Problem Set 1

All parts are due Thursday, October 1 at 11:59PM.

Name: David Walter

Collaborators: Cooper Sloan, Elizabeth Cox

Part A

Problem 1-1.

(a) Part a

$f_4 = O(f_7)$, $f_7 = O(f_2)$, $f_2 = O(f_3)$, $f_3 = O(f_5)$, $f_5 = O(f_6)$, $f_6 = O(f_1)$
Constants do not affect the behavior much as the function approaches infinity. Taking the logarithm of a function decreases the order of growth. Multiplying a function by n increases the order of growth. Squaring and cubing a function increases the order of growth. Exponentiating a function to the power of n increases the order of growth. $f_6' = n + 2nlog(n)$ and $f_1' = 12n^2$ and by L'hopital's rule, we can see that f_1'/f_6' is approximately $n/\log(n)$ and so $f_6 = O(f_1)$.

(b) Part b

$f_3 = O(f_4)$, $f_4 = O(f_2)$, $f_2 = O(f_1)$
Multiplying the exponent of a function by a positive constant greater than or equal to 1 will increase the order of growth. $e > 2$ and a larger number taken to an exponent has a higher order of growth than another number taken to that same exponent. Any number with n as an exponent will approach infinity faster than a number to the power of $\log(n)$.

(c) Part c

$f_2 = O(f_5)$, $f_5 = O(f_4)$, $f_4 = O(f_3)$, $f_3 = O(f_1)$
f_4 is approximately $O(n^5)$, and it follows that f_5 is $O(n^2)$, and by using the sterling approximation on $n! = n\log(n)$, and taking the logarithm of n^5 is $5\log(n)$, and $n\log(n) < 5\log(n)$. An exponential function approaches infinity much faster than a factorial function, and a function that increases in linear time does not approach infinity as fast as a function that increases in factorial time.

Problem 1-2.

(a) $T(n) = \theta(n)$

$$T(n) = (3/2)T(n/2) + n$$
$$f(n) = n$$
$$\log_b(a) = \log_2(3/2)$$
$$f(n) = \theta(n^{log_2(3/2)+\epsilon})$$
$$T(n) = \theta(f(n))$$
$$T(n) = \theta(n)$$

(b) $T(n) = \theta(n^2 \log(n))$

$$T(n) = 4T(n/2) + n^2$$
$$a = 4, b = 2, f(n) = n^2$$
$$\log_b(a) = \log_2(4) = 2$$
$$n^c = n^2$$
$$\log_b(a) = c = 2$$
$$T(n) = \theta(n^{\log_b(a)} \log^{k+1}(n))$$
$$T(n) = \theta(n^2 \log(n))$$

(c) $T(n) = \theta(n^2)$

$$T(n) = T(n-1) + n$$
$$T(n-1) = T(n-2) + n - 1$$
$$T(1) = T(0) + 1 = 1$$
$$T(n) = 1 + 2 + 3 + ... + n = ((1+n)n)/2$$
$$T(n) = \theta(n^2)$$

Problem 1-3.

You have the (n-1) x (n-1) matrix of stock prices organized in 2D time, call it matrix P. You also have another matrix of the same size that you will use to store the minimum stock prices, and 2D time of that minimum price at the current time you are checking at, call it matrix M. So at time (k, l) in matrix P, there will be a corresponding price, and buy time (i and j) stored at (k,l) in matrix M, and that corresponding price will be the minimum price that you could have possibly bought at before selling at time (k, l). The stock price at P[(0,0)] is your first minimum buying price and your first optimal profit will be 0, and your buy time will be (i=0,j=0), and sell time will be (k=0,l=0). When checking matrix P, you start by checking from (0,1) to (0,n-1). If you are checking at (k, l) then you find the minimum of M[(k-1, l)], M[(k, l-1)] and P[(k, l)] and you store that minimum,

and the time of that minimum at M[(k, l)]. You know to compare these three prices (or two prices if either k or m = (n-1)) because (k-1, l) and (k, l-1) are both less than or equal to (k, l) in 2D time, and the minimum prices stored at M[(k-1, l)] and M[(k, l-1)] came from buy times in prior 2D time. The last part of the check for P[(k, l)] is finding the difference between P[(k, l)] and M[(k, l)], if that is higher than the previous optimal profit, then you store your new optimal profit, along with the index of the buy time= (i, j) and the sell time = (k, l). After checking at times (0,1) to (0, n-1), you check at times (1,0) to (n-1, 0), at each time checking the minimum and checking optimal profit as described above. After that, the first row and first column is checked, and you now have the matrix from (1,1) to (n-1, n-1) to check. You then repeat the process and check from (2,1) to (n-1,1), and then from (1,2) to (1, n-1). After those rows and columns are all checked you check from (3,2) to (n-1) and from (2,3) to (2,n-1) and repeat the process until you are only left with (n-1, n-1) and you make your final check. At the end you will return the optimal profit, along with the buy time (i, j) and sell time (k, l).

Part B

Problem 1-4.

(a) *Python script implementation submitted serperately.*

```
def find_amulet(magic_map):
    a = 1
    b = 2
    targeted = False  #tells if the initial range is found
    while not magic_map(a,a+1):
        #find the range of where the amulet is at the start
        if not targeted and not magic_map(a,b): # the amulet is not
                                                #between a and b
            a = b
            b += 2*b
        else: # the index is between a and b
            targeted = True
            if magic_map(a,b-((b-a)/2)): #the index is in
                                         #the lower half
                b -= ((b-a)/2)
                continue
            else: #index is in the upper half
                a += ((b-a)/2)
```

(b) This searching algorithm is basically a binary search algorithm, with the only difference being you have to search for the initial range that you will do binary search on. But the search space is being reduced as you are initially finding the range to search

in. Binary search runs in log n time because the search space is being cut in half at every time step.