Chris Mulligan ⟨clm2186⟩
COMS3157 Data Structures & Algorithms, Hershkop
Theory 1
September 30, 2013

Theory 1

1. We use a simplified model because we're primarily interested in understanding worst case, asymptotic behavior of an algorithm. By ignoring the details of the specific implementation we can focus on the big picture of how an algorithm scales, and compare in a general sense. The difference between an algorithm that completes in $2n$ vs $3n$ does matter, but is completely dwarfed by the difference between $2n$ and n^2, for large values of n.

   A more efficient implementation, or slightly better optimization, will only make a difference on the scale of the fixed cost or multiplier ($3n + 30$ to $2.5n + 18$), it won't change the class the algorithm is.

2. The class of an algorithm is asymptotic - as n gets large, what function is an appropriate upper bound (ignoring multipliers). However this doesn't mean that on a given dataset, or for a given n of a relatively small size, the a "worse" function might not perform better. If the $\mathrm{O}(n \log n)$ class function has a high fixed cost, for small values of n that could dwarf the substantially better asymptotic behavior.

   For example, take our specific functions are $\mathrm{O}(n \log_2 n + 3000)$ and $\mathrm{O}(n^2 + 100)$. If $n = 50$ then $n \log n$ class function will take take 3282 steps, while the n^2 class function takes only 2600 steps. However at larger values of n the expected behavior arises, for $n = 5\,000$ then $n \log n$ takes 64 400 steps, while n^2 takes 25 000 100.

3.
```java
/**
 * @author Chris Mulligan <clm2186>
 * @course COMSW3137
 * @assignment Theory 1, Problem 3
 */
import java.math.BigInteger;
public class Fibonacci {
    static BigInteger TWO;
    public static BigInteger fib(BigInteger n) {
        if (n.compareTo(BigInteger.ONE) <= 0) {
            return n;
        }
        return fib(n.subtract(TWO)).add(fib(n.subtract(BigInteger.ONE)));
    }

    public static void main(String[] args) {
        TWO = new BigInteger("2");
        BigInteger n = new BigInteger(args[0]);
        System.out.print("Running fib\ttailed\tfor\t" + n + "\t...");
        long begin = System.currentTimeMillis();
        BigInteger result = fib(n);
        long end = System.currentTimeMillis();
```

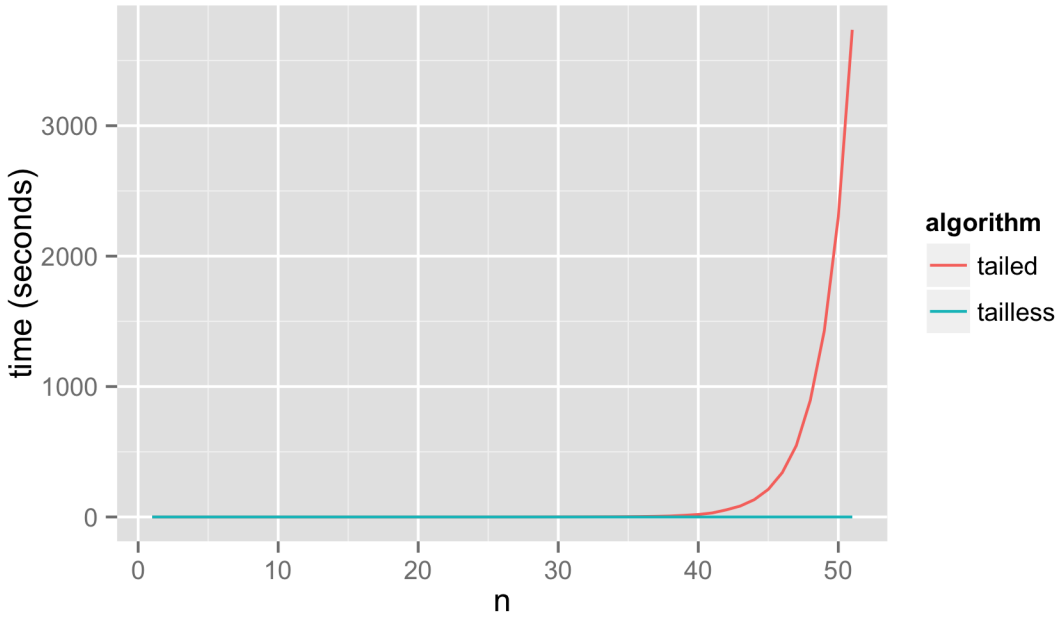
```
            System.out.println(" = " + result + ". Took\t" + (end-begin)/1000.0
                + "\tsecs");
        }
}
```

Unsurprisingly the code fails to execute in any reasonable amount of time, even for a "small" number like F_{100}. The results of my attempt at various sizes show that the function is taking exponentially longer, and is already over a minute at only F_{50}:

```
$ for((i=5;i<=51;i+=1)); do java Fibonacci $i; done
```

algorithm	10	20	30	35	40	45	50	51
tailed	0.001	0.050	0.715	2.432	18.92	211.566	2300.641	3735.449
tailless	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001

Graph of the number of seconds taken to calculate $F_n, n = 1, ..., 51$



4. (a) Given that $\mathrm{O}(n) > \mathrm{O}(\log n)$, since this is a worst case scenario we assume both even and odd execute the $\mathrm{O}(n)$ computation, thus the overall algorithm is, worst case, $\mathrm{O}(n^2)$.

   The best case is that we can assume half the time it executes $\mathrm{O}(\log n)$, so the overall runtime is $\frac{n}{2}(n + \log n)$.

(b) Without knowing anything about the data, we must assume that all locations will contain even numbers - and therefore every computation will take $\mathrm{O}(n)$ time. Therefore Algorithm C would be $\mathrm{O}(n^2)$.

5. Assume we have a function `random()` that returns a random number in $\mathrm{O}(1)$ time.

   One method would be to first create the array, A, and fill it in order, such that $a_i = i, i = 1, \ldots, N$, taking $\mathrm{O}(N)$ time. Then iterate through the array, $i = 1, \ldots, N$, for element i use `random()` to generate a random integer $j \in (1, N)$ and swap a_i with a_j. The run time for a single element is a constant for generating the random number, and a constant for the swap. Thus the overall runtime is simply $\mathrm{O}(N)$.

   Another (worse) method is to create the basic array, A as described above, where $a_i = i, i = 1, \ldots, N$, taking $\mathrm{O}(N)$. Then create an empty array of size N, B, taking $\mathrm{O}(1)$. For $i = 1, \ldots, N$ use `random()` to select a random index $r \in [1, N - i]$ of the initial array A, remove the element a_r and insert it into the next slot b_i. However array A must be collapsed after each removal so it's not sparse. This results in an overall $\mathrm{O}(N^2)$ runtime.

6. Number the elements $e_i, i = 1, ..., N$. Starting with the first element test whether each subsequent element, $e_j \mid j > 1$ can be subtracted from this element (or vice-versa) to equal K. If not, proceed with the second element and test all the remaining elements from e_3. In other words test:
   $(\forall i)(\forall j : j > i)(e_i - e_j = K \text{ OR } e_j - e_i = K)$

   Since in the worst case there is no combination that equals K every element must be tested against every other element, resulting in $\mathrm{O}(n^2)$ run time.

7. From slowest to fastest:
   e) 8
   b) $\log_3 n$
   f) $\log_2 n$
   d) $10n$
   g) $n \log_2 n$
   a) $10n^2$
   a) 2^n
   a) $n!$

   I primarily used intuition and graphs of the functions I created with R. Intuitively it was easy to bin the growth rates into: constant; $\log n$; linear; bigger than linear.

Comparing within each group was slightly more difficult, but using derivatives helped as well. The R code is below:

```
require(ggplot2)
ns <- (0:100000)/10
main <- ggplot(NULL, aes(x=x, color=name)) +
    stat_function(data=data.frame(x=ns, name="a) 10n^2"), fun=function(n)
        10*n^2) +
    stat_function(data=data.frame(x=ns, name="b) log_3(n)"),
        fun=function(n) log(n, base=3)) +
    stat_function(data=data.frame(x=ns, name="c) 2^n"), fun=function(n)
        2^n) +
    stat_function(data=data.frame(x=ns, name="d) 10n"), fun=function(n)
        10*n) +
    stat_function(data=data.frame(x=ns, name="e) 8"), fun=function(n) 8) +
    stat_function(data=data.frame(x=ns, name="f) log_2(n)"),
        fun=function(n) log2(n)) +
    stat_function(data=data.frame(x=ns, name="g) nlog_2(n)"),
        fun=function(n) n*log2(n)) +
    stat_function(data=data.frame(x=ns, name="h) n!"), fun=function(n)
        gamma(n+1)) +
    ylab("f(n)") + xlab("n")

main + coord_cartesian(y=c(0, 25))
main + coord_cartesian(x=c(0, 500), y=c(0, 1000))
main + coord_cartesian(x=c(0, 20), y=c(0, 2000))
main + coord_cartesian(x=c(0, 20), y=c(0, 10000))
main + coord_cartesian(x=c(0, 2500), y=c(0, 20000))
```

8. (a) Exactly: $1 + 1 + 3(1 + 2 + 1 + n(2 + 2 + 1)) = 17 + 18n$. Which is of the class $\mathrm{O}(n)$.

   (b) Exactly: $1 + 1 + n^2(2 + 2 + 2) = 2 + 6n^2$. Which is of the class $\mathrm{O}(n^2)$.

   (c) Exactly: $1 + n(n(1 + 2 + 1) + n \log n + 2 + 1) = 4n^2 + n^2 \log n + 3n + 1$. Which is of the class $\mathrm{O}(n^2 \log n)$.

   (d) Exact runtime depends on the value of n. The exact run time if n is even is $1 + 1 + 1 + n(2 + 2 + 1) = 3 + 5n$. If n is odd $1 + 1 + 2 = 4$. However we must assume worst case, hence this is $\mathrm{O}(n)$.

9. Delete the head. Then set `next.next = next.next.next`. Then set `this = next.next` and repeat. By bypassing every other node you'll force the garbage collector to clean up after you, and delete all the odd elements in the list.

   This will take $\mathrm{O}(n)$.

10. Take each element in the list, and as you encounter it either add it to an AVL tree that contains a tuple of (element, times), or increment the existing node. This will take $n \log n$ work. Then go through the tree and check every number to see how many times it appeared. Worst case that's n work, so overall $\mathrm{O}(n \log n)$.

11. Implementation in terms of choosing various algorithms determines your general growth rate, which greatly impacts whether certain solutions are viable or not. If the a given algorithm is $\mathrm{O}(n!)$ it effectively can't be used for any non-trivial n. However if it can be rewritten to be something like $\mathrm{O}(n \log n)$ it may be a viable solution even for reasonably large values of n. For example if $n = 100\,000$ and our system can calculate $10\,000$ items per second, then $n! \approx 2.82 \times 10^{456573}$ steps taking many of orders of magnitude longer than the universe will last, while $n \log_2 n \approx 1.66 \times 10^6$ steps taking only 2.7 minutes.