Problem Set 1 Comments

Erika Chin, Erin Golub, and Phu Le

February 6, 2006

1. Prove that $n! \in O(n^n)$

We need to prove that $n! \in c * n^n$ where $n > n_0$ and c is a constant.

Recursion relations:
Let $n! = f(n)$
$f(n) = n * f(n-1) \quad f(1) = 1$
Let n^n be $g(n)$
$g(n) = n * g(n-1) \quad g(1) = n$

Logically you can see that when written out, $n!$ has the same number of terms as n^n does. Comparing $f(n)$ and $g(n)$, we have:
$n * (n-1) * (n-2) * \ * 1$ compared to $n * n * n * \ * n$

Both have the same number of terms (n terms) but factorial's terms decrease down to 1, a constant, while n^n keeps on multiplying by n. Since the number of terms are comparable and n is always going to be larger than n-constant, every term in n^n is larger than its counterpart in $n!$ (not counting the first term, which is equal). Therefore, $n! \leq n^n$, so: $n! \in O(n^n)$. We can choose $c = 1$ and $n_0 = 1$ since we have established that $n!$ is always less than or equal to n^n.

2. Prove that $n! \in \Omega(2^n)$

We need to prove that $n! \geq c * 2^n$ where $n > no$ and c is a constant: $2^n = 2 * 2 * 2 * ... * 2$ (n number of 2's)
$n! = n * (n-1) * (n-2) * ... * 1$
Both functions have the same number of terms (n terms) but $n!$ includes terms with the value n in it, while 2^n only has 2 as the term. If n_0 is 10, then all terms in $n!$ are larger than 2 except for the last term, 1. Since the number of terms are comparable and the $n!$ terms are generally larger, then $n! \geq 2^n$ so: $n! \in \Omega(2^n)$

Another way to think about it: Say $n = 4$. So $4! = 24$. $2^4 = 16$. Clearly, $24 > 16$. As you increase n, the value of the factorial will increase by a factor of n and the value of the 2^n will increase only by a factor of 2. Since we started with $n = 4$, the factorial will always be larger than $2n$. So, n^0 can be $= 4$ and $c = 1$, and $n! \geq 2^n$, so $n! \in (2n)$.

3. Prove that $n^\alpha \in o(c^n)$ for any $\alpha > 0$ and $c > 1$

We need to prove that $var^{const} < const^{var}$. We can compare the functions as n goes off to infinite, by taking the $(n+1)^{th}$ term over the n^{th} term. This will essentially compare the slopes of the functions as n goes to infinite.

n^α: $\lim_{n\to\infty} \frac{(n+1)^\alpha}{n^\alpha} =$
$= \lim_{n\to\infty} (\frac{n+1}{n})^\alpha$
$= 1$

c^n: $\lim_{n\to\infty} \frac{c^{n+1}}{c^n}$
$= c$

Since $1 < c$ (and $c \neq 1$, as given by the parameters), that must mean that $n^\alpha < c^n$ (and not just $n^\alpha \leq c^n$), therefore $n^\alpha \in o(c^n)$.

Alternatively, you can compare two equations by putting one over the other, then taking the limit of n to infinity. If you get ∞/∞, you can take the derivative of the both terms and take the limit again. If the end result goes to $\infty/0$, than the numerator is the larger term. If the end result goes to 0, then the denominator is the larger term. This is called L'Hopital's rule.

The derivative of a polynomial, n^k is $k * n^{k-1}$. The derivative of an exponential, a^n, is $a^n * \ln a$. As you can see, if you take the derivative of a polynomial, over and over again, you eventually get a constant with no n terms. If you take the derivative of an exponential, it still goes to infinity.

$$\lim_{n\to\infty} \frac{c^n}{n^k} = \frac{\infty}{\infty}$$

so apply L'Hopital's rule to get

$$\lim_{n\to\infty} \frac{c^n}{n^k} = \lim_{n\to\infty} \frac{c^n \ln a}{kn^{k-1}} = \frac{\infty}{\infty}$$

if $k > 1$. So keep applying L'Hopital's rule until you reach

$$\lim_{n\to\infty} \frac{c^n}{n^k} = \lim_{n\to\infty} \frac{c^n (lnc)^k}{n!} = \infty$$

Thus $n^k = o(c^n)$.

4. Determine the asymptotic running time of the list `append` operation as a function of the input list size

We used the following Python code to measure the time `append` takes:

```
alist = []
for k in range (10):
    total = 0;
    for j in range (10):
        timer = Timer.Timer ()
        timer.start ()
        count = 0
        for i in range (100000):
            count += 1
            alist.append (i)
        timer.stop ()
        total += timer.elapsed ()
    total /= j + 1
    print "AVG %d: %2.6f" % (k, total)
```

Output results:

AVG 0: 0.129934
AVG 1: 0.130687
AVG 2: 0.126835
AVG 3: 0.140032
AVG 4: 0.133960
AVG 5: 0.139781
AVG 6: 0.140540
AVG 7: 0.146938
AVG 8: 0.141221
AVG 9: 0.173969

The AVG is the average time of appending 100,000 elements to the list in which the size is increasing by 100,000. The results indicate that the append time is independent from the size of the list. Therefore, from the times above, it makes sense to say that time per operation stays statistically signifficantly constant. This implies that Python keeps a pointer to the last item in the list, in which case it does not make a difference how long of a list we have. So, the average running time of append appears to be in $\Theta(1)$.

5. Determine the asymptotic complexity of `insert`

The time of an insert operation may be affected by two properties of the input: the size of the list, n and the location index where the new value is inserted, l.

We used the following Python code:

```
alist = []
size = 1000000
inc = int(.2 * size)
timer = Timer.Timer ()
for i in range (size):
```

```
    alist.append (i)
for j in range(10):
    timer.start ()
    for k in range (100):
        alist.insert(1+(j*inc), 1)
    timer.stop ()
    #total is the time of inserting 100 new element.
    total = timer.elapsed ()
    print "AVG %d: %2.6f" % (j, total)
```

Data produced is as follows:
AVG 0: 1.199500
AVG 1: 0.959843
AVG 2: 0.728636
AVG 3: 0.485241
AVG 4: 0.242937
AVG 5: 0.000317
AVG 6: 0.000187
AVG 7: 0.000211
AVG 8: 0.000192
AVG 9: 0.000201

If you graph this data, the graph will appear slanted downward with a negative slope. From this graph we can conclude that the the list appears to be implemented like a stack with the pointer at the end of the list. The length of the list makes a big difference in how long it takes to access elements at the beginning of the list.
Let n = length of the list and let i = insertion index, then the list is $O(i)$ and in the worst case $i = n$ so we predict that the average running time for `insert` is in $\Theta(n)$.

6. Speculate on how Python implements lists.

Let's consider the slice operation. Use the following python code:

```
#the code will calculate the time of 100 accesses,
#the experiment will run 10 time and each with a different index.

alist = []
size = 1000000
inc = int(.2 * size)
timer = Timer.Timer ()
for i in range (size):
    alist.append (i)
for j in range(10):
    timer.start ()
    for k in range (100):
        alist[1+(j*inc):1+(j*inc)]
```

```
    timer.stop ()
    total = timer.elapsed ()
    print "AVG %d: %2.6f" % (j, total)
```

The run of this experiment produces the following results for 100 elements:

AVG 0: 0.000136
AVG 1: 0.000144
AVG 2: 0.000141
AVG 3: 0.000138
AVG 4: 0.000156
AVG 5: 0.000141
AVG 6: 0.000143
AVG 7: 0.000158
AVG 8: 0.000144
AVG 9: 0.000157

Since all of the values are pretty much very similar, the list length did not have much impact on the results. There appears to be constant time needed to directly access a list element; therefore the access/slicing function has the time complexity in $\Theta(1)$. The list appears to have elements that are directly accessible like an array.

Given the results of the list access function time, the append function, and the insert function, we can assume that the list is most likely represented consequtively. We should be somewhat wary of this conclusion, however, based on the limited experiments we have done. There are many other implementations that would have the properties we observed, and many experiments we did not do yet that could produce results that would invalidate our guesses here. In PS4, we will examine the Python list implementation code to determine what is really going on here.

7. Implement the missing `goodnessScore` procedure. Your procedure should calculate the goodness score of two genomes with gaps inserted.

Here is an implementation:

```
# pre: U and V are strings; c and g are numbers
# post: Returns the score of the goodness of the alignment between U and V
def goodnessScore(U,V,c,g):
    #initialize the score to Zero
    score=0
    minLength= min(len(U),len(V))
    #compare "alignable" character in U and V and reflect the change in score
    for i in range(minLength):
        if U[i]==V[i]:
            score+=c
        if (U[i]== '-') or (V[i]=='-'):
            score-=g
    score -= abs(len(U) - len(V)) * g
    return score
```

8. Determine analytically the asymptotic time of the `bestAlignment` procedure.

In the bestAlignment procedure, the function will call itself 3 times recursively if the length of U and V are both not equal to zero. In the worst case where the function always calls itself 3 times, then the upper bound for the time complexity of bestAlignment is $O(3^{len(U)+len(V)})$ because each call to bestAlignment will eventually branch out 3 more calls and the length of the strings goes into the next function call is decreased by 1. However, to find a tighter bound for the time complexity of the bestAlignment function, we have to realize that even in the worst case, the function will not "branch" to 3 more function calls to itself when $len(U) = 0$ or $len(V) == 0$, therefore, if its time complexity grows exponentially, then the base will be less than 3 (more accurately, somewhere between 2 and 3). We can estimate the base of the exponential function by writing the following code to calculate the time of running bestAlignment(U,V,c,g) with different sizes of U and V:

```
def alignmentTime(n):
    timer = Timer.Timer ()
    alist = 'a'
    blist = 'b'
    for i in range(n):
        timer.start()
        bestAlignment(blist,alist,10,2)
        timer.stop()
        print(timer.elapsed())
        alist+='a'
        blist+='b'
```

And the result of running alignmentTime(10) is:

Len(U)+Len(V)	Time	Growth Factor	Sqrt(Growth Factor)
2	0.0000514		
4	0.0002282	4.4402	2.1072
6	0.0011071	4.8507	2.2024
8	0.0056306	5.0858	2.2552
10	0.0300938	5.3447	2.3119
12	0.1612440	5.3581	2.3147
14	0.8909177	5.5253	2.3506
16	4.8112517	5.4003	2.3239
18	26.9680032	5.6052	2.3675
20	150.3840907	5.5764	2.3614
Average			2.2883

From the table, we can estimate that each time the $len(U) + len(V)$ increases by 1, the time complexity increases by about 2.288. Therefore, it is more convincing now that the growth order of the time complexity is clearly exponential: $O(a^{len(U)+len(V)})$ with a is between 2 and 3 and is close to 2.288

9. A typical gene is a few thousand bases long. Predict how long it would take your `bestAlignment` procedure to align two 1000-base pair human and mouse genes.

Using the result from question 8, we can estimate the time to align two 1000-base pair human and mouse genes by the general estimation: 3^{2000}*(time of calculation of 1 function call without recursion). Or if we wanted to get a closer estimate, we can do:

$$\begin{aligned}\text{alignmentTime}(2000) &= alignmentTime(2) * (2.288)^{1000+1000-2} \\ &= alignmentTime(2) * (2.288)^{1998} \\ &\approx 0.0056306 * (2.288)^{1998}(seconds)\end{aligned}$$

10. Suggest approaches for improving the performance of `bestAlignment` and predict how they would affect your answers to 8 and 9.

There are very many ways to improve the `bestAlignment` procedure, so we provide only one small one as an example.

One small suggestion is that we can add the line to check whether or not `U[0]==V[0]` before calling the recursive calls. If they are equal, then their position will be included in the final best alignment sequences and we just need to find the bestAlignment for `U[1:]` and `V[1:0]`. That means we just save one third of the work that would be done by the old function. However, the growth order is still exponential.

See Lecture 4 for an algorithm that solves alignment in $\Theta(|U| + |V|)$.