Topic 1: RECURRENCES

1. Asymptotic notation
2. Big Oh notation: f(n) = O(g(n)) is an abbreviation for:
3. There exist **positive constants** c and n0 such that

**0<= f(n)<=c\*g(n),** for all n>= n0

1. G(n) is an **asymptotic upper bond** for f(n).
2. f(n) = O(g(n)) means **that f(n) does not grow substantially faster than g(n)** because a multiple of g(n) eventually dominates f(n).
3. c of interest will be larger than 1, thus enlarging g(n).
4. “Omega” notation: f(n) = Ω(g(n)) is an abbreviation for:
5. There exists positive constants c ang n0 such that

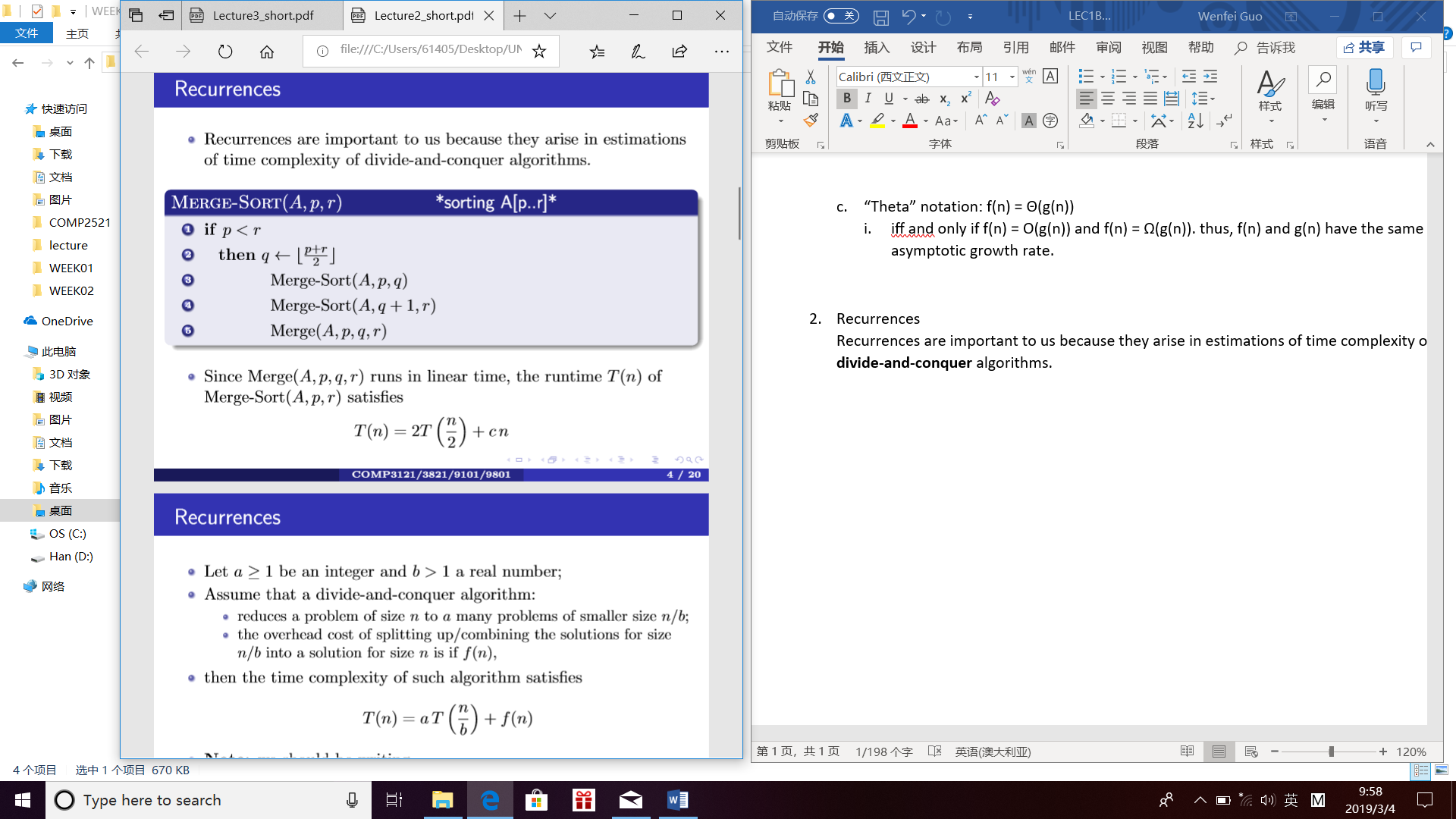
0<=c\*g(n)<=f(n), for all n>= n0

1. G(n) is an asymptotic lower bound for f(n)
2. F(n) grows at least as fast as g(n), because f(n) eventually dominates

a multiple of g(n).

1. Since c\*g(n) <= f(n) if an only if g(n) ≤ 1 cf(n), we have f(n) = Ω(g(n)) if and only if g(n) = O(f(n)).
2. “Theta” notation: f(n) = Θ(g(n))
3. iﬀ and only if f(n) = O(g(n)) and f(n) = Ω(g(n)). thus, f(n) and g(n) have the same asymptotic growth rate.
4. Recurrences

Recurrences are important to us because they arise in estimations of time complexity of **divide-and-conquer** algorithms.



Let a ≥ 1 be an integer and b > 1 a real number;

Assume that a divide-and-conquer algorithm:

reduces a problem of size n to a many problems of smaller size n/b;

the overhead cost of splitting up/combining the solutions for size n/b into a solution for size n is if f(n),

then the time complexity of such algorithm satisﬁes T(n) = aT(n/b) + f(n)

Note: we should be writing

T(n) = aT([n/b]) + f(n)

but it can be shown that assuming that n is a power of b is OK, and that the estimate produced is still valid for all n.

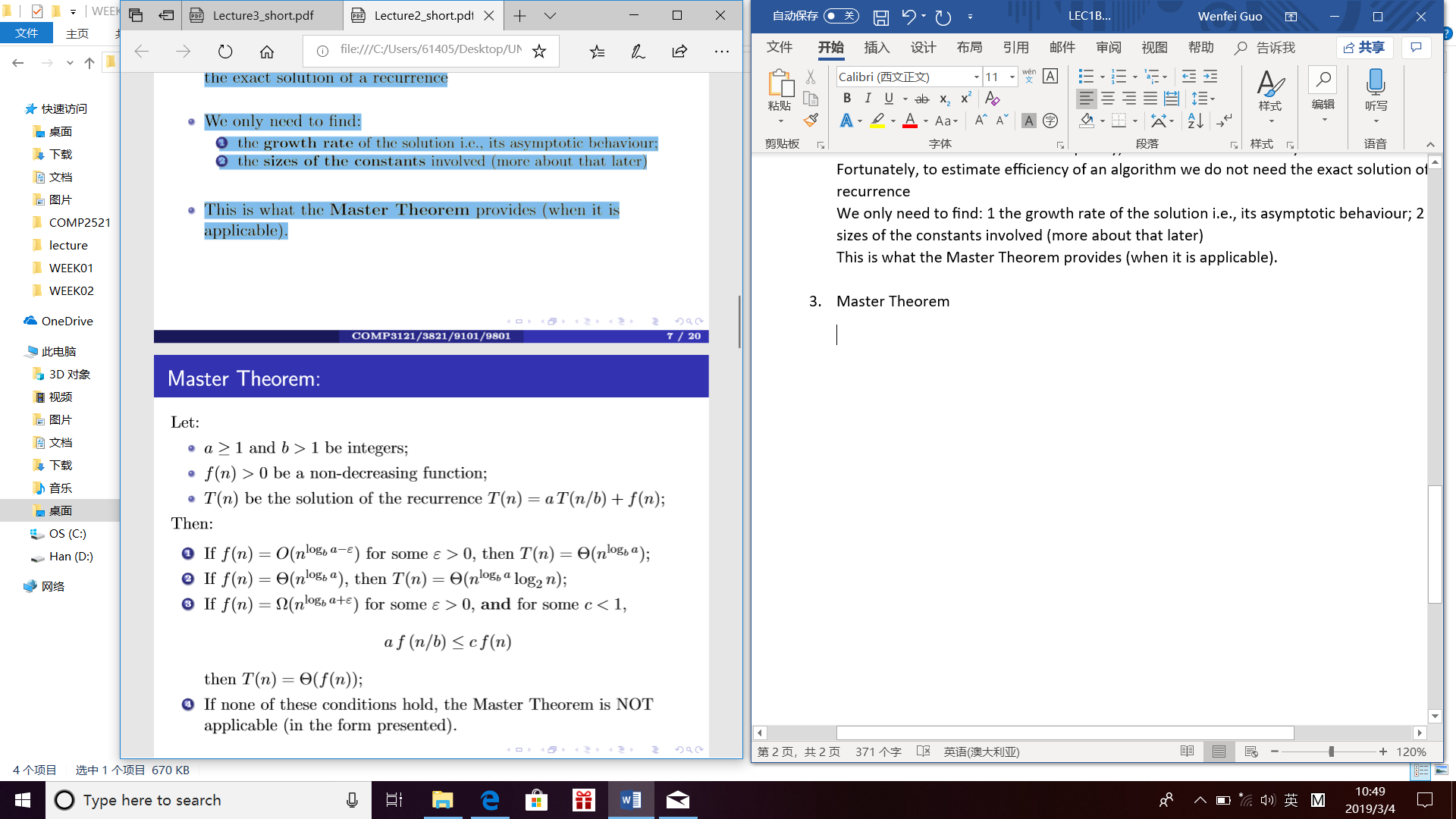
Some recurrences can be solved explicitly, but this tends to be tricky.

Fortunately, to estimate eﬃciency of an algorithm we do not need the exact solution of a recurrence

We only need to ﬁnd: 1 the growth rate of the solution i.e., its asymptotic behaviour; 2 the sizes of the constants involved (more about that later)

This is what the Master Theorem provides (when it is applicable).

1. Master Theorem



* 1. A remark
* Note that for any b > 1,

logb n = logb 2 \* log2 n

* Since b > 1 is constant (does not depend on n), we have for c = logb 2 > 0

logb n = c log2 n;

log2 n = 1/c \* logb n;

* Thus,

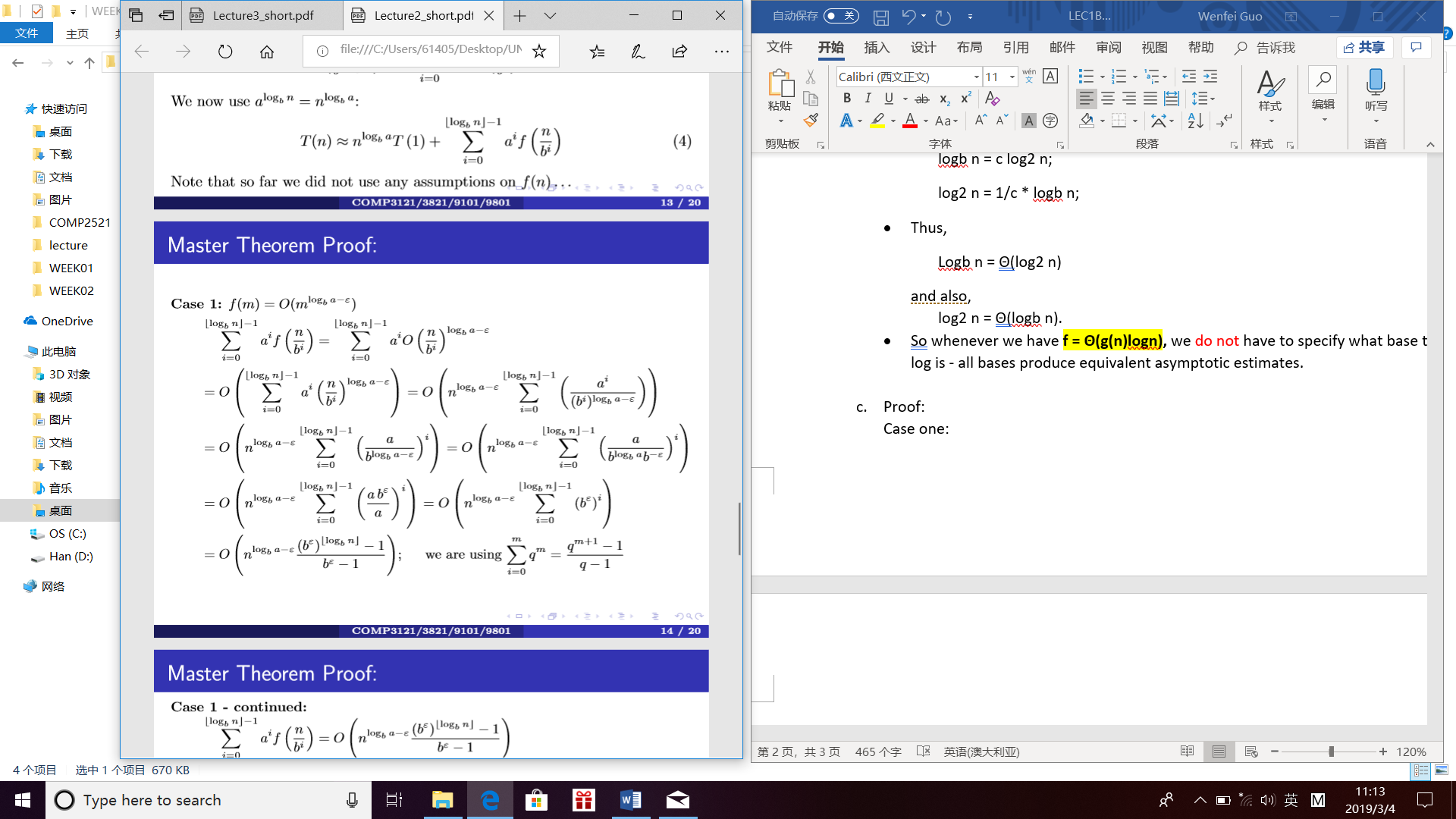
Logb n = Θ(log2 n)

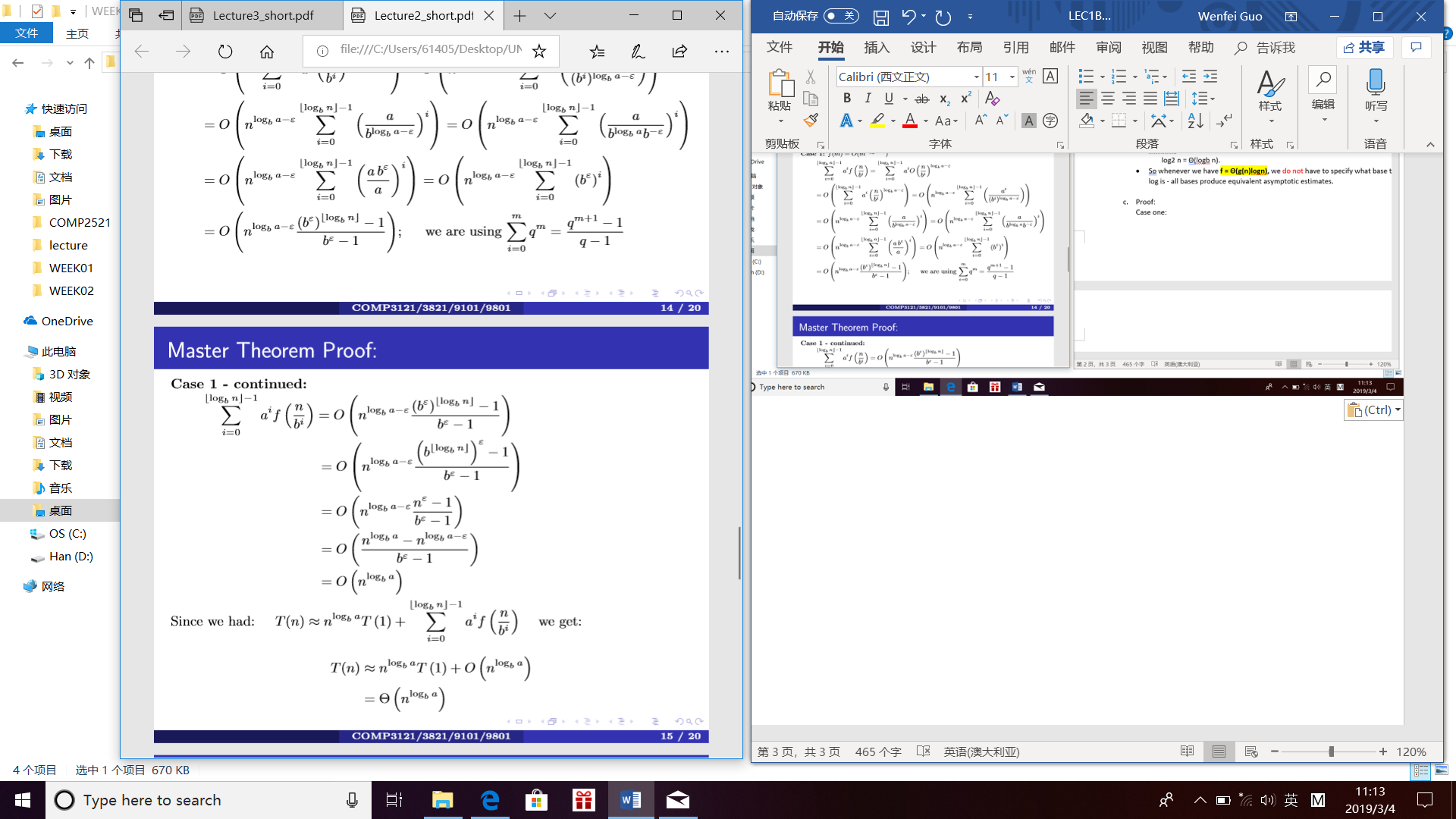
and also,

log2 n = Θ(logb n).

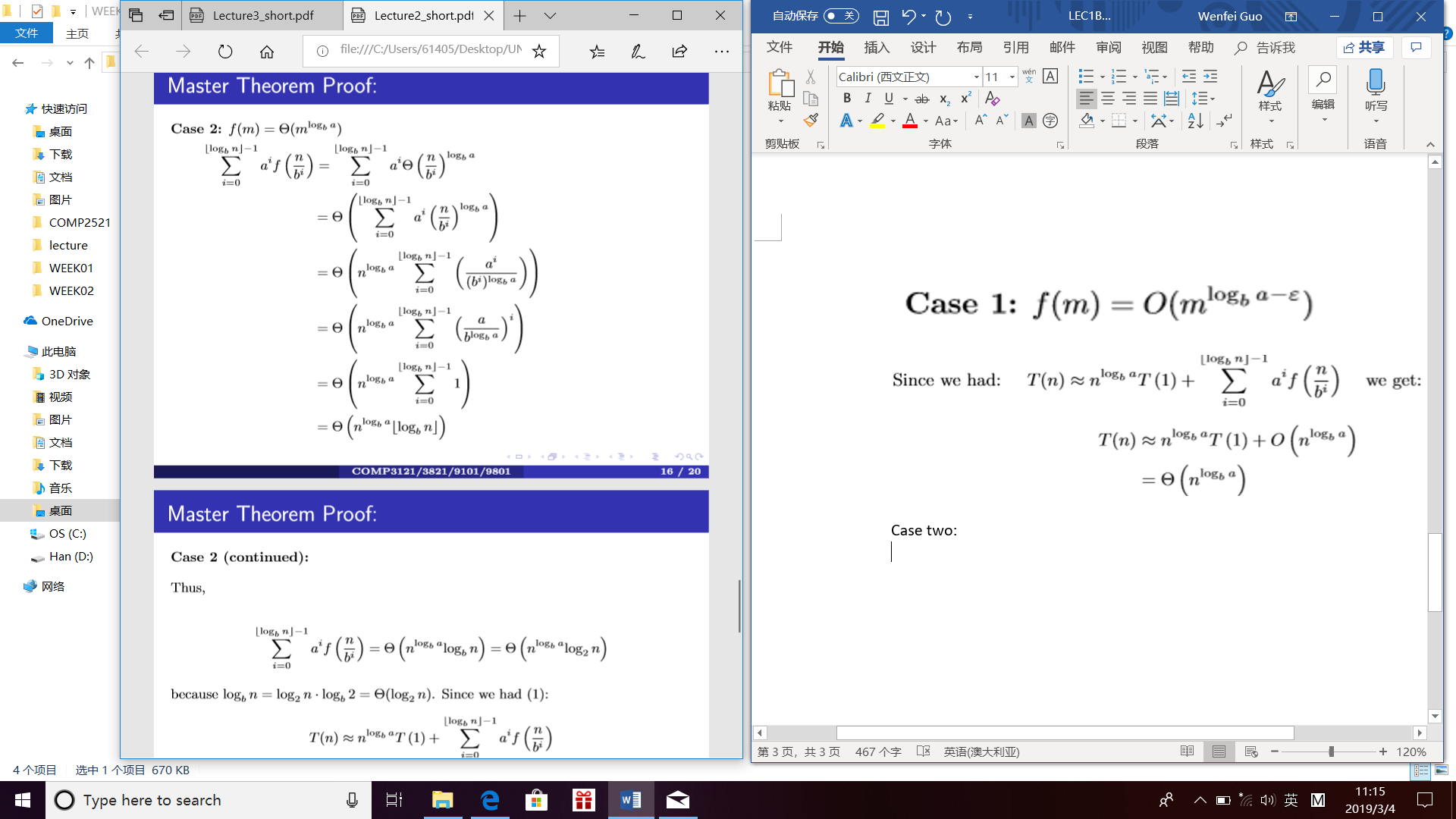
* So whenever we have **f = Θ(g(n)logn),** we do not have to specify what base the log is - all bases produce equivalent asymptotic estimates.
  1. Proof:

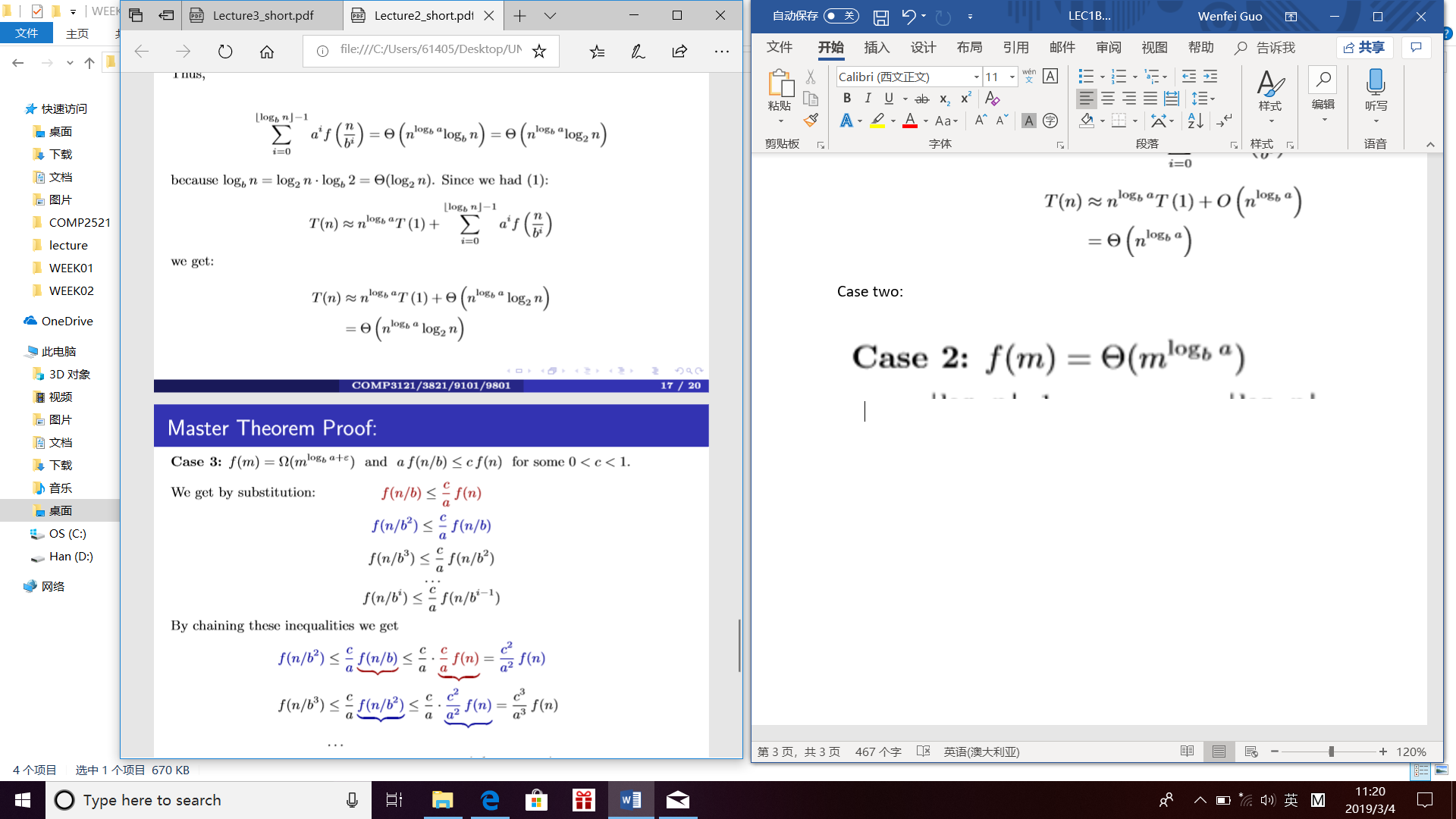
Case one:



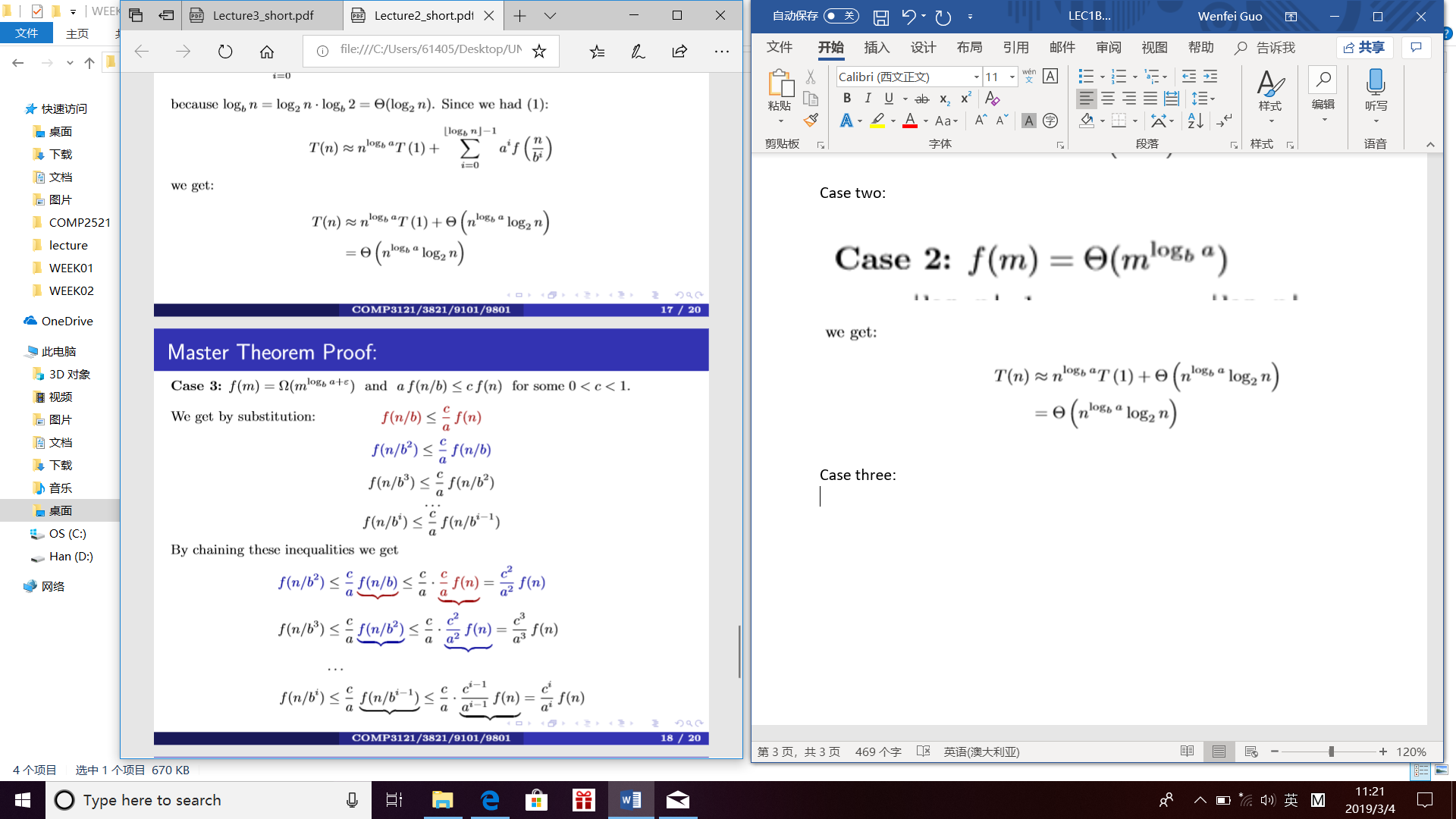


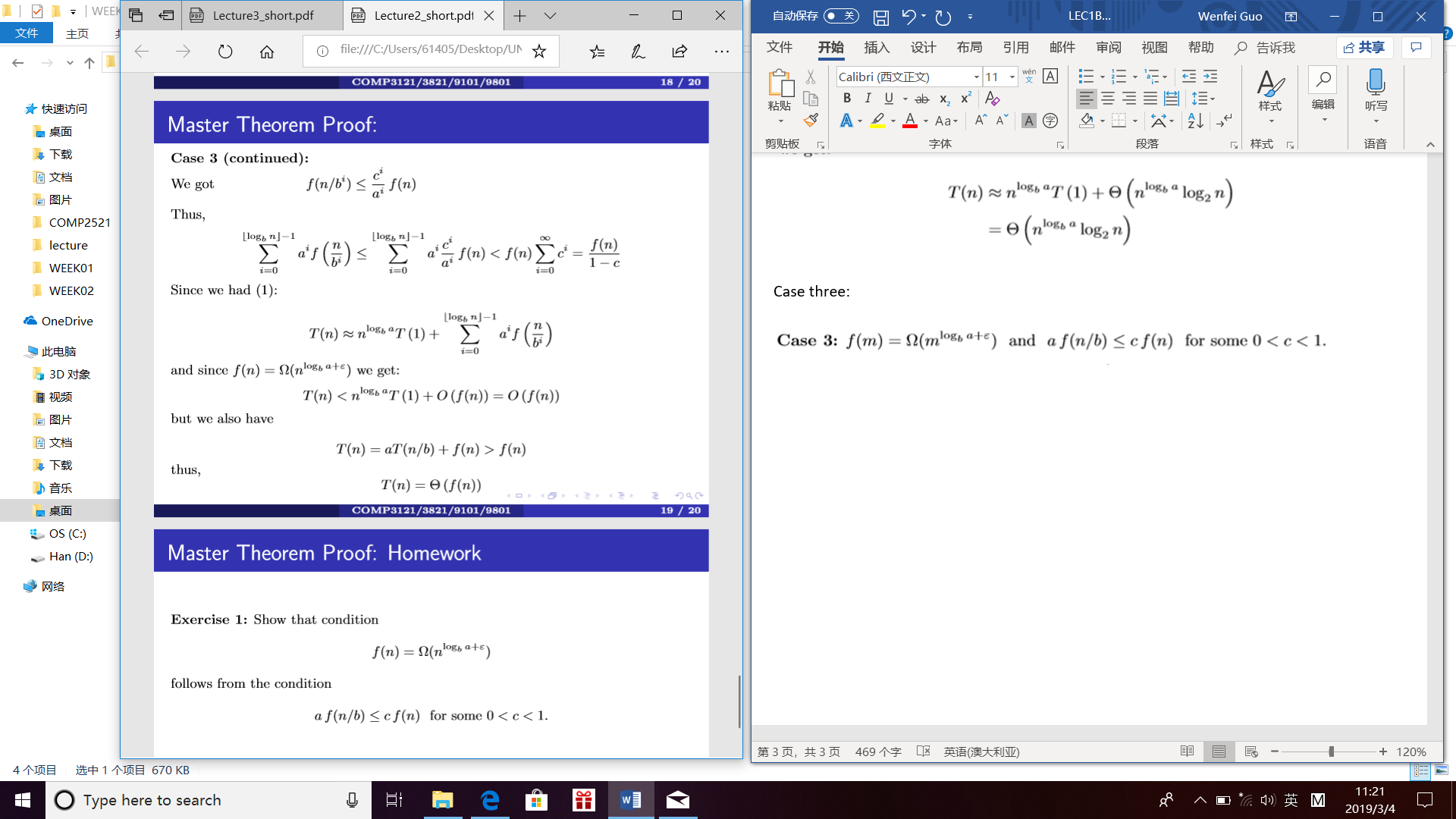
Case two:





Case three:





* 1. Examples

