Jason Lai
204995126
Dis 1B

CS 180
April 11, 2019

Homework 1

1 Problem 4

Given the list of functions

$$g_1(n) = 2^{\sqrt{\log n}}$$
$$g_2(n) = 2^n g_3(n) = n^{4/3}$$
$$g_4(n) = n(\log n)^3$$
$$g_5(n) = n^{\log n}$$
$$g_6(n) = 2^{2^n}$$
$$g_7(n) = 2^{n^2}$$

From the above list, immediately, it can be observed that g_1, g_2, g_6, and g_7, all share a common exponential base 2. From this, ranking them by growth rate depends on the ranks of their exponent; here, it is observed that

$$2^{\sqrt{\log n}} = O(2^n), \text{ since } \sqrt{\log n} = O(\log n) \text{ and } \log n = O(n)$$

$$2^n = O(2^{n^2}), \text{ since } n = O(n^2)$$
$$2^{n^2} = O(2^{2^n}), \text{ since } n^2 = O(2^n)$$

As for g_3, since $\sqrt{(\log\ n)} = O(\log\ n)$, then $2^{\sqrt{(\log\ n)}} = O(2^{\log\ n})$ and since $2^{\log\ n}$ is just n, that means $2^{\sqrt{\log\ n}}$ is sublinear. It can be observed that $n^{4/3}$ is superlinear since $4/3 > 1$, therefore

$$2^{\sqrt{\log n}} = O(n^{4/3})$$

g_4 grows slower than g_3, since if you divide both $n(\log\ n)^3$ and $n^{4/3}$ by n, you get $(\log\ n)^3$ and $n^{1/3}$, and $(\log\ n)^3 = O(n^{1/3})$, so

$$n(\log n)^3 = O(n^{4/3})$$

g_3 grows slower than g_5, since $4/3 = O(\log\ n)$, so

$$n^{4/3} = O(n^{\log n})$$

And intuitively, g_5 grows slower than g_2, since 2^n is much larger than $n^{\log\ n}$, so the following ranking of growth rate by ascending order is obtained.

$$2^{\sqrt{\log n}} < n(\log n)^3 < n^{4/3} < n^{\log n} < 2^n < 2^{n^2} < 2^{2^n} \tag{1}$$

$$g_1 < g_4 < g_3 < g_5 < g_2 < g_7 < g_6 \tag{2}$$

2 Problem 6

Given the algorithm

```
For i = 1, 2, ..., n
  For j = i+1, i+2, ..., n
    Add up array entries A[i] through A[j]
    Store the result in B[i, j]
  Endfor
Endfor
```

a) The function $f(n)$ would yield a bound

$$O(n^3)$$

This comes from the fact that the outer loop of the algorithm iterates through n elements (*O(n)*), the inner loop will iterate through at most *n-1* elements (*O(n)*), and for each element $B[i, j]$, all elements $A[i]$ through $A[j]$ must be summed (*O(n)*; this would probably require an additional loop). So the function for the algorithm in the worst case (upper bound) complexity would be

$$f(n) = n(n-1)(n) = n^3 - n^2 \quad => \quad O(n^3 - n^2) \approx O(n^3)$$

b) For the lower bound of the function $f(n)$, the outer loop must run at least n times in order to reach every row, and by extension, every element of the matrix B. The inner loop however, will always run n - i times for any given i. And for each loop of addition, to calculate the sum of each element in matrix B, the loops will run at least j - i times, so this alludes that the lower bound of the algorithm is

$$\Omega(n^3)$$

Part c on next page

c) A highly unnecessary source of inefficiency in the program is the computation of the sum $A[i]$ through $A[j]$ for every element $B[i, j]$ in the matrix B. Through each element $B[i, j]$, a new sum must be recalculated and stored, starting over from the element i to the element j in the array A. However, a better algorithm to reduce the running time of this action, would be to use the sum of the previous element in order to find the sum for the current element. This is derived from the fact that, for every element $B[i, j]$, the sum for that element is simply $B[i, j\text{-}1] + A[j]$, since $B[i, j\text{-}1]$ sums $A[i]$ through $A[j\text{-}1]$ and $B[i, j]$ sums $A[i]$ through $A[j\text{-}1] + A[j]$. So the new algorithm should maintain the two outer loops, but use a variable to hold the sum of previous element.

Proposed new algorithm

```
Previous_sum = 0
For i = 1, 2, ..., n
  Previous_sum = A[i]
  For j = i+1, i+2, ..., n
    Previous_sum += A[j]
    Store the result in B[i, j]
  Endfor
Endfor
```

3 Problem 7

Given a song belonging to the specified class of folk songs, to encode the song, the script should only contain the unique lines of that song in order. This is because, for singing the song, you don't need to repeat the lines for every verse in the script, but rather, every time you sing a verse, you just start from the top of script and read down to the new line that would have been added to the previous verse.

Since you read the lines starting with the first line, then reading the second line in addition to the first line, and so on and so forth, you obtain an arithmetic sequence that is the sum 1 + 2 + ... + k which sums to 1/2(k)(k - 1) which is approximated to $1/2(k + 1)^2$. This is the number of words n that are sung out loud for a given song, however, at most the song can have ck words, since each line can have at most c words. So the upper bound for the total number of words $n < 1/2(ck + 1)^2$. Solving for n, you obtain the function f(n)

$$n < 1/2(ck + 1)^2 \tag{3}$$

$$\sqrt{2n} < ck + 1 \tag{4}$$

$$\sqrt{2n} - 1 < ck \tag{5}$$

$$f(n) = \sqrt{2n} - 1 < ck \tag{6}$$

therefore,

$$f(n) = O(\sqrt{n}) \tag{7}$$

4 Problem 8

a) To make effective use of the $k = 2$ jar budget, an algorithm to find the highest safe rung would use the first jar to the probe the entire n rung ladder at intervals of $\sqrt{n}$. Continue doing this until either the jar breaks or the highest rung is reached. If the jar breaks, we know that the highest safe rung is somewhere within that interval of $\sqrt{n}$ rungs we had just skipped, so, using the second jar, probe those $\sqrt{n}$ rungs until the highest safe rung is found. This yields a function where, for the first jar, the runtime will take at most $n/\sqrt{n} = \sqrt{n}$, and for the second jar (if it's necessary), its runtime will be $\sqrt{n}$. So the function would be

$$f(n) = n/\sqrt{n} + \sqrt{n} = 2\sqrt{n}$$

therefore,

$$O(\sqrt{n})$$

This complexity is sublinear as, for n sufficiently large, f(n)/n will approach 0.

b) For a budget of $k > 2$ jars, an algorithm to find the highest safe rung would be to use the same algorithm as the $k = 2$ jar budget problem, but use k as the index of the root for the intervals at which probe, i.e. for a budget k, probe the entire n rung ladder at intervals of $n^{1/\mathrm{k}}$ until either the jar breaks or the highest rung is reach; if the jar breaks, then use the second jar to probe all $n^{1/\mathrm{k}}$ rungs in that interval. While this does increase the number of probes along the entire ladder, this also decreases the amount of rungs to probe in the interval when a jar breaks, so this means for every increase in budget $k > 2$, the growth rate of the function is slower.