CS062
DATA STRUCTURES AND ADVANCED PROGRAMMING

8: Analysis of Algorithms

Alexandra Papoutsaki
she/her/hers
Lecture 8: Analysis of Algorithms

- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of ArrayList operations
Different Roles

Programmer needs a working solution
User wants an efficient solution
Theoretician wants to understand
3-SUM: Given $n$ distinct numbers, how many unordered triplets sum to 0?

- Input: 30 -40 -20 -10 40 0 10 5
- Output: 4
  - 30 -40 10
  - 30 -20 -10
  - -40 40 0
  - -10 0 10
3-SUM: Brute force algorithm

```java
class ThreeSum {
  public static int count(int[] a) {
    int n = a.length;
    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++;
          }
        }
      }
    }
    return count;
  }
  public static void main(String[] args) {
    String filename = args[0];
    int fileSize = Integer.parseInt(args[1]);
    try {
      Scanner scanner = new Scanner(new File(filename));
      int[] intList = new int[fileSize];
      int i=0;
      while(scanner.hasNextInt()){
        intList[i]=scanner.nextInt();
        i++;
      }
      Stopwatch timer = new Stopwatch();
      int count = count(intList);
      System.out.println("elapsed time = " + timer.elapsedTime());
      System.out.println(count);
    } catch (IOException e) {
      throw new IllegalArgumentException("Could not open " + filename, e);
    }
  }
}
```
## Empirical Analysis

<table>
<thead>
<tr>
<th>Input size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0.081</td>
</tr>
<tr>
<td>2000</td>
<td>0.38</td>
</tr>
<tr>
<td>2000</td>
<td>0.371</td>
</tr>
<tr>
<td>4000</td>
<td>2.792</td>
</tr>
<tr>
<td>8000</td>
<td>21.623</td>
</tr>
<tr>
<td>16000</td>
<td>177.344</td>
</tr>
</tbody>
</table>
EXPERIMENTAL ANALYSIS OF RUNNING TIME

Plots and log-log plots

Regression: \( T(n) = an^b \) (power-law).

\[ \log T(n) = b \log n + \log a, \text{ where } b \text{ is slope.} \]

Experimentally: \( \sim 0.42 \times 10^{-10}n^3 \), in our example for ThreeSum.
Doubling Hypothesis

- Doubling input size increases running time by a factor of $\frac{T(n)}{T(n/2)}$.

- Run program doubling the size of input. Estimate factor of growth:
  $$\frac{T(n)}{T(n/2)} = \frac{an^b}{a(\frac{n}{2})^b} = 2^b.$$  

- E.g., in our example, for pair of input sizes 8000 and 16000 the ratio $(\frac{177.344}{21.623})$ is 8.2 or \sim8 which can be written as $2^3$, therefore $b$ is approximately 3.

- Assuming we know $b$, we can figure out $a$.
  - E.g., in our example, $T(16000) = 177.34 = a \times 16000^3$.
    - Solving for $a$ we get $a = 0.42 \times 10^{-10}$. 

<table>
<thead>
<tr>
<th>Input size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0.081</td>
</tr>
<tr>
<td>2000</td>
<td>0.38</td>
</tr>
<tr>
<td>4000</td>
<td>2.792</td>
</tr>
<tr>
<td>8000</td>
<td>21.623</td>
</tr>
<tr>
<td>16000</td>
<td>177.344</td>
</tr>
</tbody>
</table>

**EXPERIMENTAL ANALYSIS OF RUNNING TIME**
Suppose you time your code and you make the following observations. Which function is the closest model of \( T(n) \)?

A. \( n^2 \)

B. \( 6 \times 10^{-4}n \)

C. \( 5 \times 10^{-9}n^2 \)

D. \( 7 \times 10^{-9}n^2 \)

<table>
<thead>
<tr>
<th>Input size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0.0</td>
</tr>
<tr>
<td>4000</td>
<td>0.1</td>
</tr>
<tr>
<td>8000</td>
<td>0.3</td>
</tr>
<tr>
<td>16000</td>
<td>1.3</td>
</tr>
<tr>
<td>32000</td>
<td>5.1</td>
</tr>
</tbody>
</table>
EXPERIMENTAL ANALYSIS OF RUNNING TIME

ANSWER

- C. $5 \times 10^{-9}n^2$
- $T(32000)/T(16000)$ is approximately 4, therefore $b = 2$.
- $T(32000) = 5.1 = a \times 32000^2$.
- Solving for $a = 4.98 \times 10^{-9}$.

<table>
<thead>
<tr>
<th>Input size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0.0</td>
</tr>
<tr>
<td>4000</td>
<td>0.1</td>
</tr>
<tr>
<td>8000</td>
<td>0.3</td>
</tr>
<tr>
<td>16000</td>
<td>1.3</td>
</tr>
<tr>
<td>32000</td>
<td>5.1</td>
</tr>
</tbody>
</table>
Effects on Performance

- **System independent effects**: Algorithm + input data
  - Determine $b$ in power law relationships.
- **System dependent effects**: Hardware (e.g., CPU, memory, cache) + Software (e.g., compiler, garbage collector) + System (E.g., operating system, network, etc).
  - Dependent and independent effects determine $a$ in power law relationships.

- Although it is hard to get precise measurements, experiments in Computer Science are cheap to run.
Lecture 8: Analysis of Algorithms

- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of ArrayList operations
Total Running Time

- Popularized by Donald Knuth in the 60s in the four volumes of “The Art of Computer Programming”.
  - Knuth won the Turing Award (The “Nobel” in CS) in 1974.

- In principle, accurate mathematical models for performance of algorithms are available.

- Total running time = sum of cost x frequency for all operations.
- Need to analyze program to determine set of operations.
- Exact cost depends on machine, compiler.
- Frequency depends on algorithm and input data.
Cost of Basic Operations

Add < integer multiply < integer divide < floating-point add < floating-point multiply < floating-point divide.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable declaration</td>
<td>int a</td>
<td>$c_1$</td>
</tr>
<tr>
<td>Assignment statement</td>
<td>$a = b$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>Integer comparison</td>
<td>$a &lt; b$</td>
<td>$c_3$</td>
</tr>
<tr>
<td>Array element access</td>
<td>$a[i]$</td>
<td>$c_4$</td>
</tr>
<tr>
<td>Array length</td>
<td>$a$.length</td>
<td>$c_5$</td>
</tr>
<tr>
<td>1D array allocation</td>
<td>new int[n]</td>
<td>$c_6n$</td>
</tr>
<tr>
<td>2D array allocation</td>
<td>new int[n][n]</td>
<td>$c_7n^2$</td>
</tr>
<tr>
<td>String concatenation</td>
<td>$s+t$</td>
<td>$c_8n$</td>
</tr>
</tbody>
</table>
MATHEMATICAL MODELS OF RUNNING TIME

Example: 1-SUM

- How many operations as a function of $n$?

```c
int count = 0;
for (int i = 0; i < n; i++) {
    if (a[i] == 0) {
        count++;
    }
}
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable declaration</td>
<td>2</td>
</tr>
<tr>
<td>Assignment</td>
<td>2</td>
</tr>
<tr>
<td>Less than</td>
<td>$n + 1$</td>
</tr>
<tr>
<td>Equal to</td>
<td>$n$</td>
</tr>
<tr>
<td>Array access</td>
<td>$n$</td>
</tr>
<tr>
<td>Increment</td>
<td>$n$ to $2n$</td>
</tr>
</tbody>
</table>
Example: 2-SUM  

\[ 1 + 2 + 3 + \ldots + n = \frac{n(n + 1)}{2} \]

How many operations as a function of \( n \)?

```
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++;
        }
    }
}
```

<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</td>
<td>( n + 2 )</td>
</tr>
<tr>
<td>Less than</td>
<td>( \frac{(n + 1)(n + 2)}{2} )</td>
</tr>
<tr>
<td>Equal to</td>
<td>( \frac{n(n - 1)}{2} )</td>
</tr>
<tr>
<td>Array access</td>
<td>( n(n - 1) )</td>
</tr>
<tr>
<td>Increment</td>
<td>( \frac{n(n + 1)}{2} ) to ( n^2 )</td>
</tr>
</tbody>
</table>
MATHEMATICAL MODELS OF RUNNING TIME

Tilde Notation

- Estimate running time (or memory) as a function of input size $n$.
- Ignore lower order terms.
  - When $n$ is large, lower order terms become negligible.

Example 1: $\frac{1}{6}n^3 + 10n + 100 \sim n^3$

Example 2: $\frac{1}{6}n^3 + 100n^2 + 47 \sim n^3$

Example 3: $\frac{1}{6}n^3 + 100n^\frac{2}{3} + \frac{1}{2n} \sim n^3$
MATHEMATICAL MODELS OF RUNNING TIME

Simplification

- **Cost model**: Use some basic operation as proxy for running time. E.g., array accesses
- Combine it with tilde notation.

<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</td>
<td>$n + 2$</td>
<td>$\sim n$</td>
</tr>
<tr>
<td>Less than</td>
<td>$(n + 1)(n + 2)/2$</td>
<td>$\sim n^2$</td>
</tr>
<tr>
<td>Equal to</td>
<td>$n(n - 1)/2$</td>
<td>$\sim n^2$</td>
</tr>
<tr>
<td>Array access</td>
<td>$n(n - 1)$</td>
<td>$\sim n^2$</td>
</tr>
<tr>
<td>Increment</td>
<td>$n(n + 1)/2$ to $n^2$</td>
<td>$\sim n^2$</td>
</tr>
</tbody>
</table>

- $\sim n^2$ array accesses for the 2-SUM problem
Back to the 3-SUM problem

- Approximately how many array accesses 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++) {
        for (int k = j + 1; k < n; k++) {
            if (a[i] + a[j] + a[k] == 0) {
                count++;
            }
        }
    }
}
```

\[
\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n-1} 3 = \frac{1}{2} n(n^2 - 3n + 2) \sim n^3 \text{ array accesses.}
\]
Lecture 8: Analysis of Algorithms

- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of ArrayList operations
ORDER OF GROWTH CLASSIFICATION

Types of analysis

- **Best case**: lower bound on cost.
  - What the goal of all inputs should be.
  - Often not realistic, only applies to “easiest” input.

- **Worst case**: upper bound on cost.
  - Guarantee on all inputs.
  - Calculated based on the “hardest” input.

- **Average case**: expected cost for random input.
  - A way to predict performance.
  - Not straightforward how we model random input.
ORDER OF GROWTH CLASSIFICATION

Worst case analysis

- **Definition**: If \( f(n) \sim cg(n) \) for some constant \( c > 0 \), then the order of growth of \( f(n) \) is \( g(n) \).
  - Ignore leading coefficients.
  - Ignore lower-order terms.

- We will be using the big-Oh (O) notation. For example:
  - \( 3n^3 + 2n + 7 = O(n^3) \)
  - \( 2^n + n^2 = O(2^n) \)
  - \( 1000 = O(1) \)

- Yes, \( 3n^3 + 2n + 7 = O(n^6) \), but that’s a rather useless bound.
ORDER OF GROWTH CLASSIFICATION

Common order of growth classifications

- **Good news**: only a small number of function suffice to describe the order-of-growth of typical algorithms.
- 1: constant
  - Doubling the input size won’t affect the running time. Holy-grail.
- \( \log n \): logarithmic
  - Doubling the input size will increase the running time by a constant.
- \( n \): linear
  - Doubling the input size will result to double the running time.
- \( n \log n \): linearithmic
  - Doubling the input size will result to a bit longer than double the running time.
- \( n^2 \): quadratic
  - Doubling the input size will result to four times as much running time.
- \( n^3 \): cubic
  - Doubling the input size will result to eight times as much running time.
- \( 2^n \): exponential
  - When you increase the input by some constant amount, the time taken is doubled.
- \( n! \): factorial
  - Running time grows exponentially with the size of the input.
ORDER OF GROWTH CLASSIFICATION

From slowest growing to fastest growing

- $1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$
Common order of growth classifications

<table>
<thead>
<tr>
<th>Order-of-growth</th>
<th>Name</th>
<th>Example code</th>
<th>$T(n)/T(n/2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant</td>
<td>$a[i]=b+c$</td>
<td>1</td>
</tr>
<tr>
<td>$\log n$</td>
<td>Logarithmic</td>
<td>while($n&gt;1$)lbrace $n=n/2;...$rbrace</td>
<td>$\sim 1$</td>
</tr>
<tr>
<td>$n$</td>
<td>Linear</td>
<td>for(int $i=0$; $i&lt;n$; $i++$)</td>
<td>2</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>Linearithmic</td>
<td>for (i = 1; i &lt;= n; i++){</td>
<td>$\sim 2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int $x = n$; while ($x &gt; 0$) $x -= i;$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>$n^2$</td>
<td>Quadratic</td>
<td>for(int $i=0$; $i&lt;n$; $i++$) {</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for(int $j=0$; $j&lt;n$; $j++$){</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>$n^3$</td>
<td>Cubic</td>
<td>for(int $i=0$; $i&lt;n$; $i++$) {</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for(int $j=0$; $j&lt;n$; $j++$){</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>for(int $k=0$; $k&lt;n$; $k++$){</td>
<td></td>
</tr>
</tbody>
</table>
Useful approximations

- Harmonic sum: $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n \sim \ln n$
- Triangular sum: $1 + 2 + 3 + \ldots + n \sim n^2$
- Geometric sum: $1 + 2 + 4 + 8 + \ldots + n = 2n - 1 \sim n$, when $n$ power of 2.
- Binomial coefficients: $\binom{n}{k} \sim \frac{n^k}{k!}$ when $k$ is a small constant.
- Use a tool like Wolfram alpha.
Lecture 8: Analysis of Algorithms

- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of ArrayList Operations
Worst-case performance of \texttt{add()} is $O(n)$

- **Cost model**: 1 for insertion, $n$ for copying $n$ items to a new array.
- **Worst-case**: If ArrayList is full, \texttt{add()} will need to call \texttt{resize} to create a new array of double the size, copy all items, insert new one.
- **Total cost**: $n + 1 = O(n)$.

- Realistically, this won’t be happening often and worst-case analysis can be too strict. We will use amortized time analysis instead.
Amortized analysis

*Amortized cost per operation*: for a sequence of \( n \) operations, it is the total cost of operations divided by \( n \).
Amortized analysis for $n$ add() operations

- As the ArrayList increases, doubling happens half as often but costs twice as much.
- $O(\text{total cost}) = \sum (\text{“cost of insertions”}) + \sum (\text{“cost of copying”})$
- $\sum (\text{“cost of insertions”}) = n.$
- $\sum (\text{“cost of copying”}) = 1 + 2 + 2^2 + \ldots 2^{\lfloor \log_2 n \rfloor} \leq 2n.$
- $O(\text{total cost}) \leq 3n$, therefore amortized cost is $\leq \frac{3n}{n} = 3 = O^+(1)$, but “lumpy”.

<table>
<thead>
<tr>
<th>Insertion Cost</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copying Cost</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total Cost</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Amortized analysis for \( n \) \texttt{add()} operations when increasing ArrayList by 1.

\[ \sum \text{("cost of insertions")} = n. \]
\[ \sum \text{("cost of copying")} = 1 + 2 + 3 + \ldots + n - 1 = n(n - 1)/2. \]
\[ O(\text{total cost}) = n + n(n - 1)/2 = n(n + 1)/2, \text{ therefore amortized cost is } (n + 1)/2 \text{ or } O^+(n). \]

Same idea when increasing ArrayList size by a constant.

\[ \text{This is why in the lab on Friday, we saw that doubling was the fastest and linear(1) the slowest.} \]
Lecture 8: Analysis of Algorithms

- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of ArrayList operations
Readings:

- Recommended Textbook:
  - Chapter 1.4 (pages 172-205)

- Recommended Textbook Website:

Code

- Lecture 8 code

Practice Problems:

- 1.4.1-1.4.9, 1.4.32, 1.4.35-1.4.36