CSSE 220 Day 29

Analysis of Algorithms continued

Recursion

Questions?

- On Capstone Project?
  - Have your *networking spike solution* completed by *yesterday*!
    - Get my help (outside of class, make an appointment) as needed
  - Cycle 3 ends tomorrow! Ask in class if you want an extension.
  - About 30 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 **Friday** 6 a.m.
  - Designed to take about 90 minutes, you may take up to 3 hours
  - All on-the-computer.
- On anything?

Re Exam 1:

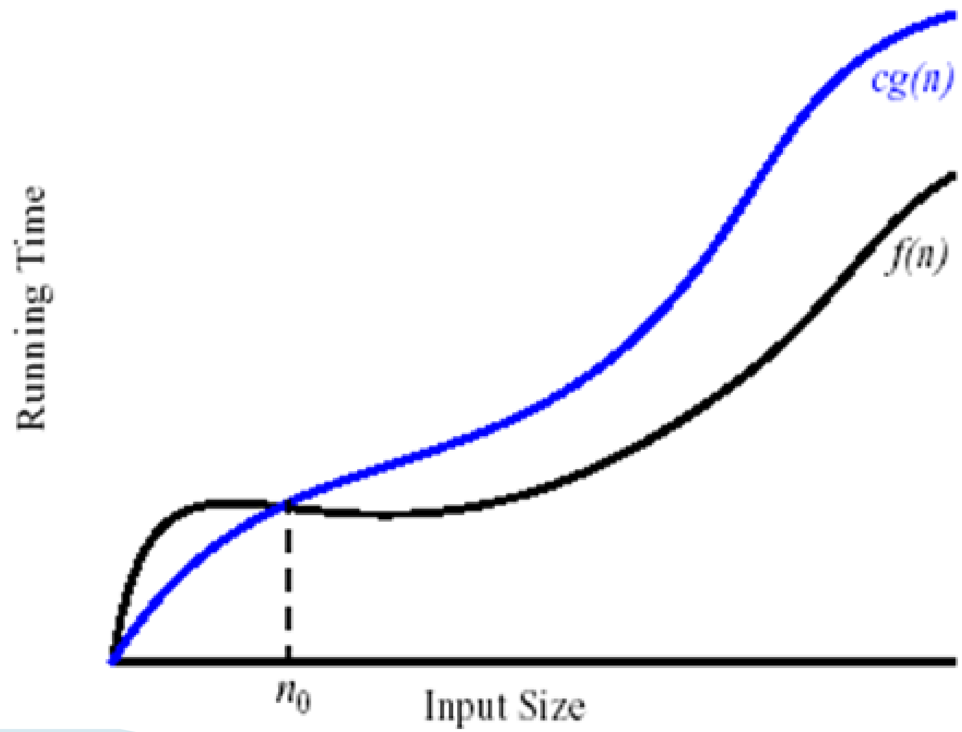
- Bad news: I have not graded all of yours.
- Good news: I will add 10 points (of 100) to your score. **50 points** if I don't have it graded by Thursday!

Outline of today's session

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

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 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: *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:	Answer: length of array
Worst-case run-time is O(______)?	Answer: O(log n)
Best-case run-time is O(______)?	Answer: O(1)
Average-case run-time is O(______)?	Answer: O(log n)

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 insert(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:	Answer: length of string
Run-time of EACH iteration of loop is:	Answer: O(n)
Run-time of string copy is O(______)?	Answer: $O(n^2)$
Would your answer change if we used character arrays instead of immutable strings?	Yes, it would be O(n)

Outline of rest of this lecture

- Introduction to *recursion*
  - Motivational example: Palindrome
  - Basic idea summarized
  - Examples:
    - Recursive definitions:
      - Fibonacci
      - Ackermann's
    - Recursion algorithms:
      - Binary search (recursive implementation)
      - Merge sort

Programming Problem

- A palindrome is a phrase that reads the same forward or backward
  - We'll ignore case, punctuation, and spaces.
  - Examples:

A man, a plan, a canal -- Panama!

Go hang a salami, I'm a lasagna hog.

- Add a recursive method to Sentence for computing whether Sentence is a palindrome

Sentence
String text
String toString() boolean equals() boolean isPalindrome

Recursive Functions

- Factorial:

Base Case

$$n! = \begin{cases} 1 & \text{if } n \le 1 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

Recursive step

- Ackermann function:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{otherwise} \end{cases}$$

Q4

Recursive Helpers

- Our isPalindrome() makes lots of new Sentence objects
- We can make it better with a "recursive helper method"
- ```
  public boolean isPalindrome() {
      return isPalindrome(0, this.text.length() – 1);
  }
  ```
  ```

Key Rules to Using Recursion

- Always have a base case that doesn't recurse
- Make sure recursive case always makes progress, by solving a smaller problem
- You gotta believe
  - Trust in the recursive solution
  - Just consider one step at a time

Course Goals for Searching and Sorting: You should be able to …

- Describe basic searching & sorting algorithms:
  - Search
    - Linear search of an UNsorted array
    - Linear seach of a sorted array (silly, but good example)
    - Binary search of a sorted array
  - Sort
    - Selection sort
    - Insertion sort
    - Merge sort
- Determine the best and worst case inputs for each
- Derive the run-time efficiency of each, for best and worst-case

Recap: Search, unorganized data

- For an ***unsorted*** / unorganized array:
  - *Linear search* is as good as anything:
    - Go through the elements of the array, one by one
    - Quit when you find the element (best-case = early) or you get to the end of the array (worst-case)
  - We'll see *mapping* techniques for unsorted but *organized* data

Recap: Search, sorted data

- For a *sorted* array:
  - Linear search of a SORTED array:
    - Go through the elements starting at the beginning
    - Stop when either:
      - You find the sought-for number, or
      - You get past where the sought-for number would be

  - But binary search (next slide) is MUCH better

Recap: Search, sorted data

```
search(Comparable[] a, int start, int stop, Comparable sought) {
    if (start > stop) {
        return NOT_FOUND;
    }

    int middle = (left + right) / 2;
    int comparison = a[middle].compareTo(sought);

    if (comparison == 0) {
        return middle;
    } else if (comparison > 0) {
        return search(a, 0, middle - 1, sought);
    } else {
        return search(a, middle + 1, stop, sought);
    }
}
```

Recap: Selection Sort

- Basic idea:
  - Think of the list as having a sorted part (at the beginning) and an unsorted part (the rest)
  - Find the smallest number in the unsorted part
  - Exchange it with the element at the beginning of the unsorted part (making the sorted part bigger and the unsorted part smaller)

Repeat until unsorted part is empty

Recap: Insertion Sort

- Basic idea:
  - Think of the list as having a sorted part (at the beginning) and an unsorted part (the rest)

  - Get the first number in the unsorted part
  - Insert it into the correct location in the sorted part, moving larger values up in the array to make room

Repeat until unsorted part is empty

Merge Sort

- Basic recursive idea:
  - If list is length 0 or 1, then it's already sorted
  - Otherwise:
    - Divide list into two halves
    - Recursively sort the two halves
    - **Merge** the sorted halves back together

Analyzing Merge Sort

- Use a recurrence relation again:
  - Let $T(n)$ denote the worst-case number of **array access** to sort an array of length n
  - Assume n is a power of 2 again, $n = 2^m$, for some m

- Or use tree-based sketch…