CS062
DATA STRUCTURES AND ADVANCED PROGRAMMING
5: Analysis of Algorithms

Alexandra Papoutsaki
she/her/hers
Lecture 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption
Different Roles

- **Programmer**: needs a working solution
- **User**: Wants an efficient solution
- **Theoretician**: Wants to understand

**You**

`needs a working solution`  `Wants an efficient solution`  `Wants to understand`
Why analyze algorithmic efficiency?

- Predict performance.
- Compare algorithms that solve the same problem.
- Provide guarantees.
- Understand theoretical basis.
- *Avoid performance bugs.*

Why is my program so slow? Why does it run out of memory?

We can use a combination of experiments and mathematical modeling.
Lecture 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption
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
public 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();
            }
            Stopwatch timer = new Stopwatch();
            int count = count(intList);
            System.out.println("elapsed time = " + timer.elapsedTime());
            System.out.println(count);
        } catch (IOException ioe) {
            throw new IllegalArgumentException("Could not open " + filename, ioe);
        }
    }
}
```
## Experimental 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>

- Input: 8ints.txt
  - Output: 4 and 0

- Input: 1Kints.txt
  - Output: 70 and 0.081

- Input: 2Kints.txt
  - Output: 528 and 0.38

- Input: 2Kints.txt
  - Output: 528 and 0.371

- Input: 4Kints.txt
  - Output: 4039 and 2.792

- Input: 8Kints.txt
  - Output: 32074 and 21.623

- Input: 16Kints.txt
  - Output: 255181 and 177.344
EXPERIMENTAL ANALYSIS OF RUNNING TIME

Plots and log-log plots

- Regression: $T(n) = an^b$ (power-law).
- $\log T(n) = b \log n + \log a$, where $b$ is slope.
- Experimentally: $\sim 0.42 \times 10^{-10}n^3$, in our example for ThreeSum.
EXPERIMENTAL ANALYSIS OF RUNNING TIME

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:
  \[\frac{T(n)}{T(n/2)} = \frac{an^b}{a(n/2)^b} = 2^b.\]
- E.g., in our example, for pair of input sizes 8000 and 16000 the ratio \(\frac{177.344}{21.623}\) is 8.2 or \(~8\) 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 \(a = 0.42 \times 10^{-10}\).
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>
Answer

- C. $5 \times 10^{-9}n^2$
- Ratio 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 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption
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>
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

How many operations as a function of $n$?

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

<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>$1/2(n + 1)(n + 2)$</td>
</tr>
<tr>
<td>Equal to</td>
<td>$1/2n(n - 1)$</td>
</tr>
<tr>
<td>Array access</td>
<td>$n(n - 1)$</td>
</tr>
<tr>
<td>Increment</td>
<td>$1/2n(n + 1)$ 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: \( \frac{1}{6}n^3 + 100n^\frac{2}{3} + \frac{1/2}{n} \sim n^3 \)

- Technically \( f(n) \sim g(n) \) means that \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1 \)
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>$1/2(n + 1)(n + 2)$</td>
<td>$\sim n^2$</td>
</tr>
<tr>
<td>Equal to</td>
<td>$1/2n(n - 1)$</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>$1/2n(n + 1)$ 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++;
            }
        }
    }
}
```

- $n^3$ array accesses.
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 \( n \) is a power of 2.
- **Binomial coefficients**: \( \begin{pmatrix} n \\ k \end{pmatrix} \sim \frac{n^k}{k!} \) when \( k \) is a small constant.

- Use a tool like Wolfram alpha.
How many array accesses does the following code make?

```java
int count = 0;
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        for (int k = 1; k < n; k=2) {
            if (a[i] + a[j] >= a[k]) {
                count++;
            }
        }
    }
}
```

A. $n^2$
B. $n^2 \log n$
C. $n^3$
D. $n^3 \log n$
Answer

- B. $n^2 \log n$
Lecture 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption
Order of growth

- **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 use this definition in the mathematical analysis of the running time of our programs as the coefficients depend on the system.
- E.g., the order of growth of the running time of the ThreeSum program is \( n^3 \).
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 run-time. Holy-grail
- **log n**: logarithmic
  - Doubling the input size, will increase the runtime by a constant.
- **n**: linear
  - Doubling the input size, will result to double the run-time.
- **n log n**: linearithmic
  - Doubling the input size, will result to a bit longer than double the run-time.
- **n^2**: quadratic
  - Doubling the input size, will result to four times as much run-time.
- **n^3**: cubic
  - Doubling the input size, will result to eight times as much run-time.
- **2^n**: exponential
  - When you increase the input by some constant amount, the time taken is doubled.
- **n!**: factorial
  - Runtime grows exponentially with the size of the input
ORDER OF GROWTH CLASSIFICATION

From slowest growing to fastest growing

- $1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$
# ORDER OF GROWTH CLASSIFICATION

## 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>$a=b+c$</td>
<td>1</td>
</tr>
<tr>
<td>$\log n$</td>
<td>Logarithmic</td>
<td><code>while(n&gt;1){n=n/2;...}</code></td>
<td>~ 1</td>
</tr>
<tr>
<td>$n$</td>
<td>Linear</td>
<td><code>for(int i=0; i&lt;n; i++{</code></td>
<td>2</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>Linearithmic</td>
<td><code>for (i = 1; i &lt;= n; I++){</code></td>
<td>~ 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>int x = n;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>while (x &gt; 0)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>x -= i;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>$n^2$</td>
<td>Quadratic</td>
<td><code>for(int i=0; i&lt;n; i++)</code></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>for(int j=0; j&lt;n; j++){</code></td>
<td></td>
</tr>
<tr>
<td>$n^3$</td>
<td>Cubic</td>
<td><code>for(int i=0; i&lt;n; i++)</code></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>for(int j=0; j&lt;n; j++){</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>for(int k=0; k&lt;n; k++){</code></td>
<td></td>
</tr>
</tbody>
</table>
Lecture 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption
Basics

› **Bit**: 0 or 1.
› **Byte**: 8 bits.
› **Megabyte (MB)**: $2^{20}$ bytes.
› **Gigabyte**: $2^{30}$ bytes.
Typical use of memory for primitives and arrays

- **boolean**: 1 byte
- **byte**: 1 byte
- **char**: 2 bytes
- **short**: 2 bytes
- **int**: 4 bytes
- **float**: 4 bytes
- **long**: 8 bytes
- **double**: 8 bytes

Array overhead: 24 bytes

- **char[n]**: 2n+24 bytes
- **int[n]**: 4n+24 bytes
- **double[n]**: 8n+24 bytes
Typical use of memory for objects

- Object overhead: 16 bytes
- Reference: 8 bytes
- Padding: padded to be a multiple of 8 bytes
- Example:
  ```java
  public class Date {
    private int day;
    private int month;
    private int year;
  }
  ```
  - 16 bytes overhead + 3x4 bytes for ints + 4 bytes padding = 32 bytes
How much memory does WeightedQuickUnionUF use as a function of $n$?

```java
public class WeightedQuickUnionUF{
    private int[] parent;
    private int[] size;
    private int count;

    public WeightedQuickUnionUF(int n) {
        parent = new int[n];
        size = new int[n];
        count = 0;
    }
}
```

A. $\sim 4n$ bytes  
B. $\sim 8n$ bytes  
C. $\sim 4n^2$ bytes  
D. $\sim 8n^2$ bytes
Answer

B. \( \sim 8n \) bytes

- 16 bytes for object overhead
- Each array: 8 bytes for reference + 24 overhead + 4n for integers
- 4 bytes for int
- 4 bytes for padding
- Total \( 88 + 8n \sim 8n \)
Lecture 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption
Readings:

- Recommended Textbook:
  - Chapter 1.4 (pages 172-196, 200-205)

- Recommended Textbook Website:

Code

- Lecture 5 code

Practice Problems:

- 1.4.1-1.4.9