*Wednesday, 12 february 2014*

DIT725 - Time Complexity

*Luuk van Egeraat, Marcus Näslunv & Philip Malm*

**1a.**

f(n) = 1000n^3

g(n) = n^3

f(n) <= cg(n), for constant integer n >= n0 and real number c > 0

let n0 = 1 and c >= 1000

f(1) = g(1)

1000 \* 1^3 = 1000

**1b.**

f(n) = 10n

g(n) = n^2

f(n) <= cg(n), for constant integer n >= n0 and real number c > 0

Base case: let n0 = 10

f(10) <= g(10)

10\*10 <= 10^2

100 <= 100

Step case: n -> n+1

f(n+1) <= g(n+1)

10\*(n+1) <= (n+1)^2

10n + 10 <= n^2 + 2n + 1

Base case proves 10n <= n^2

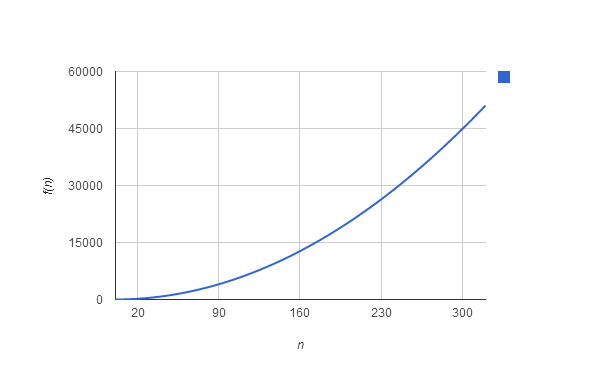
10 <= 2n + 1

This always holds, since n >= 10

**2a.**

Time complexity relates to the amount of times the for- and while-loop are executed for any given array size in the worst scenario.

f(n) = n \* ( n + 1 ) / 2 - 1



f(n) = (n^2 + n) / 2 - 1

= n^2 / 2 + n / 2 - 1

Therefore:

O(f(n)) = n^2

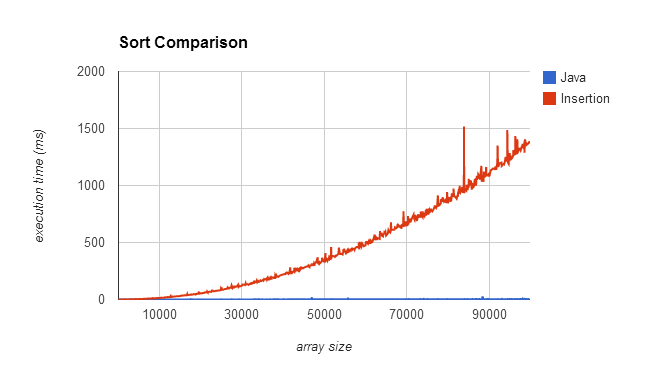
**2b.**

The worst scenario would be the input of integers in reverse order (e.g. { 3, 2, 1 }).

**2c.**

The best scenario would be the input of already sorted integers (e.g. { 1, 2, 3 }). In this case, the while loop would never execute, which results in:  
f(n) = n and O(f(n)) = n

**2d.**



**2e.**

Java’s built-in sort function is O(f(n)) = n  
Below you can see how it slowly increases, linearly, as array size goes up.