9/10/2018
Algorithms & Data Structures
Analysis of Algorithms
Email me if the office door is closed
Grades have been emailed

github.com/COSC311/assignment01-<userid>
Check out my latest commit for the comments and changes.

Click on the latest commit, and you can
This class meeting

• Algorithm Analysis
  • Big O, Big Omega, Big Theta

• Recursion

• Useful data structure: ArrayList
Algorithm: a step by step procedure

```java
ArrayList<Integer> values = new ArrayList<>();

public Average0(int windowSize) {
    this.windowSize = windowSize;
}

public double next(int val) {
    values.add(val);

    if (values.size() <= windowSize) {
        Integer sum = 0;
        for (int v : values) {
            sum = sum + v;
        }
        return sum.doubleValue() / values.size();
    }

    this.values.remove(index: 0);
    Integer sum = 0;
    for (int v : values) {
        sum = sum + v;
    }
    return sum.doubleValue() / windowSize;
}
```

Data structure: a systematic way to organize the data
Algorithms

Washtenaw Ave.
Algorithm Analysis

Analyze what? **Time** and space

Analogy: ways of transportation

Washtenaw Ave.
Different algorithms

Examples of algorithms:
Ways of sorting

elementary sorts, mergesort, **quick sort**, ...
Different algorithms

Examples of algorithms:
Ways of sorting

26 year-old
In 1959

elementary sorts, mergesort, **quick sort**, priority queue sorting
Why is the running time important?

30 trillion web pages
Why is the running time important?

30 trillion web pages
30,000,000,000,000 web pages
Why is the running time important?

30 trillion web pages
30,000,000,000,000 web pages

Suppose we have an algorithm to search these web pages
The speed of the algorithm is:
1000 web pages in 1/1000 second

How long will the algorithm take to find a web page?
Why is the running time important?

30 trillion web pages
30,000,000,000,000 web pages

Suppose we have an algorithm to search these web pages
The speed of the algorithm is:
1000 web pages in $\frac{1}{1000}$ second

How long will the algorithm take to find a web page?
~ 347 days
Analysis of Algorithm

• Most algorithms transform input objects into output objects.
• The running time of an algorithm typically grows with the input size.
• Average case time is often difficult to determine.
• We focus on the worst case running time.
• Easier to analyze
• Crucial to applications such as games, finance and robotics
How do we measure?

Experimental Studies

1 \texttt{long} startTime = System.currentTimeMillis(); // record the starting time
2 \texttt{/* (run the algorithm) */}
3 \texttt{long} endTime = System.currentTimeMillis(); // record the ending time
4 \texttt{long} elapsed = endTime - startTime; // compute the elapsed time
Experimental Process

1st step: implementation
```java
/* Uses repeated concatenation to compose a String with n copies of character c. */

public static String repeat1(char c, int n) {
    String answer = "";
    for (int j=0; j < n; j++)
        answer += c;
    return answer;
}

/* Uses StringBuilder to compose a String with n copies of character c. */

public static String repeat2(char c, int n) {
    StringBuilder sb = new StringBuilder();
    for (int j=0; j < n; j++)
        sb.append(c);
    return sb.toString();
}

Code Fragment 4.2: Two algorithms for composing a string of repeated characters.
```
Experimental Process

$2^{nd}$ step: set up monitoring method
1 long startTime = System.currentTimeMillis(); // record the starting time
2 /* (run the algorithm) */
3 long endTime = System.currentTimeMillis(); // record the ending time
4 long elapsed = endTime - startTime; // compute the elapsed time

repeat1(‘*’, 50000)
repeat2(‘*’, 50000)
Experimental Process

3rd step: run the implementation with different inputs
<table>
<thead>
<tr>
<th>( n )</th>
<th>( \text{repeat1 (in ms)} )</th>
<th>( \text{repeat2 (in ms)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>50,000</td>
<td>2,884</td>
<td>1</td>
</tr>
<tr>
<td>100,000</td>
<td>7,437</td>
<td>1</td>
</tr>
<tr>
<td>200,000</td>
<td>39,158</td>
<td>2</td>
</tr>
<tr>
<td>400,000</td>
<td>170,173</td>
<td>3</td>
</tr>
<tr>
<td>800,000</td>
<td>690,836</td>
<td>7</td>
</tr>
<tr>
<td>1,600,000</td>
<td>2,874,968</td>
<td>13</td>
</tr>
<tr>
<td>3,200,000</td>
<td>12,809,631</td>
<td>28</td>
</tr>
<tr>
<td>6,400,000</td>
<td>59,594,275</td>
<td>58</td>
</tr>
<tr>
<td>12,800,000</td>
<td>265,696,421</td>
<td>135</td>
</tr>
</tbody>
</table>
Assignment (part 1/3) – P.187, P-4.60 – deadline Friday 10 pm

```java
/** Returns an array `a` such that, for all `j`, `a[j]` equals the average of `x[0]`, ..., `x[j]`. */
public static double[] prefixAverage1(double[] x) {
    int n = x.length;
    double[] a = new double[n]; // filled with zeros by default
    for (int j=0; j < n; j++) {
        double total = 0; // begin computing `x[0] + ... + x[j]
        for (int i=0; i <= j; i++)
            total += x[i];
        a[j] = total / (j+1); // record the average
    }
    return a;
}
```
Assignment (part 2/3) – P.187, P-4.60 – deadline Friday 10 pm

```java
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[] prefixAverage2(double[] x) {
    int n = x.length;
    double[] a = new double[n]; // filled with zeros by default
    double total = 0; // compute prefix sum as x[0] + x[1] + ...
    for (int j=0; j < n; j++) {
        total += x[j]; // update prefix sum to include x[j]
        a[j] = total / (j+1); // compute average based on current sum
    }
    return a;
}
```
Assignment (part 3/3) – deadline Friday 10 pm

Perform an experimental analysis of the two algorithms prefixAverage1 and prefixAverage2, from Section 4.3.4. Visualize their running times as a function of the input size with a log-log chart (using Excel or other software tools. You don’t have to use Java to draw the chart).
break
Limitations of Experiments

• It is necessary to implement the algorithm, which may be difficult.
• Results may not be indicative of the running time on other inputs not included in the experiment.
• In order to compare two algorithms, the same hardware and software environments must be used.
Theoretical Analysis

• Uses a high-level description of the algorithm instead of an implementation
• Characterizes running time as a function of the input size, n
• Takes into account all possible inputs
• Allows us to evaluate the speed of an algorithm independent of the hardware/software environment
Assumption: Primitive Operations

• Basic computations performed by an algorithm
• Identifiable in pseudocode (high-level language, that is irrelevant to programming languages, running platforms, etc.)
• Largely independent from the programming language
• Exact definition not important (we will see why later, when we introduce big-Oh)
• Assumed to take a constant amount of time
Primitive Operations

• Examples:
  • Evaluating an expression
  • Assigning a value to a variable
  • Indexing into an array
  • Calling a method
  • Returning from a method
Counting Primitive Operations

```java
/** Returns the maximum value of a nonempty array of numbers. */
public static double arrayMax(double[] data) {
    int n = data.length;
    double currentMax = data[0]; // assume first entry is biggest (for now)
    for (int j = 1; j < n; j++) { // consider all other entries
        if (data[j] > currentMax) { // if data[j] is biggest thus far...
            currentMax = data[j]; // record it as the current max
        }
    }
    return currentMax;
}
```
Counting Primitive Operations

• Step 3: 2 ops, 4: 2 ops, 5: 2n ops, 6: 2n ops, 7: 0 to n ops, 8: 1 op

```java
/** Returns the maximum value of a nonempty array of numbers. */
public static double arrayMax(double[] data) {
    int n = data.length;
    double currentMax = data[0]; // assume first entry is biggest (for now)
    for (int j=1; j < n; j++) // consider all other entries
        if (data[j] > currentMax) // if data[j] is biggest thus far...
            currentMax = data[j]; // record it as the current max
    return currentMax;
}
```
Estimating Running Time

• Algorithm \texttt{arrayMax} executes $5n + 5$ primitive operations in the worst case, $4n + 5$ in the best case. Define:
  \begin{align*}
  a &= \text{Time taken by the fastest primitive operation} \\
  b &= \text{Time taken by the slowest primitive operation}
  \end{align*}

• Let $T(n)$ be worst-case time of \texttt{arrayMax}. Then
  \[ a \ (4n + 5) \leq T(n) \leq b(5n + 5) \]

• Hence, the running time $T(n)$ is bounded by two linear functions
Functions Graphed Using “Normal” Scale

- $g(n) = 1$
- $g(n) = \log n$
- $g(n) = n$
- $g(n) = n^2$
- $g(n) = n^3$
- $g(n) = 2^n$

Slide by Matt Stallmann included with permission.
Seven Important Functions

Seven functions that often appear in algorithm analysis:

- Constant \( \approx 1 \)
- Logarithmic \( \approx \log n \)
- Linear \( \approx n \)
- N-Log-N \( \approx n \log n \)
- Quadratic \( \approx n^2 \)
- Cubic \( \approx n^3 \)
- Exponential \( \approx 2^n \)

In a log-log chart, the slope of the line corresponds to the growth rate.

Siyuan Jiang, Sept. 2018
Big-Oh High-level idea

Limitations of Experiments

- **Need implementation**
- **Results depend on inputs** not included in the experiment.
- In order to compare two algorithms, the same **hardware and software environments** must be used.
Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$.

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$
arrayMax: T(n) is O(n)

Let $T(n)$ be worst-case time of arrayMax. Then

$$a \ (4n + 5) \leq T(n) \leq b \ (5n + 5)$$
Big-Oh Example

• Example: the function $n^2$ is not $O(n)$
  • $n^2 \leq cn$
  • $n \leq c$
  • The above inequality cannot be satisfied since $c$ must be a constant
More Big-Oh Examples

- **7n - 2**
  
  7n-2 is O(n)
  
  need \( c > 0 \) and \( n_0 \geq 1 \) such that \( 7n - 2 \leq cn \) for \( n \geq n_0 \)
  
  this is true for \( c = 7 \) and \( n_0 = 1 \)

- **3 n^3 + 20 n^2 + 5**
  
  3 n^3 + 20 n^2 + 5 is O(n^3)
  
  need \( c > 0 \) and \( n_0 \geq 1 \) such that \( 3n^3 + 20n^2 + 5 \leq cn^3 \) for \( n \geq n_0 \)
  
  this is true for \( c = 4 \) and \( n_0 = 21 \)

- **3 log n + 5**
  
  3 log n + 5 is O(log n)
  
  need \( c > 0 \) and \( n_0 \geq 1 \) such that \( 3 \log n + 5 \leq c \log n \) for \( n \geq n_0 \)
  
  this is true for \( c = 8 \) and \( n_0 = 2 \)
Big-Oh Rules

• If \( f(n) \) is a polynomial of degree \( d \), then \( f(n) \) is \( O(n^d) \), i.e.,
  1. Drop lower-order terms
  2. Drop constant factors

• Use the smallest possible class of functions
  • Say “\( 2n \) is \( O(n) \)” instead of “\( 2n \) is \( O(n^2) \)”
  • Is “\( 2n \) is \( O(n^2) \)” correct?

• Use the simplest expression of the class
  • Say “\( 3n + 5 \) is \( O(n) \)” instead of “\( 3n + 5 \) is \( O(3n) \)”
Asymptotic Algorithm Analysis

• The asymptotic analysis of an algorithm determines the running time in big-Oh notation

• To perform the asymptotic analysis
  • We find the worst-case number of primitive operations executed as a function of the input size
  • We express this function with big-Oh notation

• Example:
  • We say that algorithm arrayMax “runs in $O(n)$ time”

• Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations
Big-Oh High-level idea

Suppress constant factors, lower-order terms

Too system-dependent
Irrelevant for large inputs
Comparison of Two Algorithms

Broadly

Washtenaw Ave.
break
Exercise: Computing Prefix Averages

The $i$th prefix average of an array $X$ is average of the first $(i + 1)$ elements of $X$:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$
Exercise: Prefix Averages

```java
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[] prefixAverage1(double[] x) {
    int n = x.length;
    double[] a = new double[n]; // filled with zeros by default
    for (int j=0; j < n; j++) {
        double total = 0; // begin computing x[0] + ... + x[j]
        for (int i=0; i <= j; i++)
            total += x[i];
        a[j] = total / (j+1); // record the average
    }
    return a;
}
```
Arithmetic Progression

• The running time of `prefixAverage1` is \( O(1 + 2 + \ldots + n) \)
• The sum of the first \( n \) integers is \( n(n + 1) / 2 \)
Improvement on this algorithm?

```java
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */

public static double[] prefixAverage1(double[] x) {
    int n = x.length;
    double[] a = new double[n]; // filled with zeros by default
    for (int j=0; j < n; j++) {
        double total = 0; // begin computing x[0] + ... + x[j]
        for (int i=0; i <= j; i++)
            total += x[i];
        a[j] = total / (j+1); // record the average
    }
    return a;
}
```
Prefix Averages 2

```java
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[] prefixAverage2(double[] x) {
    int n = x.length;
    double[] a = new double[n]; // filled with zeros by default
    double total = 0; // compute prefix sum as x[0] + x[1] + ...
    for (int j=0; j < n; j++) {
        total += x[j]; // update prefix sum to include x[j]
        a[j] = total / (j+1); // compute average based on current sum
    }
    return a;
}
```

Algorithm `prefixAverage2` runs in $O(n)$ time!
Math you need to Review

- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:
  \[
  a^{(b+c)} = a^b a^c \\
  a^{bc} = (a^b)^c \\
  a^b / a^c = a^{(b-c)} \\
  b = a^{\log_a b} \\
  b^c = a^{c \log_a b}
  \]

- Properties of logarithms:
  \[
  \log_b(xy) = \log_b x + \log_b y \\
  \log_b (x/y) = \log_b x - \log_b y \\
  \log_b x a = a \log_b x \\
  \log_b a = \log_x a / \log_x b
  \]
Relatives of Big-Oh

**big-Omega**
- \( f(n) \) is \( \Omega(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that
  \[
  f(n) \geq c \, g(n) \text{ for } n \geq n_0
  \]

**big-Theta**
- \( f(n) \) is \( \Theta(g(n)) \) if there are constants \( c' > 0 \) and \( c'' > 0 \) and an integer constant \( n_0 \geq 1 \) such that
  \[
  c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0
  \]
Intuition for Asymptotic Notation

**big-Oh**
- \( f(n) \) is \( \mathcal{O}(g(n)) \) if \( f(n) \) is asymptotically **less than or equal to** \( g(n) \)

**big-Omega**
- \( f(n) \) is \( \Omega(g(n)) \) if \( f(n) \) is asymptotically **greater than or equal to** \( g(n) \)

**big-Theta**
- \( f(n) \) is \( \Theta(g(n)) \) if \( f(n) \) is asymptotically **equal to** \( g(n) \)
Some catch up

• Recursion
• ArrayList<Type>
Recursion

Factorial:

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \]

\[ 4! = 4 \times 3 \times 2 \times 1 \]

\[ 2! = 2 \times 1 \]

\[ 1! = 1 \]

\[ 0! = 1 \]
Sum
Implement Insertion(A):

*Input*: An array A of n comparable elements  
*Output*: The array A with elements rearranged in non-decreasing order

for k from 1 to n-1 do  
   Insert A[k] at its proper location within A[0], A[1], ..., A[k]