Problem 1-1. (Motivation)

Exciting Examples of Algorithms:

(a) **Protein Folding and the Schrodinger Equation** - attempt to predict the 3D structure of proteins based on their amino acid sequence. If we could do this, it would revolutionize the way we treat disease. While an algorithm derived from the Schrodinger Equation exists, at present an efficient one does not exist. Hence, it is more practical to just "let the universe compute the answer," *i.e.*, wait and see what happens, rather than attempting to *a priori* predict the outcome.

(b) **Finance** - being able to predict the stock price of a company based on past data would be very exciting. Before investing millions of dollars, however, we absolutely want to be sure that the algorithm we're using is correct. Also need it to be efficient (its not very useful if the algorithm needs 2 days to compute tomorrow's stock price).

(c) **Google Maps (Uber, Doordash, Amazon Delivery, etc.)** - computing shortest or fastest paths and routes from one destination to another. We'll be learning some of these algorithms in this class!

(d) **Security** - examples include RSA public key-private key encryption for securely transferring messages over an open channel, as well as internet security protocols (HTTPS and SSL).

(e) **Artificial Intelligence** - robot navigation (A-Star algorithm), clustering algorithms (K-means), classification algorithms (support vector machine, perceptron).

Problem 1-2. (Induction)

(a) **The Implication** Prove that if n^2+5n+1 is even for some $n \in \mathbb{N}$, then

$$(n+1)^2+5(n+1)+1$$

is also even.

Solution:

We wish to show that if n^2+5n+1 is even for some $n \in \mathbb{N}$, then $(n+1)^2+5(n+1)+1$ is also even. If n^2+5n+1 is even for some $n \in \mathbb{N}$, then that means that we can write it as

$$n^2+5n+1 = 2m$$

for some $m \in \mathbb{Z}$. Now, let us expand $(n+1)^2+5(n+1)+1$ into:

$$n^2+2n+1+5n+5+1$$

Rearranging:

$$[n^2+5n+1]+2n+6$$

Plugging in $n^2+5n+1=2m$ we get:

$$2m+2n+6=2(m+n+3)$$

But this implies that $(n+1)^2+5(n+1)+1$ is even, because we have represented it as 2 times some integer (namely, the integer $m+n+3$. Note that $m+n+3$ is indeed an integer because m,n and 1 are all integers, and the integers are a ring and hence closed under addition).

(b) The Base Case Does this prove that n^2+5n+1 is even for any $n \in \mathbb{N}$? What is the moral of this exercise?

Solution:

No! Part 1 does not imply that n^2+5n+1 is even for any $\mathbb{N}$. The only thing we have proved in part 1 is an implication, i.e. that if you manage to find some $\mathbb{N}$ such that n^2+5n+1 is even, then we guarantee $(n+1)^2+5(n+1)+1$ is also even.

The important lesson here is, the information you get from an implication is only tentative or conditional. In order to actually use it, you must also demonstrate the information that the implication is contingent on, is also true (i.e. the base case).

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.

Solution:

$$T(n) = \begin{cases} \Theta(1) & \text{if n = 0} \\ T(n) = T(n-1) + \Theta(1) & \text{otherwise} \end{cases}$$

Complexity is $\Theta(n)$ (as long as we assume multiplication of arbitrarily large numbers is $O(1)$)

(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.

Solution:

$$T(n) = \begin{cases} \Theta(1) & \text{if n = 1 or n = 2} \\ T(n) = T(n-1) + T(n-2) + \Theta(1) & \text{otherwise} \end{cases}$$

Complexity is $O(2^n)$.

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

Solution:

Yes. We can use a technique called memoization to cache results and skip redundant computation. This changes the recursive algorithm to $O(n)$.

Note that it is possible to compute Fibonacci numbers in logarithmic time; even using memoization, this is not the algorithm you should write to efficiently compute Fibonacci numbers.

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	$2^{n/2}$					
$n^{\log c}$	$c^{\log n}$					
$\log(n!)$	$\log(n^n)$					

Solution:

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
```

Solution:

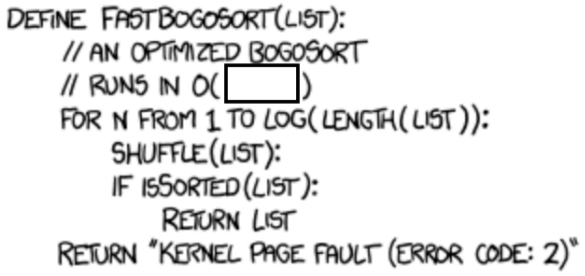
$$\Theta(n^2)$$

.

The outer loop runs $n = \Theta(n)$ times, the first inner loop runs $10 = \Theta(1)$ times per iteration of the outer loop, and the next inner loop runs $n/2 = \Theta(n)$ times per iteration of the first inner loop. Thats $\Theta(n)\Theta(n)\Theta(1) = \Theta(n^2)$

Problem 1-5. (Bogosort)

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



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

Solution:

No. It only works if `Shuffle` happens to sort the list.

(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`?

Solution:

$\Theta(n \log n)$. The outer loop is executed $\log n$ times, and `Shuffle` and `IsSorted` each take $\Theta(n)$. Thus, we have $\Theta(n \log n)$.

(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?

Solution:

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

For simplicity, we're removing repeats so we don't have to deal with lists that are correctly sorted in more than one way. With this assumption, the probability of success on each individual trial is $\frac{1}{n!}$, and we run $\log n$ trials, so, from the binomial distribution, we have

$$\sum_{i=1}^{\log n} \binom{n}{i} \left(\frac{1}{n!}\right)^i \left(1-\frac{1}{n!}\right)^{n-i}$$

Note that this can be bounded above using the union bound, or

$$\sum_{i=1}^{\log n} \frac{1}{n!} = \frac{\log n}{n!}$$

(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.

Solution:

$$\lim_{n\to\infty} \frac{\log n}{n!} / \frac{\log n}{n^n} = \lim_{n\to\infty} \frac{n^n}{n!}$$

By Stirling's formula

$$= \lim_{n\to\infty} \frac{e^n n^n}{cn^n\sqrt{n}} = \lim_{n\to\infty} \frac{e^n}{c\sqrt{n}} = \infty$$

So

$$\frac{\log n}{n!} \in \Omega\left(\frac{\log n}{n^n}\right)$$

Problem 1-6. (Merge Sort and Insertion Sort)

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

Solution:

If a list is already sorted, insertion sort runs in $\Theta(n)$ time. In general, insertion sort performs better on partially sorted lists.

(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?

Solution:

Merge sort's best case is the same as its worst case: $\Theta(n\log n)$. However, this doesn't mean that, in practice, the same number of comparisons are made on an arbitrary permutation of a list.

(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.

Solution:

Merge sort:

```
      [4, 1, 3, 2]
  [4, 1]      [3, 2]
[4]    [1] [3]    [2]
  [1, 4]      [2, 3]
     [1, 2, 3, 4]
```

Insertion sort:

```
[4, 1, 3, 2]
[1, 4, 3, 2]
[1, 3, 4, 2]
[1, 2, 3, 4]
```

(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?

Solution:

Use an in-place sort – i.e. a sorting algorithm that uses $O(1)$ extra space. Insertion sort works, but quicksort or heapsort would probably be better. Depending on the domain and stability needs, other algorithms might work better. Variants of merge sort with constant space also exist.

(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?

Solution:

In general, insertion sort performs better on small lists because it has a smaller constant factor. In addition, it works very quickly on lists that are already mostly sorted.

One thing to do is to modify merge sort to use insertion sort on runs that are smaller than some length ℓ and on longer runs that are already mostly sorted.