DATA STRUCTURES HOMEWORK 1

Benjamin Mazel bm2745

1. WEISS 2.1

Functions separated by an "and" have the same rate

$37, \frac{2}{N}, \sqrt{N}, N, Nlog(log(N)), Nlog(N)\, and\, Nlog(N^2), Nlog^2(N), N^{3/2}, N^2, N^2log(N), N^3, 2^{N/2}, 2^N$

$Nlog(N^2) = 2Nlog(N) = O(Nlog(N))$

2. WEISS 2.6

$F(N) = \$2^N$

if it takes d days to reach a fine of D dollars, $D = 2^d$

$d = log(D) = O(log(D))$.

3. BIG-OH PROBLEMS

a)

The first loop (over i) runs 23 times,$T_1 = O(1)$. The second loop (over j) runs n times $T_2 = O(n)$. The contents of the inner loop have one operation $T_3 = O(1)$. The total run-time is a product of these three $T = O(n)$.

b)

We can rewrite the number of steps for the loops mathematically as $\sum_{j=1}^{n} j$ because it is doing $n - i$ operations for i from 0 to n, which is effectively this sum in reverse order. We know $T = \sum_{j=1}^{n} j = \frac{n(n+1)}{2} = \frac{n^2+n}{2} = O(n^2)$.

c)

We know that $T(n) = 2 + T(n-1)$ as the method itself only runs one Boolean check and one action before returning or calling itself. We also know the method diminishes its input by a factor of k each time, so the most it can run is $log_k(n)$, which gives how many times n can be reduced by *a factor of* k before becoming 1. Thus $T = log_k(n) = \frac{log(n)}{log(k)} = O(log(n))$

4. WEISS 2.11

Let S be the time for a single step so $S * O(n)$ gives the total time

a)$0.5 = S * 100$

$x = S * 500 = 5 * (S * 100) = 5 * (0.5) = 2.5ms$

b)$0.5 = S * 100 * log(100)$

$x = S*500*log(500) = 5S*100*[log(100)+log(5)] = (S*100*log(100))*[5*(1+\frac{log(5)}{log(100)})]$

$x = 0.5 * (5 + 5\frac{log(5)}{log(100)}) = 3.37371ms$

c)$0.5 = S * 100^2$

$x = S * 500^2 = S * 100^2 * 25 = 12.5ms$

d)$0.5 = S * 100^3$

$x = S * 500^3 = S * 100^3 * 125 = 62.5ms$

5. WEISS 2.15

First, one must note that this array contains only integers, and each entry is strictly greater than the last. Thus as a rule, $A_{i+1} \geq A_i + 1$. So as the index i increases by 1, A_i increases by *at least* 1. Given some j, if $A_j > j$, then this relationship will remain true for all $i \geq j$ because $A_i \geq A_j + (i-j) > j + (i-j) = i$ so if we reach a point where $A_i > i$ then we dont even have to bother looking in the further half of the array. Likewise, if $A_j < j$, then this relationship will remain true for all $i \leq j$ because $A_i \leq A_j - (j-i) < j - (j-i) = i$ so if we reach a point where $A_i < i$ then we dont even have to bother looking in the previous half of the array.

Our algorithm now develops as follows: We check the midpoint of the array to determine the relationship between A_i and i. If $A_i > i$ then we look exclusively at the first half of the array and if $A_i < i$ we look exclusively at the second half of the array, again determining the relationship and making a choice as to which half of that array to look at. At each midpoint, if we find $A_i = i$ we return yes and the location (if it is needed). If we reach a point where the section of the array were looking at has length of 1 and $A_i \neq i$ at its midpoint, we can safely return no.

Because each step of this algorithm reduces the length of the array N by a factor of 2, and the check itself is a single step $O(1)$, the maximum number of steps this algorithm takes is the number of times N can be divided by 2 (plus some constant number of steps) thus the runtime of this algorithm is $O(log(n))$.