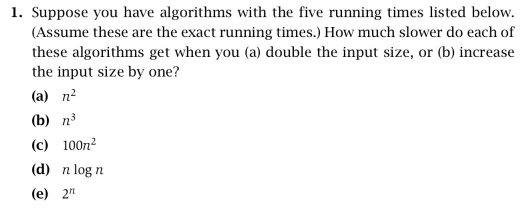
Justin Cabral

CS 5084

Assignment 2

Basics of Algorithm Analysis



N2: **(a)** The code will run slower by a factor of 4.

(**b)** An additive of 2n + 1.

N3**: (a)** The code will run slower by a factor of 8

**(b)** An additive of 3n2 + 3n + 1

100n2: **(a)** The code will run slower by a factor of 4.

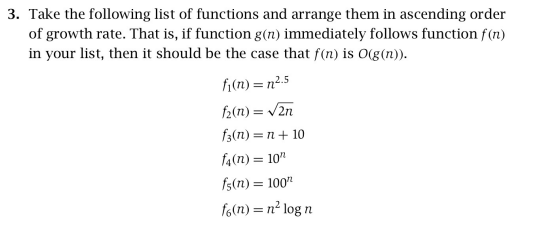
**(b)** An additive of 200n + 100

n log n: **(a)** The code will run slower by a factor of 2 plus an additive of 2n.

**(b)** An additive of log(n + 1) + n[log(n + 1) - log n]

2n:  **(a)** The code will run slower by the square of the previous run time.

**(b)** A factor of 2



**1 – f2** : 2n1/2 = O(n1) and 1 > ½

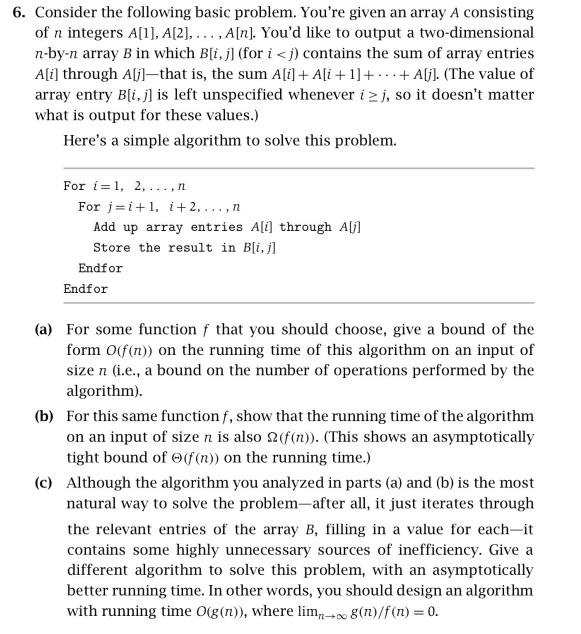
**2 – f3** : n1 + 10 = O (n2 log n) and 2 > 1

**3 – f6 :** n2 log n = O(n2.5) and 2.5 > 2

**4 – f1 :** n2.5 = O(10n) based on the reading from section 2.9 in our book

**5 – f4 :** 10n = O(100n) and 100 > 10

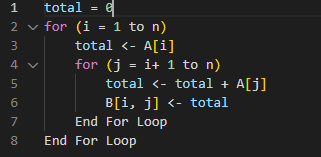
**6 – f5**



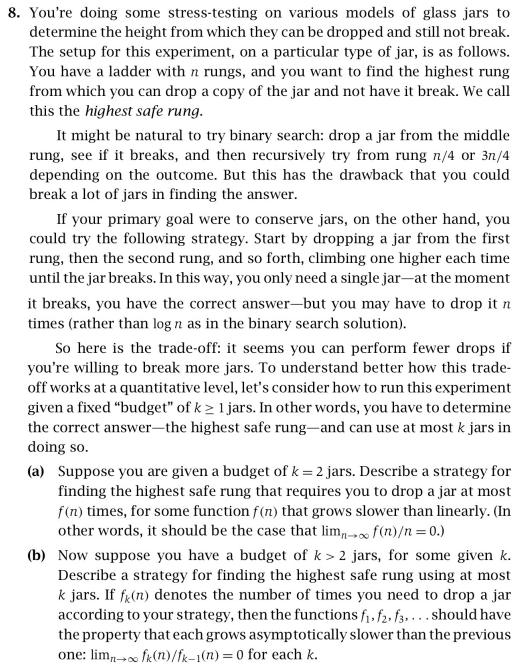
**(a)** For the given algorithm, the outer for loop will run for n iterations, and the inner for loop will run for at most n iterations each time. Knowing this, the line of code that takes entries A[i] through A[j] is executed at most n2 times. Adding the array entries, A[i] through A[j] will sum at most (j – j) + 1 times, which means it will execute O(n) operations. The result stored in B[i,j] requires just constant time, so the entire running time of the algorithm is at most n2 \* O(n), which means O(n3) is the upper bound of our algorithm.

**(b)** The total number of operations done by the outer loop is n, and the total number of operations done by the inner loop is (n) + (n-1) + … ½ n((n + 1) - 1) = ½ n2. With the addition inside the inner loop, it will always run n-1 times because it must iterate from the starting index of A[i], and add all elements until A[j] or the end of the array. Therefore, Ω(n3) is the lower bound for the algorithm.

**(c)**



The Algorithm here is now O(n2) since the addition here was changed to take only constant time.



**(a)** Start by dividing the total amount of rungs, n, into multiples of min(n1/2). Then drop the first jar from the first multiple of n and if it breaks, we will start from that rung and drop the second jar going from each rung up to the rung from which it previously broke, thus getting the highest safe rung. By doing this we have a smaller range to try, limiting the runtime to sublinear.

If the first jar does not break at the first multiple of n, then we’d drop it from the next multiple and continue until it breaks, or we reach the final rung. If it broke, we would do the linear drop sequence with the second jar from the first multiple or previous, all the way up to the rung from which the first jar broke. By doing this there is at most min(n1/2) + C drops and we’d only have to use 2 jars. This means our Big-O time complexity would be O(n1/2), which grows slower than O(n).

**(b)** Start by dividing the total amount of rungs, n, into multiples of min(nk-1/k). The first jar would get dropped from at most 2n1/k and if it breaks at any multiple of min(nk-1/k) then we know it is between min(nk-1/k) rungs. We know this since it would be from either 1 to min(nk-1/k) or a previous multiple of min(nk-1/k) and the multiple it happens to break at.