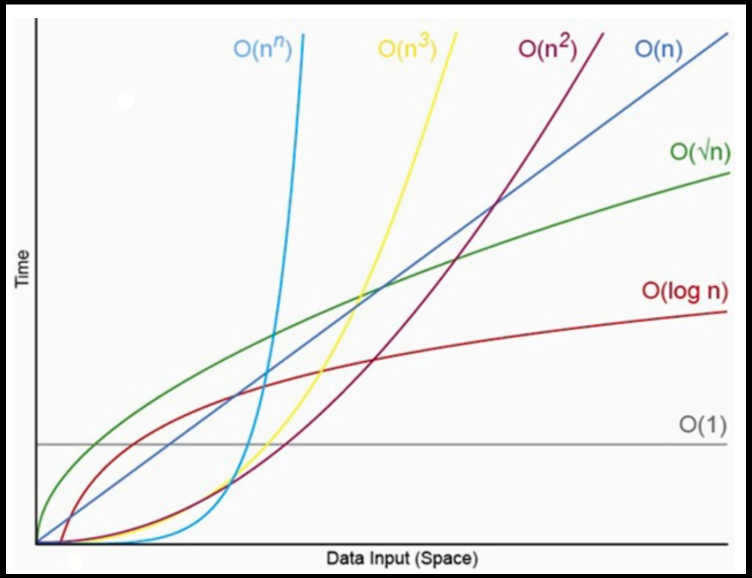
Algorithm Efficiency



Tiziana Ligorio
Hunter College of The City University of New York

Recap

We implemented a Bag ADT

Using an Array data structure

Next using a Linked data structure

But first…

Today's Plan

Algorithm Efficiency

Algorithm Efficiency

Scenario 1

You are using an application but it won’t complete some operation…
whatever it is doing it’s taking way too long…

Scenario 1

You are using an application but it won’t complete some operation…
whatever it is doing it’s taking way too long…

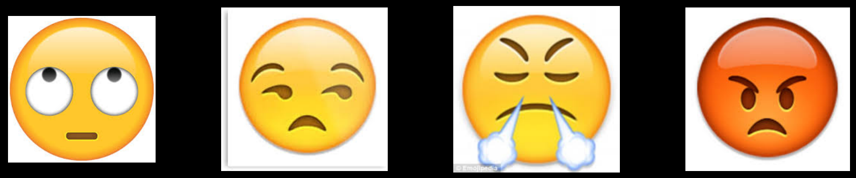
how “*long*” does that have to be for you to become ridiculously frustrated?

Scenario 1

You are using an application but it won’t complete some operation…
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 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 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 (large) input it's taking an awfully long time to complete execution…

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)

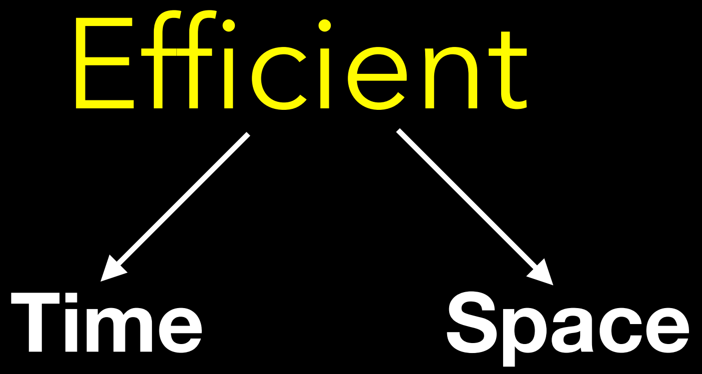
What is a good solution?

Correct

If it's not correct it is not a solution at all

What is a good solution?

Correct


Efficient

Time

Space

What is a good solution?

Correct

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 actual runtime for comparison

What computer are you using?

- Runtime is highly sensitive to hardware

Problems with actual 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

How should we measure execution time?

How should we measure execution time?

Number of “steps” or “operations” as a function of the size of the input

How should we measure execution time?

Constant

Number of "steps" or "operations" as a function of the size of the input

Variable

```
template<class T>
int ArrayBag<T>::getFrequencyOf(const T& an_entry) const
{
   int frequency{0};
   int current_index{0};       // array index currently being inspected
   while (current_index < item_count_)
   {
      if (items_[current_index] == an_entry)
      {
         frequency++;
      }  // end if
      current_index ++;       // increment to next entry
   }  // end while
   return frequency;
} // end getFrequencyOf
```

What are the operations?
Let **n** be the number of items in the array

```
template<class T>
int ArrayBag<T>::getFrequencyOf(const T& an_entry) const
{
   int frequency{0};
   int current_index{0};       // array index currently being inspected
   while (current_index < item_count_)
   {
      if (items_[current_index] == an_entry)
      {
         frequency++;
      }  // end if
      current_index ++;       // increment to next entry
   }  // end while
   return frequency;
} // end getFrequencyOf
```

What are the operations?
Let **n** be the number of items in the array

initialization

comparison

```
template<class T>
int ArrayBag<T>::getFrequencyOf(const T& an_entry) const
{
   int frequency{0};
   int current_index{0};        // array index currently being inspected
   while (current_index < item_count_)
   {
      if (items_[current_index] == an_entry)
      {
         frequency++;
      }  // end if
      current_index ++;        // increment to next entry
   }  // end while
   return frequency;
} // end getFrequencyOf
```

increment

return

What are the operations?
Let **n** be the number of items in the array

initialization

comparison

```
template<class T>
int ArrayBag<T>::getFrequencyOf(const T& an_entry) const
{
   int frequency{0}; C0
   int current_index{0}; C1    // array index currently being inspected
   while (current_index < item_count_) C2
   {
      if (items_[current_index] == an_entry) C3
      {
         frequency++; C4
      }  // end if
      current_index ++; C5    // increment to next entry
   }  // end while
   return frequency; C6
} // end getFrequencyOf
```

increment

return

C_i is some constant number

What are the operations?
Let **n** be the number of items in the array

initialization

comparison

```
template<class T>
int ArrayBag<T>::getFrequencyOf(const T& an_entry) const
{
   int frequency{0}; C0
   int current_index{0}; C1    // array index currently being inspected
   while (current_index < item_count_) C2
   {
      if (items_[current_index] == an_entry) C3
      {
         frequency++; C4
      }  // end if
      current_index ++; C5    // increment to next entry
   }  // end while
   return frequency; C6
} // end getFrequencyOf
```

n

increment

return

C_i is some constant number
n is the number of items

What are the operations?
Let **n** be the number of items in the array

initialization

comparison

```
template<class T>
int ArrayBag<T>::getFrequencyOf(const T& an_entry) const
{
   int frequency{0}; C0
   int current_index{0}; C1    // array index currently being inspected
   while (current_index < item_count_) C2
n  {
      if (items_[current_index] == an_entry) C3
      {
         frequency++; C4
      }  // end if
      current_index ++; C5    // increment to next entry
   }  // end while
   return frequency; C6
} // end getFrequencyOf
```

increment

return

$$C_0+ C_1 + n\,(C_2 + C_3 + C_4 + C_5) + C_6 = C_7 + nC_8$$ operations

What are the operations?
Let **n** be the number of items in the array

initialization

comparison

```
template<class T>
int ArrayBag<T>::getFrequencyOf(const T& an_entry) const
{
   int frequency{0}; C0
   int current_index{0}; C1    // array index currently being inspected
   while (current_index < item_count_) C2
n  {
      if (items_[current_index] == an_entry) C3
      {
         frequency++; C4
      }  // end if
      current_index ++; C5    // increment to next entry
   }  // end while
   return frequency; C6
} // end getFrequencyOf
```

increment

return

$C_0 + C_1 + n\,(C_2 + C_3 + C_4 + C_5) + C_6 = \boxed{C_7 + nC_8}$ operations

How should we measure execution time?

Constant

Number of “steps” or “operations” as a function of the size of the input

Variable

Lecture Activity

Identify the steps and write down an expression for execution time

```
template<class T>
int ArrayBag<T>::getIndexOf(const T& target) const
{
    bool found = false;
    int result = -1;
    int search_index = 0;

    // If the bag is empty, item_count_ is zero, so loop is skipped
    while (!found && (search_index < item_count_))
    {
        if (items_[search_index] == target)
        {
          found = true;
          result = search_index;
         }
         else
         {
           search_index ++;
        }   // end if
    } // end while
    return result;
}  // end getIndexOf
```

Lecture Activity

Identify the steps and write down an expression for execution time

```
template<class T>
int ArrayBag<T>::getIndexOf(const T& target) const
{
    bool found = false;
    int result = -1;
    int search_index = 0;

    // If the bag is empty, item_count_ is zero, so loop is skipped
    while (!found && (search_index < item_count_))
    {
        if (items_[search_index] == target)
        {
          found = true;
          result = search_index;
         }
         else
         {
           search_index ++;
        }  // end if
    } // end while
    return result;
}  // end getIndexOf
```

Was this tricky?

Identify the steps and write down an expression for execution time

```
template<class T>
int ArrayBag<T>::getIndexOf(const T& target) const
{
    bool found = false;
    int result = -1;
    int search_index = 0;

    // If the bag is empty, item_count_ is zero, so loop is skipped
    while (!found && (search_index < item_count_))
    {
        if (items_[search_index] == target)
        {
          found = true;
          result = search_index;
         }
         else
         {
           search_index ++;
        }  // end if
    } // end while
    return result;
}  // end getIndexOf
```

n here is the size of the ArrayBag

Identify the steps and write down an expression for execution time

```
template<class T>
int ArrayBag<T>::getIndexOf(const T& target) const
{
    bool found = false;
    int result = -1;
    int search_index = 0;

    // If the bag is empty, item_count_ is zero, so loop is skipped
    while (!found && (search_index < item_count_))
    {
        if (items_[search_index] == target)
        {
          found = true;
          result = search_index;
         }
         else
         {
           search_index ++;
        }  // end if
    } // end while
    return result;
}  // end getIndexOf
```

Maybe stop in the middle

Maybe stop at end of loop

n here is the size of the ArrayBag

Identify the steps and write down an expression for execution time

```
template<class T>
int ArrayBag<T>::getIndexOf(const T& target) const
{
    bool found = false;
    int result = -1;
    int search_index = 0;

    // If the bag is empty, item_count_ is zero, so loop is skipped
    while (!found && (search_index < item_count_))
    {
        if (items_[search_index] == target)
        {
          found = true;
          result = search_index;
         }
         else
         {
           search_index ++;
        }  // end if
    } // end while
    return result;
}   // end getIndexOf
```

In the
WORST CASE

Execution completes in **at most:**

$c_0 n + c_1$ operations

Types of Analysis

Best case analysis: running time under best input (e.g., in linear search item we are looking for is the first) - not reflective of overall performance)

Average case analysis: assumes equal probability of input (usually **not** the case)

Expected case analysis: assumes probability of occurrence of input is known or can be estimated, and if it were possible may be too expensive

Worst case analysis: running time under worst input, gives upper bound, it can't get worse, good for sleeping well at night!

Identify the steps and write down an expression for execution time

```
template<class T>
int ArrayBag<T>::getIndexOf(const T& target) const
{
    bool found = false;
    int result = -1;
    int search_index = 0;

    // If the bag is empty, item_count_ is zero, so loop is skipped
    while (!found && (search_index < item_count_))
    {
        if (items[search_index] == target)
        {
          found = true;
          result = search_index;
         }
         else
         {
           search_index ++;
        }  // end if
    } // end while
    return result;
}  // end getIndexOf
```

Execution completes in **at most:**

$c_0 n + c_1$ operations

Some constant number of operations repeated inside the loop

Some constant number of operations performed outside the loop

Identify the steps and write down an expression for execution time

```
template<class T>
int ArrayBag<T>::getIndexOf(const T& target) const
{
    bool found = false;
    int result = -1;
    int search_index = 0;

    // If the bag is empty, item_count_ is zero, so loop is ski
    while (!found && (search_index < item_count_))
    {
        if (items[search_index] == target)
        {
          found = true;
          result = search_index;
         }
         else
         {
           search_index ++;
        }  // end if
    } // end while
    return result;
}  // end getIndexOf
```

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

Execution completes in **at most:**

$c_0 n + c_1$ operations

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 `c`$_i$

4**n** + 1000

n + 137

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

Big-O Notation

Ignores everything except **dominant term**

Examples:

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

Notation: describes the overall behavior

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

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

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

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

Big-O Notation

T(n) is the running time

n is the size of the input

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 behavior

Big-O Notation

Ignores everything except **dominant term**

Examples:

$\boxed{T(n) = 4n + 4} = \boxed{O(n)}$

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

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

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

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

Big-O describes the overall behavior

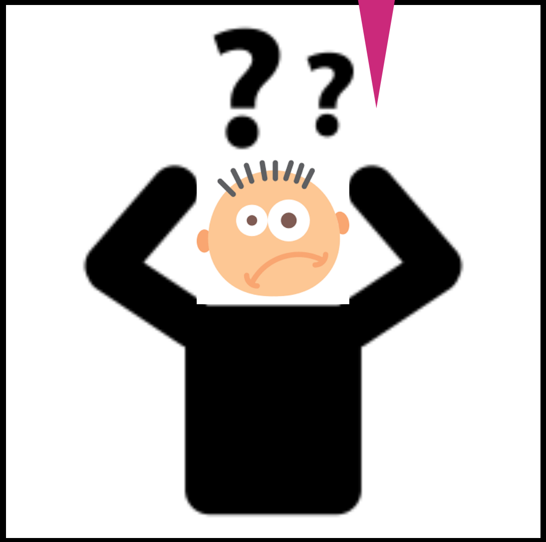
Let $T(n)$ be the ***running time*** of an algorithm measured as number of operations given **input of size n**.

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

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

Big-O Notation

But
164n+35 > n !!??!!



Ignores everything except **dominant term**

Examples:

$$\boxed{T(n) = 4n + 4} = \boxed{O(n)}$$

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

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

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

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

Big-O describes the overall 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 behavior

Examples:

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

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

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

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

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

More formally:

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

if there exist constants **k** and **n_0**

such that for all **$n \geq n_0$**

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

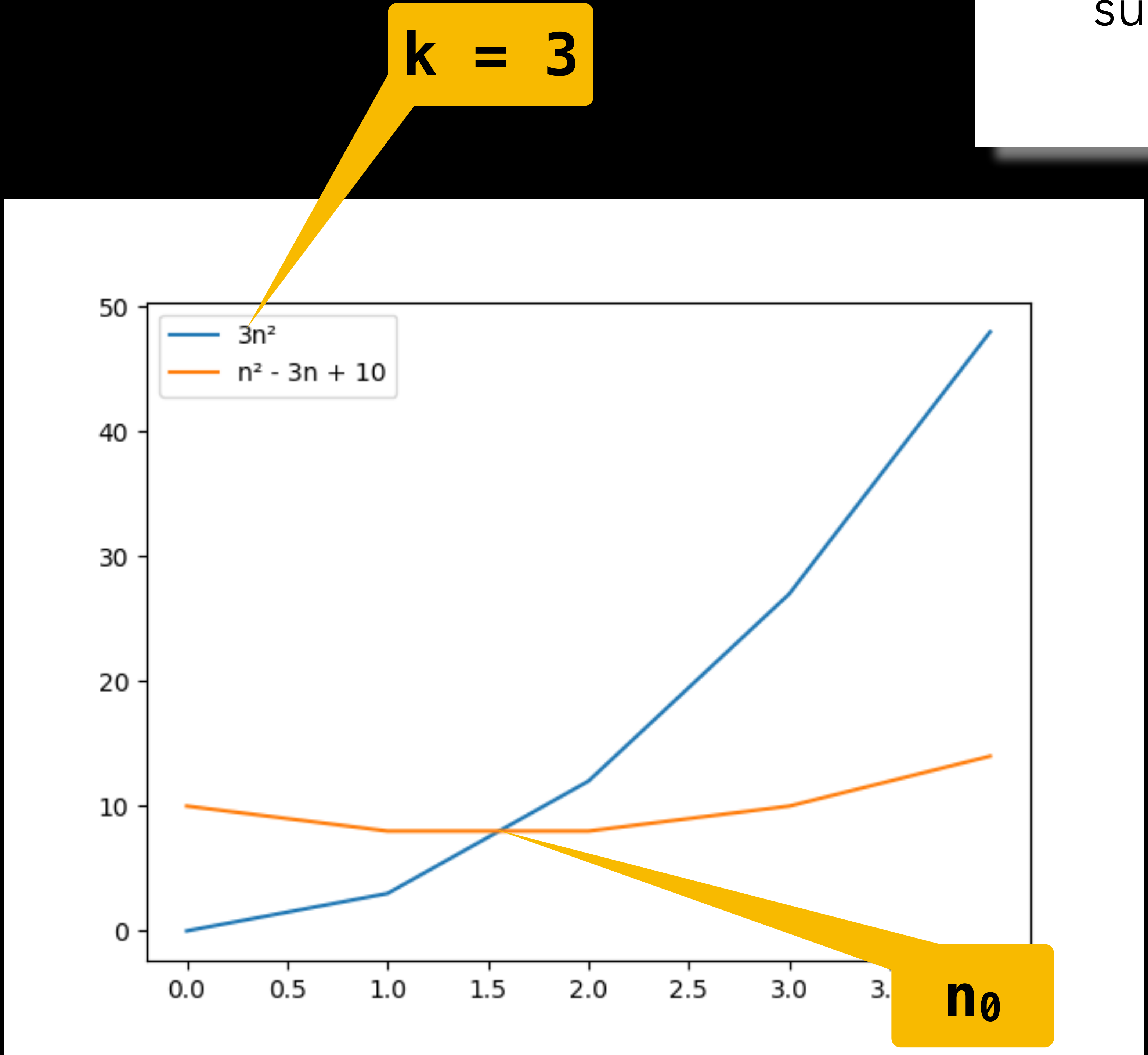
More formally:

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

if there exist constants k and n_0

such that for all $n \geq n_0$,

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



k = 3

n_0

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

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

For k=3 and n≥1.5

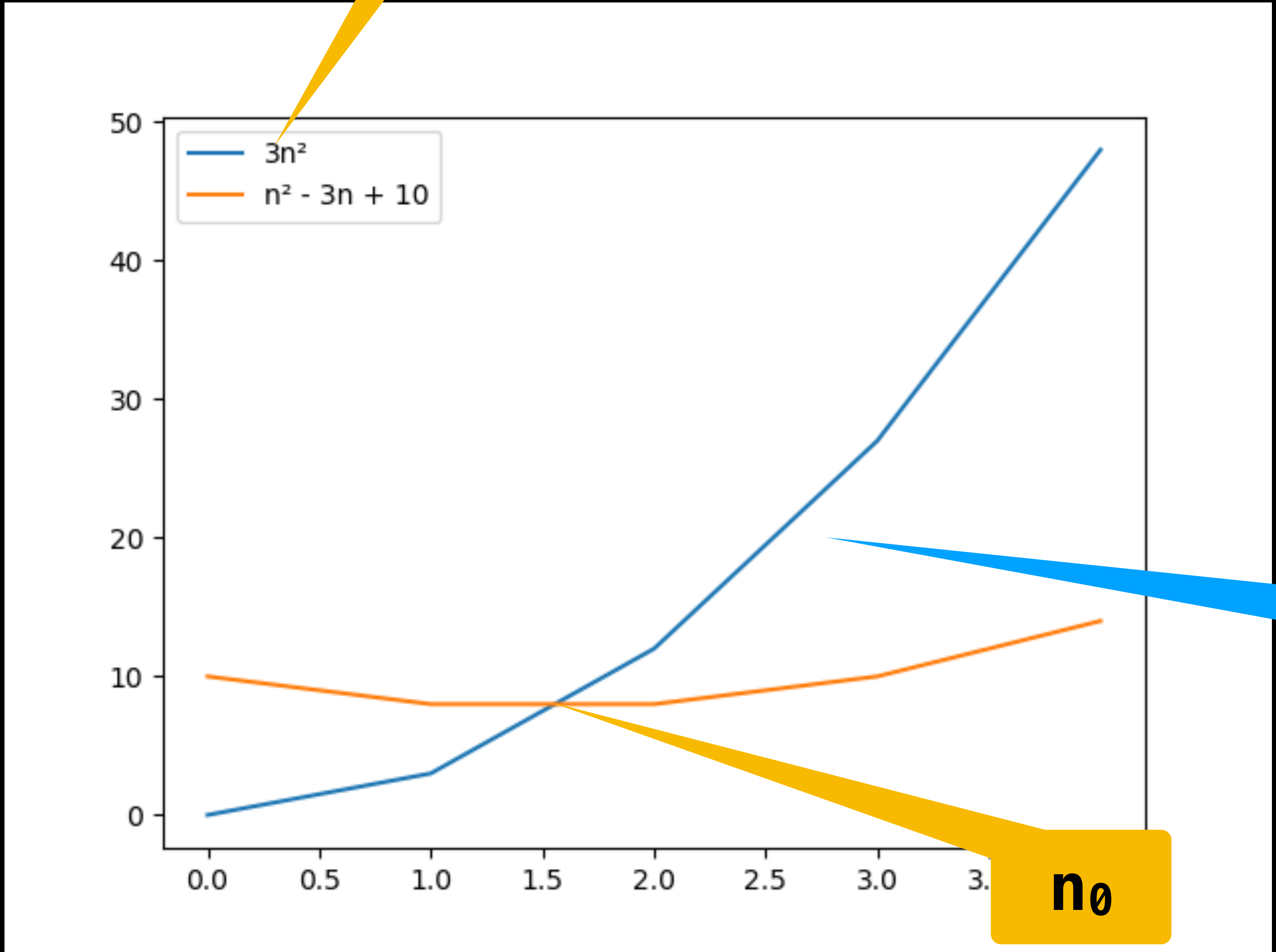
More formally:

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

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

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

$k = 3$



n_0

$T(n) = n^2 - 3n + 10$
$T(n)$ 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 overall growth rate of an algorithms for **large n**

Proving Big-O Relationship

Apply definition of Big-O to prove that $T(n)$ is $O(f(n))$ for particular functions T and f

Do so by choosing k and n_0 s.t. for all $n \geq n_0$, $T(n) \leq kf(n)$

Proving Big-O Relationship

Example:

Suppose $T(n) = (n+1)^2$

We can say that $T(n)$ is $O(n^2)$

To prove it must find k and n_0 s.t. for all $n \geq n_0$,

$(n+1)^2 \leq kn^2$

Proving Big-O Relationship

Example:

Suppose $T(n) = (n+1)^2$

We can say that $T(n)$ is $O(n^2)$

To prove it must find k and n_0 s.t. for all $n \geq n_0$,

$(n+1)^2 \leq kn^2$

Expand $(n+1)^2 = n^2 + 2n + 1$

Observe that, as long as $n \geq 1$, $n \leq n^2$ and $1 \leq n^2$

Proving Big-O Relationship

Example:

Suppose $T(n) = (n+1)^2$

We can say that $T(n)$ is $O(n^2)$

To prove it must find k and n_0 s.t. for all $n \geq n_0$,

$(n+1)^2 \leq kn^2$

Expand $(n+1)^2 = n^2 + 2n + 1$

Observe that, as long as $n \geq 1$, $n \leq n^2$ and $1 \leq n^2$

Thus if we choose $n_0 = 1$ and $k = 4$ we have

$n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$ ✓

T(n)

k

f(n)

Proving Big-O Relationship

Example:

Suppose $T(n) = (n+1)^2$

We can say that $T(n)$ is $O(n^2)$

To prove it must find k and n_0 s.t. for all $n \geq n_0$,

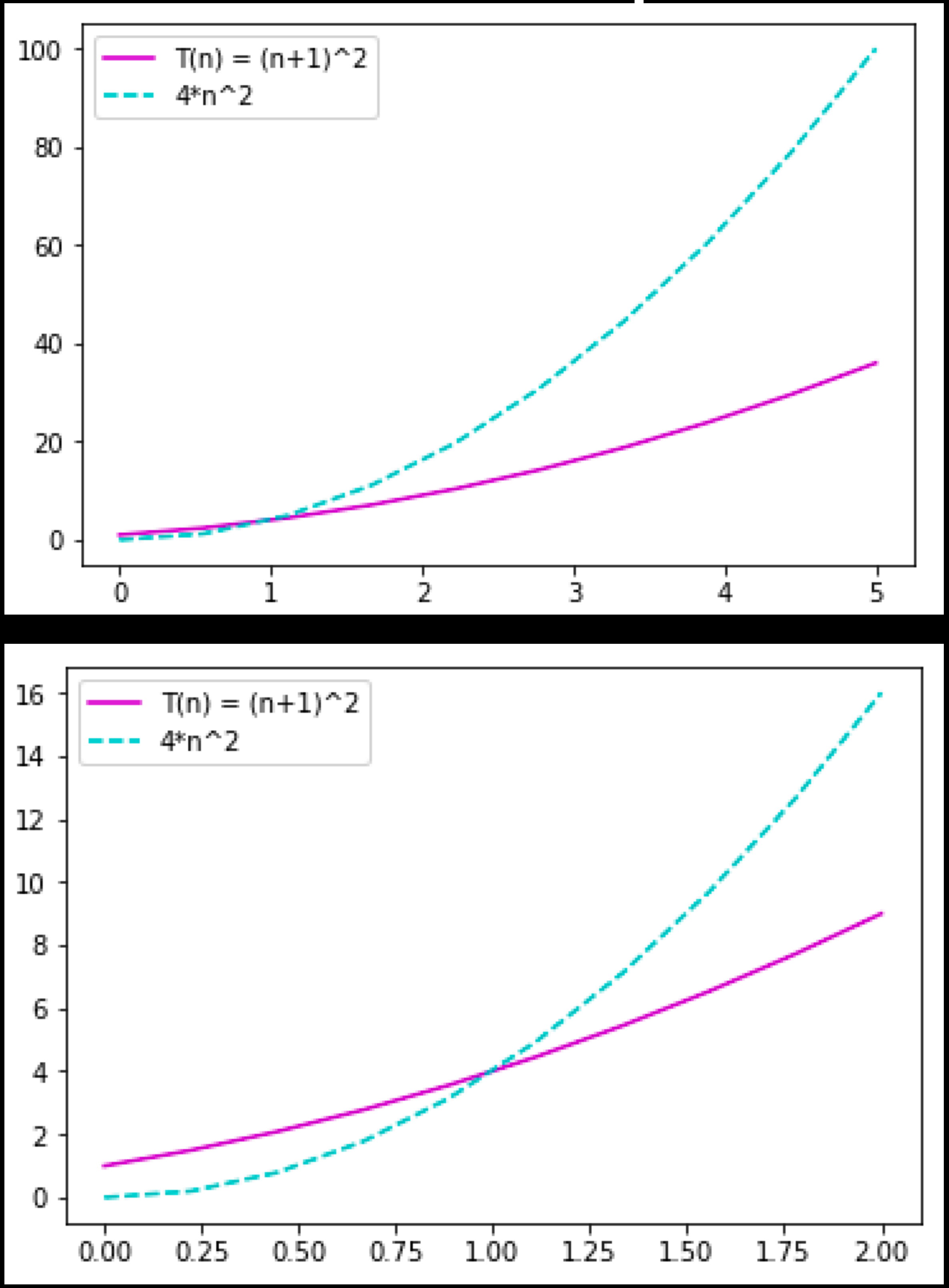
$(n+1)^2 \leq kn^2$

Expand $(n+1)^2 = n^2 + 2n + 1$

Observe that, as long as $n \geq 1$, $n \leq n^2$ and $1 \leq n^2$

Thus if we choose $n_0 = 1$ and $k = 4$ we have

$n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$ ✓



Proving Big-O Relationship

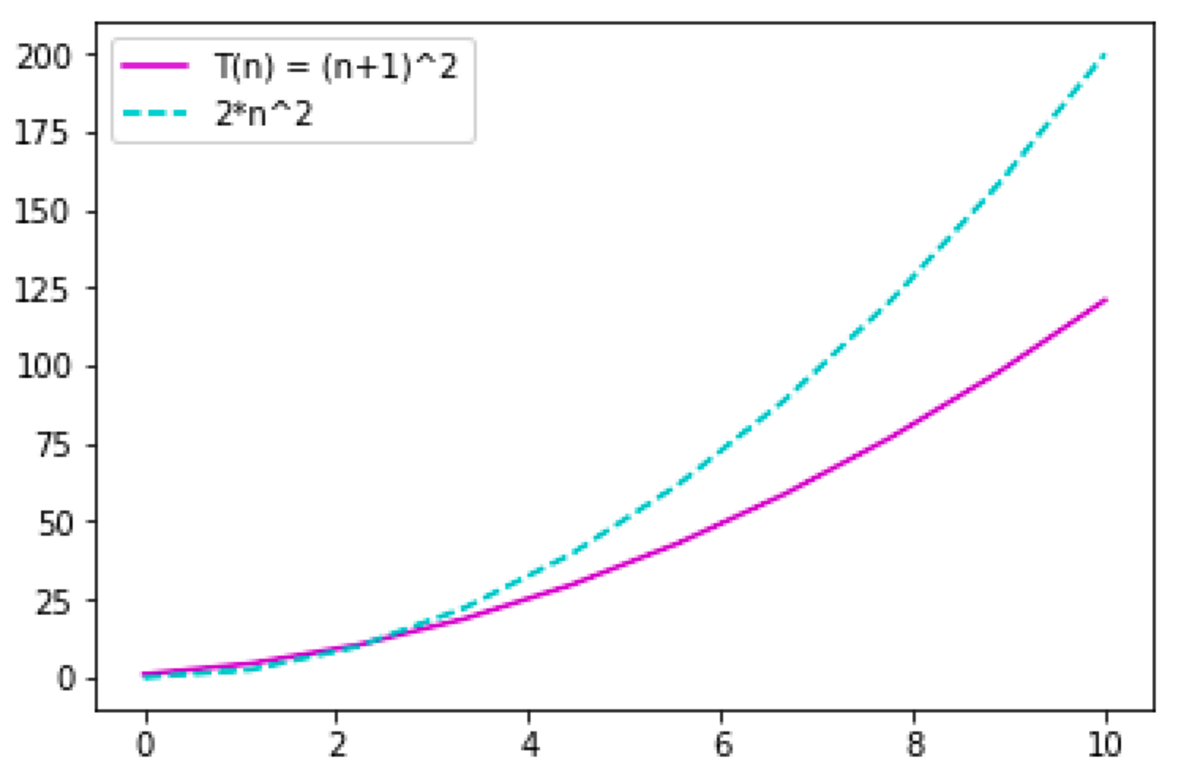
Not Unique:

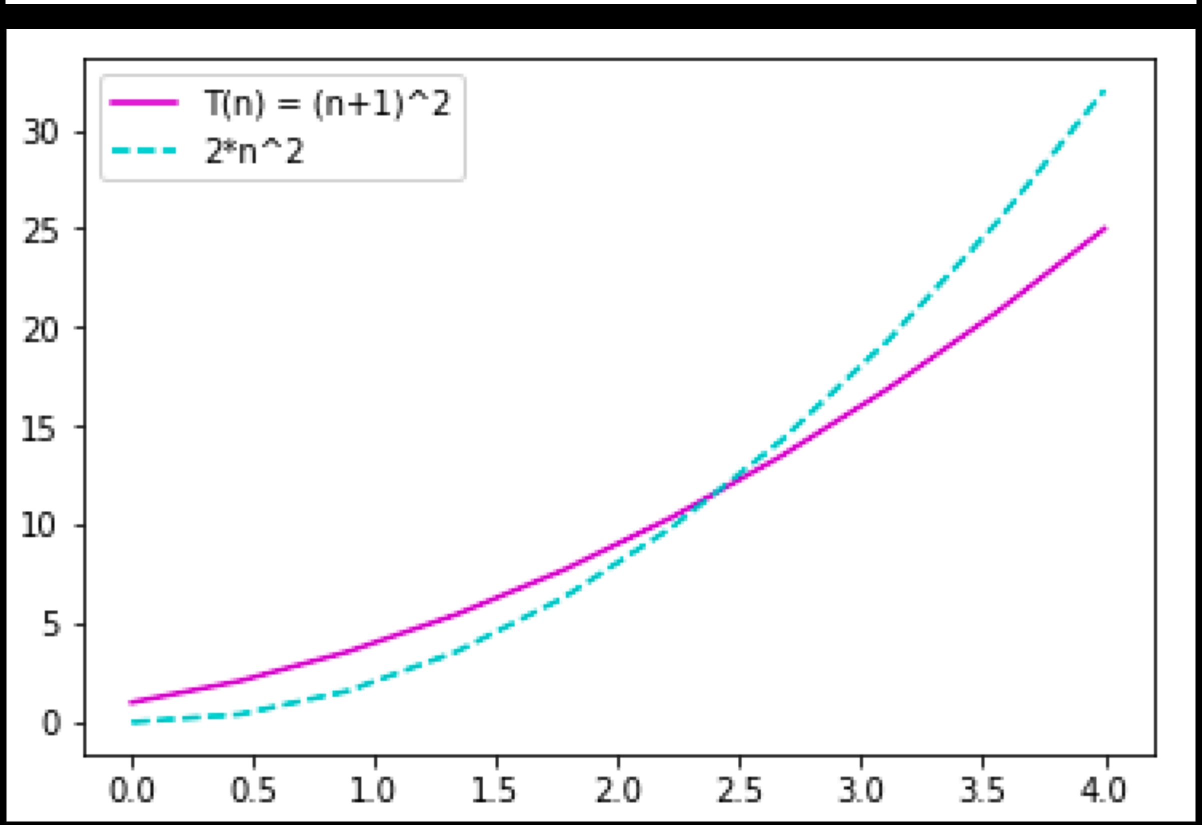
Could also choose $n_0 = 3$ and

$k = 2$ because

$(n+1)^2 \leq 2n^2$ for all $n \geq 3$

For proof one is enough





Complexity classes

$O(1)$: **Constant** worst-case running time

$O(\log n)$: **Logarithmic** worst-case running time

$O(n)$: **Linear** worst-case running time

$O(n \log n)$: **Log-Linear** worst-case running time

$O(n^2)$: **Quadratic** worst-case running time

$O(n^3)$: **Cubic** worst-case running time

$O(n^k)$: **Polynomial** worst-case running time

$O(c^n)$: **Exponential** worst-case running time (too slow!)

Examples

O(1): Hello world! (Does not depend on input)

O(logn): `for(int i = n; i > 1; i= i/2)`

O(n): `for(int i = 0; i < n; I++)`

O(n logn):

```
for(int i = 0; i < n; i++)
    for(int i = 100; i > 1; i= i/2)
```

$O(n^2)$:

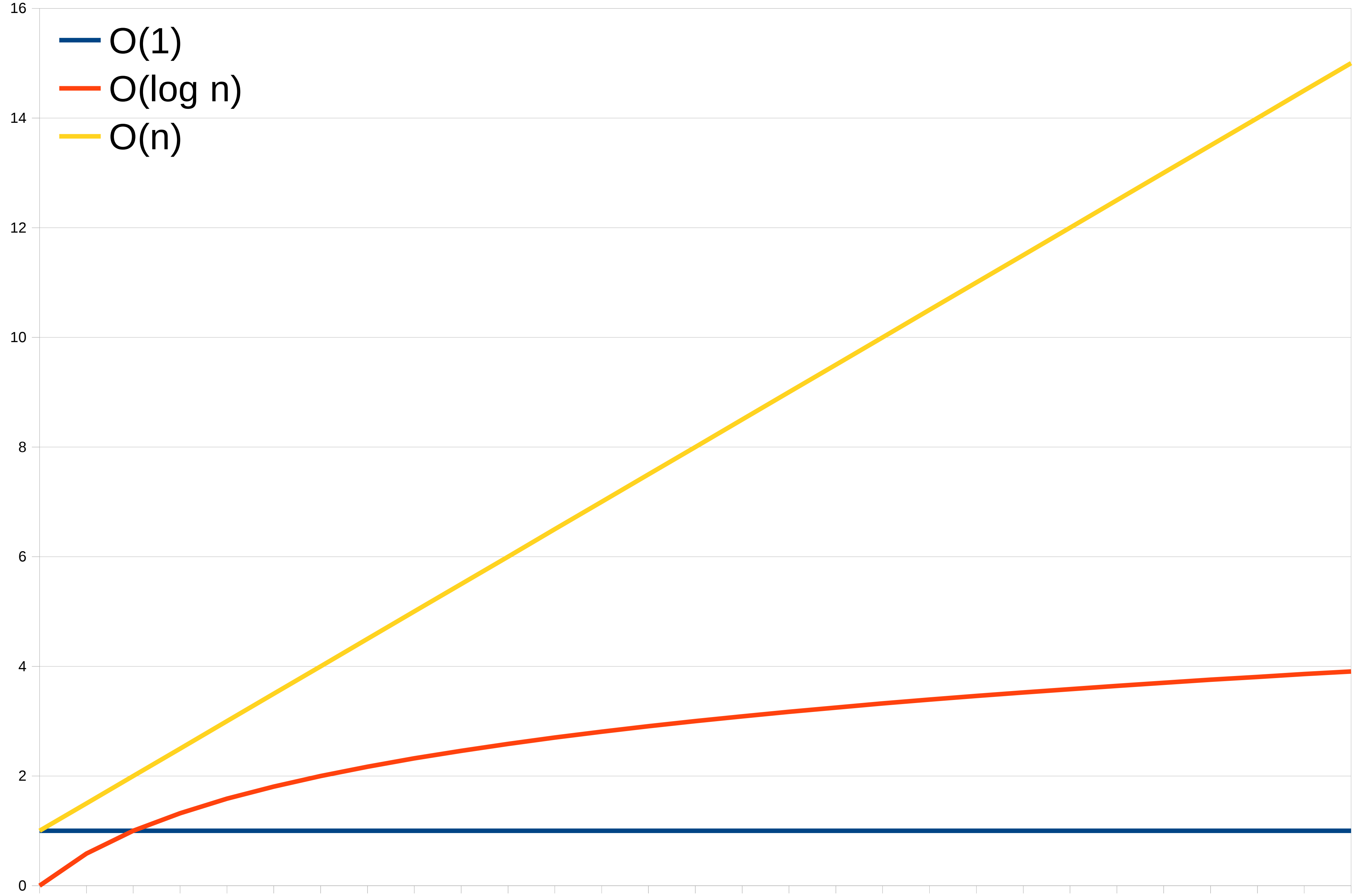
```
for(int i = 0; i < n; i++)
    for(int i = 0; i < n; I++)
```

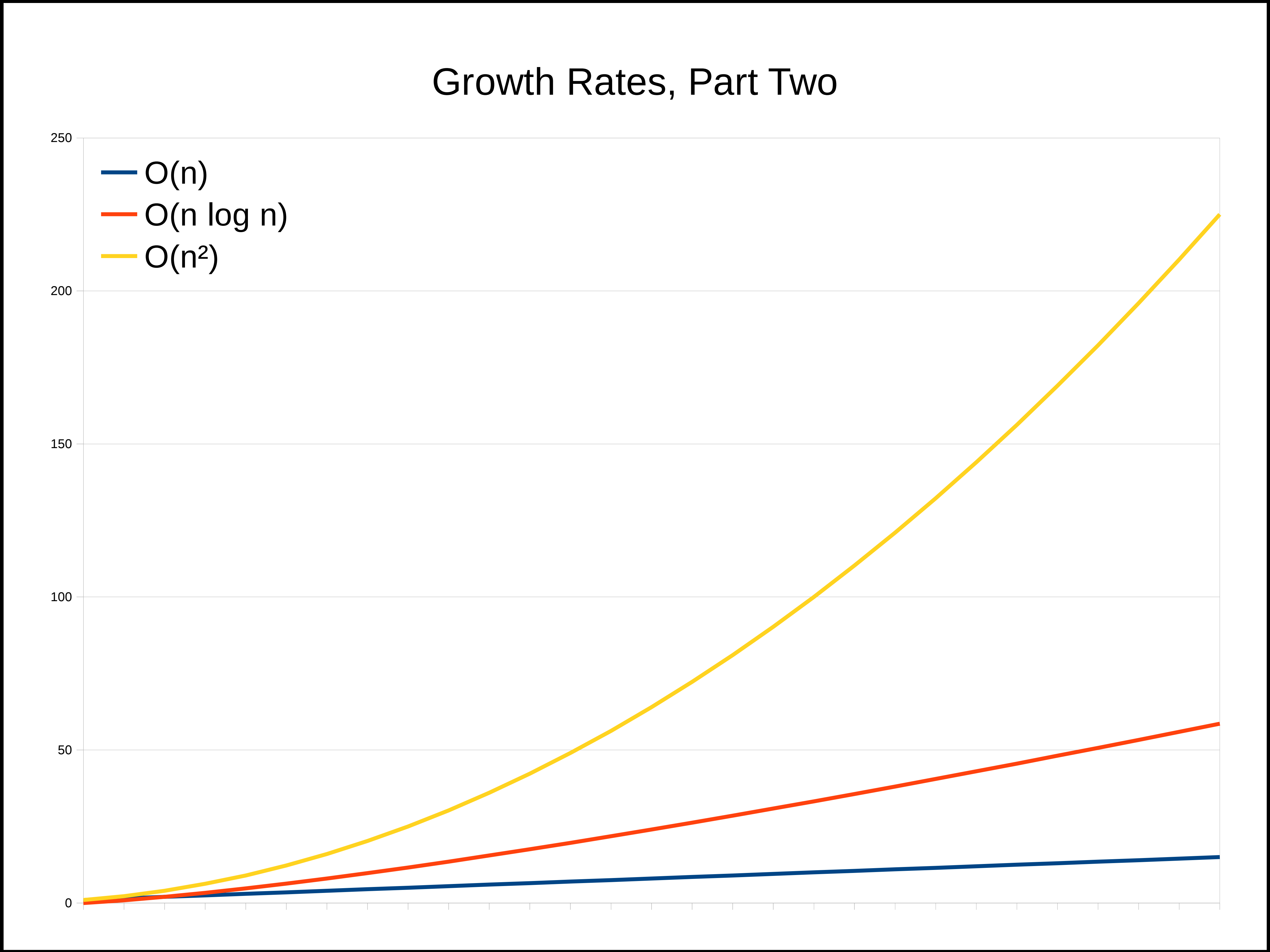
$O(2^n)$: Combinations - find all possible combinations of n elements

e.g. n=3: ({}, {a}, {b}, {c}, {a,b}, {a,c}, {b,c}, {a,b,c}) = 8 = 2^3

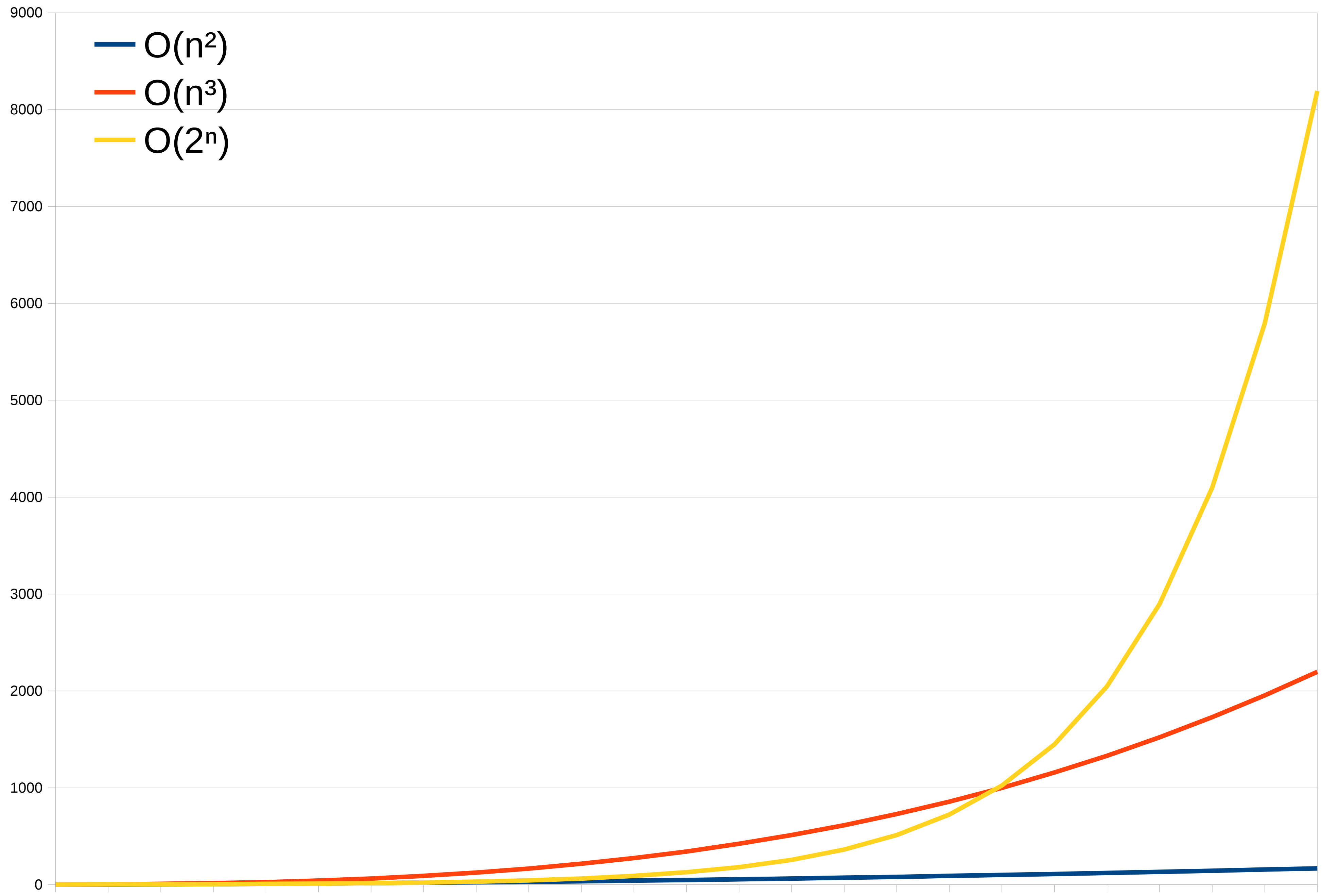
A visual comparison of growth rates

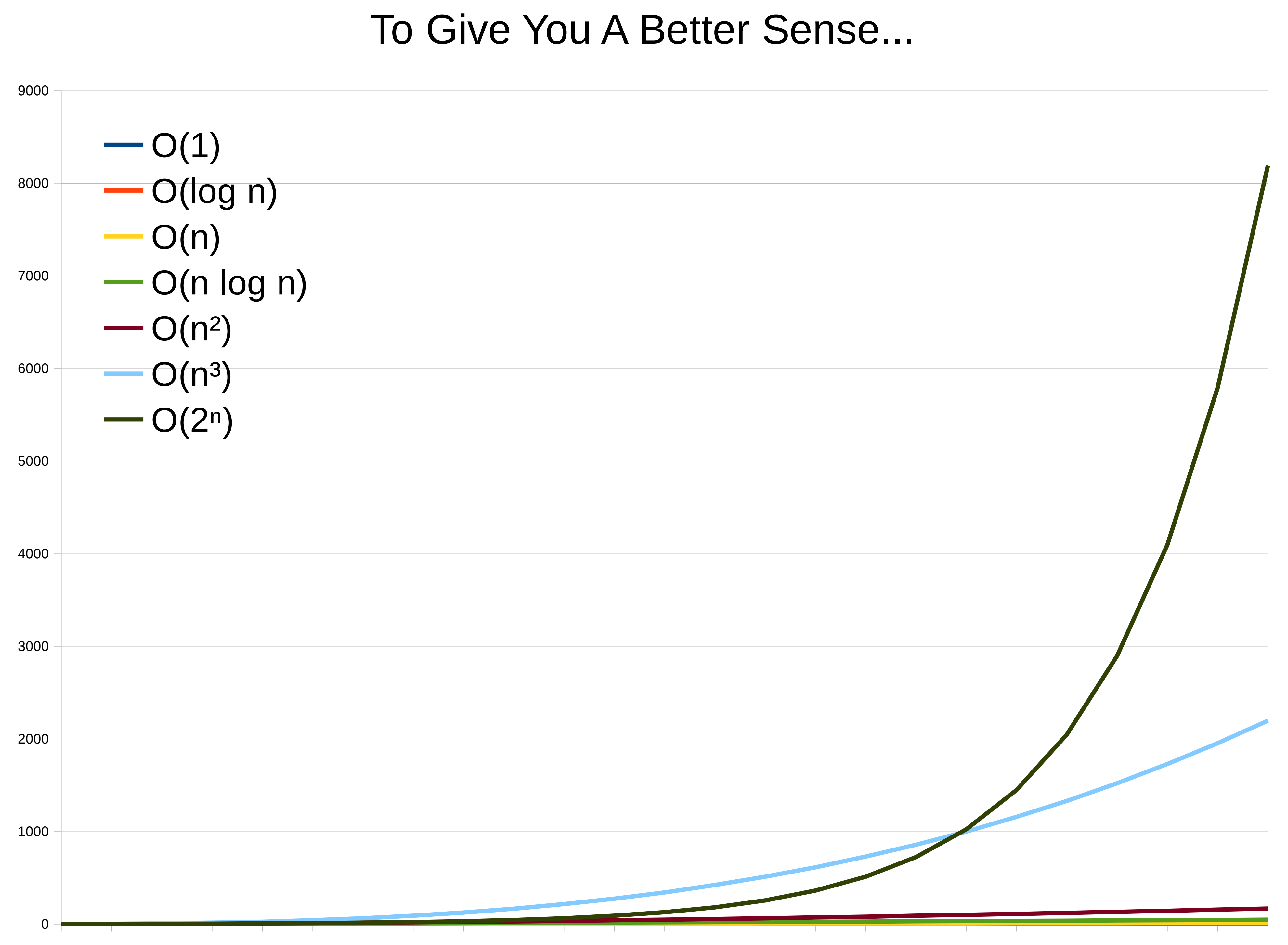
Growth Rates, Part One




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

Growth Rates, Part Three




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)

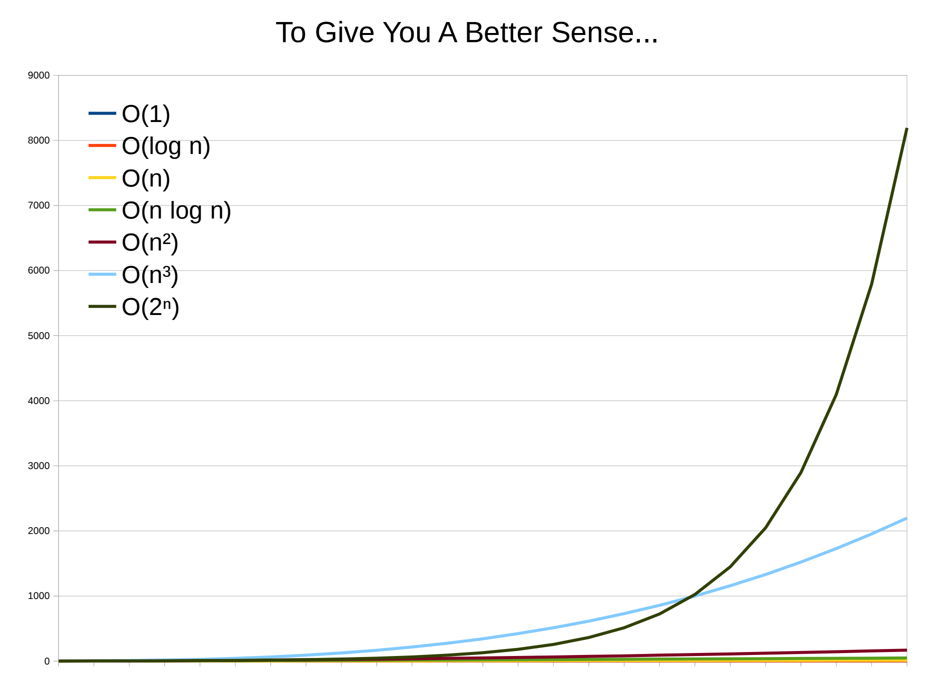
Tight is more meaningful

If $T(n)$ is $O(n)$

It is also true that $T(n)$ is $O(n^3)$

And it is also true that $T(n)$ is $O(2^n)$

But what does it mean???



The closest Big-O is the most descriptive of the overall worst-case behavior

Tightening the bounds

Big-O: upper bound

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

if there exist constants **k** and **n_0** such that for all $n \geq n_0$ $T(n) \leq k\, f(n)$

Grows no faster than f(n)

Tightening the bounds

Big-O: upper bound

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

if there exist constants **k** and **n_0** such that for all $n \geq n_0$ $T(n) \overset{\downarrow}{\leq} k\, f(n)$

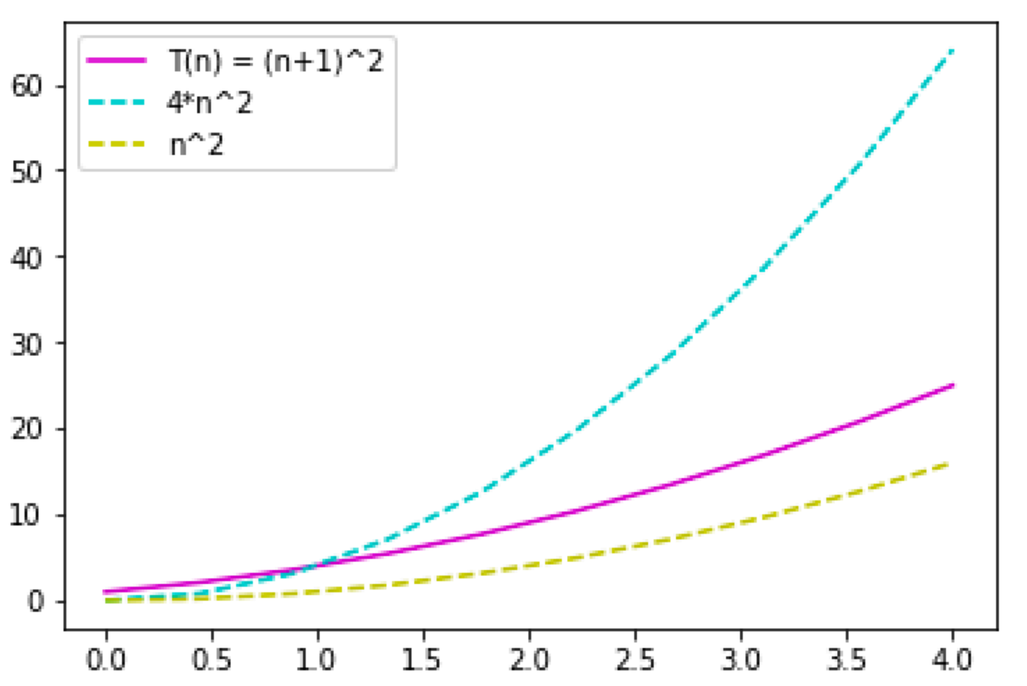
Grows no faster than f(n)

Omega: lower bound

T(n) is Ω(f(n))

if there exist constants **k** and **n_0** such that for all $n \geq n_0$ $T(n) \overset{\downarrow}{\geq} k\, f(n)$

Grows at least as fast as f(n)

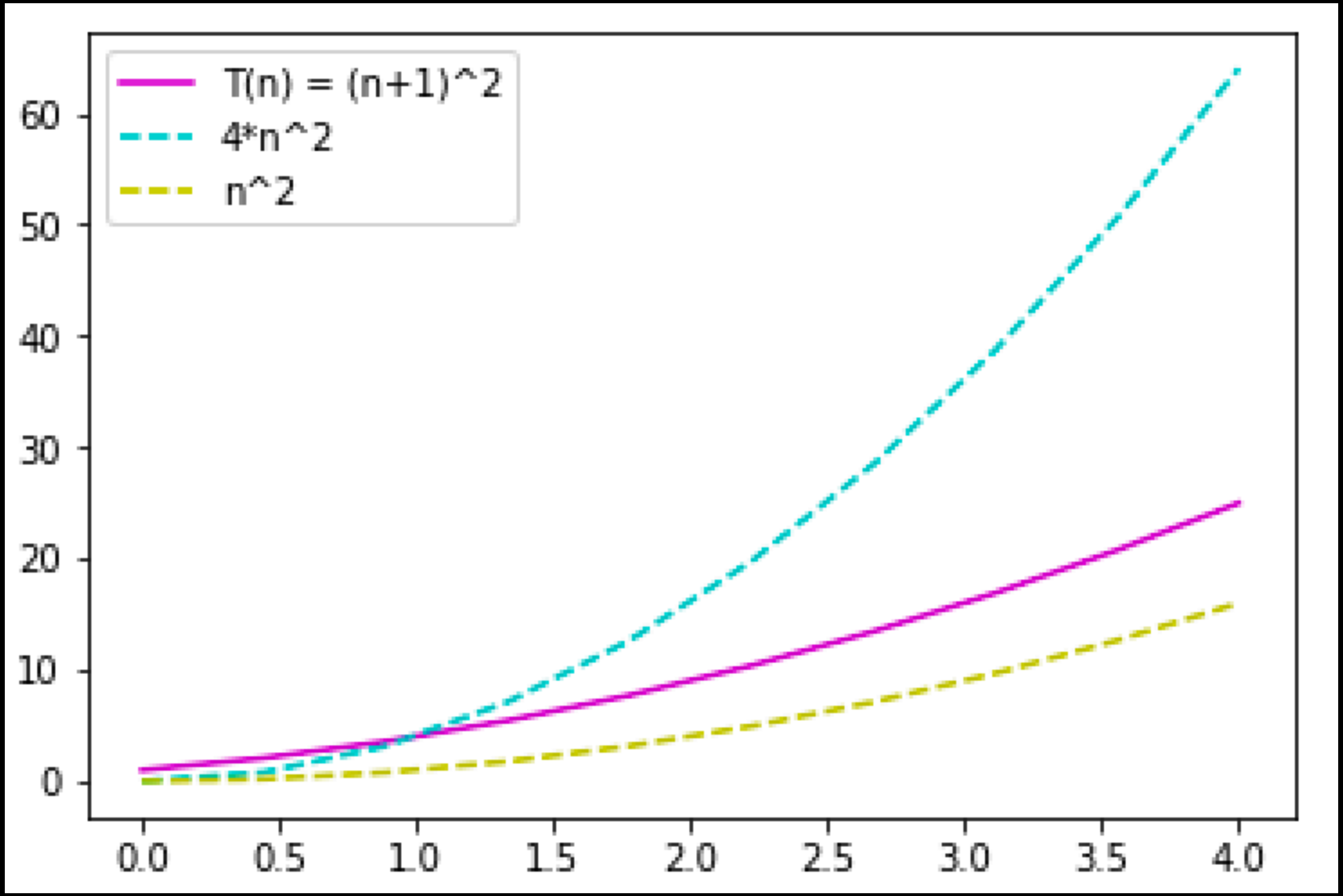


Tightening the bounds

Theta: tight bound

T(n) is Θ(f(n))

Grows at the same rate as f(n) : iff both T(n) is O(f(n)) and Ω(f(n))



A numerical comparison of growth rates

n f(n)	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), 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

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

$10^6 = O(1)$

Space Complexity

Similarly, you can think about the space complexity

How much space in memory (as a function of the size of the input)?

Examples later in the course.

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

From now on, you will think about every algorithm in these terms!!!

Next time Pointers