

Lecture 5

Algorithm Performance Analysis

Zonghua Gu (based on Jianchen Shan)
Department of Computer Science
Hofstra University

Lecture Goals

- Calculate the **big-O** class of complicated code snippets.
- Define **worst case**, **average case**, and **best case** performance and describe why each of these is used.
- State and justify the asymptotic performance for **linear search**, **binary search**, **selection sort**, **insertion sort**, **merge sort**, and **quick sort**.
- Recognize and avoid some common **pitfalls** in asymptotic analysis.
- Use Java timing libraries to measure **execution time**.
- Use runtimes from a **real system** to reason about performance.
- Identify **components** of real systems which impact execution time.

Motivation

Algorithm: a strategy for solving a problem.

Performance: how good that strategy is.

Algorithm with good performance can answer very hard questions in very short amount of time. We need to have a sense of how good our algorithm is without just running it.

There is hereby imposed on the taxable income of every individual (other than a surviving spouse as defined in section 2(a) or the head of a household as defined in section 2(b)) who is not a married individual (as defined in section 2(c)) the tax determined in accordance with the following table:

If you are single, never married, and not the head of a household, you pay taxes according to the following table:

How long dose this take?

Use **flesch score** to measure of text readability

$$\text{FleschScore} = 206.835 - 1.015\left(\frac{\# \text{ words}}{\# \text{ sentences}}\right) - 84.6\left(\frac{\# \text{ syllables}}{\# \text{ words}}\right)$$



Problem with just looking at the “stopwatch” time.

- different computers
- different compilers
- different libraries/optimizations

The time for running the specific code on a specific machine on a specific input

Is NOT a good representation of how good our algorithm is.

Performance Analysis Overview

- What an algorithm can control?

The number of operations

#1: Count operations instead of time

Start at first index of array/list

While current index is less than length:

count syllables

- large input, more operations
- small input, less operations



#2: Focus on how performance scales

If list is **twice** as long,
how much **more time** does it take to search it?

Asymptotic Performance Analysis



Is data size all that matters?

#3: Go beyond input size

We'd like our performance analysis to be able to capture not just the size of the input but also what might happen because of internal structure to the input.

Worst, Best, and Average
Performance Analysis

Count Operations

```

public static boolean hasLetter(String word, char letter)
{
    initial step      final check
    for (int i = 0; i < word.length(); i++) {
        if (word.charAt(i) == letter) {
            return true;
        }
    }
    return false;
}

```

Linear search

0	1	2	3	4
H	a	p	p	y

Is the number of operations the same every time we run `hasLetter(String word, char letter)`?

Search for the letter "a"
in the word "Happy"

How many operations
get executed?

Total operations so far: 7

Search for the letter "x"
in the word "Happy"

Each iteration (in the middle of the
algorithm) contains 3 operations

Total iterations: 5

Total operations: 18

NO

```

hasLetter("happy", "a");
hasLetter("happy", "x");
hasLetter("apple", "a");

```

Introduction to Asymptotic Analysis

- What counts as an operation?

Basic unit that doesn't change as the input changes

- We don't need to worry about anything irrelative to input size

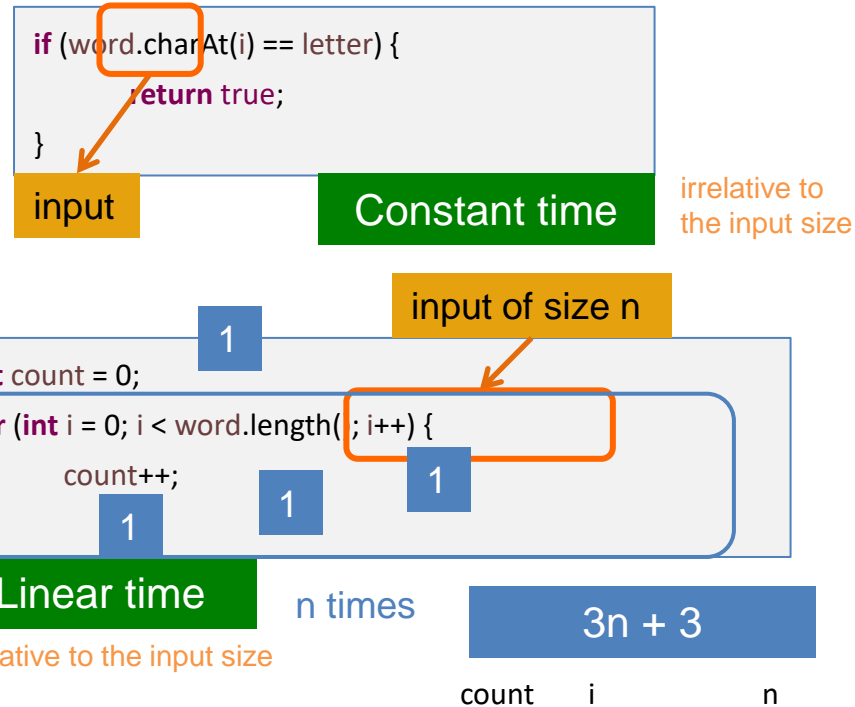
Implementations of specific operations

Initialization time

- Focus on how performance scale with the increase of input size

If input is **twice** as big, how many **more operations** do we need?

Asymptotic Analysis



```

public static int count_a(String s1, int threshold) {
    int total = 0;
    for(int i = s1.length() - 100; i < s1.length(); i++) {
        char c = s1.charAt(i);
        if (c == 'a')
            total++;
    }
}

```

Constant time

Big-O Classes

The goal is to look at the code and pick up its big-O classes. Don't worry about the formal definition too much.

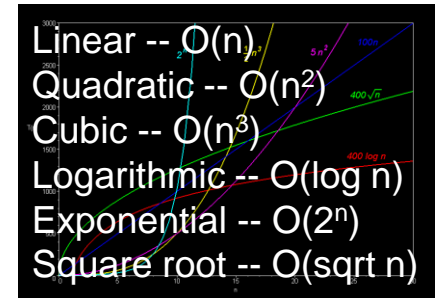
$$f(n) = O(g(n))$$

means

$f(n)$ is big-O of $g(n)$ and they grow in same way as their input grows

there are constants N and c so that for each $n > N$, $f(n) \leq C g(n)$

FORMAL



- We use **big-O classes** as a tool to phrase how algorithm performance scale.
- Two functions are in the same big-O class if they have the same rate of growth.
- Other notations represent a **finer-grained** asymptotic analysis, such as lower and upper bound. We focus on the big-O as shorthand for the **tightest bound**.
- How to compute big O?

Drop constants

$$10000000 = O(1)$$

Example: initialization cost, whose number of steps doesn't change with input size n

Keep only dominant term

$$3n+3 = O(3n) = O(n)$$

Fastest growing

Asymptotic comparison operator	Numeric comparison operator
Our algorithm is $o(\text{something})$	A number is $< \text{something}$
Our algorithm is $O(\text{something})$	A number is $\leq \text{something}$
Our algorithm is $\Theta(\text{something})$	A number is $= \text{something}$
Our algorithm is $\Omega(\text{something})$	A number is $\geq \text{something}$
Our algorithm is $\omega(\text{something})$	A number is $> \text{something}$

$$g(n) = 100 + n^2 + 2^n \quad f(n) = 4 \log_2(n) + 3n \log_2(n) + n$$

$$g(n) = O(2^n) \quad f(n) = O(n \log_2(n))$$

Compute Big O for Consecutive Code

```
public static void reduce (int[] vals) {
```

```
    int minIndex = 0;    O(1) +
```

```
    for (int i=0; i < vals.length; i++) {
        if (vals[i] < vals[minIndex]){
            minIndex = i;
        }
    }
```

O(n) +

```
    int minVal = vals[minIndex];
```

O(1) +

```
    for (int i=0; i < vals.length; i++){
        vals[i] = vals[i] - minVal;
    }
```

O(n) +

```
}
```

$$1 + n + 1 + n = 2n + 2 = 2n + 2$$

Total: O(n)

Linear Algorithm

[1,2,5,3] → [0,1,4,2]

The first for loop finds the minimum value of the array. The second for loop reduces each value in the array by the minimum value.

- Run times are independent
- These doesn't depend on the input size(n)
- How the operations depend on the size of the input?
- There will be n loop iterations
- Each iteration will take constant time

Compute Big O with Nested Operations

```
public static int maxDifference (int[] vals) {
```

```
    int max = 0;    O(1) +
```

```
    for (int i=0; i < vals.length; i++) {
```

```
        for (int j=0; j < vals.length; j++) {
```

```
            if (vals[i] - vals[j] > max) {
```

```
                max = vals[i] - vals[j];
```

```
            }
```

```
        }    O(1)
```

```
    }    O(n) X
```

Multiplication

```
    return max;    O(1) +
```

```
    }    O(n2) +
```

```
}
```

$$1 + n^2 + 1 = n^2 + 2$$

Total: $O(n^2)$

Quadratic Algorithm

[1,7,2,4,6,8]

-> 7

The nested for loops look for the maximum difference between any two array elements. This biggest difference will be between 1 and 8.

- Run times are independent
- These doesn't depend on the input size(n)
- How the operations depend on the size of the input?
- Count from inside out
- There will be n inner loop iterations and each takes constant time
- There will be n outer loop iterations and each takes linear time $O(n)$

Sort Algos // Michael Sambol

- https://www.youtube.com/playlist?list=PL9xmBV_5YoZOZSbGAXAPIq1BeUf4j20pl

1 Merge Sort in 3
3:03

2 Quick Sort in 4
4:24

3 Bubble Sort in 2
2:10

4 Insertion Sort in 2
2:19

5 Selection Sort in 3
2:43

6 Heap Sort in 4
4:13

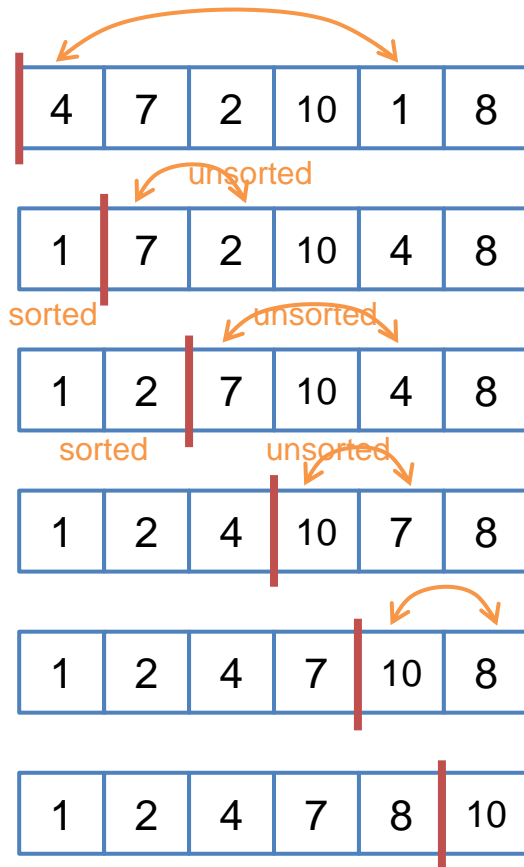
Practice: Analyze Big-O Class of Selection Sort

The **idea** is to find the smallest value in the remaining unsorted array and put that at the start. And then just keep repeating that process over and over.

$$n + (n-1) + (n-2) + \dots + 1 = \frac{n \times (n+1)}{2}$$

(Gauss sum)

O(n-i)*O(n)? NO



```

public static void selectionSort(int[] vals) {
    int indexMin;           O(1)
    for(int i = 0; i < vals.length-1; i++) {
        indexMin = i;       O(1)
        for(int j = i + 1; j < vals.length; j++) {
            if(vals[j] < vals[indexMin]) {
                indexMin = j;
            }
        }
        swap(vals, indexMin, i);
    }
}
  
```

Annotations for the code analysis:

- outer loop runs n times**: The outer loop `for(int i = 0; i < vals.length-1; i++)` runs n times.
- nested loop**: The inner loop `for(int j = i + 1; j < vals.length; j++)` is nested within the outer loop.
- inner loop runs n - (i+1) times**: The inner loop runs $n - (i+1)$ times for each iteration of the outer loop.
- O(1)**: The operations `indexMin = i;`, `indexMin = j;`, and `swap(vals, indexMin, i);` are constant time operations.
- O(n-i)**: The inner loop's execution time is $O(n-i)$ for each iteration of the outer loop.
- O(n²)**: The total time complexity of the selection sort algorithm is $O(n^2)$.

Total: O(n²)

To swap:

```

temp = vals[indexMin];
vals[indexMin] = vals[i];
vals[i] = temp;
  
```

Best case, Average case, Worst case

- How does the algorithm behave for all inputs?



```

public static boolean hasLetter(String word, char letter)
{
    for (int i = 0; i < word.length(); i++) {
        if (word.charAt(i) == letter) {
            return true;
        }
    }
    return false;
}
  
```

input

when is the least number of operations be executed?

output

when will the largest amount of operations be executed?

(same input size)

Best case: word starts with letter $O(1)$

- just 1 loop iteration
- special case

hasLetter("happy", "x");

hasLetter("happy", "y");

Worst case : letter at the end (or missing) $O(n)$

- n loop iterations

- algorithm performance depends on the combination of both inputs

- How can we account for this variability?

(lower bound)

sandbox

(upper bound)

Best case

\leq

Worst case

Best possible performance of algorithm for any input (of fixed size n)

Worst possible performance of algorithm for any input (of fixed size n)

(realistic, but too hard)

Average case

Performance of algorithm on average, consider all possible inputs of size n

Analyze Search Algorithms

	Best Case	Worst Case
Linear Search	$O(1)$	$O(n)$
Binary Search*	$O(1)$	$O(\log(n))$

times to half size?

How many times can we divide by 2 before we get to 1?

* Assuming data is sorted sorting cost?

Linear Search: Basic Algorithm

Start at the first **index** in the array

```
while index < length of the array:
    if toFind matches current value,
        return true
    increment index by 1
```

return false

E.g. hasLetter(String word, **char** letter)

Binary Search: Basic Algorithm

Initialize **low = 0**, **high = length of list**

while low <= high:

mid = (high+low)/2

if toFind matches value at mid,
return true

if toFind < value at mid

high = mid-1 first half

else low = mid+1 second half

return false

- cuts search
base in half

Worst case: don't find!

$$\log_{10}(n) = O(\log_2(n)) \quad \text{true or false?} \quad \text{Log base conversion formula: } \log_{10}(n) = \frac{\log_2(n)}{\log_2(10)}$$

Analyze Sorting Algorithms

	Best Case	Worst Case
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)^*$

when already sorted

1 2 3 4 5 6

when in reverse order

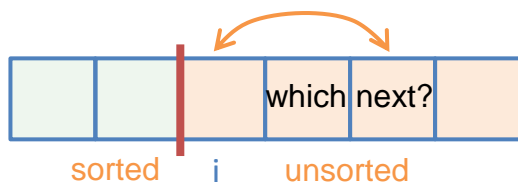
6 5 4 3 2 1

* similar to selection sort analysis

Selection Sort: Basic Algorithm

For each **position i** from **0** to **length-2**

Find smallest element in **positions i to length-1**
Swap it with element in **position i**



Best, average, worst?

```
public static void insertionSort(int[] vals) {
```

```
    int currInd;
```

```
    for(int pos=1; pos < vals.length; pos++) {
```

```
        currInd = pos;
```

```
        while (currInd > 0 &&
```

```
            vals[currInd] < vals[currInd-1]) {
```

```
            swap(vals, currInd, currInd-1);
```

```
            currInd = currInd - 1;
```

```
        }
```

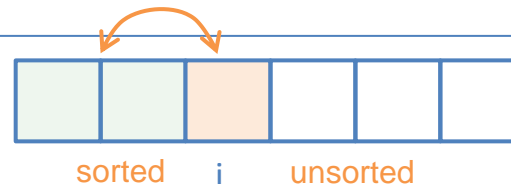
```
}
```

- # of loop iterations depends on the values of the pairs of consecutive elements

Insertion Sort: Basic Algorithm

For each **position i** from **1** to **length-1**

Swap successive pairs to put value in **position i** in correct location relative to earlier values



1	8	4	3	7	2	pos 1
1	4	8	3	7	2	pos 2
1	3	4	8	7	2	pos 3
1	3	4	7	8	2	pos 4
1	2	3	4	7	8	pos 4

Analyze Merge Sort

Merge Sort: Basic Algorithm

If list has one element, return.

Divide list in half

Keep dividing by two
until lists have size 1
 $\log_2(n)$

Sort first half

Sort second half

Each time we divide,
we call MergeSort on
two (smaller) lists

HOW? Divide and conquer.
Recursion!

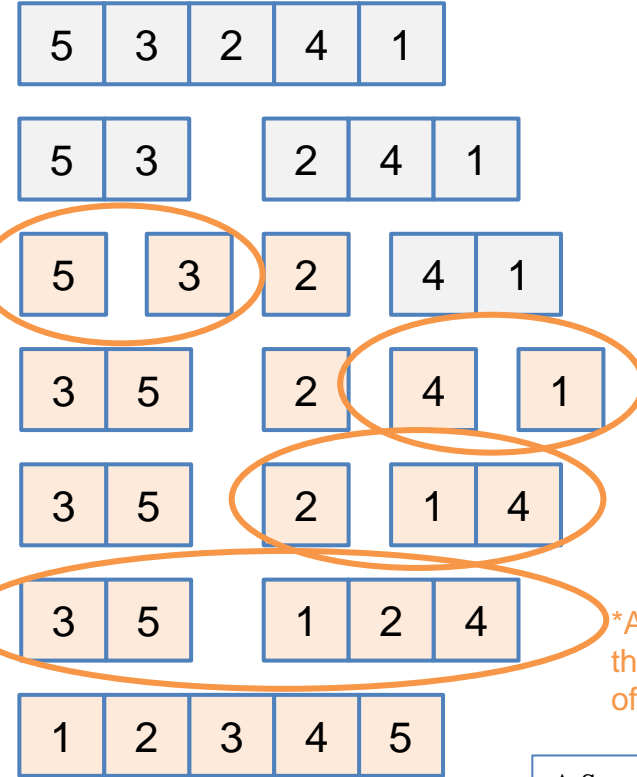
Merge sorted lists

$O(n)$ work to merge all
the lists on one level

compare the head of each list

Performance?

$O(n \log(n))$

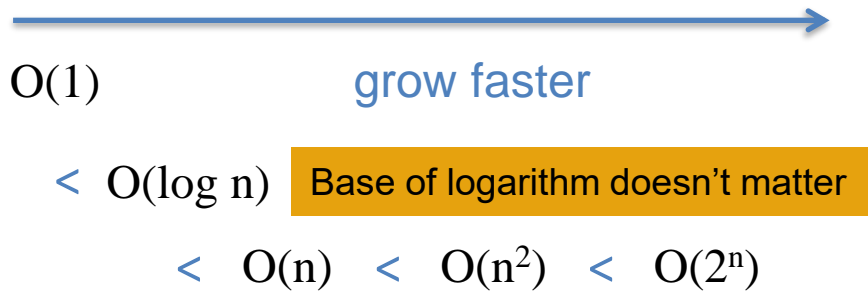


*Asymptotics is not
the only measure
of performance

A Summary of Sorting

	Best	Average	Worst
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$
*Quick Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$

Common Pitfalls in Asymptotic Analysis

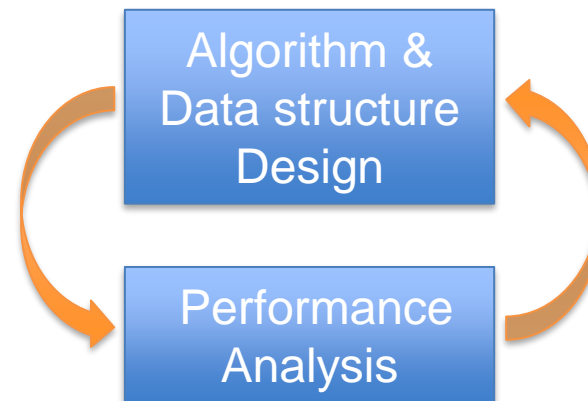


what input will I be working with?

- Will Algorithm 1 always use fewer operations than Algorithm 2?
 - No. $100000 \cdot \log n > 1 \cdot n^2$ if $n=10$

```
public static boolean hasLetter(String word, char letter)
{
    for (int i = 0; i < word.length(); i++) {
        if (word.charAt(i) == letter) {
            return true;
        }
    }
    return false;
}
```

Beware of method calls



Introduction to Benchmarking



www.speedtest.net

Your Java
Code
Version A

~10 seconds

Your Java
Code
Version B

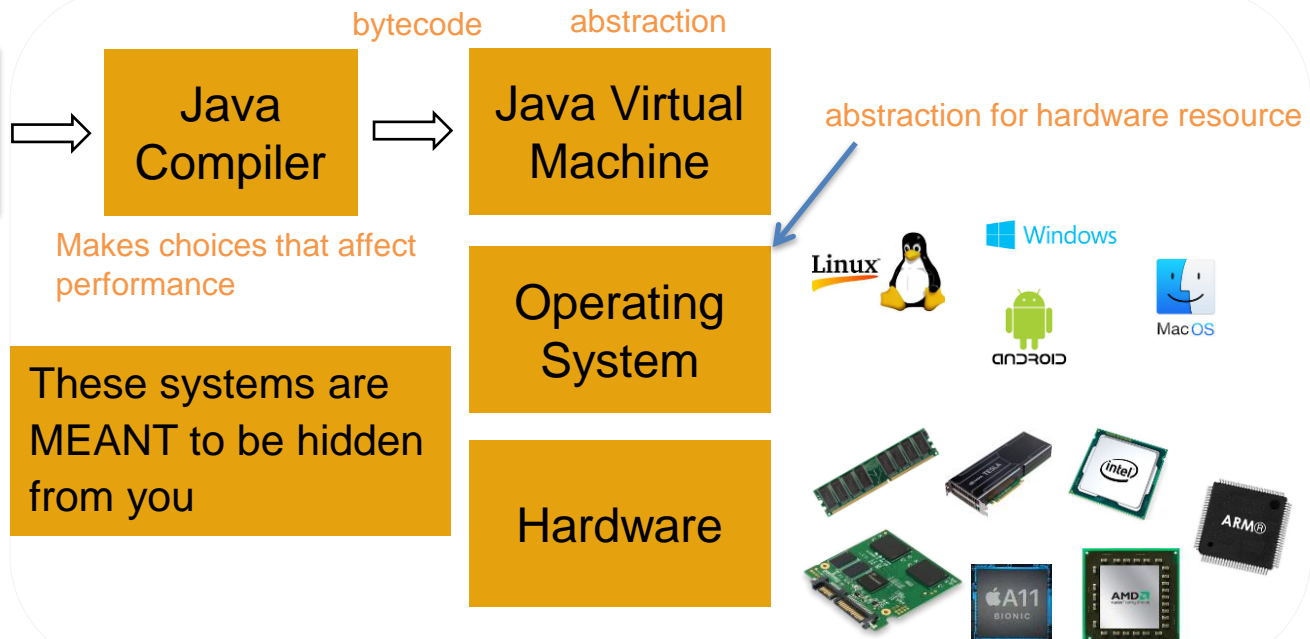
~5 seconds

Times might not be
consistent...

The running time of a
program is influenced
by many things!



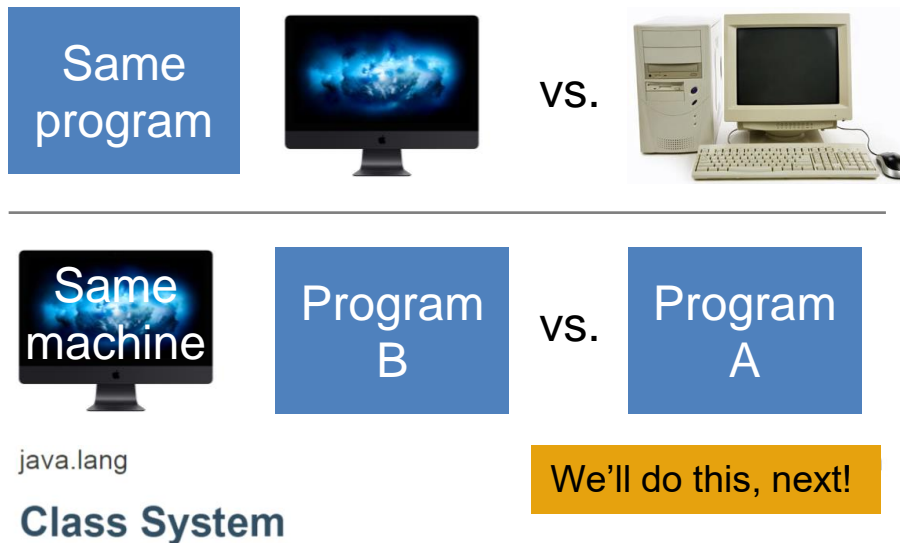
Your Java
Code



So how do we reason
about how long it takes for
a program to run on real
systems? Couldn't we just
time how long our
programs take? **YES!**

Details of Benchmarking (Using Java Timing API)

- Just means running programs on real machines and measuring performance
- For us, “performance” is just how long it takes for something to execute.
- Allows us to **compare machines** by running the same program
- Allows us to **compare programs** on a single machine



```
static long
```

```
nanoTime()
```

Returns the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.

```
public static void main(String [] args) {
    // set some size n
    double array[] = new double[n];
    // fill the array with contents (random)
    long startTime = System.nanoTime();
    selectionSort(array);
    long endTime = System.nanoTime();
    double estTime = (endTime-startTime) /
        1000000000.0;
    System.out.println(estTime);
}
```

we just want this time

How long does
selection sort run?

Idea for Analyzing Our Sorts

For increasing sizes of n

Print n

Create a randomized array of size n

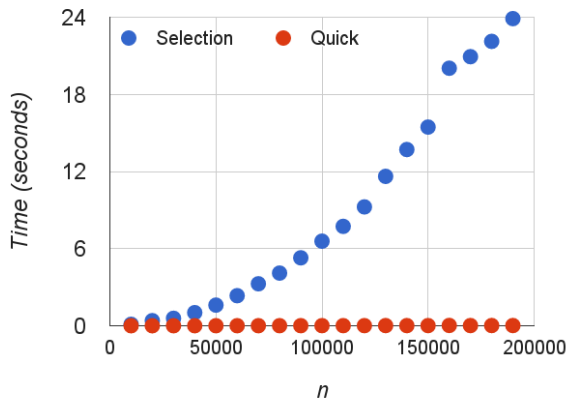
Time **selection sort**, print outcome

Create a randomized array of size n

Time **quick sort**, print outcome

n	Selection (s)	Quick (s)
10000	0.112887621	0.001323534
20000	0.397227565	0.001568662
30000	0.580318935	0.002420492
40000	1.020979179	0.003304295
50000	1.605557659	0.004232703
60000	2.340087449	0.004983088
70000	3.264979954	0.006035047
80000	4.097073897	0.006989112
90000	5.285101776	0.007900941
100000	6.57904119	0.008538038

Quick vs. Selection

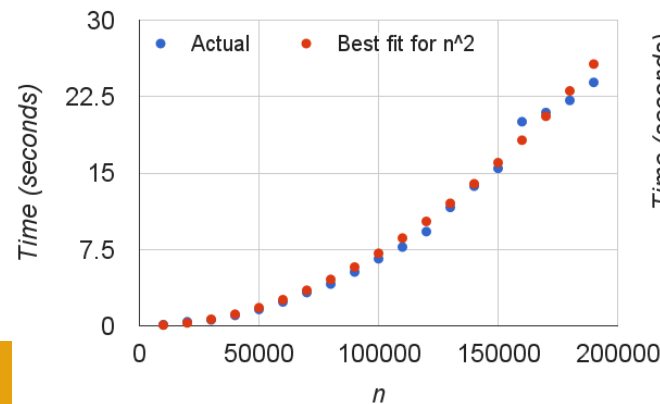


Quicksort is fantastic

Select: Looks like n^2 growth

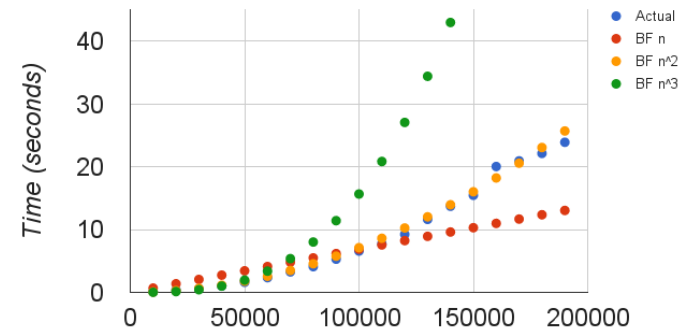
By “best fit” I just found a good value for constant “ k ”

Actual vs. $k^*(n^2)$



Won't all “best fits” look really good?

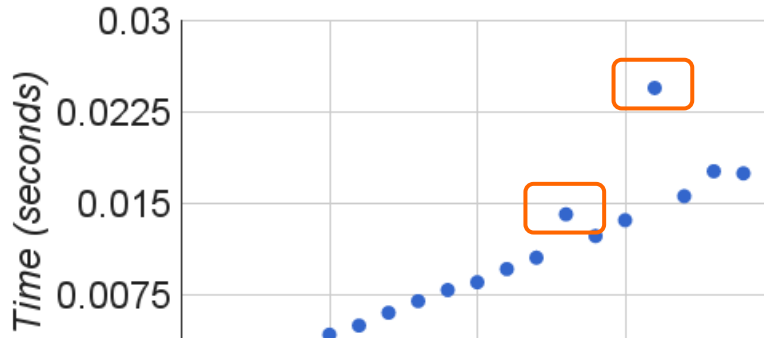
Actual vs. Best Fits



n^2 is best, matching our high-level analysis

Idea for Analyzing Our Sorts (Contd.)

Quick Sort Actual



Zoom in on quick sort

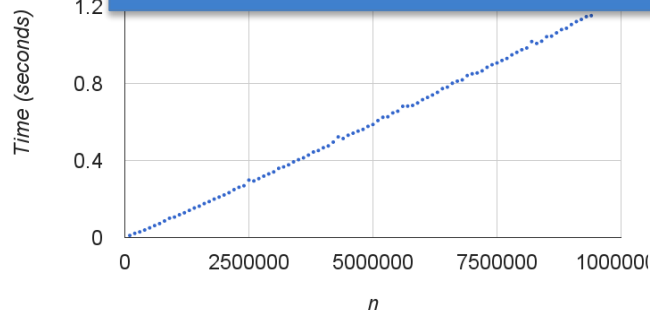
Real data...

- input
- system task

Looks linear?

n	$\log_2 n$
10,000,000	?

- We can use real runtimes to reason about performance
- Be prepared for real system data to be noisy
- Can be really useful when we want to understand actual performance on a real system

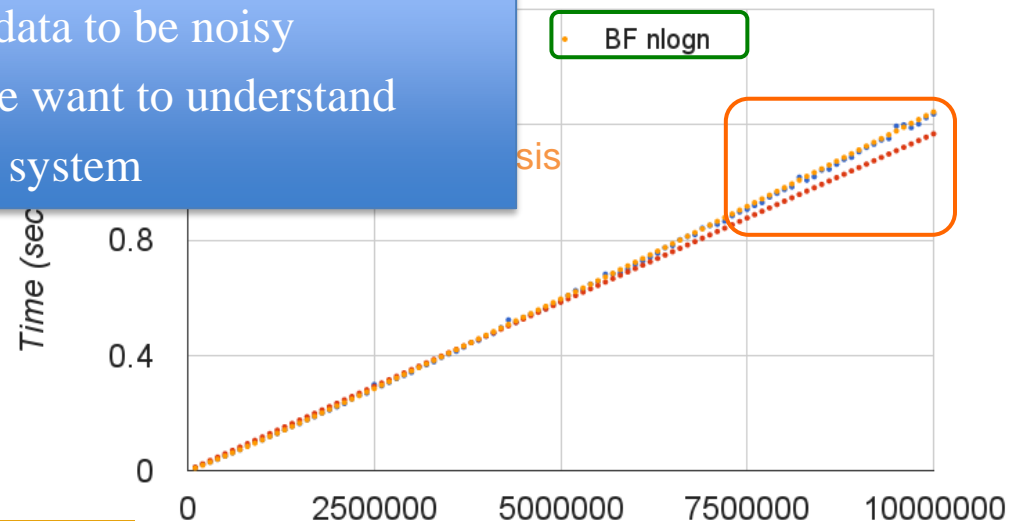


Get more data...

Still appears linear

Best Fits

~23



$\log(n)$ is just really small relative to n

Additional Resources

■ Big-O analysis

- http://web.mit.edu/16.070/www/lecture/big_o.pdf -- Big O handout from MIT
- <https://www.interviewcake.com/article/java/big-o-notation-time-and-space-complexity> -- explanation of Big O with examples
- <http://discrete.gr/complexity/> -- "A Gentle Introduction to Algorithm Complexity Analysis" Gives a lot more detail than what we provided.

■ Sorting algorithms

- <http://www.java2novice.com/java-sorting-algorithms/> -- 5 different sort algorithm explanation with codes
- <https://www.cs.cmu.edu/~adamchik/15-121/lectures/Sorting%20Algorithms/sorting.html> -- different search algorithms with solid examples

■ Timing code in Java

- <http://stackoverflow.com/questions/180158/how-do-i-time-a-methods-execution-in-java> -- many ways offered by many people