} \right)^i = 2
\]

\[
\int_{x=0}^{\infty} \left( \frac{1}{2} \right)^x \, dx = \frac{1}{\ln 2} \approx 1.4427
\]

Caveat. Integral trick doesn't always work!
Estimating a discrete sum

**Q.** How to estimate a discrete sum?

**A3.** Use Maple or Wolfram Alpha.
Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.

\[
T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E
\]

\[
A = \text{array access}
\]
\[
B = \text{integer add}
\]
\[
C = \text{integer compare}
\]
\[
D = \text{increment}
\]
\[
E = \text{variable assignment}
\]

Bottom line. We use approximate models in this course: \( T(N) \sim c N^3 \).
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 $N^3$.

```c
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.
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.

Common order-of-growth classifications

**Typical orders of growth**

- **log-log plot**
- **standard plot**
- **cubic**
- **quadratic**
- **linearithmic**
- **linear**
- **exponential**
- **logarithmic**
- **constant**
## 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)) { (N = N / 2; ) ... }</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++)) { ... } \for ((int j = 0; j &lt; N; j++)) { ... }</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++)) \for ((int j = 0; j &lt; N; j++)) \for ((int k = 0; k &lt; N; k++)) { ... }</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>
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

<table>
<thead>
<tr>
<th>6</th>
<th>13</th>
<th>14</th>
<th>25</th>
<th>33</th>
<th>43</th>
<th>51</th>
<th>53</th>
<th>64</th>
<th>72</th>
<th>84</th>
<th>93</th>
<th>95</th>
<th>96</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

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.

```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;
}
```

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 $\leq N$.

**Binary search recurrence.** $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.

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

\[
T(N) \leq T(N/2) + 1 \quad \text{[ given ]}
\]
\[
\leq T(N/4) + 1 + 1 \quad \text{[ apply recurrence to first term ]}
\]
\[
\leq T(N/8) + 1 + 1 + 1 \quad \text{[ apply recurrence to first term ]}
\]
\[
\vdots
\]
\[
\leq T(N/N) + 1 + 1 + \ldots + 1 \quad \text{[ stop applying, $T(1) = 1$ ]}
\]
\[
= 1 + \lg N
\]
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**

<table>
<thead>
<tr>
<th>Pair</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-40, -20)</td>
<td>60</td>
</tr>
<tr>
<td>(-40, -10)</td>
<td>50</td>
</tr>
<tr>
<td>(-40, 0)</td>
<td>40</td>
</tr>
<tr>
<td>(-40, 5)</td>
<td>35</td>
</tr>
<tr>
<td>(-40, 10)</td>
<td>30</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(-20, -10)</td>
<td>30</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(-10, 0)</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(10, 30)</td>
<td>-40</td>
</tr>
<tr>
<td>(10, 40)</td>
<td>-50</td>
</tr>
<tr>
<td>(30, 40)</td>
<td>-70</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.1</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
</tbody>
</table>

**ThreeSum.java**

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.14</td>
</tr>
<tr>
<td>2,000</td>
<td>0.18</td>
</tr>
<tr>
<td>4,000</td>
<td>0.34</td>
</tr>
<tr>
<td>8,000</td>
<td>0.96</td>
</tr>
<tr>
<td>16,000</td>
<td>3.67</td>
</tr>
<tr>
<td>32,000</td>
<td>14.88</td>
</tr>
<tr>
<td>64,000</td>
<td>59.16</td>
</tr>
</tbody>
</table>

**ThreeSumDeluxe.java**

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