Algorithm Efficiency (More formally)

Tiziana Ligorio
tligorio@hunter.cuny.edu

Today's Plan

Announcements

Midterm solution

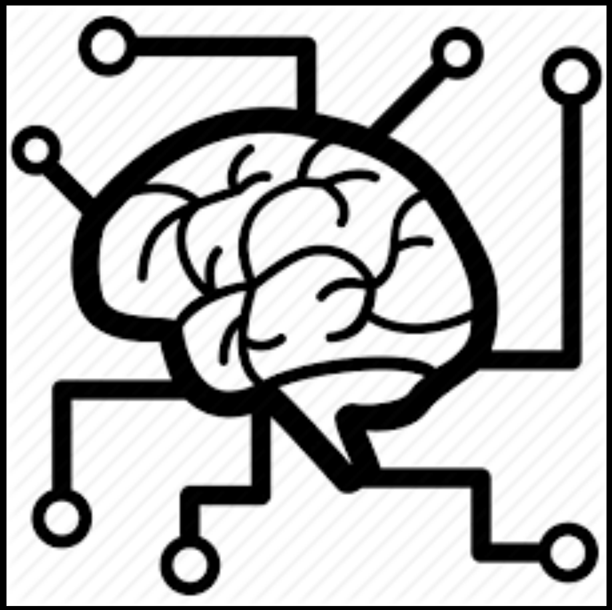
Project 4 How are we doing?

Algorithm Efficiency

Announcements and Syllabus Check

Revised tentative schedule

Midterm Solution



Project 4:
How are we doing?

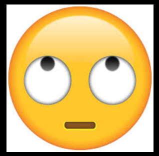
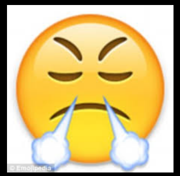
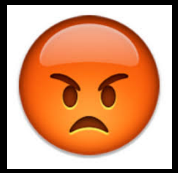
Algorithm Efficiency

Scenario 1

You are using an app and suddenly it stalls…
whatever it is doing it's taking way too long…

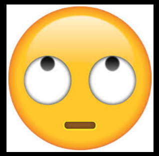
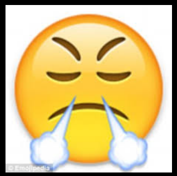
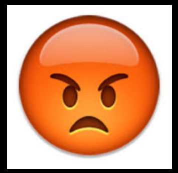
Scenario 1

You are using an app and suddenly it stalls…
whatever it is doing it's taking way too long…

how "*long*" does that have to be for you to become ridiculously frustrated?    

Scenario 1

You are using an app and suddenly it stalls…
whatever it is doing it's taking way too long…

how "*long*" does that have to be for you to become ridiculously frustrated?    

… probably not that long

Scenario 2

At your next super high-end job with the company/research-center of your dreams you are given a very difficult problem to solve

You work hard on it, find a solution, code it up and it works!!!!

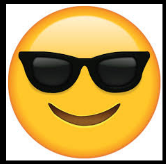
Proudly you present it the next day 

but…

Scenario 2

At your next super high-end job with the company/research-center of your dreams you are given a very difficult problem to solve

You work hard on it, find a solution, code it up and it works!!!!

Proudly you present it the next day 

but…

Given some new input it keeps stalling…

Well… sorry but your solution is no good!!!  

You need to have a means to estimate/predict the efficiency of your algorithms on unknown input.

What is a good solution?

How can we compare solutions to a problem? (Algorithms)

What is a good solution?

Exact

What is a good solution?

Exact

Efficient

Time

Space

What is a good solution?

Exact

Efficient

Time

Space

We are going to focus on time

How can we measure time efficiency?

How can we measure time efficiency?

Runtime?

Problems with runtime for comparison

What computer are you using?

- Runtime is highly sensitive to hardware

Problems with runtime for comparison

What computer are you using?

Runtime is highly sensitive to hardware

What implementation are you using?

Implementation details may affect runtime but are not reflective of algorithm efficiency

Problems with runtime for comparison

What computer are you using?

Runtime is highly sensitive to hardware

What implementation are you using?

Implementation details may affect runtime but are not reflective of algorithm efficiency

What data are you using?

Algorithm A may run faster on one dataset while algorithm B runs faster on another

How should we measure execution time?

How should we measure execution time?

Number of “steps” or “operations”

```cpp
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>*  ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

What are the operations?
Let **n** be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>*  ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

What are the operations?
Let **n** be the number of nodes

1 node instantiation and assignment upon entering the loop

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>*  ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

What are the operations?
Let **n** be the number of nodes

1 node instantiation and assignment upon entering the loop

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>*  ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

pointer comparison

What are the operations?
Let **n** be the number of nodes

1 node instantiation and assignment upon entering the loop

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>*  ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

call to `getNext()`

pointer comparison

What are the operations?
Let **n** be the number of nodes

1 node instantiation and assignment upon entering the loop

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>*  ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

call to `getNext()`

pointer assignment

pointer comparison

What are the operations?
Let **n** be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>*  ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

1 node instantiation and assignment upon entering the loop

call to `getNext()`

pointer assignment

pointer comparison

call to `getItem()`

write to the console

What are the operations?
Let **n** be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>*  ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

K_0 1 node instantiation and assignment upon entering the loop

K_1 call to `getNext()`

K_2 pointer assignment

K_3 pointer comparison

K_4 call to `getItem()`

K_5 write to the console

What are the operations?
Let **n** be the number of nodes

```
template<class ItemType>
void List<ItemType>::traverse()
{
    for(Node<ItemType>*  ptr = first; ptr != nullptr; ptr = ptr->getNext())
    {
        std::cout << ptr->getItem() << std::endl;
    }
}
```

K_0 1 node instantiation and assignment upon entering the loop

K_1 call to `getNext()`

K_2 pointer assignment

K_3 pointer comparison

K_4 call to `getItem()`

K_5 write to the console

$$\text{Operations} = K_0 + n(K_1+K_2+K_3+K_4+K_5)$$

In-Class Task

Identify the steps and write down an expression for execution time

```
bool linearSearch(const std::string& str, char ch)
{

    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

In-Class Task

Identify the steps and write down an expression for execution time

```
bool linearSearch(const std::string& str, char ch)
{

    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

Was this tricky?

n here is the length of the string

```
bool linearSearch(const std::string& str, char ch)
{
     // 1 int assignment upon entering loop
    for (int i = 0; i < str.length(); i++)
    { // call to length() and increment
        if (str[i] == ch) { // Comparisons
            return true; //return operation, maybe
        }
    }
    return false; //return operation, maybe
}
```

n here is the length of the string

```
bool linearSearch(const std::string& str, char ch)
{
     // 1 int assignment upon entering loop
    for (int i = 0; i < str.length(); i++)
    { // call to length() and increment
        if (str[i] == ch) { // Comparisons
            return true; //return operation, maybe
        }
    }
    return false; //return operation, maybe
}
```

Maybe stop in the middle

Maybe stop at end of loop

n here is the length of the string

```
bool linearSearch(const std::string& str, char ch)
{
    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

In the
WORST CASE
it will be **n**

Execution completes in **at most:**

$k_0 n + k_1$ operations

n here is the length of the string

```
bool linearSearch(const std::string& str, char ch)
{
    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

Execution completes in **at most:**

$k_0 \mathbf{n} + k_1$ operations

Some constant number of operations repeated inside the loop

Some constant number of operations performed outside the loop

n here is the length of the string

```
bool linearSearch(const std::string& str, char ch)
{
    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

Execution completes in **at most:**

$k_0 n + k_1$ operations

The number of times the loop is repeated, i.e. the size of `str`

Some constant number of operations repeated inside the loop

Some constant number of operations performed outside the loop

Observation

Don't need to explicitly compute the constants $\mathtt{k}_{\mathtt{i}}$

$4\mathbf{n} + 1000$

$\mathbf{n} + 137$

Dominant term is sufficient to explain overall behavior (in this case linear)

Big-O Notation

Ignores everything except **dominant term**

Examples:

$T(n) = 4n + 4 = O(n)$

$T(n) = 164n + 35 = O(n)$

$T(n) = n^2 + 35n + 5 = O(n^2)$

$T(n) = 2n^3 + 98n^2 + 210 = O(n^3)$

$T(n) = 2^n + 5 = O(2^n)$

Notation: describes the overall **WORST** behavior

Big-O Notation

T(n) is the running time

N is the size of the input

Ignores everything except **dominant term**

Examples:

Notation: describes the overall **WORST** behavior

$T(n) = 4n + 4 = O(n)$

$T(n) = 164n + 35 = O(n)$

$T(n) = n^2 + 35n + 5 = O(n^2)$

$T(n) = 2n^3 + 98n^2 + 210 = O(n^3)$

$T(n) = 2^n + 5 = O(2^n)$

Big-O Notation

Ignores everything except **dominant term**

Examples:

$T(n) = 4n + 4 = O(n)$

$T(n) = 164n + 35 = O(n)$

$T(n) = n^2 + 35n + 5 = O(n^2)$

$T(n) = 2n^3 + 98n^2 + 210 = O(n^3)$

$T(n) = 2^n + 5 = O(2^n)$

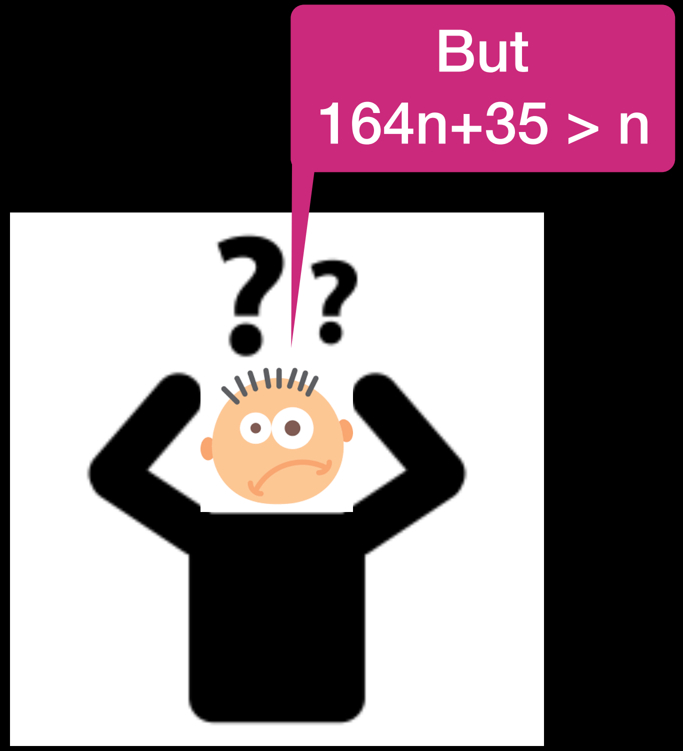
Big-O describes the overall WORST behavior

Let $T(n)$ be the ***running time*** of an algorithm measured as number of operations given **input of size n**.

$T(n)$ is $O(f(n))$

if it grows **no faster** than $f(n)$

Big-O Notation



Ignores everything except **dominant term**

Big-O describes the overall **WORST** behavior

Examples:

$T(n) = 4n + 4 = O(n)$

$T(n) = 164n + 35 = O(n)$

$T(n) = n^2 + 35n + 5 = O(n^2)$

$T(n) = 2n^3 + 98n^2 + 210 = O(n^3)$

$T(n) = 2^n + 5 = O(2^n)$

Let $T(n)$ be the ***running time*** of an algorithm measured as number of operations given **input of size n**.

$T(n)$ is $O(f(n))$

if it grows **no faster** than $f(n)$

Big-O Notation

Ignores everything except **dominant term**

Examples:

$T(n) = 4n + 4 = O(n)$

$T(n) = 164n + 35 = O(n)$

$T(n) = n^2 + 35n + 5 = O(n^2)$

$T(n) = 2n^3 + 98n^2 + 210 = O(n^3)$

$T(n) = 2^n + 5 = O(2^n)$

Notation: describes the overall WORST behavior

More formally:

T(n)* is *O(f(n))

if there exist constants **k** and $\mathbf{n_0}$

such that for all $\mathbf{n \geq n_0}$

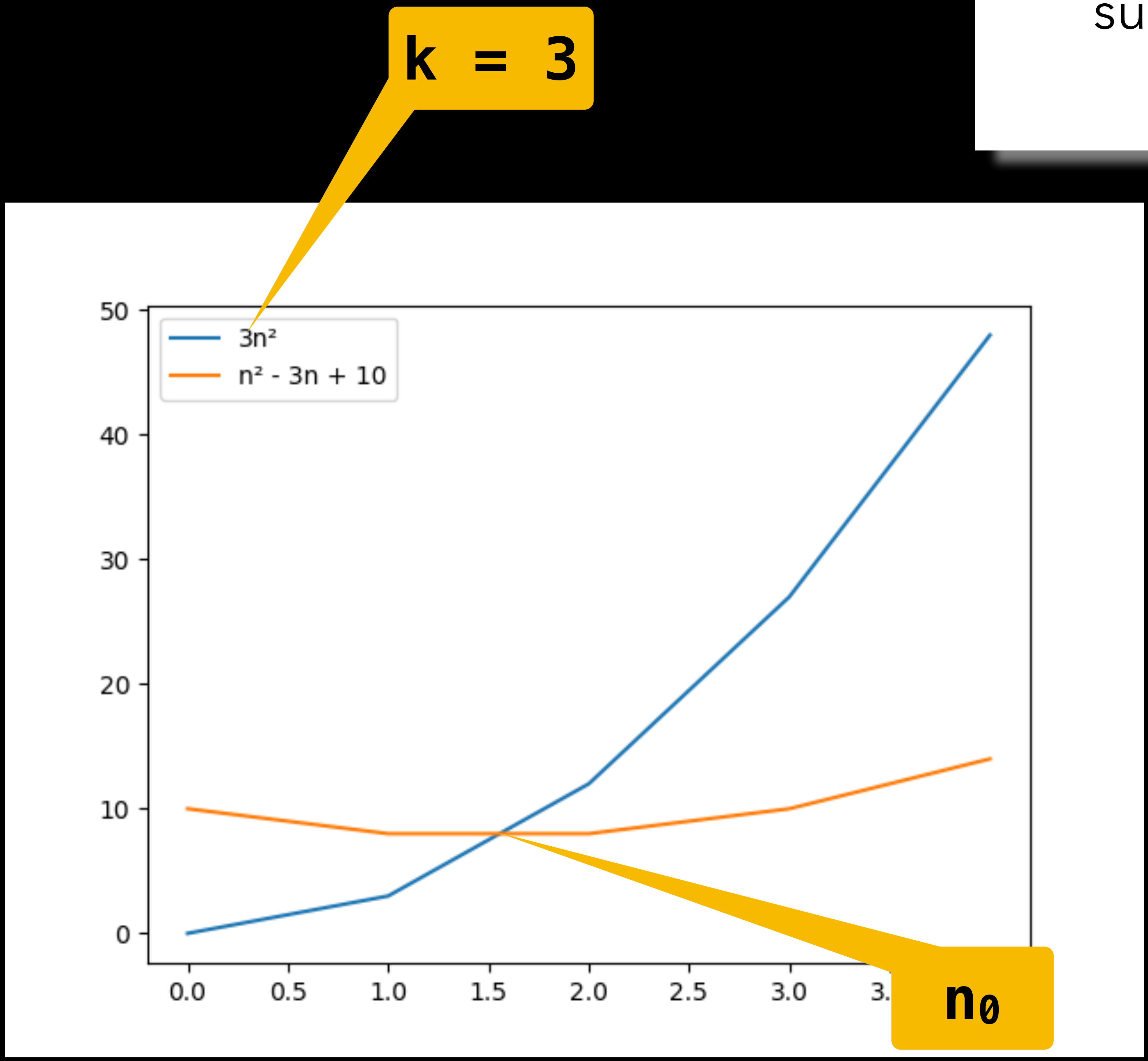
$T(n) \leq kf(n)$

More formally:

$T(n)$ is $O(f(n))$

if there exist constants k and n_0 such that for all $n \geq n_0$,

$T(n) \leq kf(n)$



$T(n) = n^2 - 3n + 10$

$T(n)$ is $O(n^2)$

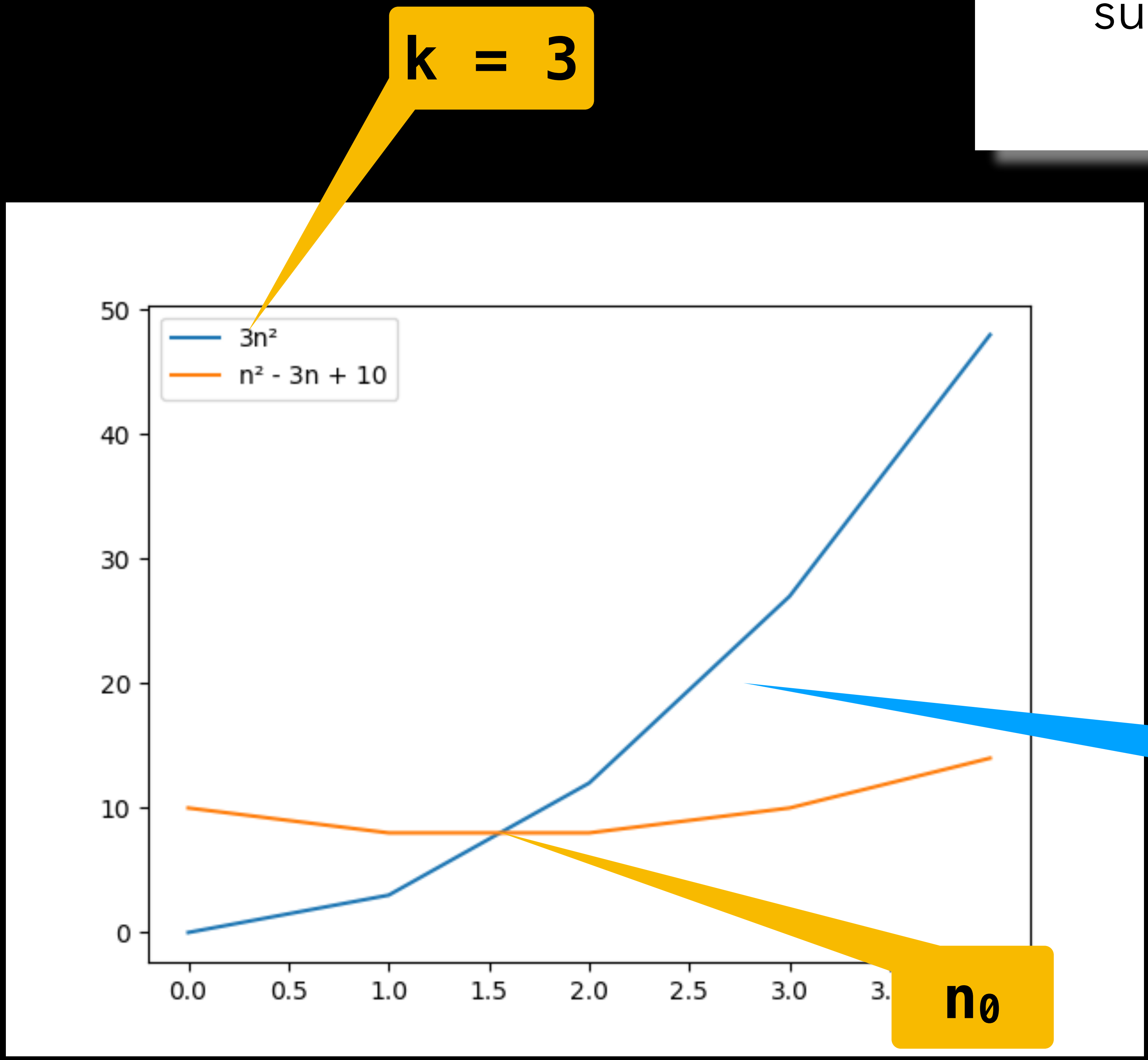
For $k=3$ and $n \geq 1.5$

More formally:

$T(n)$ is $O(f(n))$

if there exist constants k and n_0 such that for all $n \geq n_0$,

$T(n) \leq kf(n)$



$T(n) = n^2 - 3n + 10$

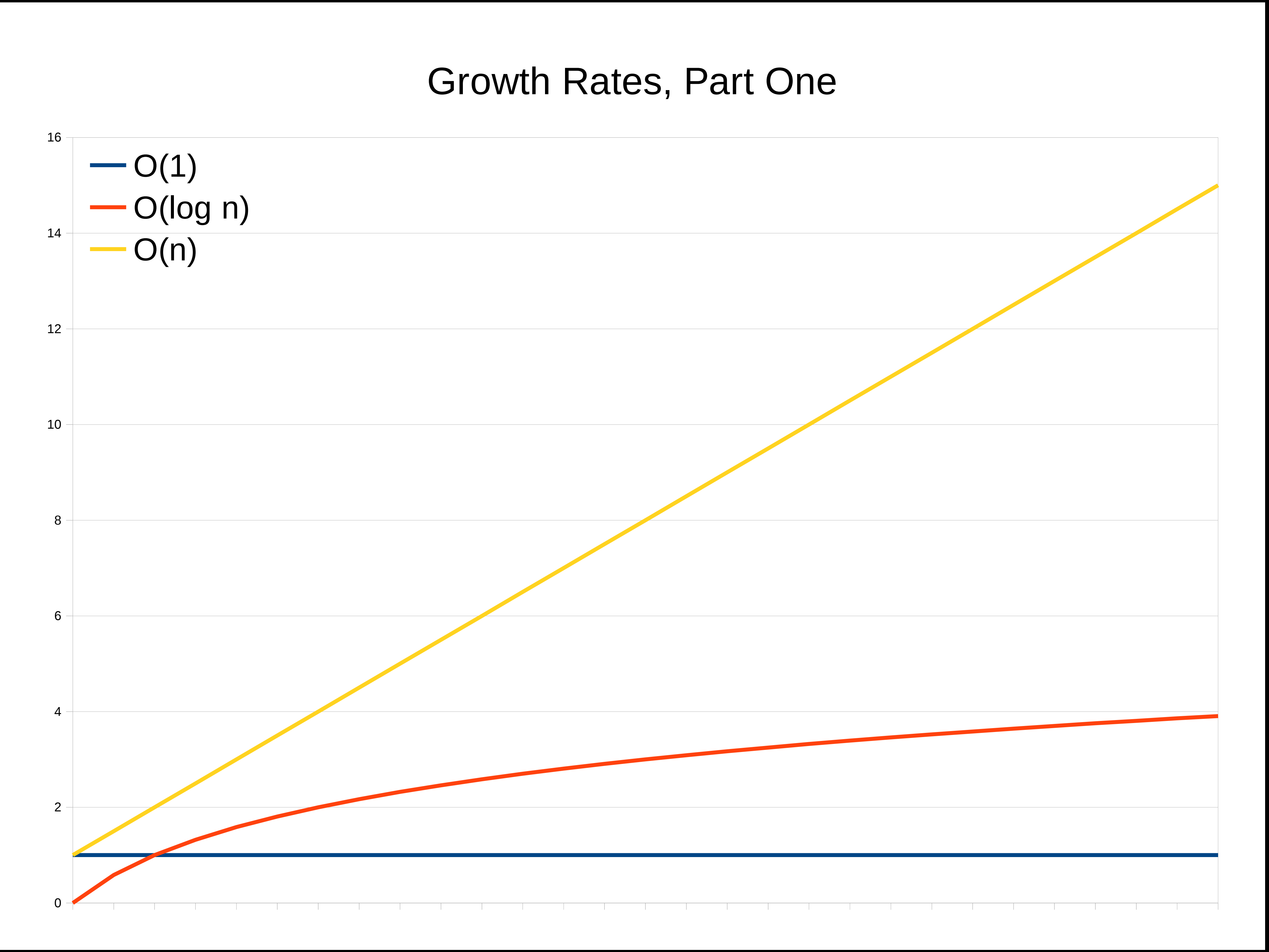
$T(n)$ is $O(n^2)$

For $k=3$ and $n \geq 1.5$

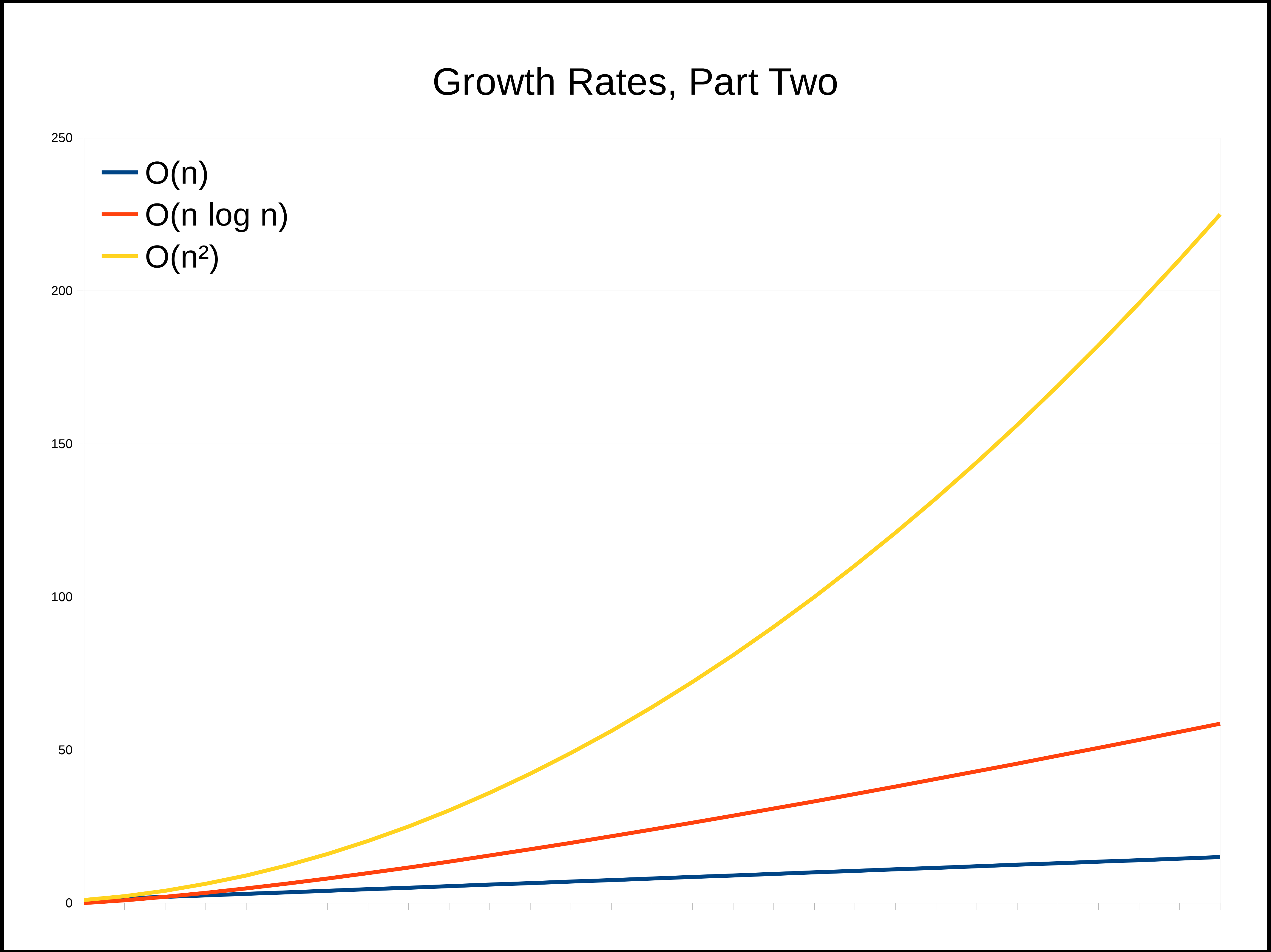
This is why we can look at dominant term only to explain behavior

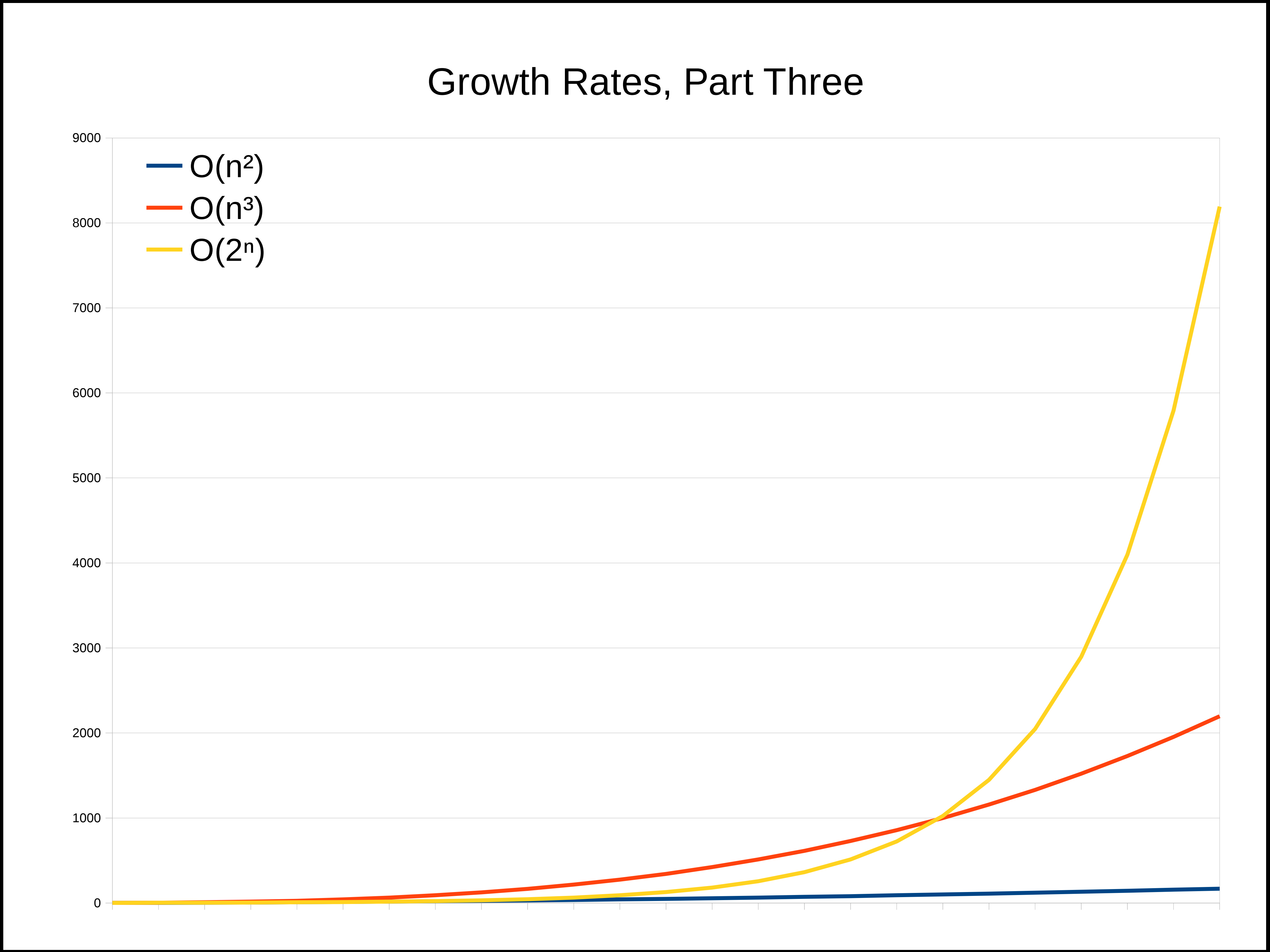
Big-O describes the worst-case overall growth rate of an algorithms for large n

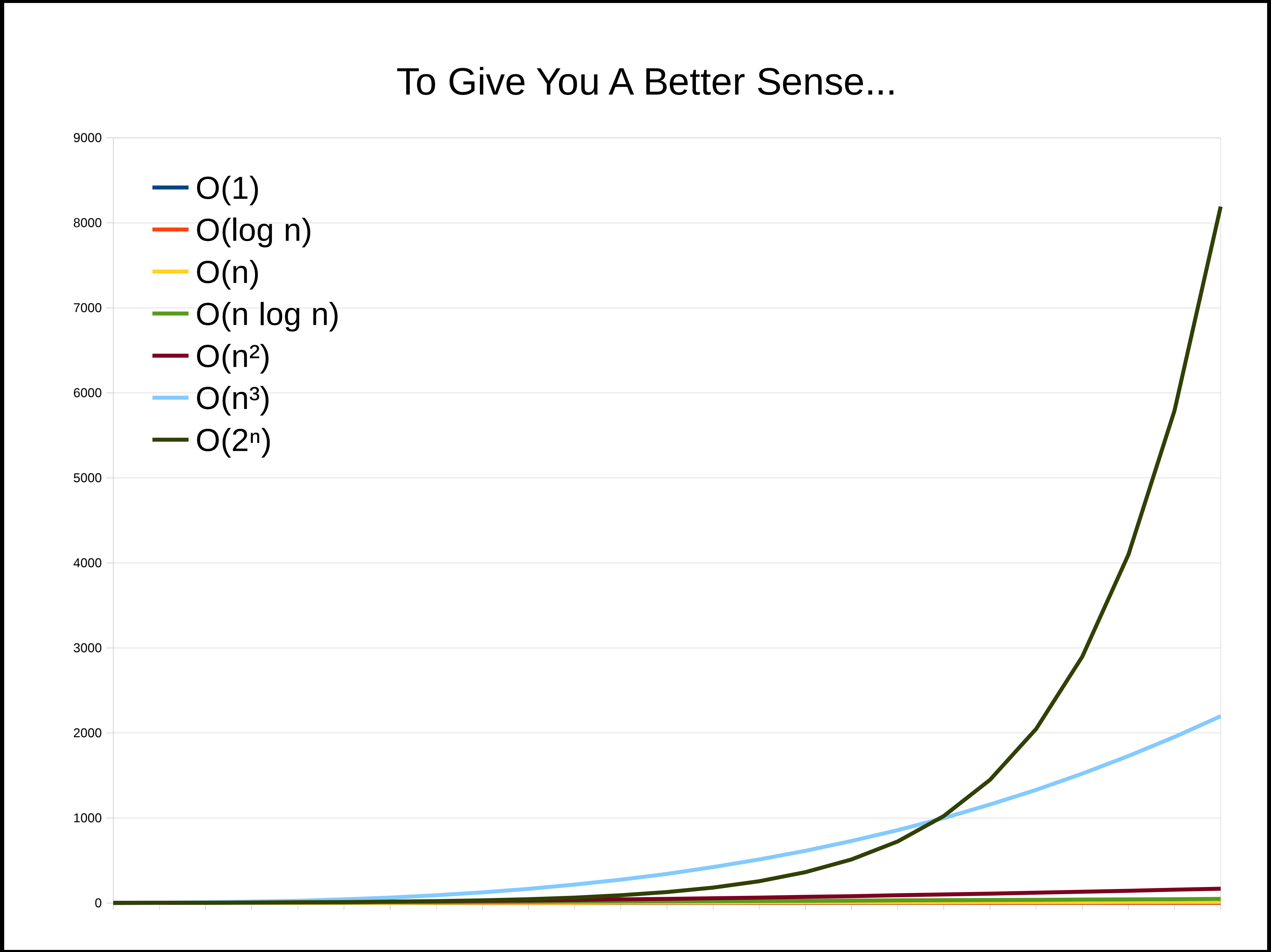
A visual comparison of growth rates



Growth Rates, Part One
16
14
12
10
8
6
4
2
0
O(1)
O(log n)
O(n)


Growth Rates, Part Two
250
200
150
100
50
0
O(n)
O(n log n)
O(n²)


Growth Rates, Part Three
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
O(n²)
O(n³)
O(2ⁿ)


To Give You A Better Sense...
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
O(1)
O(log n)
O(n)
O(n log n)
O(n²)
O(n³)
O(2ⁿ)

A numerical comparison of growth rates

n Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

What does Big-O describe?

“Long term” behavior of a function

Compare behavior of 2 algorithms

If algorithm A has runtime O(n) and algorithm B has runtime O(n^2), for large inputs A will always be faster.

If algorithm A has runtime O(n), for large inputs doubling the size of the input will double the runtime

Analyze algorithm behavior with growing input

What can't Big-O describe?

The actual runtime of an algorithm

$10^{100}n = O(n)$

$10^{-100}n = O(n)$

How an algorithm behaves on small input

$n^3 = O(n^3)$

$10^6 = O(1)$

Types of Analysis

Big-O: worst-case analysis

- T(n) is O(f(n))
- Grows no faster than f(n)
- It can't get worst than this, good for "sleeping well at night"

Types of Analysis

Big-O: worst-case analysis

- T(n) is O(f(n))
- Grows no faster than f(n)
- It can't get worst than this, good for "sleeping well at night"

Omega: best-case analysis

- T(n) is Ω(f(n))
- Grows at least as fast as f(n)
- Useful to know if algorithm performs well in some cases

Types of Analysis

Big-O: worst-case analysis

- T(n) is O(f(n))
- Grows no faster than f(n)
- It can't get worst than this, good for "sleeping well at night"

Omega: best-case analysis

- T(n) is Ω(f(n))
- Grows at least as fast as f(n)
- Useful to know if algorithm performs well in some cases

Theta: average-case analysis

- T(n) is Θ(f(n))
- Grows at the same rate as f(n) : is both O(f(n)) and Ω(f(n))
- The most exact but far more difficult, must determine relative probabilities of problem sizes or distribution of data values

To summarize Big-O

It is a means of describing the growth rate of a function

It ignores all but the dominant term

It ignores constants

Allows for quantitative ranking of algorithms

Allows for quantitative reasoning about algorithms

More examples next time