CSSE 220 Day 28

Analysis of Algorithms continued

Questions?

- On Capstone Project?
  - Have your *networking spike solution* completed by *today*!
    - Get my help (outside of class, make an appointment) as needed
  - Cycle 2 ends today. Complete reports.
  - About 25 minutes today to work on Capstone.

- On Exam 2?
  - www.rose-hulman.edu/class/csse/csse220/200930/Projects/Exam2/instructions.htm
  - Take-home.
  - Open everything *except* human resources.
  - Released Wednesday 6 a.m. Complete by Thursday 6 a.m.
  - Designed to take about 90 minutes, you may take up to 3 hours
  - Most (maybe all) on-the-computer.

- On anything?

Re Exam 1:

- Bad news: I have not graded all of yours.
- Good news: I will add 10 points (of 100) to your score. 50 points if I don't have it graded by Thursday!

Outline of today's session

- Algorithm analysis, continued
  - Review: Context, motivation
  - Review: Definition of big-Oh
  - Applications of big-Oh:
    - Loops
    - Search
      - Linear
      - Binary (iterative implementation)
    - Sort
      - Selection Sort
      - Insertion Sort
- Work on Capstone

What makes a program "good"?

- Correct – meets specifications
- Easy to understand, modify, write
- Uses reasonable set of resources
  - Time (runs fast)
  - Space (main memory)
  - Hard-drive space
  - Peripherals
  - …
- Here we focus on "runs fast" – **how much CPU time does the program / algorithm / problem take?**
  - Others are important too!

Big-Oh motivation: why profiling is not enough

- Results from profiling depend on:
  - Power of machine you use
    - CPU, RAM, etc
  - Operating system of machine you use
  - State of machine you use
    - What else is running? How much RAM is available? …
  - What inputs do you choose to run?
    - Size of input
    - Specific input

Big-Oh motivation: what it provides

- Big-Oh is a mathematical definition that allows us to:
  - Determine how fast a program is (in big-Oh terms)
  - Share results with others in terms that are universally understood
- Features of big-Oh
  - Allows paper-and-pencil analysis
  - Is much easier / faster than profiling
  - Is a function of the *size of the input*
  - Focuses our attention on *big* inputs
  - Is machine independent

Informal definition of big-Oh
As applied to run-time analysis

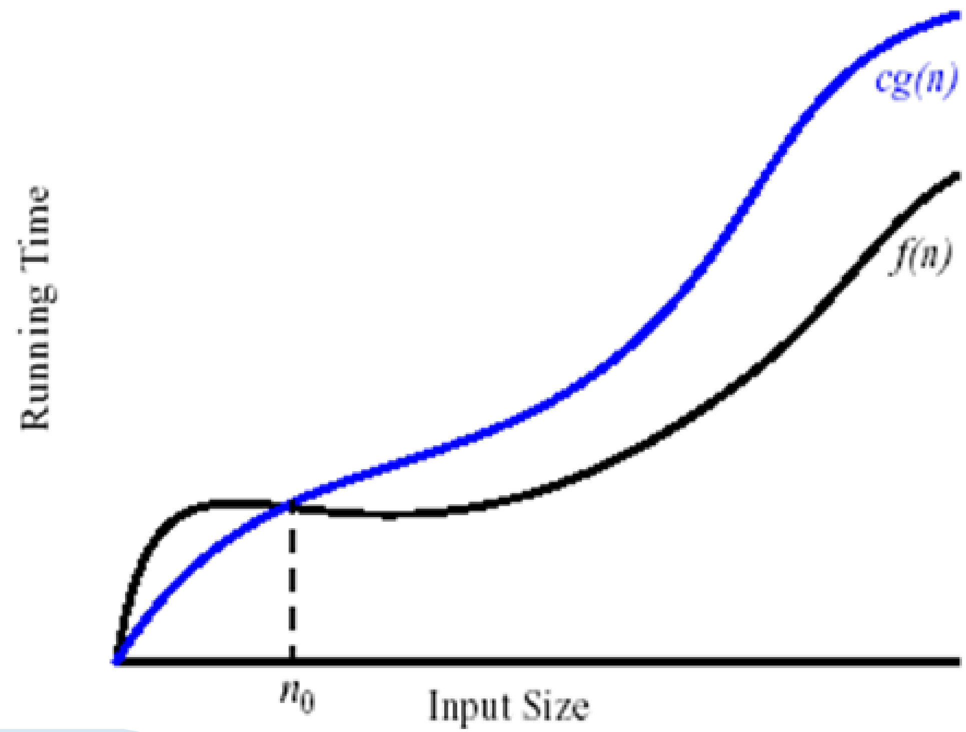
- Run-time of the algorithm of interest on a worst-case input of size n is at most a constant times *blah*, for large n
  - Example:
    - You are given K people (so K is the size of the problem)
    - You have a K x K matrix whose (j, k) entry gives how "close" person j is to person k
    - Find the pair of people who are farthest apart.
    - This is O(_____)?

Answer: $O(K^2)$

- Alternatives to:
  - Run-time: space required, …
  - Algorithm of interest: Problem of interest
  - Worst-case input: Average-case, best-case
  - At most: At least => Ω and "exactly" (i.e. one constant for at least and another for at most) => Θ

Definition of big-Oh

- Formal:
  - We say that f(n) is O(g(n)) if and only if
  - there exist constants c and n_0 such that
  - for every $n \geq n_0$ we have
  - $f(n) \leq c \times g(n)$
- Informal:
  - f(n) is roughly proportional to g(n), for large n



Examples: Loops

- Loop 1: *n* is size of input

```
int sum = 0;
for (int k = 0; k < n; ++k) {
    sum += k * k * k * k;
}
```

Run-time is
O(______)?

Answer:

O(n)

Examples: Loops

- Loop 2: *m* is size of input

```
int sum = 0;
for (int j = 0; j < m; ++j) {
    for (int k = 0; k < m; ++k) {
        sum += Math.sin(j)
                * Math.cos(k);
    }
}
```

Run-time is
O(______)?

Answer:

$O(m^2)$

Note: the above code has bad style, in that it could trivially be made more efficient. How?

- Answer: move the computation of sin(j) outside the k loop
- Would this change our big-Oh answer?
- No.

Examples: Loops

- Loop 3: *m* is size of input

Run-time is
O(______)?

Answer:

$O(m^2)$

This code is equivalent to the code in the previous example.

```
int sum = 0;
for (int k = 0; k < m; ++k) {
     sum += Math.sin(k) * Blah.foo(m);
}


public static int foo(int n) {
     int sum = 0;

     for (int j = 0; j < n; ++j) {
          sum += Math.cos(j);
     }

     return sum;
}
```

Examples: Loops

Run-time is
O(______)?

Answer:

$O(m^2)$

Key fact: sum of k from 0 to n - 1 is
n (n - 1) / 2

hence $O(n^2)$

- Loop 4: *m* is size of input

```
int sum = 0;
for (int k = 0; k < m; ++k) {
    sum += Math.sin(k) * Blah.foo(k);
}
```

Same as previous example except previous example had ***m*** here.

```
public static int foo(int n) {
    int sum = 0;

    for (int j = 0; j < n; ++j) {
        sum += Math.cos(j);
    }

    return sum;
}
```

Examples: Loops

- Loop 5: *n* is size of input

```
int sum = 0;

for (int k = 0; k < n; ++k) {
    sum += k * k * k * k;
}

for (int k = 0; k < n; ++k) {
    sum += k * k * k * k;
}
```

Run-time is
O(______)?

Answer:

O(n)

So two principles:

1. Loop followed by loop: take bigger big-Oh
2. Loop inside loop: multiply big-Oh's

Example: *Linear Search* of a sorted array of Comparable's

```
for (int i = 0; i < a.length; i++) {
  if (a[i].compareTo(soughtItem) > 0) {
      return NOT_FOUND;
  } else if (a[i].compareTo(soughtItem) == 0) {
      return i;
  }
}
return NOT_FOUND;
```

There is a better algorithm for finding an element in a *sorted* array. What is it?

	Answer
Input size is n, which is:	length of array
Worst-case run-time is O(______)?	O(n)
Best-case run-time is O(______)?	O(1)
Average-case run-time is O(______)?	O(n)

Not obvious, and depends on the input distribution

Example: *Binary Search* of a sorted array of Comparable's

```
int left = 0;     int right = a.length;     int middle;
while (left <= right) {
    middle = (left + right) / 2;
    int comparison = a[middle].compareTo(soughtItem);

    if (comparison == 0) {
                                  return middle;
        __________________
    } else if (comparison > 0) {
        ______________________    right = middle - 1;
    } else {
        ______________________    left = middle + 1;
    }
}
return NOT_FOUND;
```

Example: *Binary Search* of a sorted array of Comparable's

```
int left = 0;    int right = a.length;    int middle;

while (left <= right) {
    middle = (left + right) / 2;
    int comparison = a[middle].compareTo(soughtItem);

    if (comparison == 0) {
        return middle;
    } else if (comparison > 0) {
        right = middle – 1;
    } else {
        left = middle + 1;
    }
}
return NOT_FOUND;
```

For worst & average-case, how big a gain is this over linear search? Try some numbers!

Average case is not obvious and depends on the input distribution.

Input size is n, which is:
Worst-case run-time is O(______)?
Best-case run-time is O(______)?
Average-case run-time is O(______)?

Answer: length of array
Answer: O(log n)
Answer: O(1)
Answer: O(log n)

Example: *Selection Sort* of an array of Comparable's

```
for (int k = 0; k < a.length; ++k) {
    int m = smallest(a, k);
    swap(a, k, m);
}

// Returns the index of the smallest element in the given array
// from index k to the end of the array.
//    Algorithm:  linear search of UNsorted array.
public static int smallest(Comparable<T>[] a, int k);

// Swaps the elements in the array at the given indices.
public static void swap(Comparable<T>[] a, int j, int k);
```

Input size is n, which is:
Worst-case run-time is O(_____)?
Best-case run-time is O(_____)?
Average-case run-time is O(_____)?

Answer: length of array
Answer: $O(n^2)$
Answer: $O(n^2)$
Answer: $O(n^2)$

Example: *Insertion Sort* of an array of Comparable's

```
for (int k = 1; k < a.length; ++k) {
   insert(a, k);
}

// Inserts a[k] into its correct place in the given array.
// Precondition:  The given array is SORTED from indices 0 to k-1, inclusive.
// Postcondition:  The given array is SORTED from indices 0 to k, inclusive.
public static int smallest(Comparable<T>[] a, int k) {
    int j;
    Comparable<T> x = a[k];

    while (int j = k - 1; j >= 0; --j) {
        if (a[k].compareTo(a[j]) < 0) {
            a[j + 1] = a[j];
        } else {
            break;
    }
    a[j + 1] = x;
}
```

Example: *Insertion Sort* of an array of Comparable's

```
for (int k = 1; k < a.length; ++k) {
   insert(a, k);
}
```

Worst-case is ? Its run-time is ?
Best-case is ? Its run-time is ?
Average-case is ? [Nonsense!]
Average-case run-time is ?

```
// Inserts a[k] into its correct place in the given array.
// Precondition: The given array is SORTED from indices 0 to k-1, inclusive.
// Postcondition: The given array is SORTED from indices 0 to k, inclusive.
public static int smallest(Comparable<T>[] a, int k) {
    int j;
    Comparable<T> x = a[k];

    while (int j = k - 1; j >= 0; --j) {
        if (a[k].compareTo(a[j]) < 0) {
            a[j + 1] = a[j];
        } else {
            break;
        }
    }
    a[j + 1] = x;
}
```

Worst-case is backwards sorted array. Its run-time is $O(n^2)$.

Best-case is sorted array. Its run-time is $O(n)$.

Average-case run-time, under most reasonable input distributions, is $O(n^2)$.

Example: *String copy*

```
public static String stringCopy(String s) {
  String result = "";

  for (int i = 0; I < s.length(); i++)
     result += s.charAt(i);

  return result;
}
```

Reminder: Strings are immutable.

Input size is n, which is:
Run-time of EACH iteration of loop is:
Run-time of string copy is O(______)?
Would your answer change if we used character arrays instead of immutable strings?

Answer: length of string
Answer: O(n)
Answer: $O(n^2)$
Yes, it would be O(n)