Mid semester feedback

What do computers do?

- Computers spend a lot of time *searching* for data
- Searching is easier if the data is *sorted*
- So, computers spend a lot of time *sorting* data
- Sorting is also a building block for many other algorithms
- Sorting is fascinating and well-studied: there are lots of *algorithms*
- The sorting algorithm *matters*

Efficiency of Algorithms

(Some solutions are better than others!)

How long does it take to…

…to see if three side lengths form a triangle?

```
public static boolean
isTriangle(double a, double b, double c)
{

      boolean case1, case2, case3, all3;
      case1 = a + b > c;
      case2 = b + c > a;
      case3 = c + a > b;
      all3 = case1 && case2 && case3;

      return all3;

}
```

It takes just a few basic computational steps

- Each step takes just a few microseconds, a few basic steps:
  - True for arithmetic operations
  - Also true for logical operations
  - Also true for initiating a method call
  - Also true for returning from a method call
  - Also true for declaring primitive variables
- So, it takes altogether in the order of, say, 20 basic steps
  - It is so fast we cannot easily measure the time.
- We say it takes "order of a constant number of steps":
  - O(20) = O(1)

…to find a specific Account in the Bank?

```
public Account findAccount(int aNum){

    for(int i = 0; i < accCount; i++)
    {
       Account account = allAccounts[i];
       if(account.getAccountNumber() == aNum)
            return allAccounts[i];
    }

    return null;

}
```

Well, it depends on `accCount` the number of accounts in the Bank…

- Of course, it also depends on **which** account we search for:
  - If the account is in location 0, it takes just a few steps.
  - If it is in location 1000, it takes 1000 times of a few steps
  - If it is in location 2000, it takes 2000 times of a few steps
  - If it is in location 1000000000, it takes… you know…
- If it is not found among the `accCount`.accounts, it takes… order of `accCount` steps
- We say it takes "order of `N` steps" where `N` is the number of accounts: the "order of the size of the input"
  - In general: O(`accCount`) = O(`N`)
  - When you double `accCount` , the time doubles! **Linear**!

Well, you may get lucky and find it right away. So?

- We are looking for the **worst-case** guarantees, when we are not lucky.
- Sometimes, we are interested in looking for the average-case scenario, but it is harder to compute than the worst-case.
  - What is “average”?
- So, Big-Oh is measuring the **worst-case** performance of an algorithm.

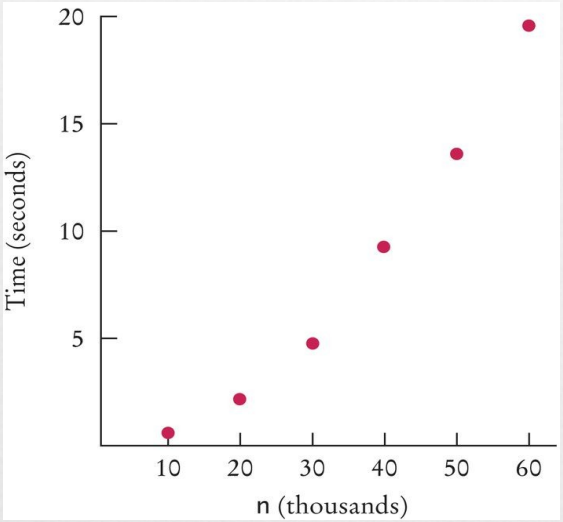
…to compute the sums of all pairs of numbers in an array?

```
public static int[]
twoSum(int[] nums, int target) {

    int[] indices = new int[2];

    for (int i=0; i < nums.length; i++){
       for (int j=i+1; j < nums.length; j++){
          if (nums[i] + nums[j] == target){
             indices[0] = i; indices[1] = j;
          }}}

     return indices;
}
```

...?

```
public static int[]
twoSum(int[] nums, int target) {

    int[] indices = new int[2];

    for (int i=0; i < nums.length; i++){
        for (int j=i+1; j < nums.length; j++){
            return;

}
    return indices;


}
```

Well, let's measure it for different array sizes **n**



n	Milliseconds
10,000	786
20,000	2,148
30,000	4,796
40,000	9,192
50,000	13,321
60,000	19,299

- Doubling the size of the array, more than doubles the time… much more…
- How long would it take with 80,000 numbers?

Why?

- The nested-for loops!
- For the 1st number, you need to form **`n-1`** sums.
- For the 2nd number, you need to form **`n-2`** sums
- For the 3rd number…
- For the nth number …
- All together?

So, what is the “order of complexity” for sumOf2Nums?

- Let’s compute it:

- .

Determining the Efficiency of Algorithms

- **Analysis of algorithms** is a major field that provides math **tools** for evaluating the efficiency of algorithmic solutions to problems
- What is an **efficient** algorithm?
  - Taking less time (fewer steps) is better
  - Using less space is better

    If you need to get data in/out of main memory it takes time
- Algorithm efficiency is **independent** of
  - Specific implementations and coding tricks (programming language, control statements)
  - Specific Computers (hardware chip, OS, clock speed)
- The **size of the input data** should matter
  - But a particular set of data should not matter

Single-statement Execution

- A statement that the computer can execute in one or a few (fixed number of) instructions, we count it as 1 step :

  `O(1) = "order of one"`

- This includes arithmetic operations, logical operations, assignments, but not necessarily function calls, recursive steps, etc.
- For example:

```
// code with O(1) steps
int i = 100;
if (data[i] > maxNum) {
            maxNum = data[i];
            maxIndex = i;
        }
```

Each of these statements is O(1).

Thus the overall order of k simple operations is
k * O(1) = O(1)

where k is the number of individual statements

Analyzing Single Loop Execution

- Need to determine how often a set of statements gets executed to determine the order of an algorithm
- To analyze loop execution, first determine the order of the body of the loop, and then multiply that by the number of times the loop will execute

```
// n = numbers.length is the size of the array
for(int i = 0; i < n; i++) {

    if (numbers[i] == 0) {
        count++;
    }
}
```

The loop executes n times, and the body of the loop is O(1).
Thus, the overall order is O(n) * O(1) = O(n)

Analyzing Nested Loop Execution

- When loops are nested, we must multiply the complexity of the outer loop by the complexity of the inner loop

```
// n = numbers.length is the size of the array
for(int i = 0; i < n; i++) {

   for(int j = i + 1; j < n; j++){

      if (numbers[i] + numbers[j] == 0)
         count++;
      }
   }
}
```

Both the inner and outer loops have complexity of $O(n)$

When multiplied together, the order becomes $n*O(n) = O(n^2)$

Analyzing Recursive Algorithms is harder

- Determining the order of a recursive algorithm
  - determine the **order of the recursion** (# of times recursive definition is followed) and **multiply it by the order of the body** of the recursive method
- Example: Consider the recursive method to compute the product of integers from 1 to some n > 1

```
public int fact (int n){
    int result;
    if (n == 1)
        result = 1;
    else
        result =
            n * fact(n-1);
    return result;
}
```

Size of the problem is n, the number of values to be multiplied

Operation of interest is the multiplication operation

The body of the method performs one multiplication, therefore is O(1)

Each time the recursive method is invoked, n is decreased by 1, thus…

…the recursive method is called n times

The order of the entire algorithm is O(n)

Algorithm "Growth Rates"

- An algorithm's time requirements is measured as a function of the problem size **n (first thing we have to figure out!)**

- Function's growth rate enables the comparison between algorithms

– Examples
  - **Searching** a **sorted** array requires time — proportional to **lg n**
  - **Searching** a list requires time — proportional to **n**
  - Computing **sumOf2Nums** requires time — proportional to $\mathbf{n^2}$

– Notation: **Big-Oh** aka "**order of**"
  - **Searching** a **sorted** array requires time — O(**lg n**)
  - **Searching** a list requires time — O(**n**)
  - Computing **sumOf2Nums** requires O($\mathbf{n^2}$)

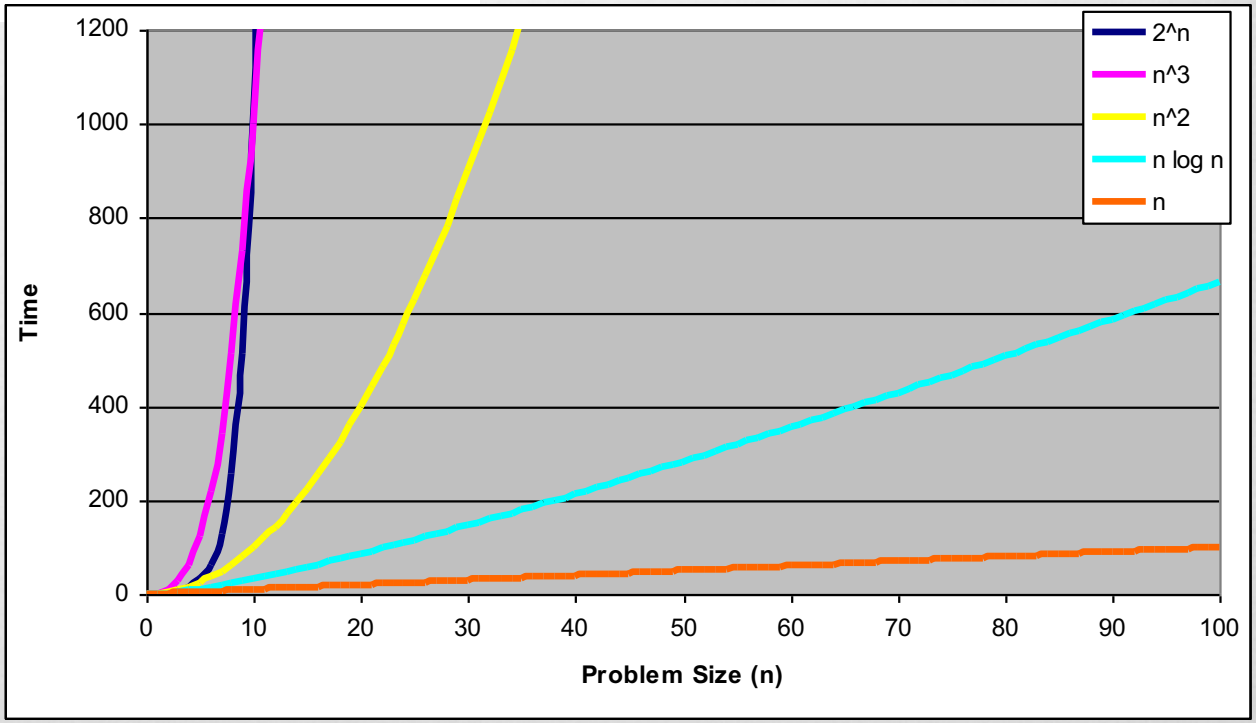
- Algorithm efficiency is typically a concern for large problems only (as **n** grows…)

Comparison of Growth Rates

(a)

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Time units needed to solve problem of size



Order-of-Magnitude Analysis and "Big-Oh"

- Definition of the order of an algorithm:

  Algorithm **A** is order *f(n)*, denoted ***A = O(f(n))***,
  if constants k and n_0 exist such that
  A requires no more than $k * f(n)$ time units to solve a problem of size $n \geq n_0$

- Growth-rate function
  - A mathematical function used to specify an algorithm's order in terms of the size of the problem

- "Big-Oh" notation
  - A notation that uses the capital letter O to specify an algorithm's order
  - Example: *O(n), O(**n**²*log n), O(**n**⁴)*, in general, *O(f(n))*

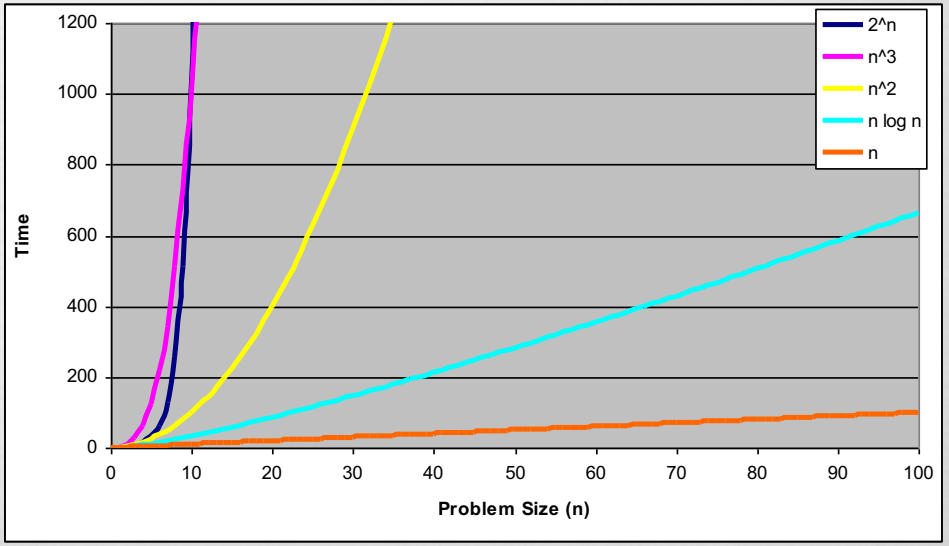
Order-of-Magnitude Analysis and "Big-Oh"

- Order of growth of some common functions

$$O(1) < O(\log_2 n) < O(n) < O(n * \log_2 n) < O(n^2) < O(n^3) < O(2^n)$$

Note that < is **not** the arithmetic "less than" but it means "smaller order"

Where would you place:
$O(n^6)$?
$O((\log_2 n)^2)$?



Order-of-Magnitude Analysis and "Big-Oh"

Properties of growth-rate functions

- Summing orders is dominated by the larger order
  $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\ max\{f(n),g(n)\}\ )$
  - Therefore, you can ignore low-order terms
    $O(n^2 + n) = O(n^2)$

- Multiplying orders means multiplying terms
  $O(f(n)) * O(g(n)) = O(f(n) * g(n))$
  - Therefore, you can ignore multiplicative constants
    $O(12*n^2) = O(n^2)$

What is the running time?

```
// Q1: What is the running time of the whole removeCD_IfFound() method in big-O notation
public boolean removeCD_IfFound(String title) {
      boolean found = false;
      if (count == 0) return found;   // Q2: Give the big-O notation of this statement
      int cd = 0;
      while (cd < count)          // Q3: Give the big-O notation of this statement
      {    // Q4: Give the big-O notation of the line below
           if (title.equals( (collection[cd]).getTitle() ) && !found)
           {     // CD was found. Now remove by replacing it by the last one
                 found = true;
                 collection[cd] = collection[count-1];    // Q5: Give the big-O notation of this statement
                 count--;
           }
           cd = cd+1;                                          // Q6: Give the big-O notation of this statement
      }
      return found;                                      // Q7: Give the big-O notation of this statement
}
```

What is the running time?

```
/**
 * Adds a CD to the collection of n CDs, increasing the
 *  size of the collection if necessary
 */

public void addCD (String title, String artist, double cost,
                                int tracks) {
      if (count == collection.length)
            increaseSize();
      collection[count] = new CD(title, artist, cost, tracks);
      totalCost += cost;
      count++;
}
```

Need the running time of…

```
/**
 * Increases the capacity of the collection by
 * creating a larger array and copying
 */

private void increaseSize(){
        CD[] temp = new CD[collection.length * 2];

        for (int cd = 0; cd < collection.length; cd++){
                temp[cd] = collection[cd];
        }
        collection = temp;
}
```

Worst Case and Average Case Analyses

- An algorithm can require different times to solve different problems of the same size
  - **Worst-case analysis**
    - A determination of the **maximum** amount of time that an algorithm requires to solve problems of size n
    - Big-O uses worst-case analysis
  - **Average-case analysis**
    - A determination of the **average** amount of time that an algorithm requires to solve problems of size n

Keeping your Perspective

- Throughout the course of an analysis,
  keep in mind that you are interested only in **significant differences** in efficiency
- When choosing an ADT's implementation, consider
  how frequently particular ADT operations occur in an application
- Some seldom-used but **critical** operations must be efficient
- If the problem size is always small,
  you can probably ignore an algorithm's efficiency
- Weigh the **trade-offs** between
  an algorithm's time requirements and its memory requirements
- Compare algorithms for both style and efficiency
- Order-of-magnitude analysis focuses on large problems