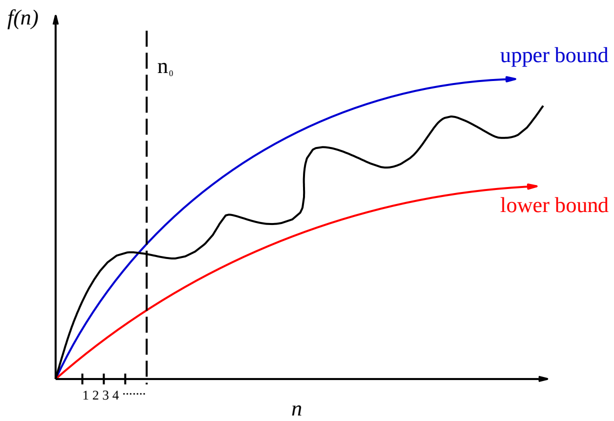
Algorithms

Bounding functions. Big Oh

Lecture 9 by *Marina Barsky*

Still exact analysis can be hard!

Best, worst, and average case are all difficult to deal with because the ***precise*** function details may be complicated:

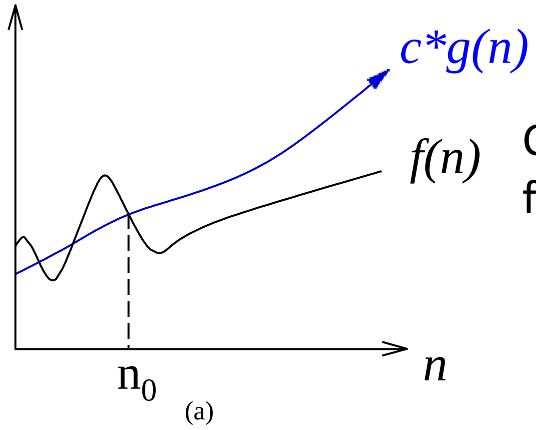


It is easier to talk about *upper* and *lower bounds* of a function

Asymptotic notation (O, Θ, Ω) allows us to describe complexity functions in terms of these bounds

Bounding from above: Big Oh

$f(n) = \boldsymbol{O}(g(n))$ if there are positive constants n_0 and c such that to the right of n_0 the value of $f(n)$ always lies on or below $c \cdot g(n)$

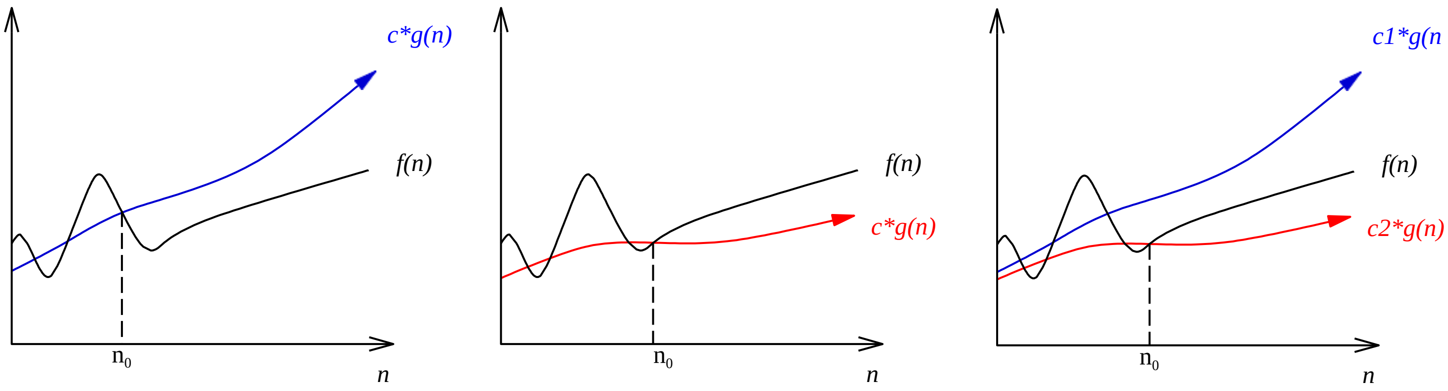


Bounding from above: big Oh

Complexity function

Other bounding functions

Big O | Big Ω omega | Big Θ theta



(a) (b) (c)

- The definitions imply a constant n_0 beyond which they are satisfied
- We do not care about small values of *n*

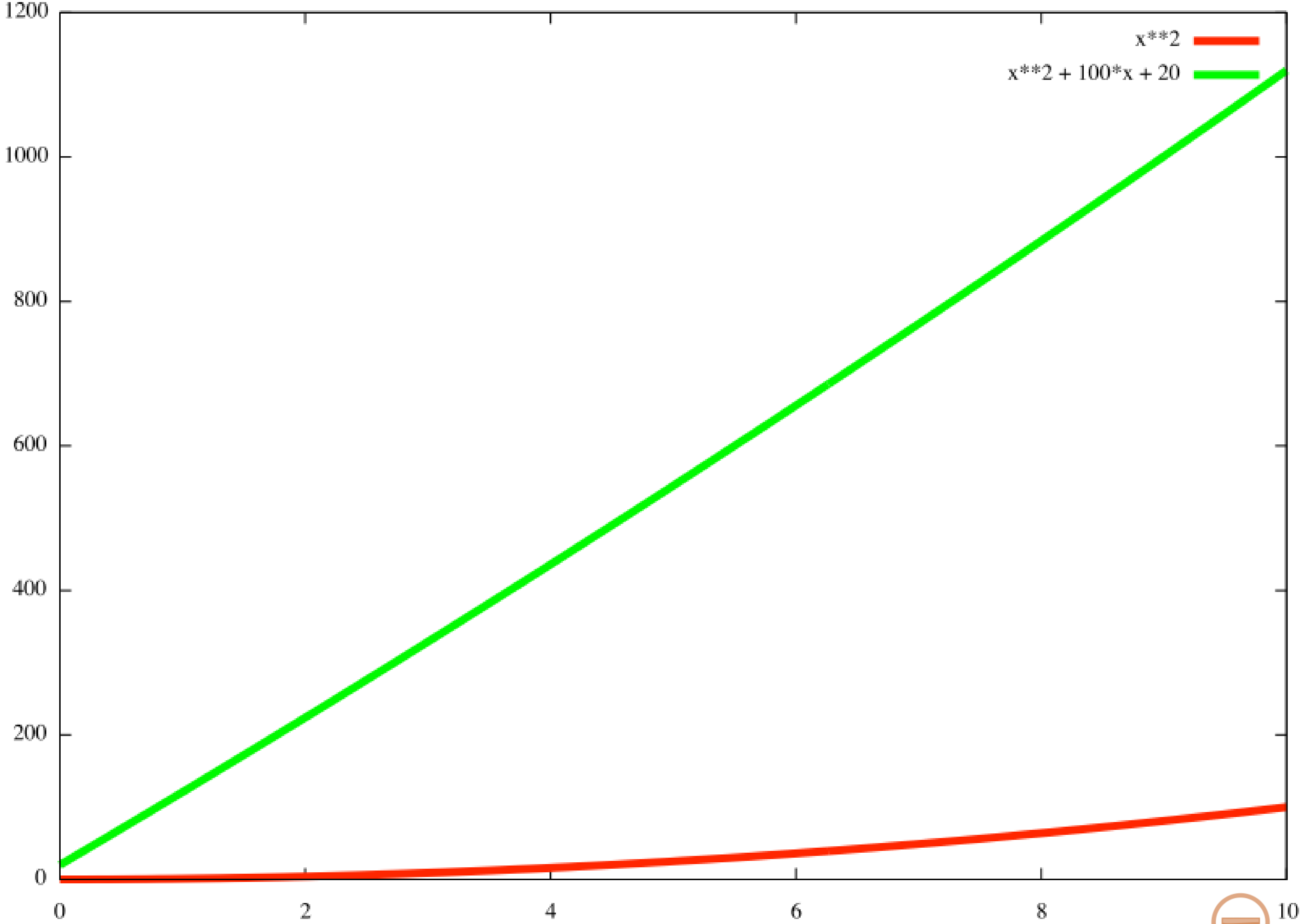
Big Oh guarantees

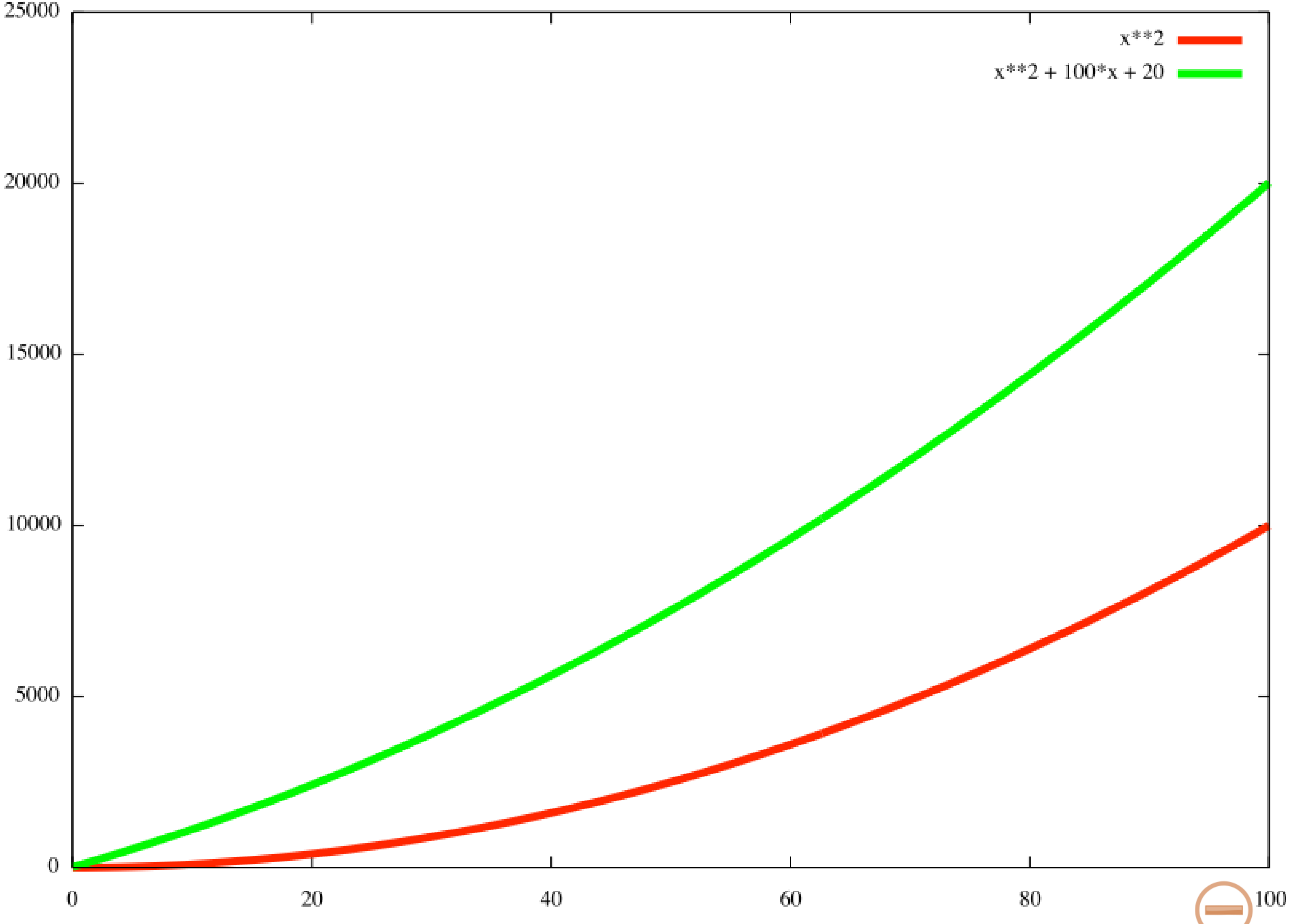
- Big O guarantees that for a given input size *n* the algorithm never exceeds the value of some function on *n*

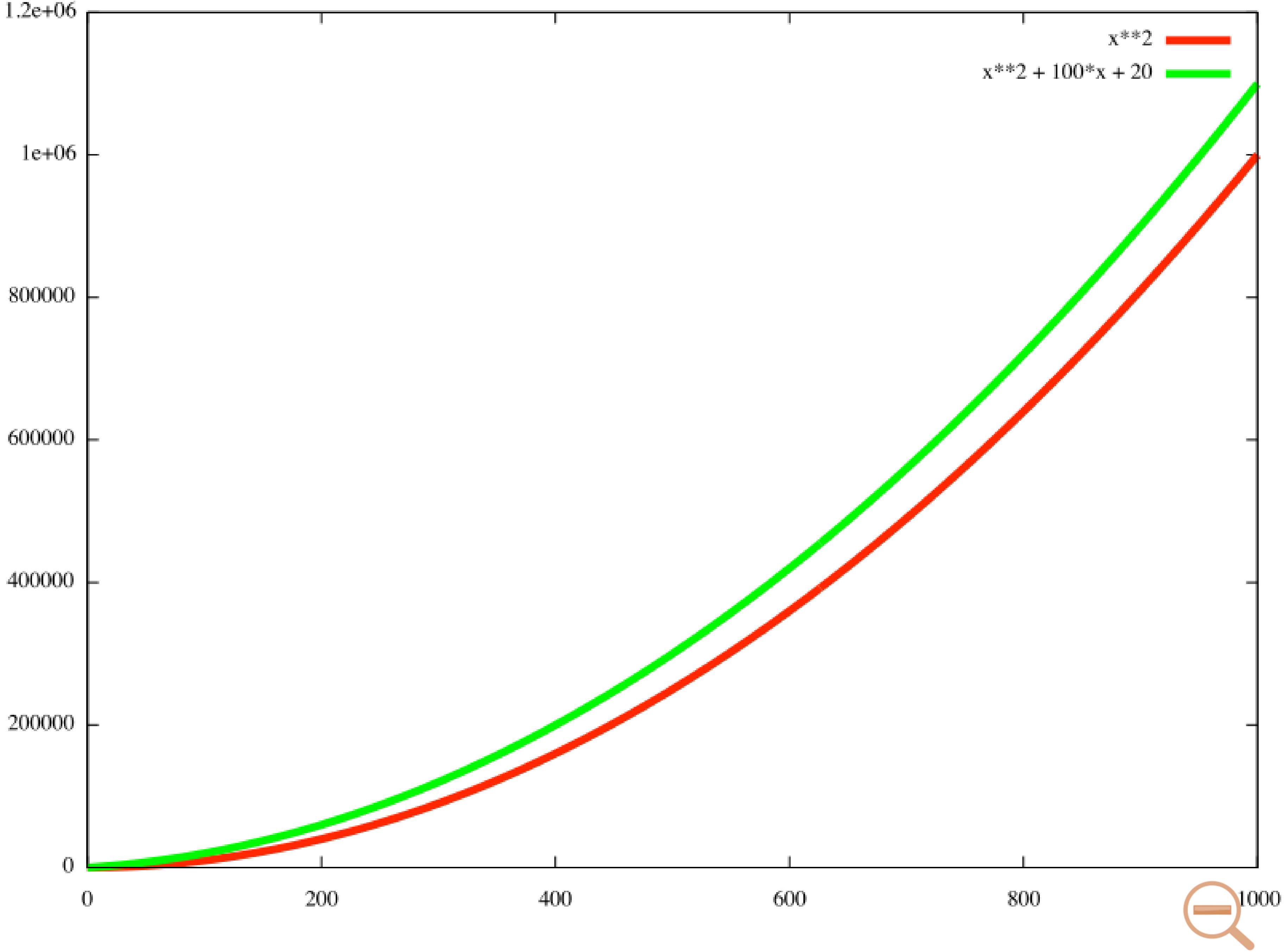
Big Oh ignores low order terms

- For Big-O analysis, we care more about the part that grows fastest as the input grows, because everything else is quickly becomes negligible small as *n* gets very large

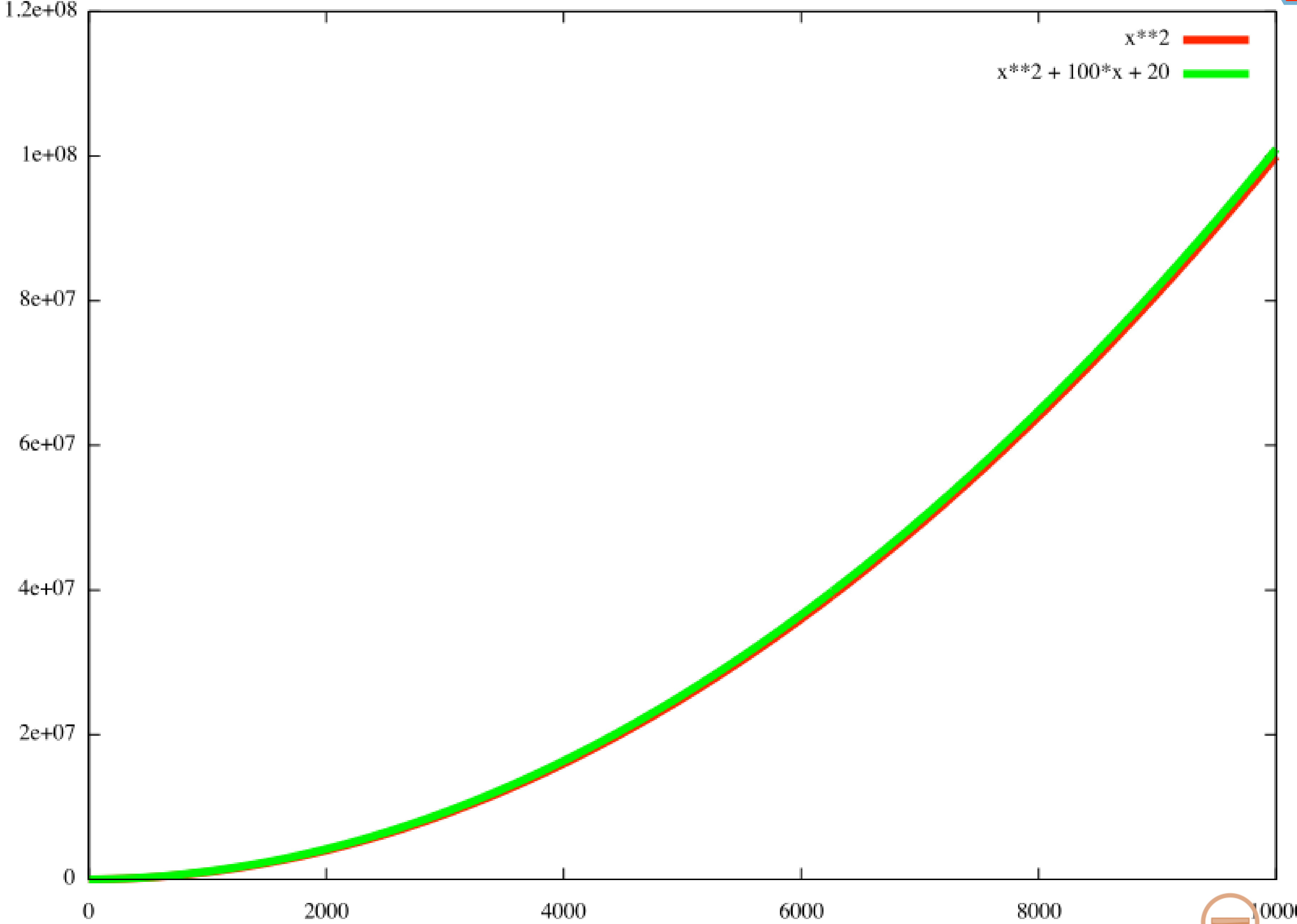
Low order terms are quickly eclipsed by higher-order terms


1200
1000
800
600
400
200
0
0
2
4
6
8
10
x**2
x**2 + 100*x + 20


x**2
x**2 + 100*x + 20
25000
20000
15000
10000
5000
0
0
20
40
60
80
100



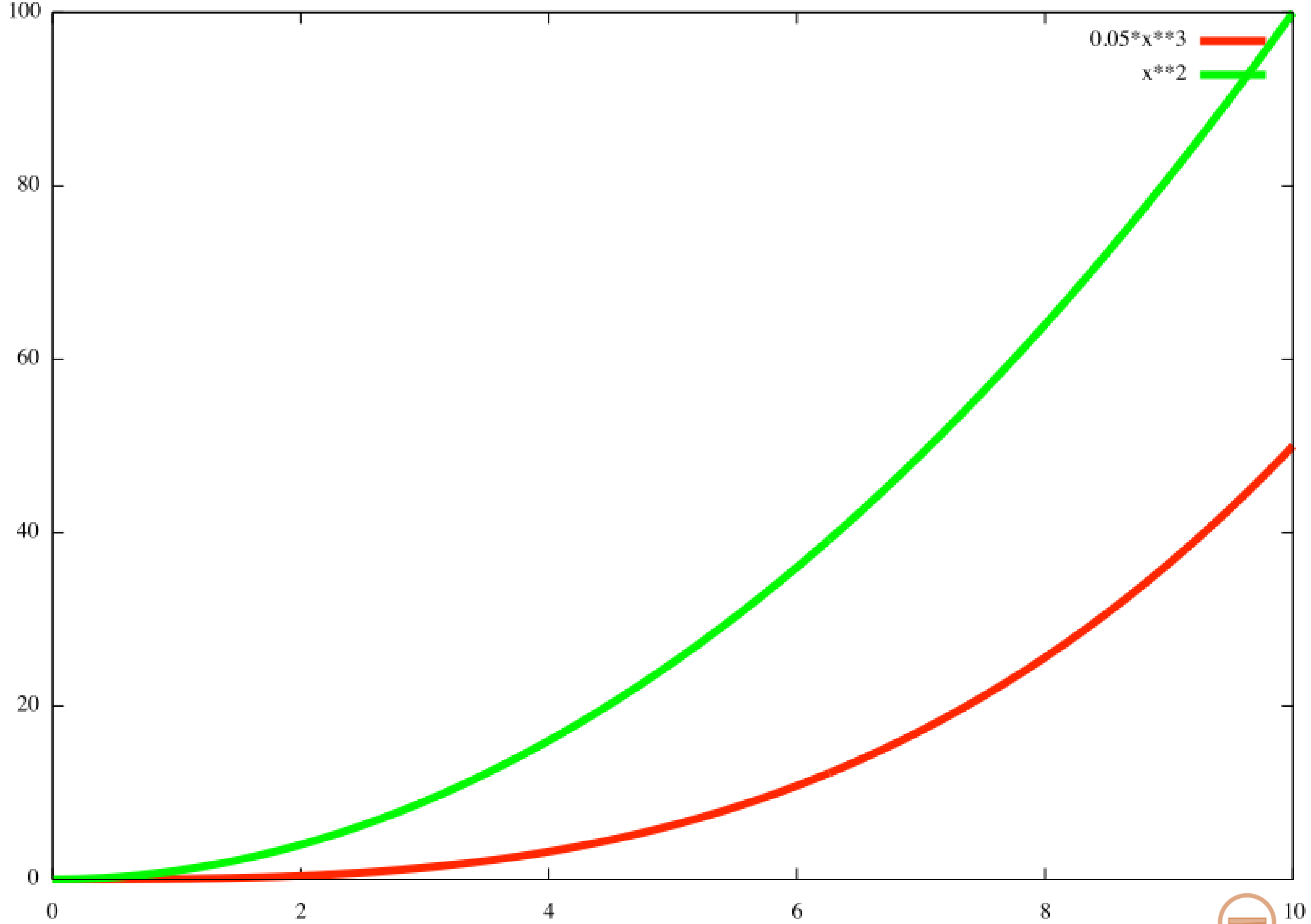
1.2e+06
1e+06
800000
600000
400000
200000
0
0
200
400
600
800
1000
x**2
x**2 + 100*x + 20


x**2
x**2 + 100*x + 20
1.2e+08
1e+08
8e+07
6e+07
4e+07
2e+07
0
0
2000
4000
6000
8000
10000

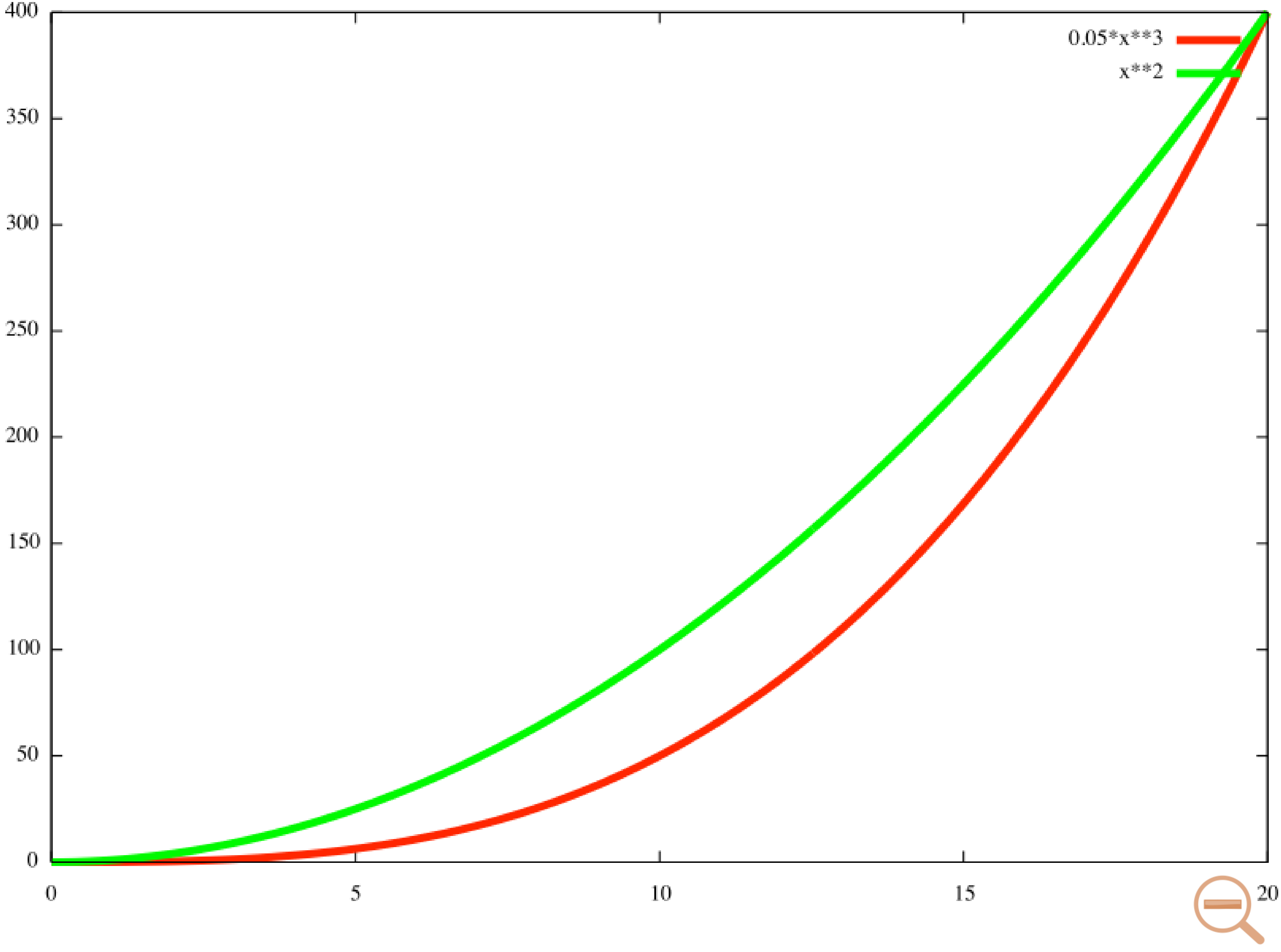
Big Oh ignores constants

- For big values of *n*, the terms that contain variable *n* quickly dominate all the values that stay constant
- Thus to compare $g(n) = 1000n$ and $h(n)= 0.05n^2$ we should ignore constants and compare $O(n)$ with $O(n^2)$. The second algorithm is slower than the first

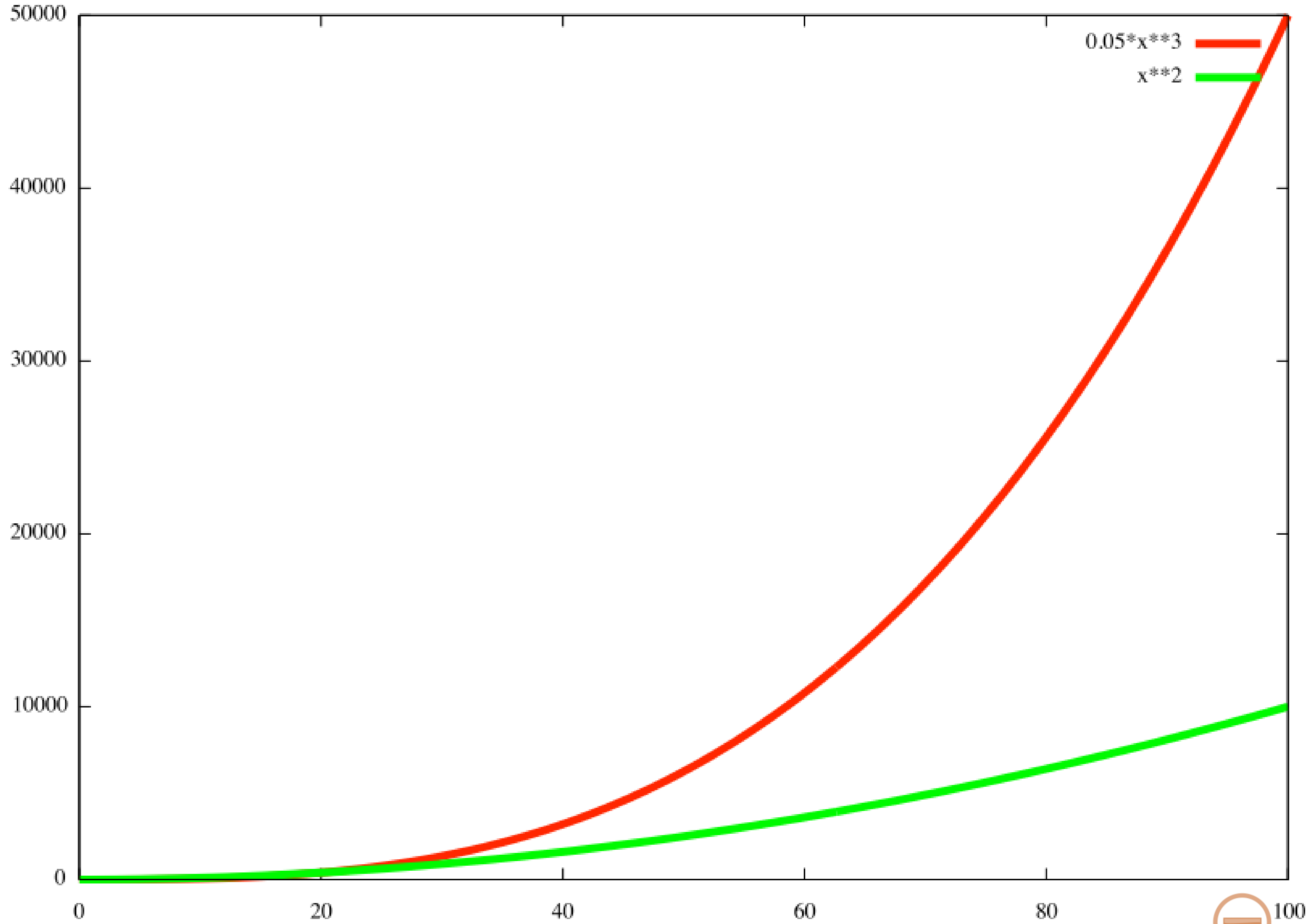
We ignore constants



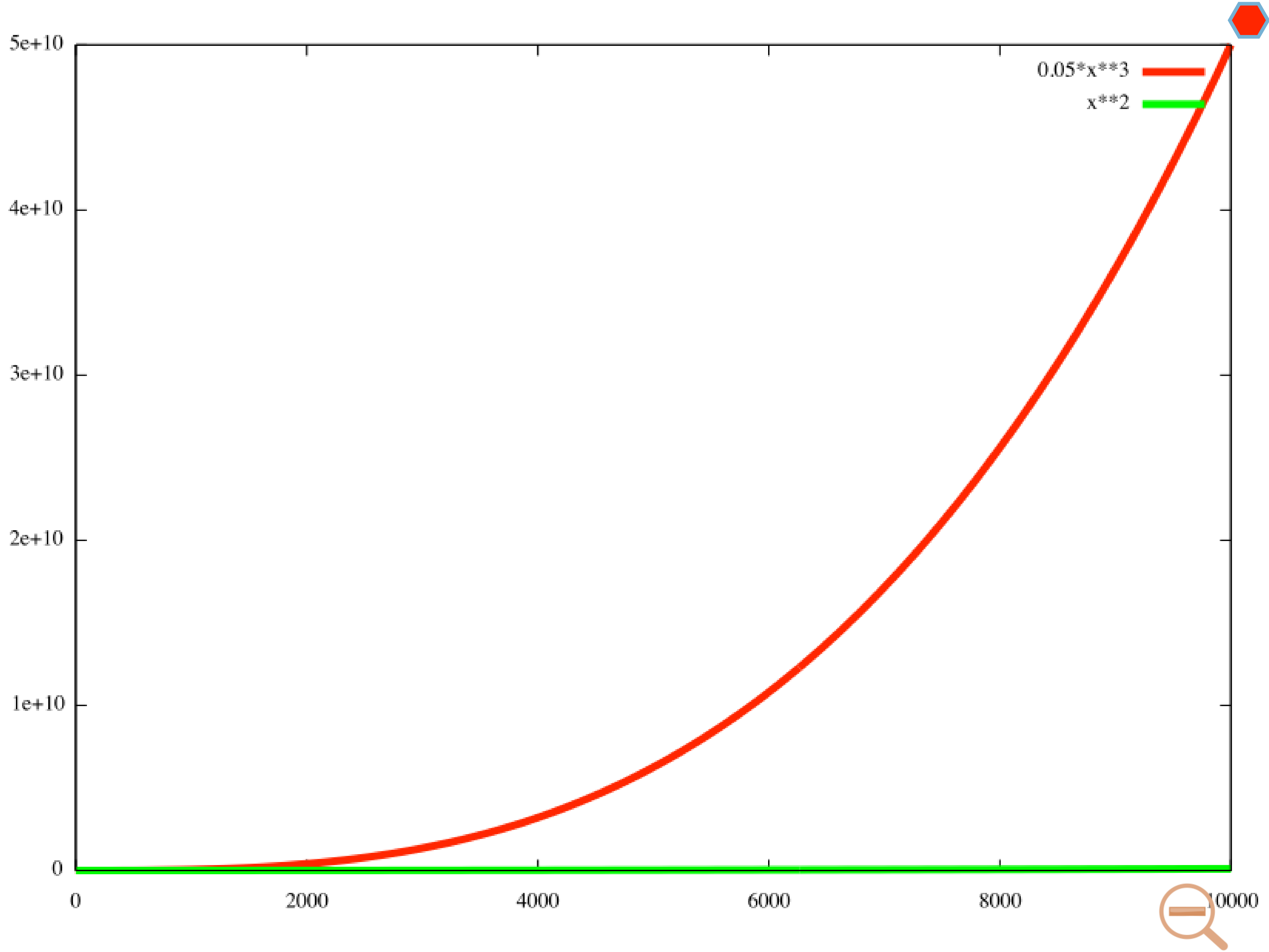
100
80
60
40
20
0
0
2
4
6
8
10
0.05*x**3
x**2



400
350
300
250
200
150
100
50
0
0
5
10
15
20
0.05*x**3
x**2



50000
40000
30000
20000
10000
0
0
20
40
60
80
100
0.05*x**3
x**2



5e+10
4e+10
3e+10
2e+10
1e+10
0
0
2000
4000
6000
8000
10000
0.05*x**3
x**2

Big Oh represents the rate of growth

- We use Big O Notation to talk about how quickly the runtime grows with the increase in the input size
- For example, let's say the algorithm runs in total $2n(n\text{-}1)$ steps
  - For one thing, for REALLY large values of n, such as n=1,000,000 $2n(n\text{-}1)$ is pretty much the same thing as $2n^2$.
  - For another, what happens if we increase the size of the list by a factor of k, from n to kn?
  - The number of basic operations will increase by a factor of $2(kn)^2/2n^2 = k^2$

Big Oh represents the rate of growth

- We use Big O Notation to talk about how quickly the runtime grows with the increase in the input size
- Big O bounds the speed of growth:

  so we can say things like the runtime grows “on the order of the size of the input” ($O(n)$) or “on the order of the square of the size of the input” ($O(n^2)$)

Reasoning about time complexity

- When you *intuitively* understand an algorithm, the reasoning about the run-time of an algorithm can be done in your head
- But it is usually much easier to estimate complexity given a precise-enough pseudocode (or a code)

- To get big-Oh:
  - Count all the elementary operations
  - Ignore (remove) lower order (i.e. slower growing) terms
  - Remove constant factors

- For example: $5n^3+3n^2+177$ is still $O(n^3)$

What is O() of $f(n) = n(n+1)/2$

A. $O(n^2)$
B. $O(2n^2)$
C. $O((n^2 + n)/2)$
D. $O(n^3)$
E. I don't know

Let’s do some Big-Oh analysis!

A. Algorithm that sums numbers from 1 to n

```
int A(int n) {
   int sum = 0;
   for (int i=1; i <= n; i++)
      sum += i;
   return sum;
}
```

A. O(log n)

B. O(n)

C. O(n^2)

D. O(n+1)

A. Algorithm that sums numbers from 1 to n

```
int A(int n) {
   int sum = 0;
   for (int i=1; i <= n; i++)
      sum += i;
   return sum;
}
```

A. O(log n)

B. O(n)

C. O(n^2)

D. O(n+1)

Answer B: One loop with n iterations. O(n).
Answer D: O(n+1) is also correct, but we usually remove constants

B. Nested loop?

```
int B(int n) {
   int sum = 0;
   for (int i=1; i <= 2*n; i++)
      for (int j=0; j < 5; j++)
         sum += j;
   return sum;
}
```

A. O(log n)
B. O(n)
C. O(n^2)
D. O(5*2*n)

B. Nested loop?

```
int B(int n) {
   int sum = 0;
   for (int i=1; i <= 2*n; i++)
      for (int j=0; j < 5; j++)
         sum += j;
   return sum;
}
```

A. O(log n)
B. O(n)
C. O(n^2)
D. O(5*2*n)

Analysis: The inner for-loop (on j) always adds 5 numbers together, and the outer loop (on i) does this 2*n times. So this is O(5*2*n) = O(n).
Answers B and D are both correct, but answer B is better.

C. Nested loop?

```
int C(int n) {
   int sum = 0;
   for (int i=1; i <= n; i++)
      for (int j=0; j <= n; j++)
         sum += j;
   return sum;
}
```

A. $O(n)$
B. $O(n^2)$
C. $O(n^n)$
D. The answer depends on the value of n

C. Nested loop?

```
int C(int n) {
   int sum = 0;
   for (int i=1; i <= n; i++)
      for (int j=0; j <= n; j++)
         sum += j;
   return sum;
}
```

A. O(n)
B. $O(n^2)$
C. $O(n^n)$
D. The answer depends on the value of n

Analysis: The inner loop (on j) has n steps
It runs n times: for each value of i from 1 to n. Altogether this is n+n+n+ …+ n steps.
So this is B: $O(n^2)$.

D. Loops

```
int D(int n) {
   int sum = 0;
   for (int i=1; i <= n; i++)
      sum += i*i;
   for (int j=0; j < n; j++)
      sum-= j;
   for (int k = 0; k < 2*n; k++)
      sum = sum*k;
   return sum;
}
```

A. $O(n)$
B. $O(n^2)$
C. $O(n^3)$
D. $O(n^n)$

D. Loops

```
int D(int n) {
   int sum = 0;
   for (int i=1; i <= n; i++)
      sum += i*i;
   for (int j=0; j < n; j++)
      sum-= j;
   for (int k = 0; k < 2*n; k++)
      sum = sum*k;
   return sum;
}
```

A. $O(n)$
B. $O(n^2)$
C. $O(n^3)$
D. $O(n^n)$

Analysis: Note that the loops are sequential, not nested. The loop on i does n additions. After it finished, the loop on j does n subtractions. Then the loop on k does 2n multiplications. Altogether there are 4n steps. This is A: O(n)

E. While…

```
int E(int n) {
   int count = 0;

   for (int i=n/2; i<n; i++){
      int j = 0;

      while  (j + n/2 <= n){
         int k = 1;

         while (k <= n){
           count = count + 1;
           k = k*2;
         }

       j = j + 1;
   }
}
```

A. $O(n^3)$

B. $O(n \log^2 n)$

C. $O(n^2 \sqrt{n})$

D. $O(n^2 \log n)$

E. While…

```
int E(int n) {
   int count = 0;

   for (int i=n/2; i<n; i++){
      int j = 0;

      while  (j + n/2 <= n){
         int k = 1;

         while (k <= n){
           count = count + 1;
           k = k*2;
         }

       j = j + 1;
   }
}
```

A. $O(n^3)$

B. $O(n \log^2 n)$

C. $O(n^2 \sqrt{n})$

D. $O(n^2 \log n)$

Analysis: 3 nested loops. We start with the innermost loop – loop on k. It runs log n times
The next loop is on j. It runs n/2 times
The outer loop also runs n/2 times.
The loops are nested so we multiply: n/2*n/2*log n = $O(n^2 \log n)$

F. Break

```
int F(int n) {
   if (n == 1) return  1;

   for (int i = 0; i<n; i++){
      for(int j= 0; j<n; j++) {
          System.out.println("*");
          break;
      }
   }
}
```

A. $O(n^2)$
B. $O(n)$
C. $O(1)$
D. $O(2n)$

F. Break

```
int F(int n) {
   if (n == 1) return  1;

   for (int i = 0; i<n; i++){
      for(int j= 0; j<n; j++) {
          System.out.println("*");
          break;
      }
   }
}
```

A. $O(n^2)$

B. $O(n)$

C. $O(1)$

D. $O(2n)$

Correct answer is B