Algorithm Efficiency, Big O Notation, and Javadoc

- Algorithm Efficiency
- Big O Notation
- Role of Data Structures
- Javadoc
- **Reading:**
  - L&C 2.1-2.4
  - `http://algs4.cs.princeton.edu/14analysis`
  - HTML Tutorial

Algorithm Efficiency

- Let's look at the following algorithm for initializing the values in an array:

```
final int N = 500;
int [] counts = new int[N];
for (int i=0; i<counts.length; i++)
     counts[i] = 0;
```

- The length of time the algorithm takes to execute depends on the value of **N**

Algorithm Efficiency

- In that algorithm, we have one loop that processes all of the elements in the array
- Intuitively:
  - If **N** was half of its value, we would expect the algorithm to take half the time
  - If **N** was twice its value, we would expect the algorithm to take twice the time
- That is true, and we say that the algorithm efficiency relative to **N** is *linear*

Algorithm Efficiency

- Let's look at another algorithm for initializing the values in a different array:

```
final int N = 500;
int [] [] counts = new int[N][N];
for (int i=0; i<counts.length; i++)
  for (int j=0; j<counts[i].length; j++)
     counts[i][j] = 0;
```

- The length of time the algorithm takes to execute still depends on the value of **N**

Algorithm Efficiency

- However, in the second algorithm, we have two nested loops to process the elements in the two dimensional array
- Intuitively:
  - If **N** is half its value, we would expect the algorithm to take one quarter the time
  - If **N** is twice its value, we would expect the algorithm to take quadruple the time
- That is true and we say that the algorithm efficiency relative to **N** is *quadratic*

Big-O Notation

- We use a shorthand mathematical notation to describe the efficiency of an algorithm relative to any parameter n as its “Order” or **Big-O**
  - We can say that the first algorithm is **O(n)**
  - We can say that the second algorithm is **$O(n^2)$**
- For any algorithm that has a function **g(n)** of the parameter n that describes its length of time to execute, we can say the algorithm is **O(g(n))**
- We only include the fastest growing term and ignore any multiplying by or adding of constants

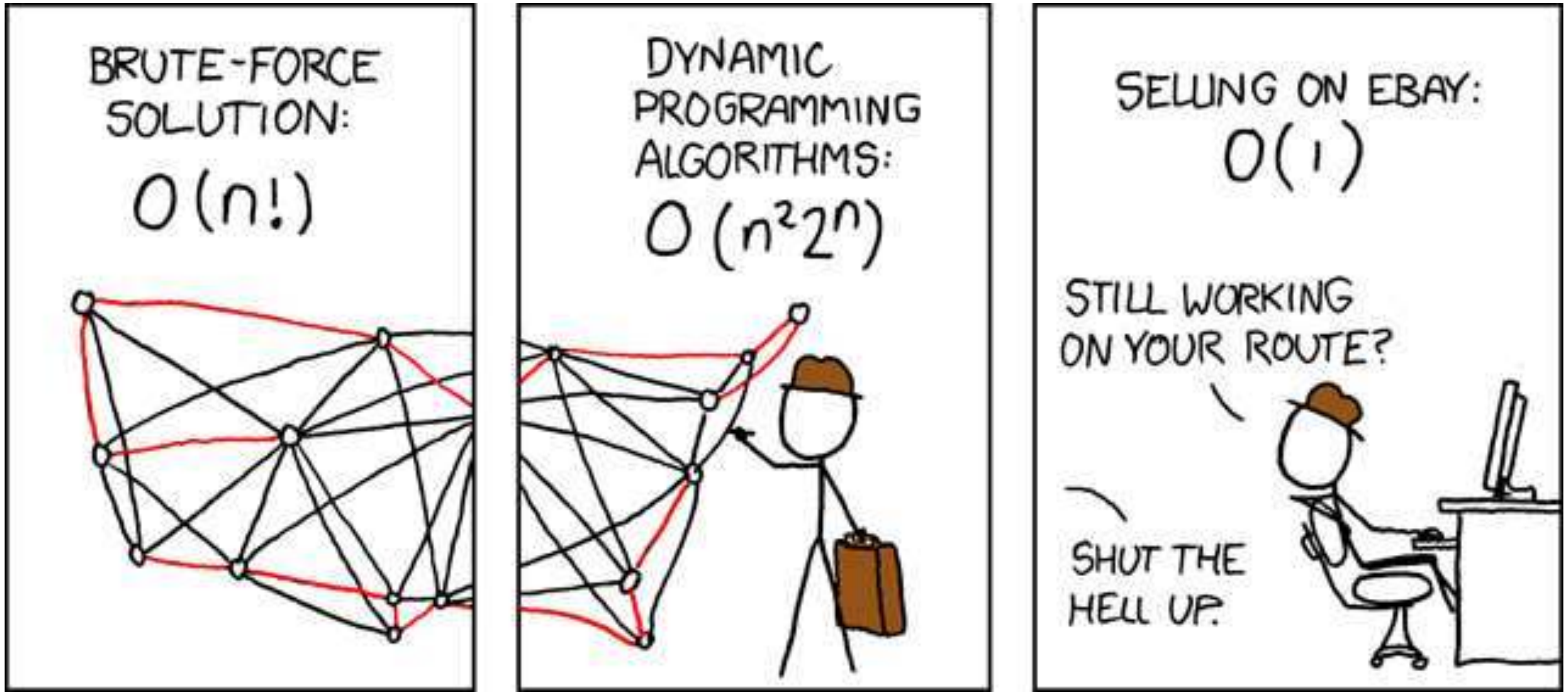
Eight Growth Functions

- Eight functions O(n) that occur frequently in the analysis of algorithms (in order of increasing rate of growth relative to n):
  - Constant ≈ $\boldsymbol{1}$
  - Logarithmic ≈ log $\boldsymbol{n}$
  - Linear ≈ $\boldsymbol{n}$
  - Log Linear ≈ $\boldsymbol{n}$ log $\boldsymbol{n}$
  - Quadratic ≈ $\boldsymbol{n}^2$
  - Cubic ≈ $\boldsymbol{n}^3$
  - Exponential ≈ $\mathbf{2}^n$
  - Exhaustive Search ≈ $\boldsymbol{n}!$

Growth Rates Compared

	n=1	n=2	n=4	n=8	n=16	n=32
1	1	1	1	1	1	1
logn	0	1	2	3	4	5
n	1	2	4	8	16	32
nlogn	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
n!	1	2	24	40320	20.9T	Don’t ask!

Travelling Salesman Problem Joke



Big-O for a Problem

- **O(g(n))** for *a problem* means there is some **O(g(n))** algorithm that solves the problem
- Don't assume that the specific algorithm that you are currently using is the best solution for the problem
- There may be other correct algorithms that grow at a smaller rate with increasing **n**
- Many times, the goal is to find an algorithm with the *smallest possible* growth rate

Role of Data Structures

- That brings up the topic of the *structure of the data* on which the algorithm operates
- If we are using an algorithm manually on some amount of data, we intuitively try to organize the data in a way that *minimizes* the number of steps that we need to take
- Publishers offer *dictionaries* with the words listed in alphabetical order to minimize the length of time it takes us to look up a word

Role of Data Structures

- We can do the same thing for algorithms in our computer programs
- ***Example***: Finding a numeric value in a list
- If we assume that the list is **un**ordered, we must search from the beginning to the end
  - On *average*, we will search half the list
  - *Worst* case, we will search the entire list
  - Algorithm is **O(n)**, where **n** is size of array

Role of Data Structures

- Find a match with **value** in an unordered list

```
int [] list = {7, 2, 9, 5, 6, 4};

for (int i=0; i<list.length, i++)
   if (value == list[i])
       statement; // found it
// didn't find it
```

Role of Data Structures

- If we assume that the list is ordered, we can still search the entire list from the beginning to the end to determine if we have a match
- But, we do not **need** to search that way
- Because the values are in numerical order, we can use a **binary search** algorithm
- Like the old parlor game "Twenty Questions"
- Algorithm is **O(log2n)**, where **n** is size of array

Role of Data Structures

- Find a match with **value** in an ordered list

```
int [] list = {2, 4, 5, 6, 7, 9};
int min = 0, max = list.length-1;
while (min <= max) {
   if (value == list[(min+max)/2])
        statement;  // found it
   else
        if (value < list[(min+max)/2])
             max = (min+max)/2 - 1;
        else
             min = (min+max)/2 + 1;
}
statement;             // didn't find it
```

Role of Data Structures

- The difference in the structure of the data between an unordered list and an ordered list can be used to reduce algorithm **Big-O**
- This is the role of data structures and why we study them
- We need to be as clever in *organizing our data efficiently* as we are in figuring out an algorithm for processing it efficiently

Role of Data Structures

- The only data structure implemented in the Java language itself is the **array** using [ ]
- All other data structures are implemented in classes – either **our own** or **library** classes
- To properly use a class as a data structure, we must know the **Application Programmer's Interface (API)**
- The **API** for a class is documented using Javadoc comments in the source code that can be used to auto-create a web page

Javadoc

- **Javadoc** is a JDK tool that creates *HTML* user documentation for your classes and their methods
- In this case, user means a programmer who will be writing Java code using your classes
- You can access Javadoc via the **JDK CLI**:

```
> javadoc MyClass.java
```

- You can access Javadoc via **Dr Java** menu:

  Tools > Javadoc All Documents

  Tools > Preview Javadoc for Current Document

Javadoc

- The Javadoc tool scans your source file for specialized multi-line style comments:

```
/**
 * <p>HTML formatted text here</p>
 */
```

- Your Javadoc text is written in HTML so that it can appear within a standardized web page format

Block Tags for Classes

- At the class level, you must include these *block tags* with data (each on a separate line):

```
/**
 *  @author Your Name
 *  @version Version Number or Date
 */
```

- You should include HTML text describing the use of this class and perhaps give examples

Block Tags for Methods

- At the ***method*** level, you must include these block tags with data (each on a separate line):

```
/**
 *  @param  HTML text for 1st parameter
 *  @param HTML text for 2nd parameter
 *  . . .
 *  @return HTML text for return value
 */
```

- If there are no parameters or return type, you can omit these Javadoc block tags

In Line Tags

- At any point in your Javadoc HTML text, you may use I***n-Line Tags*** such as @link:

```
/**
 * <p>See website {@link name url}
 * for more details.</p>
 */
```

- In-Line tags are always included inside **{ }**
- These `{ }` are inside the `/**` and `*/` so the *compiler* does not see them