CS161: Algorithm Design and Analysis Recitation Section 1
Stanford University Week of 11 January, 2016

Problem 1-1. (Bogosort)

Consider this clever sorting algorithm from `xkcd.com/1185/`:

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(    )
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

(a) Is `FastBogoSort` a correct sorting algorithm? (i.e. does it return a correctly sorted list for every possible input?)

(b) Assume that we have a $\Theta(N)$ time algorithm for `Shuffle`[1] and a $\Theta(N)$ time algorithm for `IsSorted`. What's the time complexity of `FastBogoSort`?

(c) Assume that valid input to `FastBogoSort` contains only lists of numbers without repeats. On input `list` of length n, what's the probability that `FastBogoSort` returns the correct answer?

(d) An upper-bound for the solution to (c) is $\frac{\log n}{n!}$. Is this bound $O(\frac{\log n}{n^n})$, $\Omega(\frac{\log n}{n^n})$, or both ($\Theta(\frac{\log n}{n^n})$)?

Hint: use Stirling's formula.

Problem 1-2. (Merge Sort and Insertion Sort)

(a) What's the best case asymptotic efficiency for insertion sort? What kind of input makes insertion sort efficient?

(b) What's the best case asymptotic efficiency for merge sort? Does your answer suggest that merge sort on some computer will sort all permutations of a list in the same amount of time?

(c) Consider the list `[4, 1, 3, 2]`. Trace out the steps that insertion sort and merge sort each take to sort the list. Assume that merge sort splits the list in the middle.

(d) At a new software engineering job, you're writing a program to sort many lists with hundreds of elements each. You cleverly chose to use merge sort, because you know that it's $O(n \log n)$ – much better than insertion sort, which is $O(n^2)$. You look at logs for your servers, and you realize that you don't have any memory available and can't afford more servers! What do you do?

[1] Shuffling is a little trickier than it seems. If you're interested, Google the Fisher-Yates shuffle.

(e) Timsort is a (fairly complicated) combination of merge sort and insertion sort that happens to be the default sorting algorithm in the Python programming language. If you were to design a hybrid sorting algorithm like Timsort, how would you combine merge sort and insertion sort?

Problem 1-3. (Recurrences)

(a) Consider the naive algorithm to calculate a factorial:

```
function Factorial(n):
    if n == 0:
        return 1
    return n * Factorial(n - 1)
```

Write a recurrence for this algorithm, and state its complexity.

(b) Consider this algorithm to calculate the n-th Fibonacci number:

```
function Fibonacci(n):
    if n == 1 or n == 2:
        return 1
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```

Write a recurrence for this algorithm, and state its complexity.

Complexity is $O(2^n)$.

(c) Do you think this complexity could improve if you saved intermediate results?

Problem 1-4. (More practice with asymptotic notation)

(a) Let $k \geq 1$, $\varepsilon > 0$, $c > 1$ be constants. For each blank, indicate whether A_i is in O, o, Ω, ω, or Θ of B_i. More than one space per row can be valid.

A	**B**	**O**	**o**	Ω	ω	Θ
$\log^k n$	n^ε					
n^k	c^n					
2^n	$2^{n/2}$					
$n^{\log c}$	$c^{\log n}$					
$\log(n!)$	$\log(n^n)$					

(b) What's the asymptotic runtime of the following algorithm, which takes as input a natural number n?

```
function DoSomeLoops(n):
    count = 0
    For i in 1..n:
        For j in 1..10:
            For k in 1..n/2:
                count += log(n)
    return count
```

Problem 1-5. (Finding the median)

(a) Describe an $O(n \log n)$ algorithm to find the median in an n-element list.

(b) Consider the following algorithm:

```
function FindKthElement(list, k):
    pivot = random element of list
    (left, right) = partition(list, pivot)
    if k == length(left) + 1:
        return list[k]
    if k <= length(left):
        return FindKthElement(left, k)
    else:
        return FindKthElement(right, k-(length(left) + 1))
```

Explain why `FindKthElement(l, k)` returns the kth largest element of l. (Don't have to prove it.)

(c) Write a recurrence for `FindKthElement`

(d) Modify the recurrence to find the worst case complexity of FindKthElement.

(e) Try to modify your solution to find the recurrence for the best-case complexity of FindKthElement. In fact, as you'll see later, this best-case happens to also be the average case[2].

[2] It turns out you can modify this algorithm to be linear time in the worst case, but it requires finding 5 medians and is generally unwieldy.