countEvens(n): the i for-loop runs len(A) times and each instance of the loop takes constant time. So the algorithm takes linear time in which n=len(A). Its graph also looks like a linear function.

myMin(n): the i for-loop runs len(A)-1 times and each instance of the loop takes constant time. So the algorithm takes linear time in which n=len(A)-1. Its graph also looks like a linear function.

myMax(n): the i for-loop runs len(A)-1 times and each instance of the loop takes constant time. So the algorithm takes linear time in which n=len(A)-1. Its graph also looks like a linear function.

median(n): the whole loop do altogether (1+n)\*n/2 times because the i for-loop do times which decrease by one in each j for-loop. So the algorithm do O(n^2) times and its graph also looks like a n^2 function.

secondBiggest(n): max(A) is a linear function as showed before. So if we do twice max(A) in secondBiggest, we also get a linear function.

LIS(n): the i for-loop runs len(A)-1 times and each instance of the loop takes constant time. So the algorithm takes linear time in which n=len(A)-1. Its graph also looks like a linear function.

dot(n): the i for-loop runs len(A) times and each instance of the loop takes constant time. So the algorithm takes linear time in which n=len(A). Its graph also looks like a linear function.

Insterect1(n): the i for-loop runs len(A)-1 times and each instance of the loop takes constant time. The j for-loop runs len(B) times and each instance is len(A)-1 time. So the whole time is similar to n^2 times. The graph also looks like a n^2 function.

Instersect2(n): The two sorting list procedure both do O(nlog(n)) times and the finding common element takes n times. So the whole procedure takes O(log(n)) times.

fib(n): the recursive Fibonacci function goes to 2^n times because in each loop, each element is multiplied by two. The graph also looks like a 2^n function.