# Comparison of running times

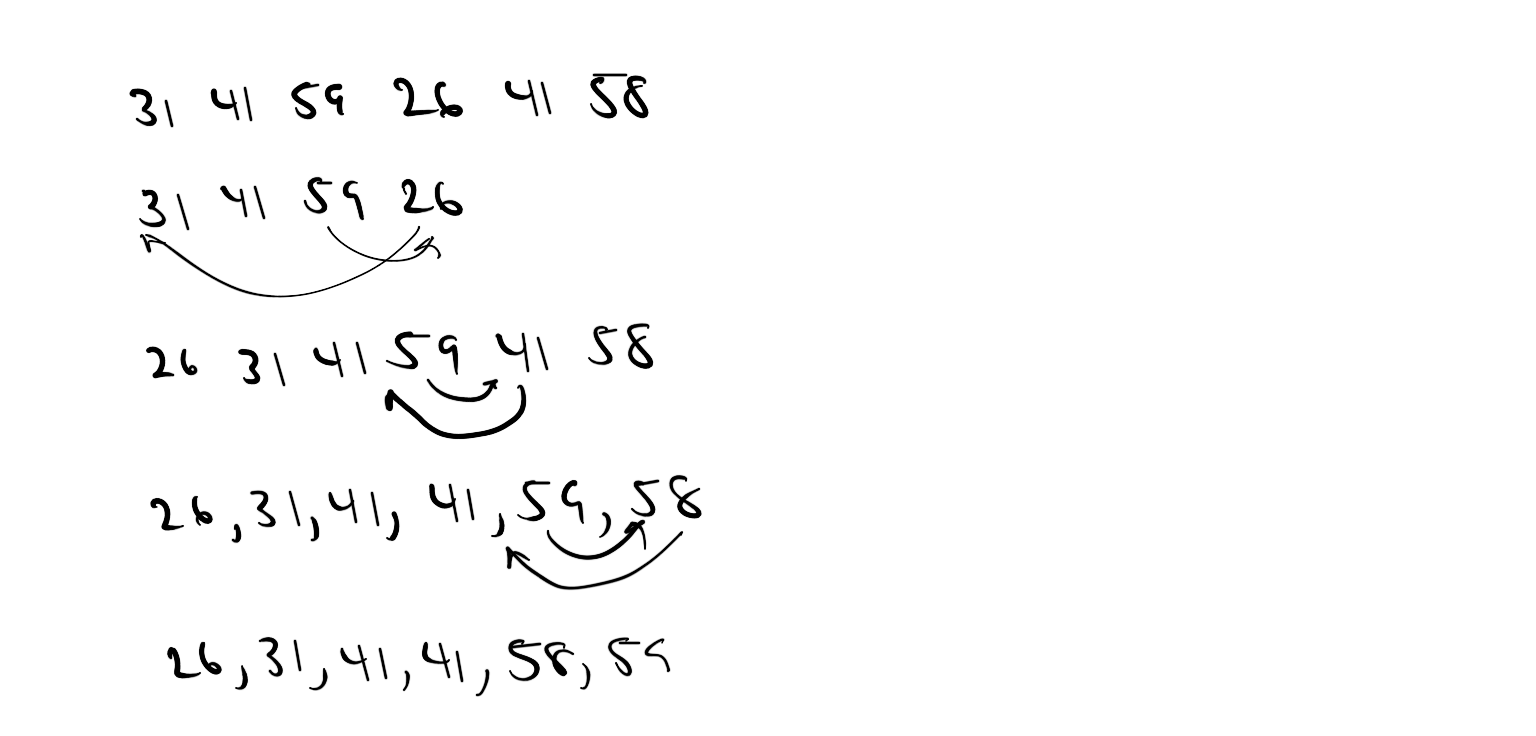
For each function f(n) and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes f(n) microseconds.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 second | 1 minute | 1 hour | 1 day | 1 month | 1 year | 1 century |
|  |  |  |  |  | Really big | Really big | Really big |
|  |  |  | 1.3\*1019 | 7.5\*1021 | 6.8\*1024 | 9.9\*1026 | 9.9\*1030 |
|  | 1.0\*106 | 6.0\*107 | 3.6\*109 | 8.64\*1010 | 2.6\*1012 | 3.11\*1013 | 3.11\*1015 |
|  | 6.3x104 | 2.8x106 | 1.3x108 | 2.7x109 | 6.7x1010 | 8.0x1011 | 6.9x1013 |
|  | 1000 | 7.7x103 | 6.0x104 | 2.9x105 | 1.6x106 | 5.5x106 | 5.5x107 |
|  | 100 | 391 | 1532 | 4420 | 13750 | 31581 | 146589 |
|  | 19 | 25 | 31 | 36 | 41 | 48 | 51 |
|  | 9 | 11 | 12 | 14 | 15 | 16 | 17 |

# 2.1

## - 1

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array A = <31, 41, 59, 26, 41, 58>.



## 2

Rewrite the INSERTION-SORT procedure to sort into non-increasing instead of non-decreasing order



## -3

Consider the searching problem:

**Input**: A sequence of n numbers *A* = <*a*1, *a*2, …., *a*4> and a value *v*.

**Output**: An index *i* such that *v* = *A*[*i*]or the special value NIL if *v* does not appear in A.

Write pseudocode for ***linear search***, which scans through the sequence, looking for *v*. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.



**Loop Invariant**:Ateach iteration of the for loop, A[*j*] != v where *j* < *i*

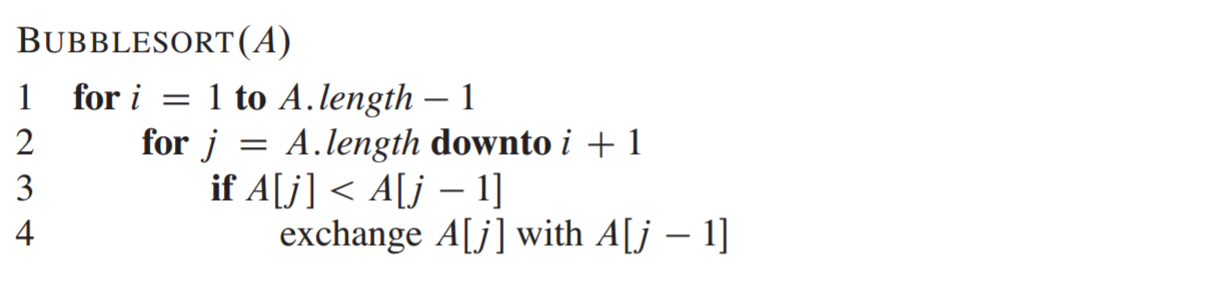
**Initialization**: Before the first loop, the invariant is true as the array is empty

**Maintenance**: The invariant is maintained at each iteration unless A[j] is equal to v, which will result in j being returned, thus ending the loop. Thus there will be no i-th iteration of the loop

**Termination**: The loop ends in two cases, the first being that there aren’t any values where A[*i*] = v and thus NIL is returned. The second being the scenario where A[i] == v, thus *i* is returned, ending the for loop.

# 2.2 *Correctness of bubblesort*

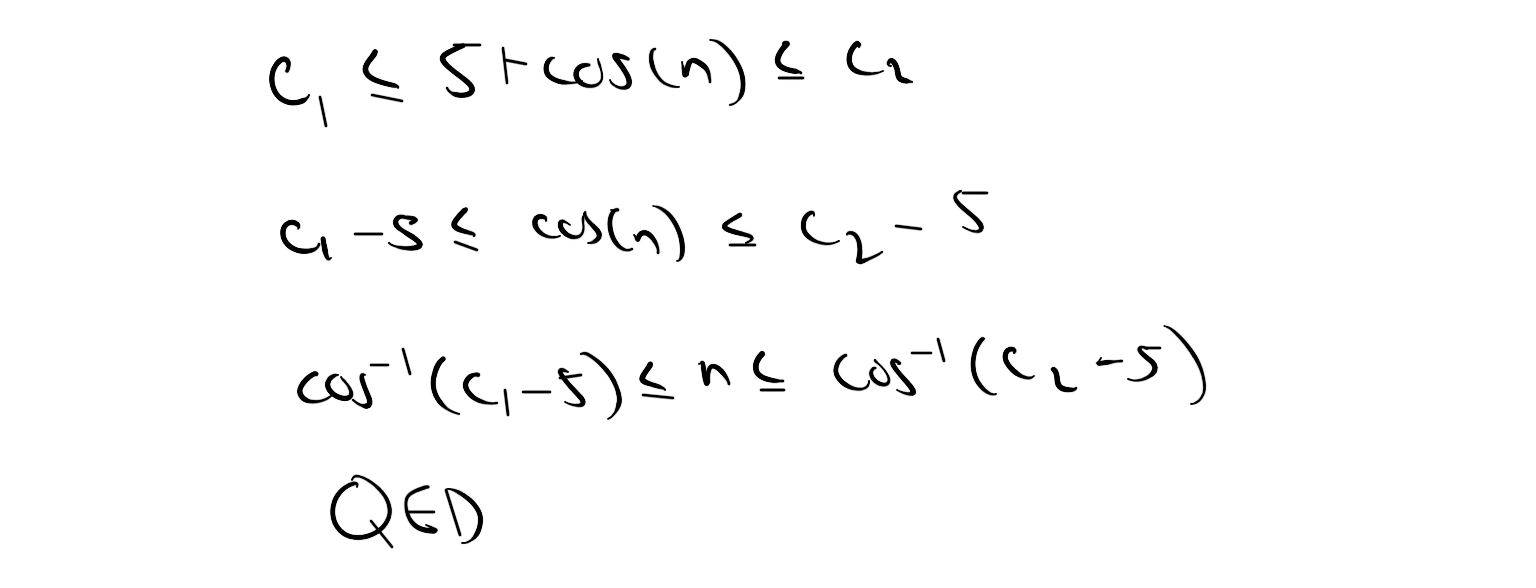
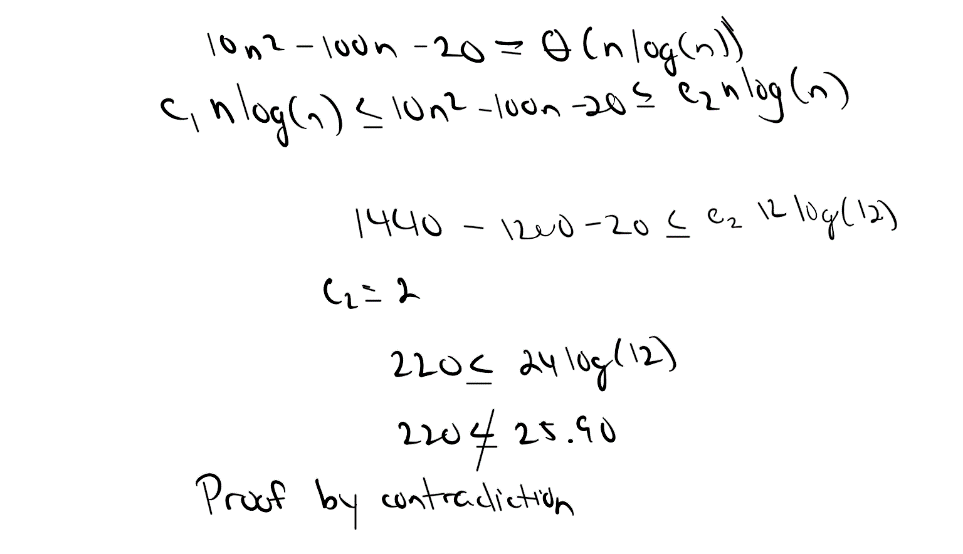
Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order



1. Let A’ denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that A’[1] <= A’[2] <= … <= A’[n] Where n = A.length. In order to show that BUBBLESORT actually sorts, what else do we need to prove?
   1. We need to show that all values contained in the array are the original values, just in sorted order
2. State precisely a loop invariant for the **for** loop in lines 2-4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.
   1. Beginning with each iteration of the for loop, the elements of A[j.. i] are elements originally in the array such that A[j] is the smallest value in the array.
   2. **Initialization**: The invariant is true as A[j] is the last element in the list and the only one existing at the moment, thus correct
   3. **Maintenance:** At each iteration, we swap A[j] with A[j – 1] if A[j] is less than A[j-1]. The invariant holds since A[j-1] becomes the smaller value
   4. **Termination:** After the loop ends, j = i. This means that A[i] is the smallest value in the array and that the array contains the elements in sorted order
3. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1-4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.
   1. **Loop Invariant**: At the beginning of each for loop iteration A[1..i-1] has sorted elements, which are less than or equal to those of A[i..A.length -1]
   2. **Initialization**: At the start, the array is empty, thus holding true
   3. **Maintenance**: When the next iteration occurs, values in A[1..i-1] are in sorted order (smallest to largest in that subarray). Thus they will be smaller than the elements inside A[i..A.*length* - 1]
   4. **Termination**: When the loop ends i=j, thus all values will be in sorted order. We can conclude this since A[i..n] only has elements larger than that of A[1…i-1].
4. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?
   1. The running time is or
   2. Insertion sorts has the same worst-case complexity at . However, both have the same best-case complexity at
      1. Both sorting algorithms have complexity

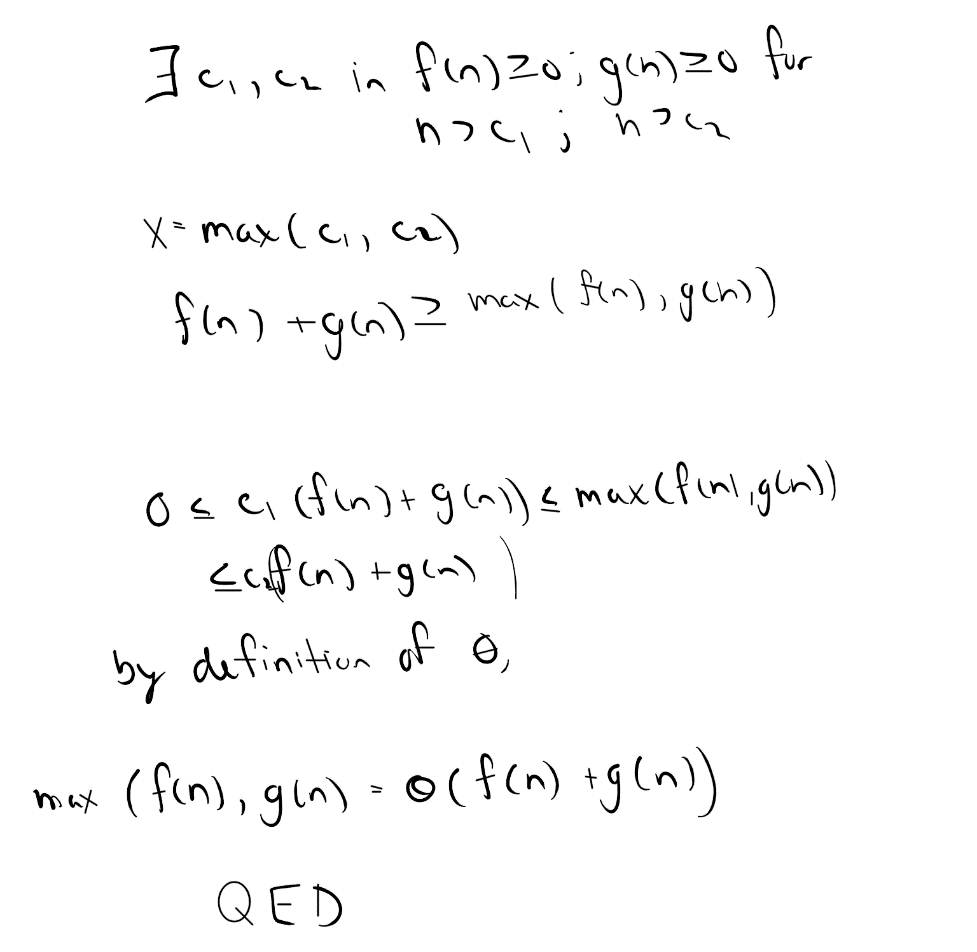
# Prove that the following properties are true:

# Prove the following

* + 
  + 

# Exercise 3.1-1

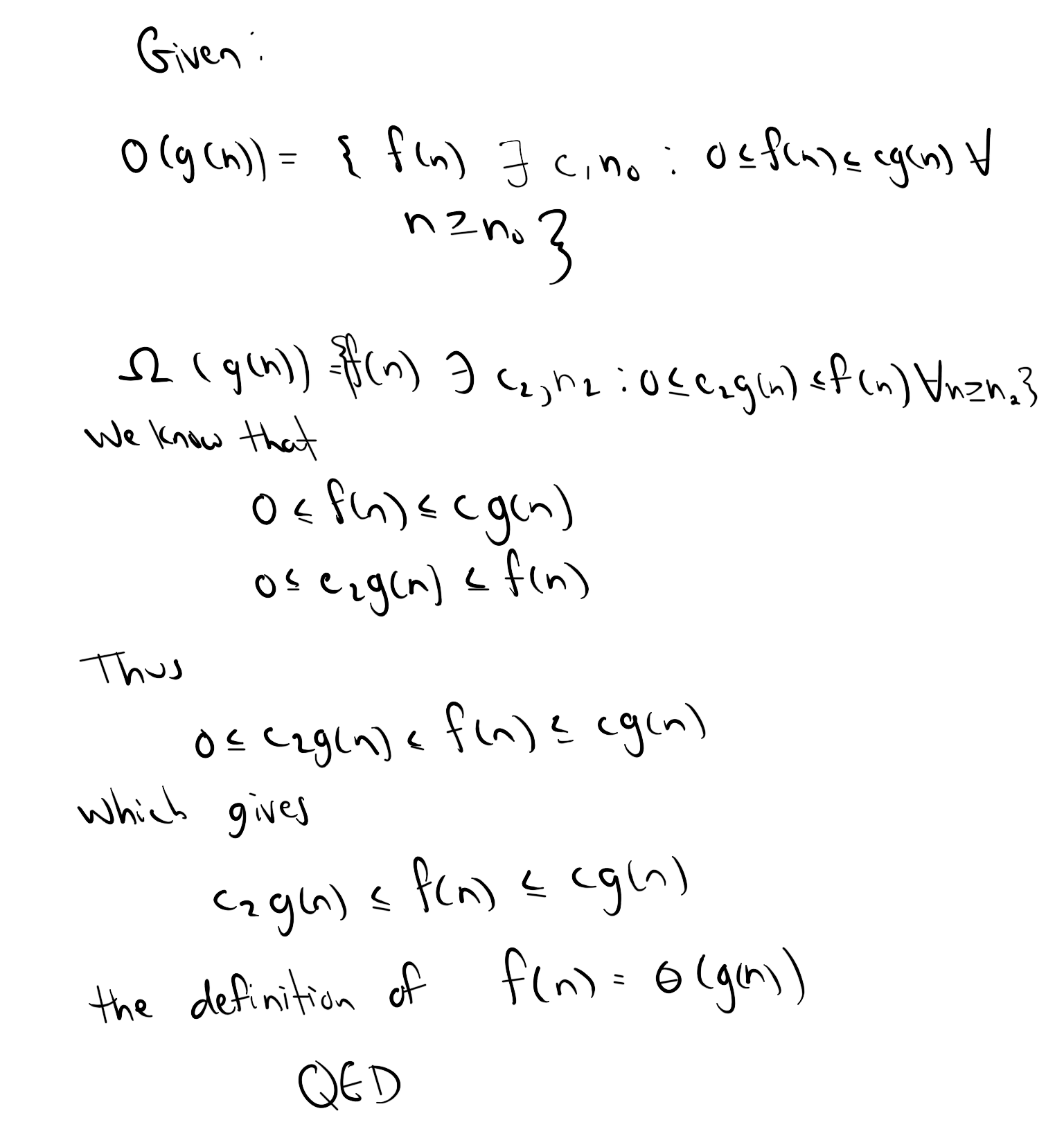
Let and be asymptotically nonnegative functions. Using the basic definition of – notation, prove that



# Exercise 3.1-5

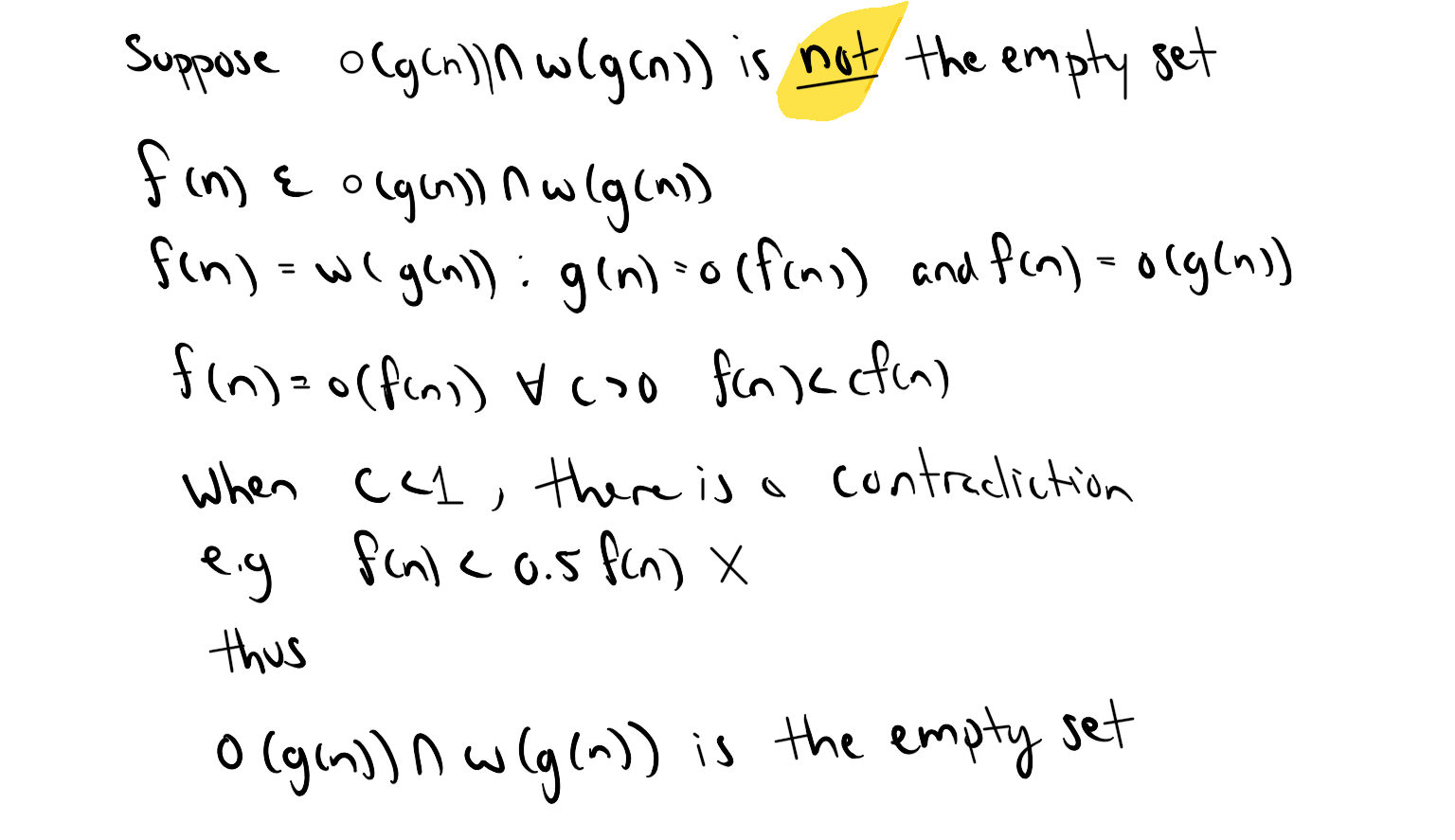
Prove Theorem 3.1.

For any two functions and , we have if and only if and .



# Exercise 3.1-7

Prove that is the empty set.



# Problem 3.2 with short arguments for each answer

Indicate, for each pair of expressions (A, B) in the table below, whether *A* is , , , , or of *B*. Assume that , and are constants. Your answer should be in the form of the table with “yes” or “no” written in each box.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
| *a* |  |  | *yes* | yes | No | No | No |
| *b* |  |  | *yes* | yes | No | No | No |
| *c* |  |  | *No* | No | No | No | No |
| *d* |  |  | *No* | No | Yes | Yes | No |
| *e* |  |  | *Yes* | No | Yes | No | Yes |
| *f* |  |  | *Yes* | No | Yes | No | Yes |

*Short arguments*

A



B



C

Sin prevents convergence to any particular value, thus it’s never always true for any of the notations

D



E

They’re the same function, e.g. they’ll have the same run time

F

Not entirely the same run time, but same concept so the big – Oh, Omega, and Theta are yes