

CALCULATING BIG-O EXPERIMENTALLY & MATHEMATICALLY

Paul Doliotis (*PhD*)
Adjunct Assistant Professor
Portland State University

How many shortest-length paths are there to get from your house to the doughnut shop?

$\binom{n}{k} = \frac{n!}{(n-k)!k!}$

$e^{i\pi} + 1 = 0$

\uparrow up's
 \rightarrow right's

$\binom{11}{7} = \binom{11}{4} = 330$ paths

B_4

Δ^3

Onto

One-to-One

Find $7 + 12 + 17 + 22 + \dots + 342$

$S_n = 7 + 12 + 17 + 22 + \dots + 342$

$+S_n = 342 + 337 + 332 + 327 + \dots + 7$

$2S_n = 349 + 349 + 349 + 349 + \dots + 349$

$2S_n = 349 \cdot 68$

$S_n = \frac{349 \cdot 68}{2}$

$S_n = 11866$

Original:
 $\exists x \forall y (x \geq 2y \rightarrow x > y + 1)$

Converse:
 $\exists x \forall y (x > y + 1 \rightarrow x \geq 2y)$

Negation:
 $\neg [\exists x \forall y (\neg (x \geq 2y) \vee x > y + 1)]$

$\forall x \exists y (x \geq 2y \wedge x \leq y + 1)$

Contrapositive:
 $\exists x \forall y (x \leq y + 1 \rightarrow x < 2y)$

$v - e + f = 2$

P.I.E. Example:

$6! - \left[\binom{6}{1}5! - \binom{6}{2}4! + \binom{6}{3}3! - \binom{6}{4}2! + \binom{6}{5}1! \right]$

$(A \cup B \cup C) \cup (A \cap B \cap C)$

$7, 11, 15, 19, 23, \dots$

$a_1 - a_2 = 4$

$a_2 - a_3 = 4$

$a_3 - a_4 = 4$

\vdots

$+ a_n - a_{n+1} = 4$

$a_n - a_1 = 4n$

$a_n = a_1 + 4n$

$K_{3,3}$

$A = \{2, 4, 10, \frac{1}{2}\}$

Lecture Outline

- Find Duplicate Algorithm
 - A first analysis on time complexity
- Experimental Analysis
- Mathematical Analysis
 - Closed form runtime using sums
 - Asymptotic Analysis
 - Calculating Big-O



FIND DUPLICATE ALGORITHM

A first analysis on time complexity

Why Algorithms? An Example

- Let's discuss a problem that frequently arises when developing a web search engine.
- Every day, we have a list A of web pages that we have to visit.

Why Algorithms? An Example

- Let's discuss a problem that frequently arises when developing a web search engine.
- Every day, we have a list A of web pages that we have to visit.
 - What does “visiting” a web page mean?

Why Algorithms? An Example

- Let's discuss a problem that frequently arises when developing a web search engine.
- Every day, we have a list A of web pages that we have to visit.
 - "visiting" a web page means that our program has downloaded that web page and processed it, so that it can show up in search results.

Why Algorithms? An Example

- Let's discuss a problem that frequently arises when developing a web search engine.
- Every day, we have a list A of web pages that we have to visit.
 - "visiting" a web page means that our program has downloaded that web page and processed it, so that it can show up in search results.
- Question: are there any duplicate links in list A?

Why Algorithms? An Example

- Let's discuss a problem that frequently arises when developing a web search engine.
- Every day, we have a list A of web pages that we have to visit.
 - "visiting" a web page means that our program has downloaded that web page and processed it, so that it can show up in search results.
- Question: are there any duplicate links in list A ?
- Why is this a useful question?

Why Algorithms? An Example

- Why is this a useful question?
 - Most times list A contains a great amount of duplicate links.

Why Algorithms? An Example

- Why is this a useful question?
 - Most times list A contains a great amount of duplicate links.
 - It is a **huge waste of resources** to revisit those links.

Why Algorithms? An Example

- Why is this a useful question?
 - Most times list A contains a great amount of duplicate links.
 - It is a **huge waste of resources** to revisit those links.
 - If we can find duplicate links eventually we can create a new list with unique entries.

Why Algorithms? An Example

- Recap:
 - A list A that contains strings representing webpages.
 - List A can have duplicate entries.
 - Define $\text{FindDuplicate}(A)$, return true if there is a duplicate, false otherwise.
- Question: how do we compute $\text{FindDuplicate}(A)$?
- Any ideas?

FindDuplicate(A) – First Version

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
for i=1 to n:
    for j=1 to n:
        if (i != j) AND A[i] == A[j]
            return true
return false
```

FindDuplicate(A) – First Version

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
for i=1 to n:
    for j=1 to n:
        if (i != j) AND A[i] == A[j]
            return true
return false
```

- What can we say about how fast this would run? How many comparisons?

FindDuplicate(A) – First Version

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
for i=1 to n:
    for j=1 to n:
        if (i != j) AND A[i] == A[j]
            return true
return false
```

- This algorithm will compare every item in the list against the entire list itself.
- The size of A is n , so we need n^2 comparisons

FindDuplicate(A) – First Version

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
for i=1 to n:
    for j=1 to n:
        if (i != j) AND A[i] == A[j]
            return true
return false
```

- This algorithm will compare every item in the list against the entire list itself.
- The size of A is n , so we need n^2 comparisons
- runtime function $f(n) \approx n^2$, with input variable n

FindDuplicate(A) – First Version

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
for i=1 to n:
    for j=1 to n:
        if (i != j) AND A[i] == A[j]
            return true
return false
```

- runtime function $f(n) \approx n^2$, with input variable n
- This is our first analysis of **time complexity!**

FindDuplicate(A) – First Version Speed

- Will it always run all the way through?

FindDuplicate(A) – First Version Speed

- Will it always run all the way through?
- What if the first two elements are duplicates?

FindDuplicate(A) – First Version Speed

- Will it always run all the way through?
- What if the first two elements are duplicates? This is the best case.
Constant running time.
 - Always the same regardless of input size n

FindDuplicate(A) – First Version Speed

- Will it always run all the way through?
- What if the first two elements are duplicates? This is the best case.
Constant running time.
 - Always the same regardless of input size n
- What if the last two elements are duplicates?

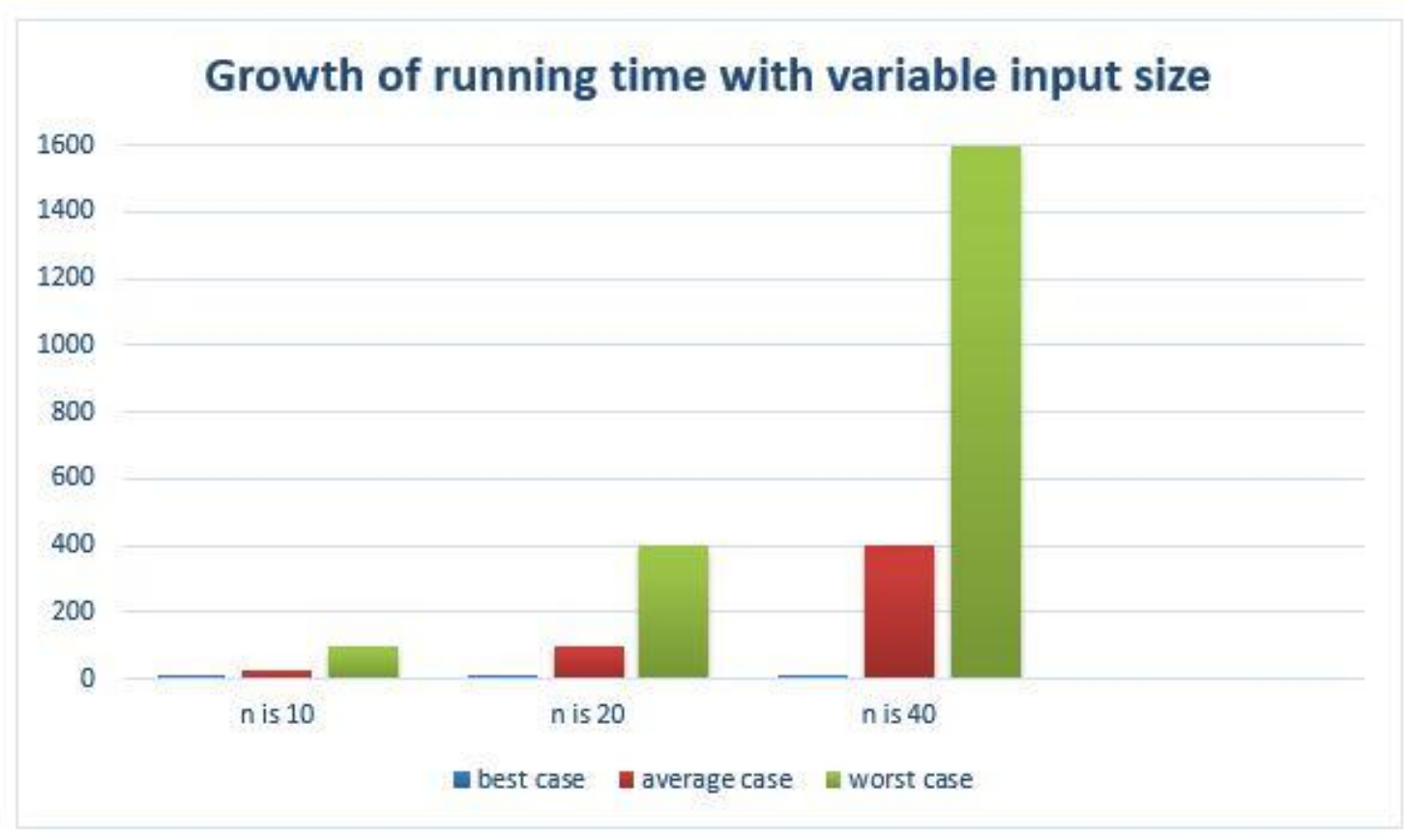
FindDuplicate(A) – First Version Speed

- Will it always run all the way through?
- What if the first two elements are duplicates? This is the best case.
Constant running time.
 - Always the same regardless of input size n
- What if the last two elements are duplicates? This is the worst case
 $\approx n^2$
 - Running time increases with n

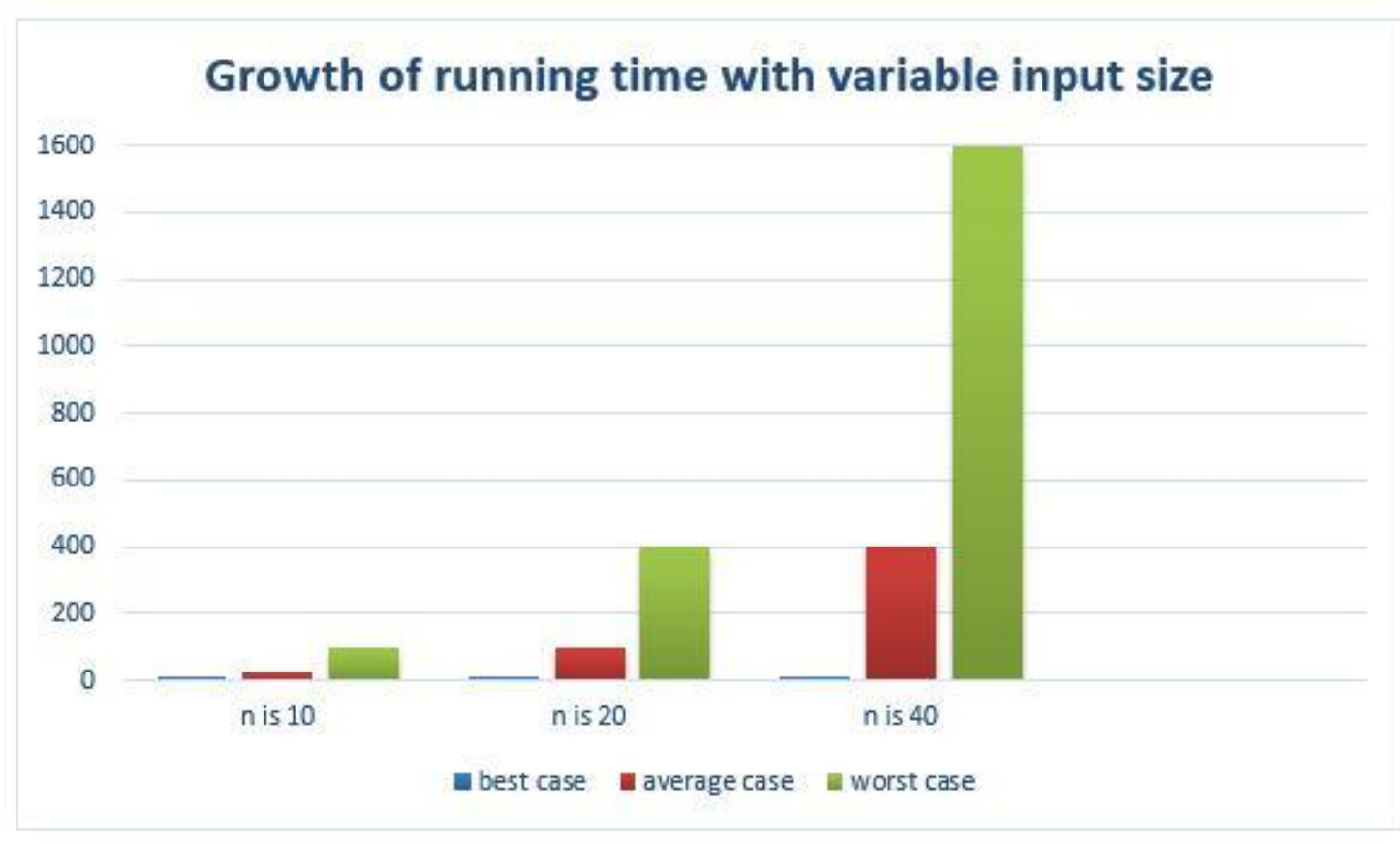
FindDuplicate(A) – First Version Speed

- Will it always run all the way through?
- What if the first two elements are duplicates? This is the best case.
Constant running time.
 - Always the same regardless of input size n
- What if the last two elements are duplicates? This is the worst case
 $\approx n^2$
 - Running time increases with n
- What about average case running time? Often difficult to determine but can be useful.

FindDuplicate(A) – First Version Speed



FindDuplicate(A) – First Version Speed



- Worst case complexity is what we will use mostly from now on.
- Easier to analyze
- **Very useful as input n grows very large**
- Asymptotic analysis

FindDuplicate(A) – First Version Speed

- What does all this mean in practice?
- We need n^2 comparisons in the worst case.
- Suppose A has 5 million items. We need to do 25 trillion comparisons.

FindDuplicate(A) – First Version Speed

- What does all this mean in practice?
- We need n^2 comparisons in the worst case.
- Suppose A has 5 million items. We need to do 25 trillion comparisons.
- On a computer that can do 1 billion comparisons per second

FindDuplicate(A) – First Version Speed

- What does all this mean in practice?
- We need n^2 comparisons in the worst case.
- Suppose A has 5 million items. We need to do 25 trillion comparisons.
- On a computer that can do 1 billion comparisons per second
 - This would take about 7 hours
 - This is very optimistic, in practice, it would be at least a few dozen hours

FindDuplicate(A) – First Version Speed

- Suppose A has 1 billion items. We need to do 1 quintillion comparisons.
- On a computer that can do 1 billion comparisons per second

FindDuplicate(A) – First Version Speed

- Suppose A has 1 billion items. We need to do 1 quintillion comparisons.
- On a computer that can do 1 billion comparisons per second
 - This would take 11574 days!
- Input size n grew only by 1000 but running time went from 7 hours to 11574 days!

FindDuplicate(A) – First Version Speed

- Suppose A has 1 billion items. We need to do 1 quintillion comparisons.
- On a computer that can do 1 billion comparisons per second
 - This would take 11574 days!
- Input size n grew only by 1000 but running time went from 7 hours to 11574 days!
 - **CAN WE DO BETTER?**

FindDuplicate(A) – First Version Speed

- Suppose A has 1 billion items. We need to do 1 quintillion comparisons.
- On a computer that can do 1 billion comparisons per second
 - This would take 11574 days!
- Input size n grew only by 1000 but running time went from 7 hours to 11574 days!
 - **CAN WE DO BETTER?**
 - Hint: sort list A

FindDuplicate(A) – Second Version

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
sort A
for i=1 to n-1:
    if A[i] == A[i+1]
        return true
return false
```

FindDuplicate(A) – Second Version

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
sort A
for i=1 to n-1:
    if A[i] == A[i+1]
        return true
return false
```

- What can we say about its speed? What takes time?

FindDuplicate(A) – Second Version

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
sort A
for i=1 to n-1:
    if A[i] == A[i+1]
        return true
return false
```

- What can we say about its speed? What takes time?
- We need to: sort A and execute the for loop.

FindDuplicate(A) – Second Version Speed

- We need to:
 - sort A
 - execute the for loop.
- How many calculations does it take to sort A?
 - We will learn that the number of calculations is proportional to $n \cdot \log(n)$

FindDuplicate(A) – Second Version Speed

- We need to:
 - sort A
 - execute the for loop.
- How many calculations does it take to sort A?
 - We will learn that the number of calculations is proportional to $n \cdot \log(n)$
- How many iterations does the for loop take?
 - no more than n .

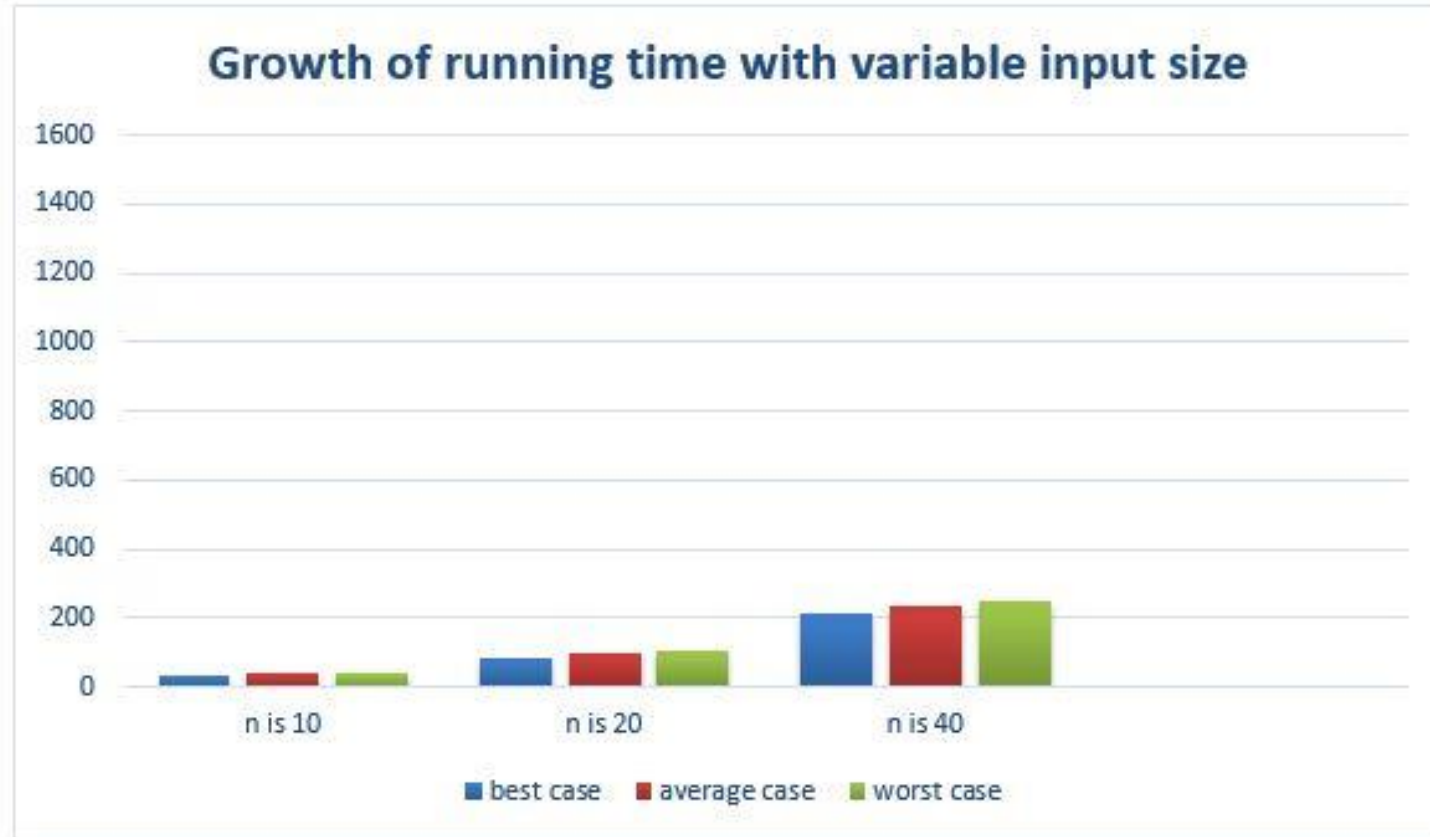
FindDuplicate(A) – Second Version Speed

- We need to:
 - sort A
 - execute the for loop.
- How many calculations does it take to sort A?
 - We will learn that the number of calculations is proportional to $n \cdot \log(n)$
- How many iterations does the for loop take?
 - no more than n .
- Total iterations $n \cdot (\log(n) + 1) \approx n \cdot \log(n)$, constants do not matter!

FindDuplicate(A) – Second Version Speed

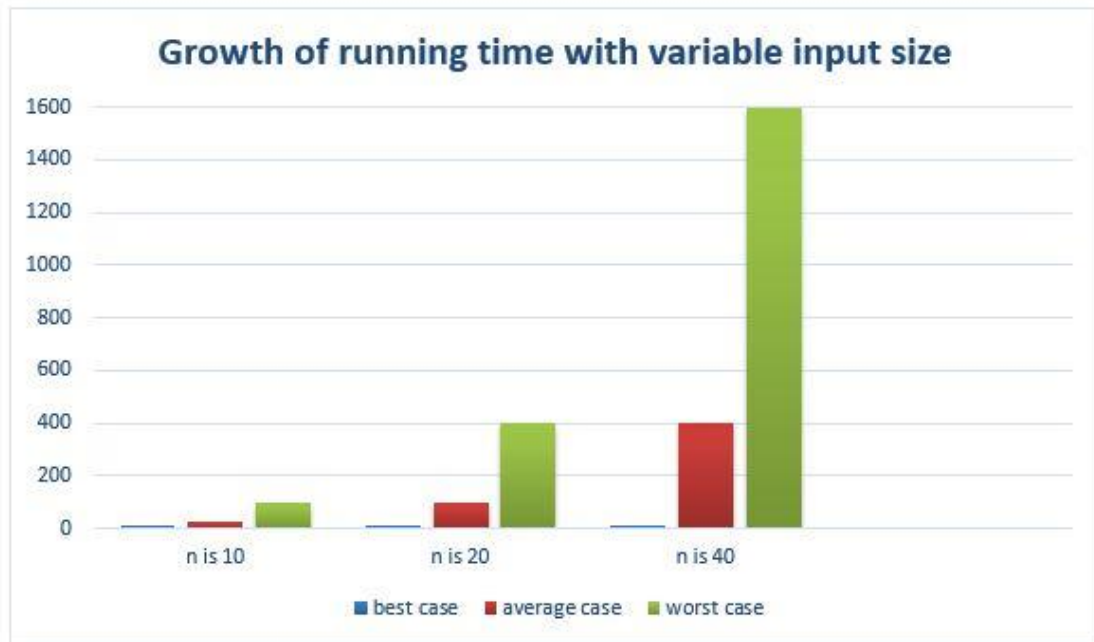
- Will it always run all the way through?
- What if the first two elements of sorted array are duplicates? This is the best case.
 - $\approx n \cdot \log(n) + 1$, we always have to sort the array
- What if the last two elements of sorted array are duplicates? This is the worst case
 - $\approx n \cdot \log(n) + n$
- What about average case running time? Proportional to $n \cdot \log(n)$

FindDuplicate(A) – Second Version Speed

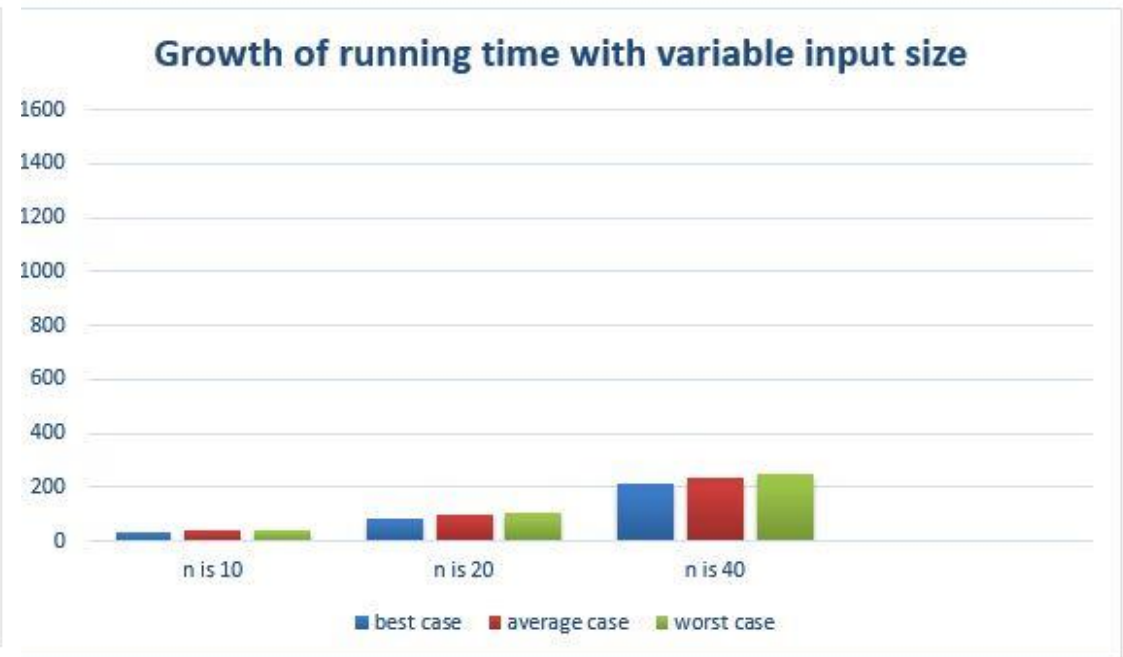


FindDuplicate(A) – Second Version Speed

$$f_1(n) \approx n^2$$



$$f_2(n) \approx n * \log(n)$$



FindDuplicate(A) – Second Version Speed

- The worst case number of calculations is approximately $n * \log(n)$
- Suppose A has 5 million items.
 - $\log(n)$ is 22.253
 - We need to do 111.27 million comparisons.

FindDuplicate(A) – Second Version Speed

- The worst case number of calculations is approximately $n * \log(n)$
- Suppose A has 5 million items.
 - $\log(n)$ is 22.253
 - We need to do 111.27 million comparisons.
- On a computer that can do 1 billion comparisons per second, this would take about 110 milliseconds

FindDuplicate(A) – Second Version Speed

- The worst case number of calculations is approximately $n * \log(n)$
- Suppose A has 5 million items.
 - $\log(n)$ is 22.253
 - We need to do 111.27 million comparisons.
- On a computer that can do 1 billion comparisons per second, this would take about 110 milliseconds
- Suppose A has 1 billion items. On a computer that can do 1 billion comparisons per second, this would take about 30 seconds

FindDuplicate(A) – Second Version Speed

	Worst case running time for first version $\approx n^2$	Worst case running time for second version $\approx n*\log(n)$
Input size $n = 5*10^6$	7 hours	110 milliseconds
Input size $n = 10^9$	11574 days	30 seconds

FindDuplicate(A) – Second Version Speed

	Worst case running time for first version $\approx n^2$	Worst case running time for second version $\approx n*\log(n)$
Input size $n = 5*10^6$	7 hours	110 milliseconds
Input size $n = 10^9$	11574 days	30 seconds

- Running two alternative algorithms on the same data and comparing the running times can be a useful tool.

FindDuplicate(A) – Second Version Speed

	Worst case running time for first version $\approx n^2$	Worst case running time for second version $\approx n*\log(n)$
Input size $n = 5*10^6$	7 hours	110 milliseconds
Input size $n = 10^9$	11574 days	30 seconds

- Running two alternative algorithms on the same data and comparing the running times can be a useful tool.
- 1 second vs. one minute is an easy-to-notice difference

FindDuplicate(A) – Second Version Speed

	Worst case running time for first version $\approx n^2$	Worst case running time for second version $\approx n*\log(n)$
Input size $n = 5*10^6$	7 hours	110 milliseconds
Input size $n = 10^9$	11574 days	30 seconds

- Running two alternative algorithms on the same data and comparing the running times can be a useful tool.
- 1 second vs. one minute is an easy-to-notice difference
- However, sometimes this empirical analysis is not a good option.

FindDuplicate(A) – Second Version Speed

	Worst case running time for first version $\approx n^2$	Worst case running time for second version $\approx n*\log(n)$
Input size $n = 5*10^6$	7 hours	110 milliseconds
Input size $n = 10^9$	11574 days	30 seconds

- However, sometimes this empirical analysis is not a good option.
 - For example, if it would take days or weeks to run the programs

FindDuplicate(A) – Second Version Speed

	Worst case running time for first version $\approx n^2$	Worst case running time for second version $\approx n \cdot \log(n)$
Input size $n = 5 \cdot 10^6$	7 hours	110 milliseconds
Input size $n = 10^9$	11574 days	30 seconds

- However, sometimes this empirical analysis is not a good option.
 - For example, if it would take days or weeks to run the programs
 - When comparing running times of two implementations, we must make sure the comparison is fair. Same hardware and software environments must be used



EXPERIMENTAL ANALYSIS

Experimental Analysis

- According to our time complexity analysis from the previous section, the brute-force implementation for FindDuplicate(A) requires n^2 comparisons.

Experimental Analysis

- According to our time complexity analysis from the previous section, the brute-force implementation for FindDuplicate(A) requires n^2 comparisons.
- How accurately does this approximation represent the actual running time function?

Experimental Analysis

- According to our time complexity analysis from the previous section, the brute-force implementation for FindDuplicate(A) requires n^2 comparisons.
- How accurately does this approximation represent the actual running time function?
- Can we experimentally come up with a more accurate representation for the running time function?

Experimental Analysis

Code a program implementing the given algorithm



Generate datasets with varying size/composition



Run the program with those datasets as input



Measure of the actual running time

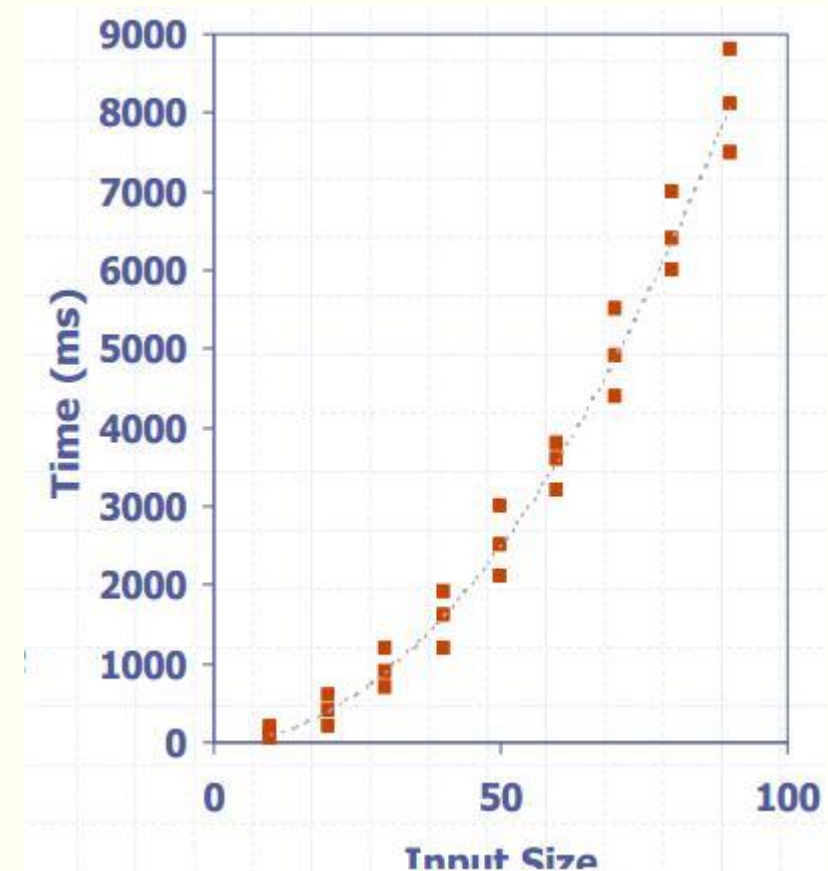
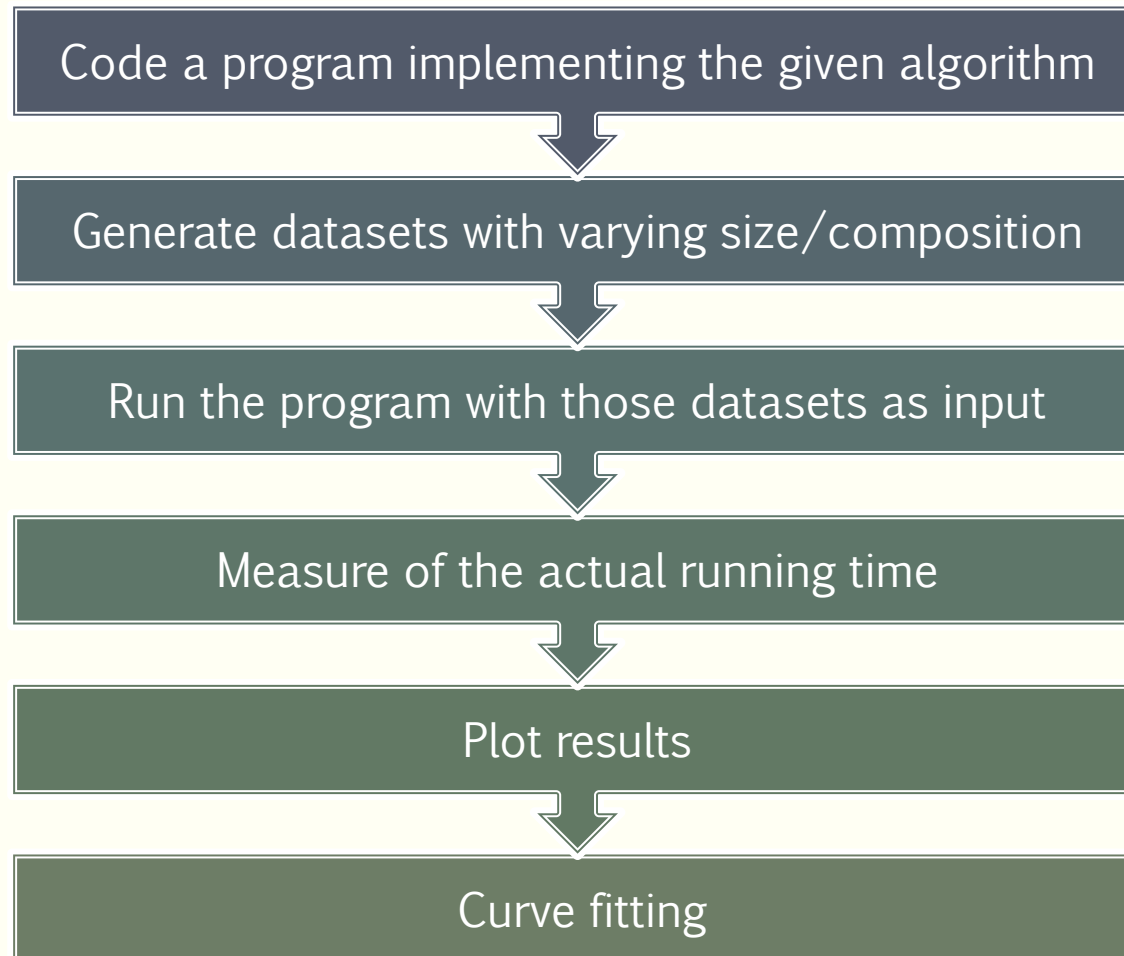


Plot results



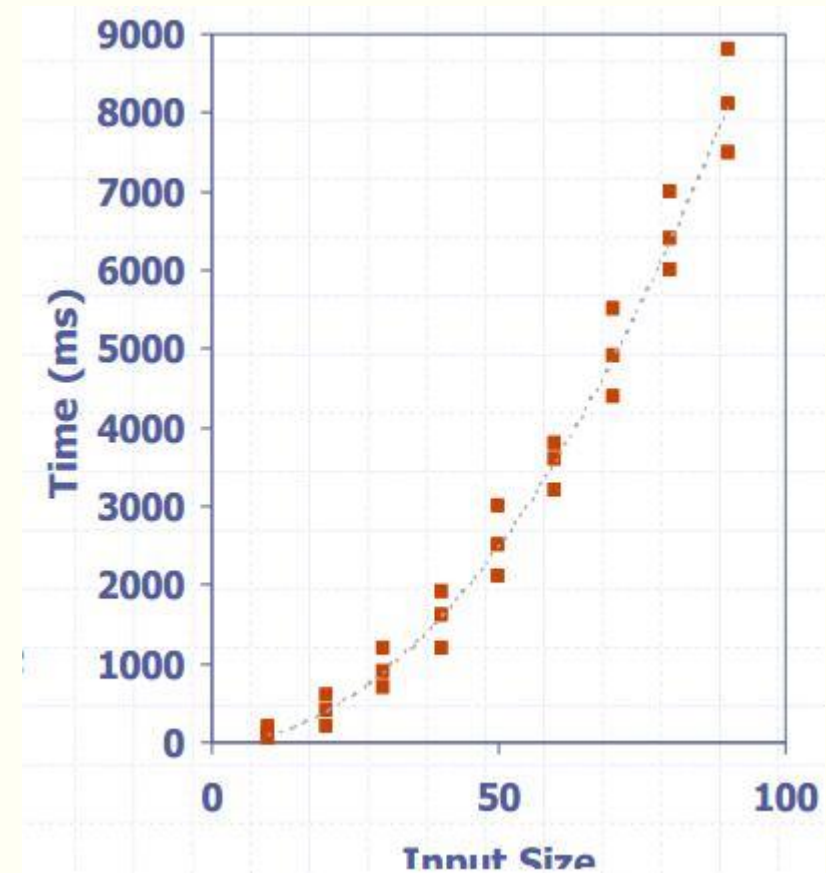
Curve fitting

Experimental Analysis



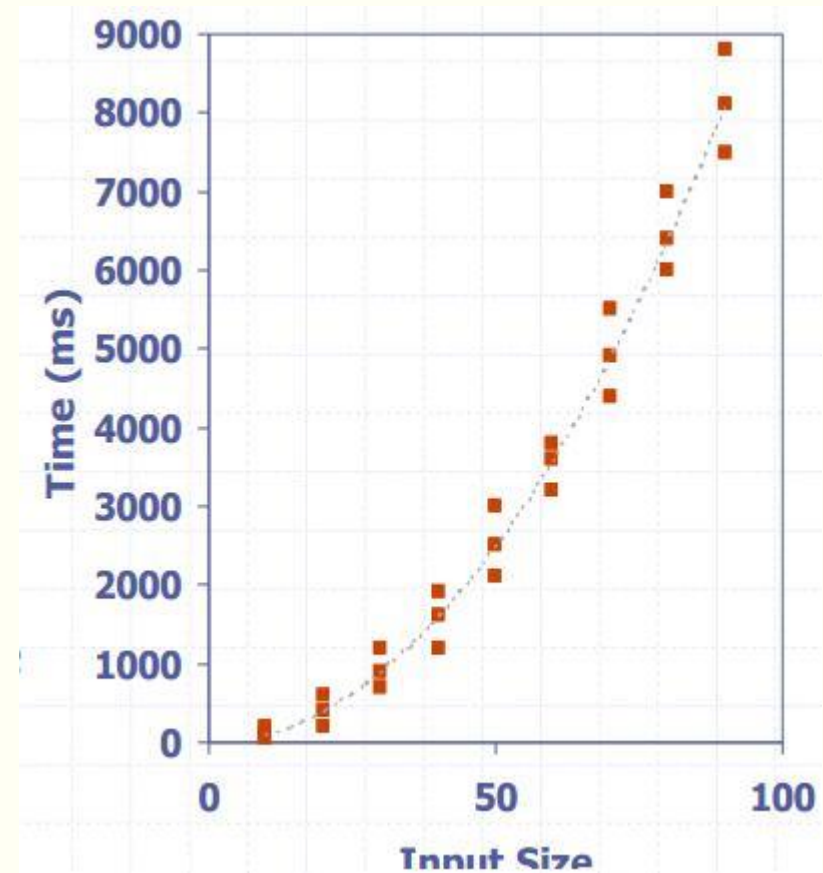
Experimental Analysis

- Use curve fitting techniques to approximate the function (e.g., least squares fitting)



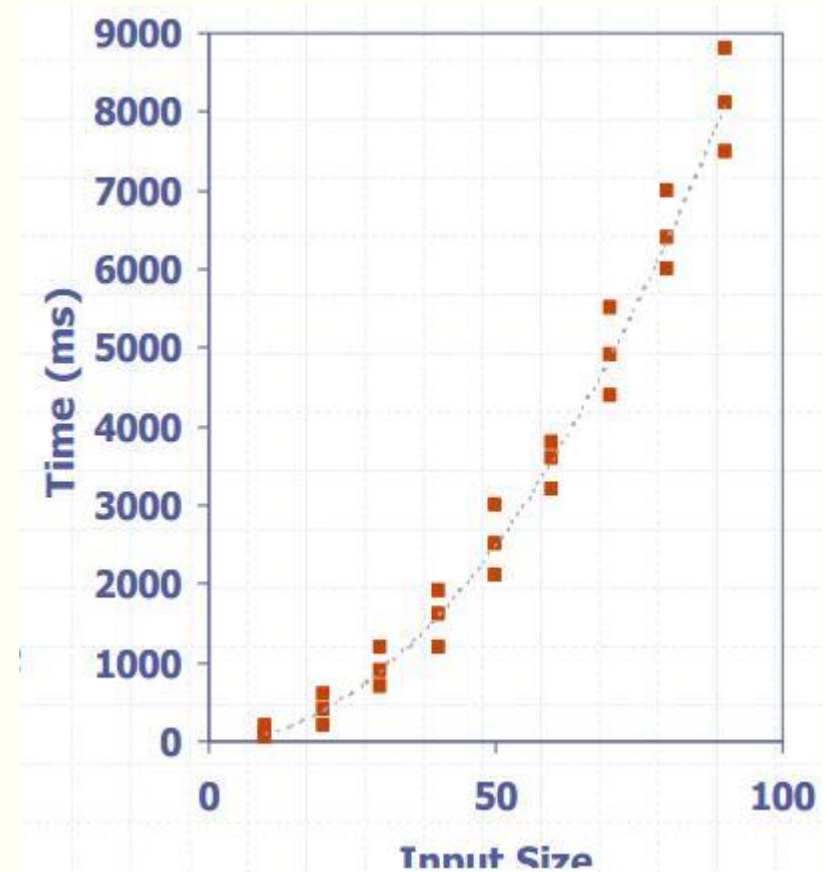
Experimental Analysis

- Use curve fitting techniques to approximate the function (e.g., least squares fitting)
- $f_1(n) = c_1 * n^2 + c_2 * (n+1)$



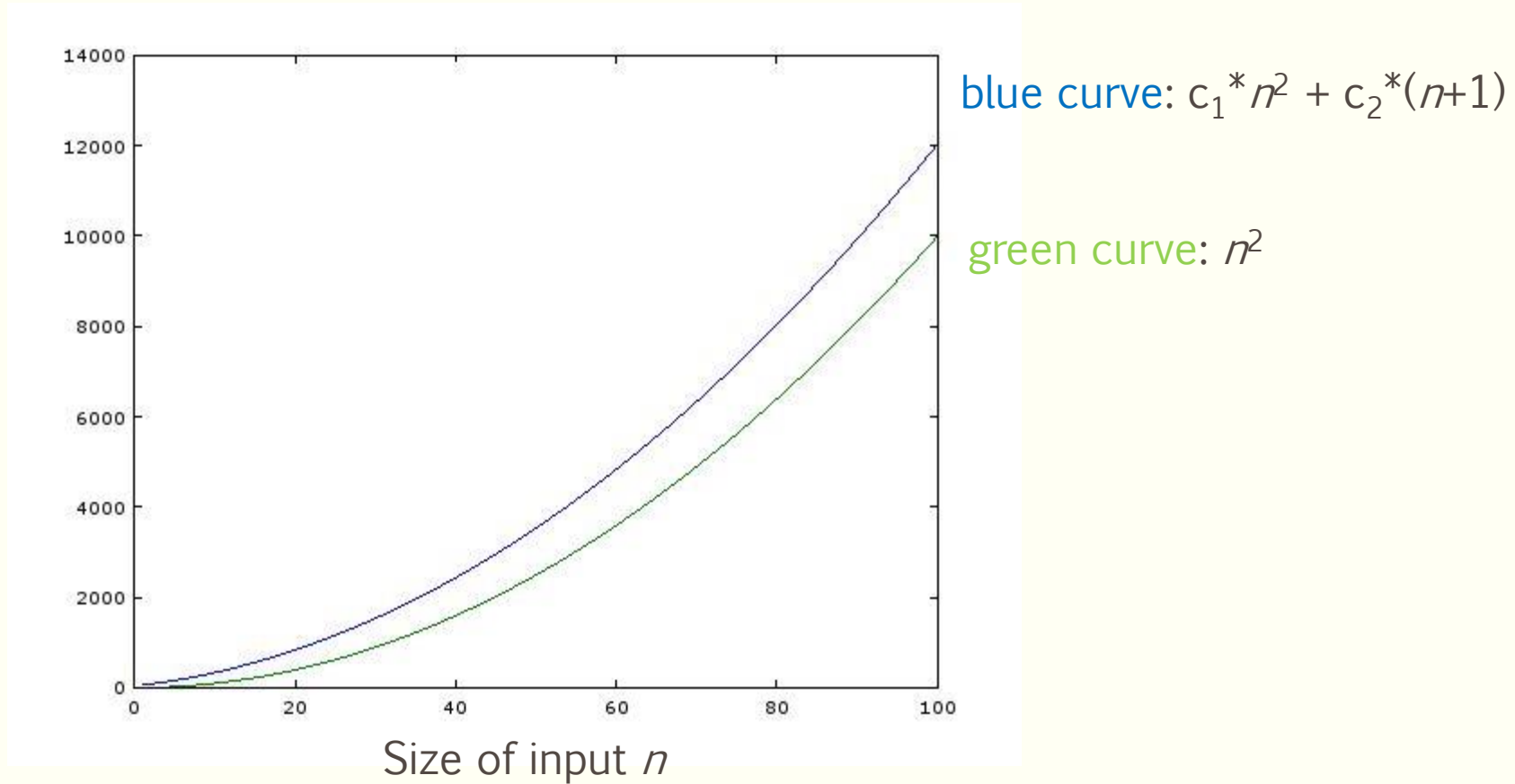
Experimental Analysis

- Use curve fitting techniques to approximate the function (e.g., least squares fitting)
- $f_1(n) = c_1 * n^2 + c_2 * (n+1)$
- How does this compare with time complexity $\approx n^2$ from the previous section?



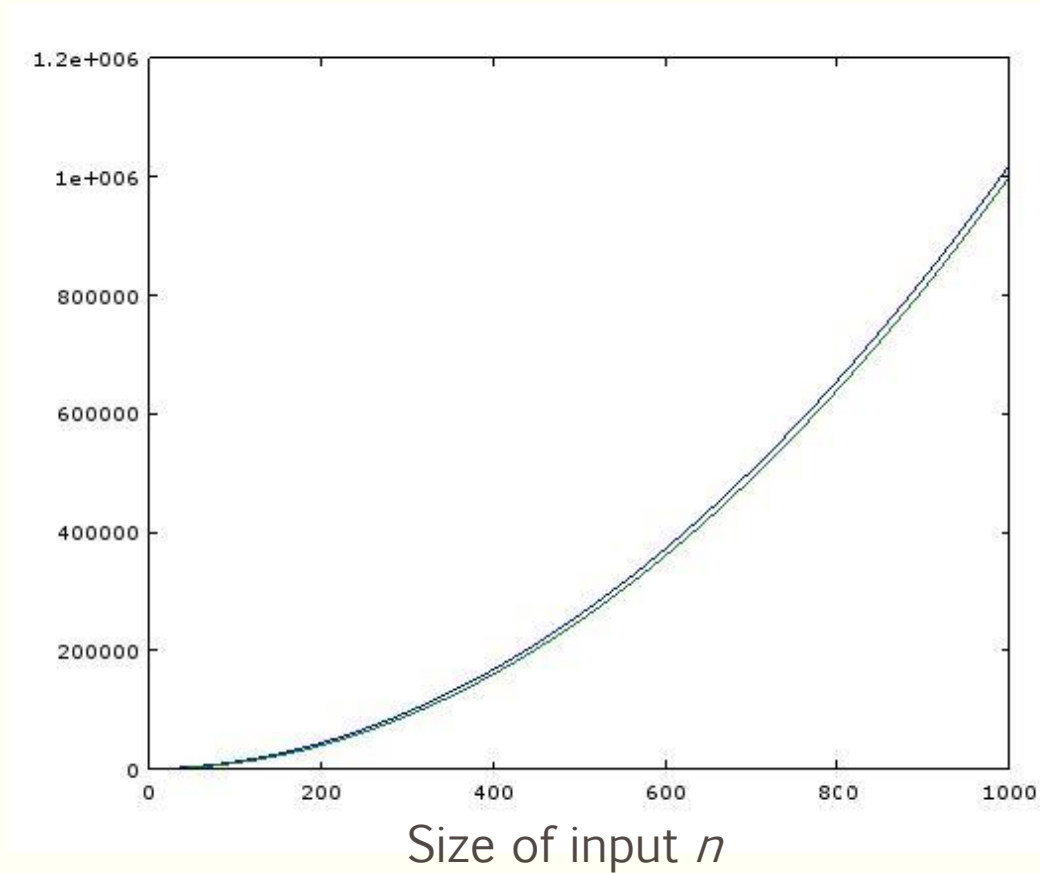
Experimental Analysis

Running time



Experimental Analysis

Running time

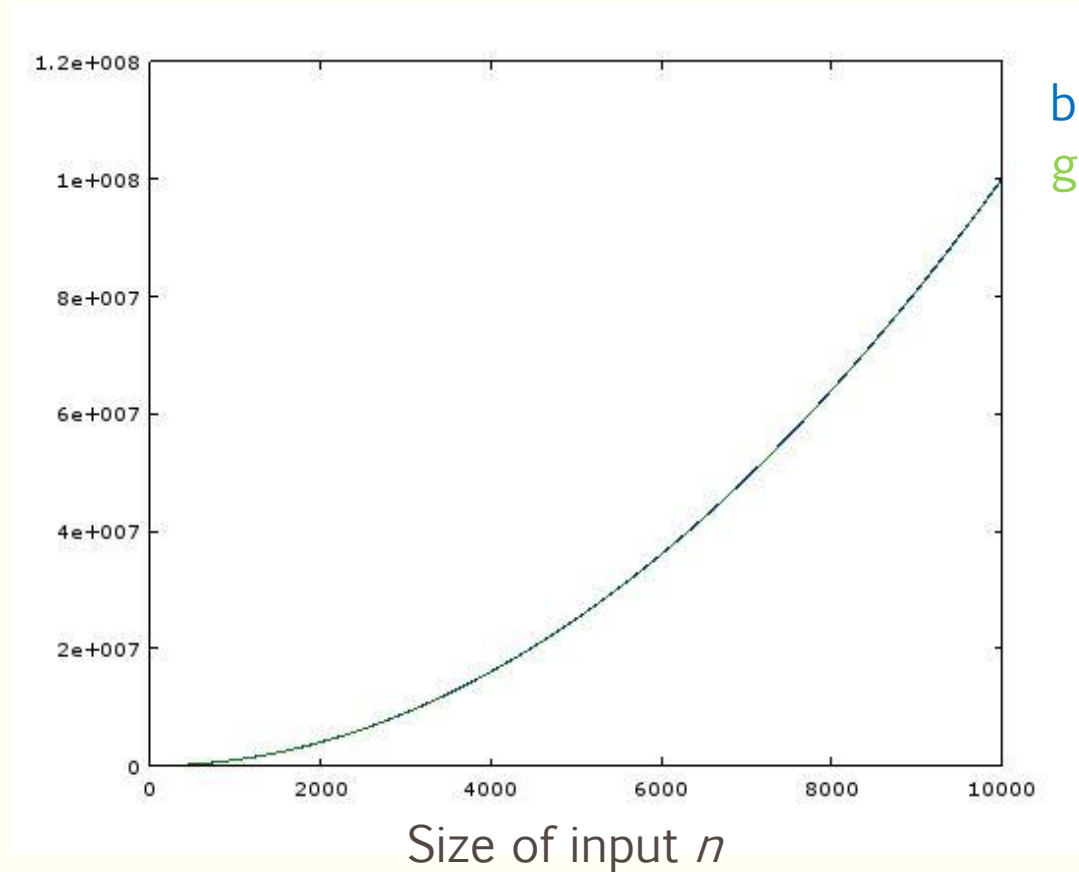


blue curve: $c_1 \cdot n^2 + c_2 \cdot (n+1)$

green curve: n^2

Experimental Analysis

Running time

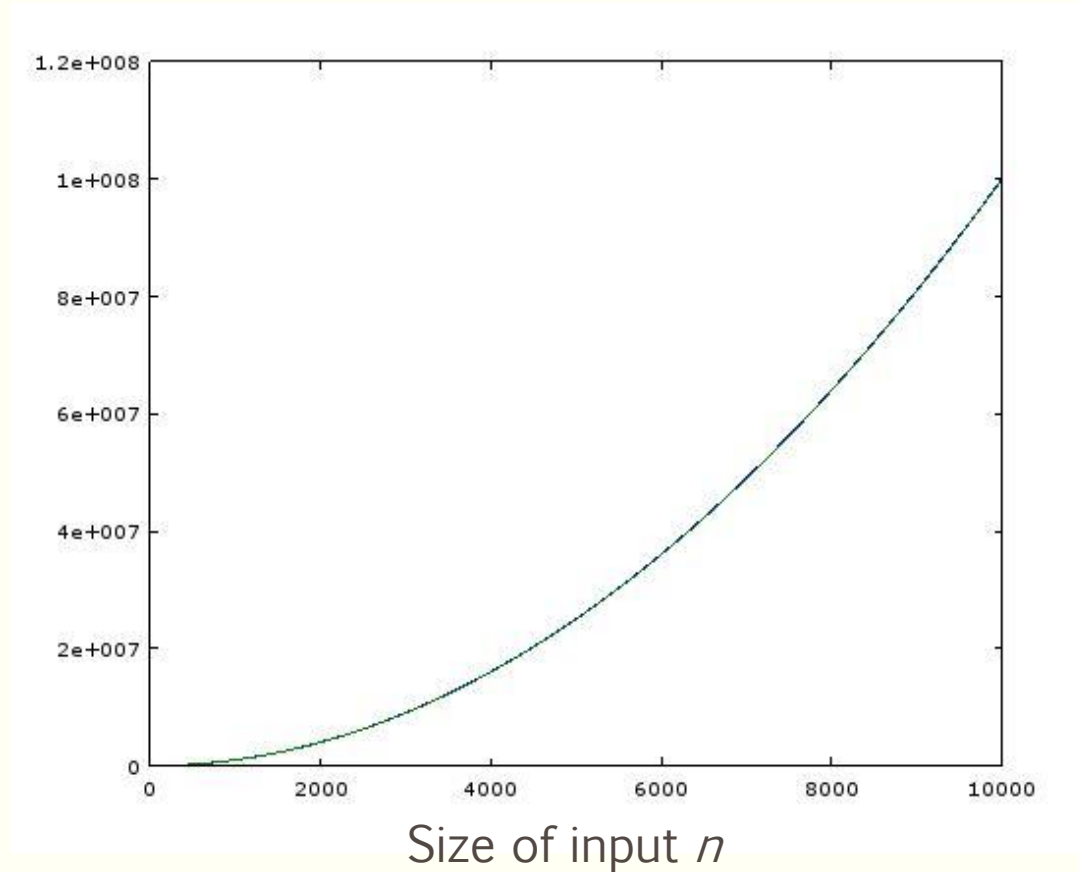


blue curve: $c_1 * n^2 + c_2 * (n+1)$

green curve: n^2

Experimental Analysis

Running time



- Plotting the functions can provide us quickly with some useful information
- As n grows very large the two functions have similar behavior
- Plotting functions is not an accurate scientific method
- Need for mathematical analysis

Experimental Analysis Conclusion

- However, experimental analysis is not always a good option.
 - For example, if it would take days or weeks to run the programs

Experimental Analysis Conclusion

- However, experimental analysis is not always a good option.
 - For example, if it would take days or weeks to run the programs
 - Collecting or generating the datasets can be challenging. Results may not be indicative of the running time on other inputs not included in the experiment.

Experimental Analysis Conclusion

- However, experimental analysis is not always a good option.
 - For example, if it would take days or weeks to run the programs
 - Collecting or generating the datasets can be challenging. Results may not be indicative of the running time on other inputs not included in the experiment.
 - Same hardware and software environments must be used
 - Implementations using different programming languages may tell us more about the difference between the languages than the difference between implementations.



MATHEMATICAL ANALYSIS

Mathematical Analysis

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
for i=1 to n:
    for j=1 to n:
        if (i != j) AND A[i] == A[j]
            return true
return false
```

Mathematical Analysis

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
for i=1 to n:
    for j=1 to n:
        if (i != j) AND A[i] == A[j]
            return true
return false
```

- Measure input size
 - find key quantity to measure the **size of the data** that determine the running time.

Mathematical Analysis

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
for i=1 to n:
    for j=1 to n:
        if (i != j) AND A[i] == A[j]
            return true
return false
```

- Measure input size
 - find key quantity to measure the **size of the data** that determine the running time.
- Sometimes obvious: length of array n
- For graph problems can be nodes and edges

Mathematical Analysis

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
for i=1 to n:
    for j=1 to n:
        if (i != j) AND A[i] == A[j]
            return true
return false
```

- Identify Algorithm's basic operation
 - Identify the important operations/instructions in the program
 - Basic operation: the operation that uses the most runtime overall (typically, most expensive operation of innermost loop)

Mathematical Analysis

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
for i=1 to n:
    for j=1 to n:
        if (i != j) AND A[i] == A[j]
            return true
return false
```

■ Identify Algorithm's basic operation

- integer comparison - c_0
- Array access - c_1
- Conditional - c_2
- Loop - c_l

Mathematical Analysis

Input: a length n array A of strings

Output: *true* if the array contains a duplicate, *false* otherwise

```
for i=1 to n:
    for j=1 to n:
        if (i != j) AND A[i] == A[j]
            return true
return false
```

- Identify Algorithm's basic operation

- Sometimes more useful to clump operations:

$$c_{op} = 2 * c_0 + c_1 + \dots$$

Mathematical Analysis

Decide on a parameter (or parameters) for measuring the size of the input



Identify the Algorithm's basic operation



Determine if best, worst, and average case will be different



Set up a sum representing the number of times the operation happens



Find a closed form formula for the expression

Mathematical Analysis

- Closed form runtime:

$$\begin{aligned}C_{\text{OUTERLOOP}} &= C_l + \sum_{i=1}^n C_{\text{INNERLOOP}} = C_l + \sum_{i=1}^n (C_l + \sum_{j=1}^n C_{op}) \\ &= C_{op} * n^2 + C_l * (n + 1)\end{aligned}$$

Mathematical Analysis

- We just proved mathematically that:

running time function $f_1(n) = C_{op} * n^2 + C_l * (n + 1)$

Mathematical Analysis

- We just proved mathematically that:

running time function $f_1(n) = C_{op} * n^2 + C_l * (n + 1)$

- Now lets prove that $f_1(n) \approx n^2$ or more formally $f_1(n) \in O(n^2)$

Motivation for Big-O Notation

- When we analyze an algorithm, our goal is to find a function $g(n)$, such that the running time of the algorithm is **proportional** to $g(n)$.

Motivation for Big-O Notation

- When we analyze an algorithm, our goal is to find a function $g(n)$, such that the running time of the algorithm is **proportional** to $g(n)$.
- Why **proportional** and not **equal**?

Motivation for Big-O Notation

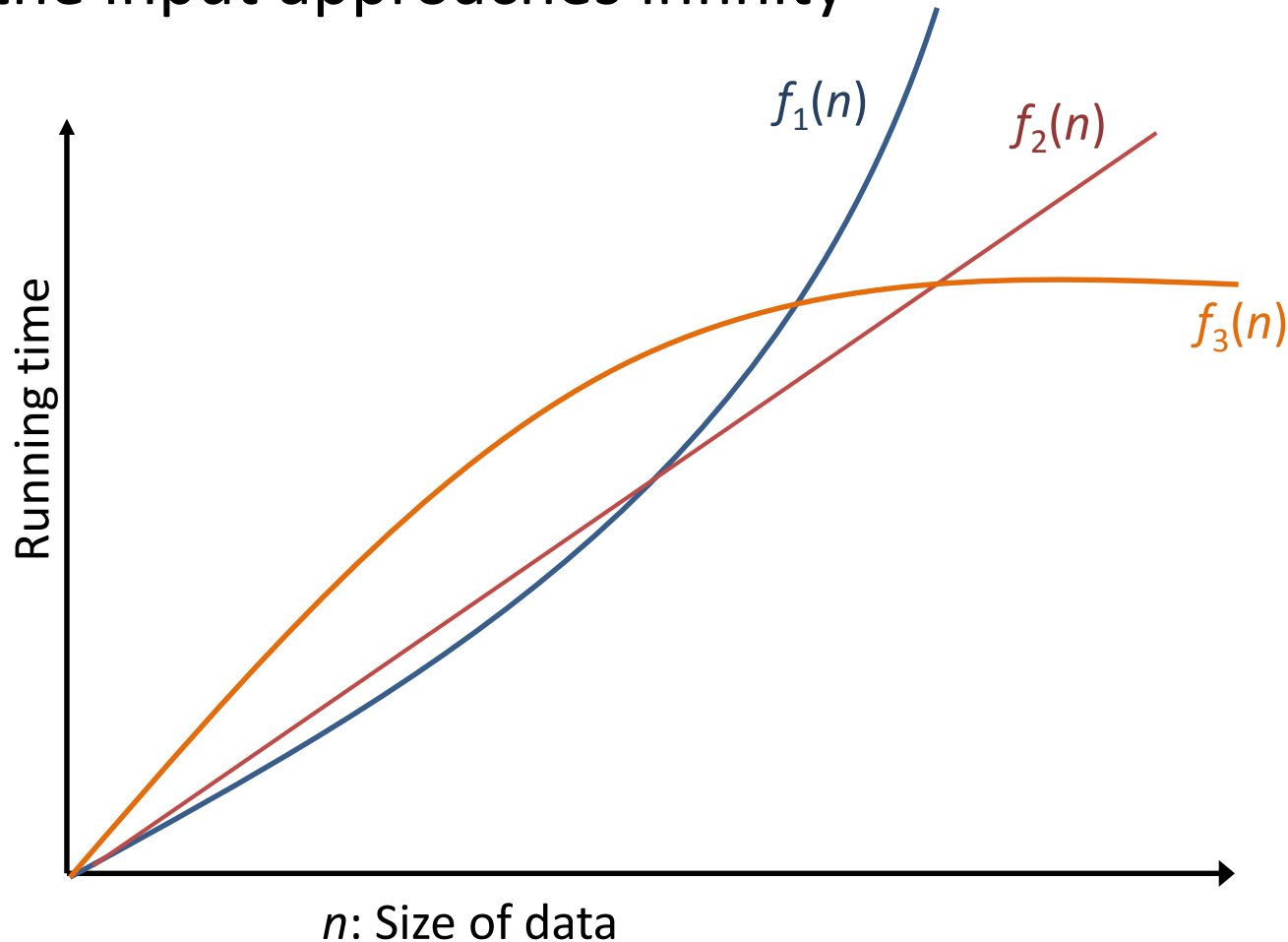
- When we analyze an algorithm, our goal is to find a function $g(n)$, such that the running time of the algorithm is **proportional** to $g(n)$.
- Why **proportional** and not **equal**?
- Because **the actual running time is not a defining characteristic of an algorithm**
 - Running time depends on programming language, actual implementation, compiler used, machine executing the code, ...

Motivation for Big-O Notation

- There are some details that we would actually NOT want $g(n)$ to include, because they can make a function unnecessarily complicated.
 - Constants, are not important!
 - Behavior fluctuations on small data.
- The Big-Oh notation, which we will see in a few slides, achieves that, and greatly simplifies algorithmic analysis.

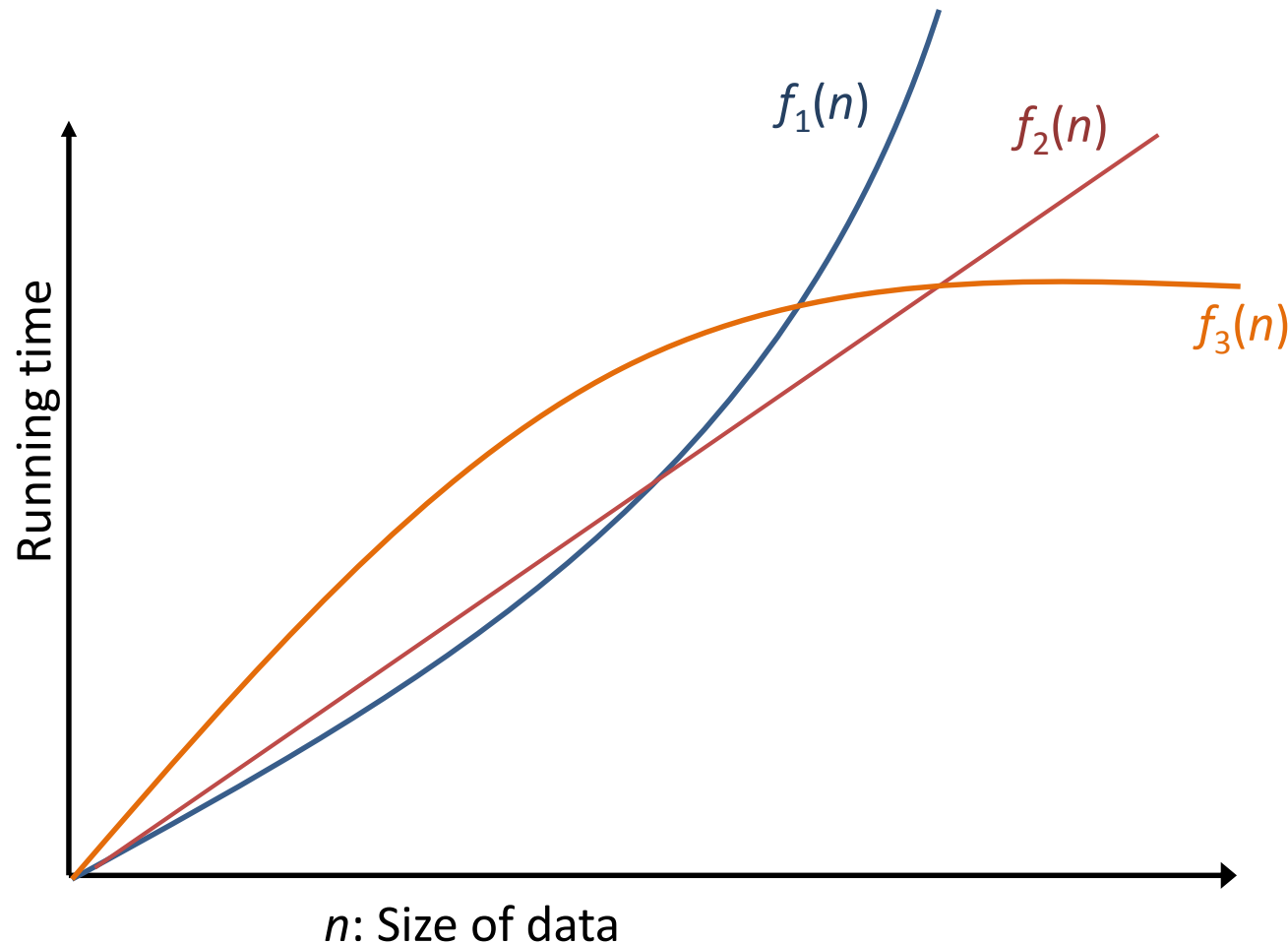
Why Asymptotic Behavior Matters

- Asymptotic behavior: The behavior of a function as the input approaches infinity



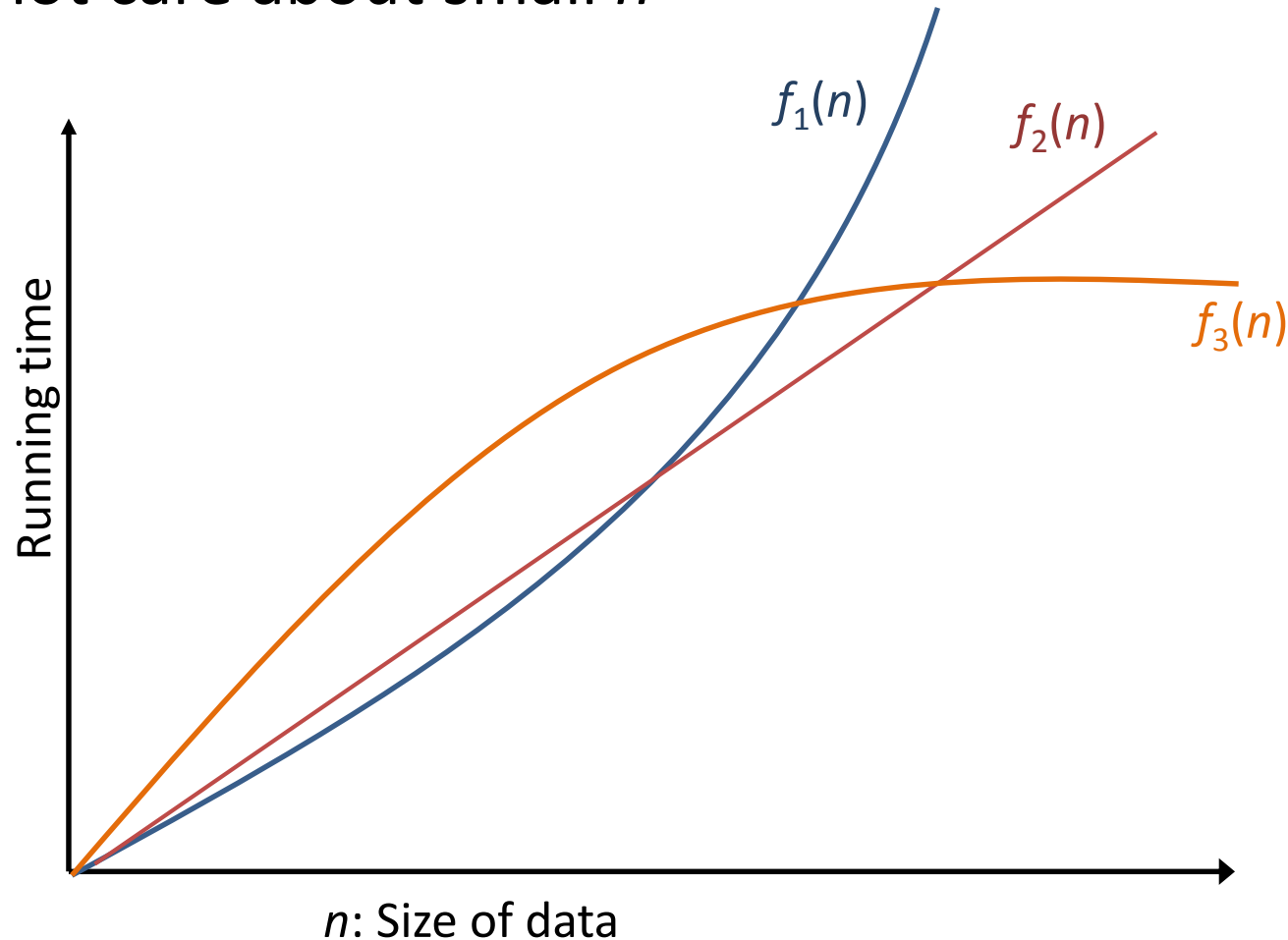
Why Asymptotic Behavior Matters

- Which of these functions works best asymptotically?



Why Asymptotic Behavior Matters

- f_3 seems to grow VERY slowly after a while. We do not care about small n



Definition of Big-O

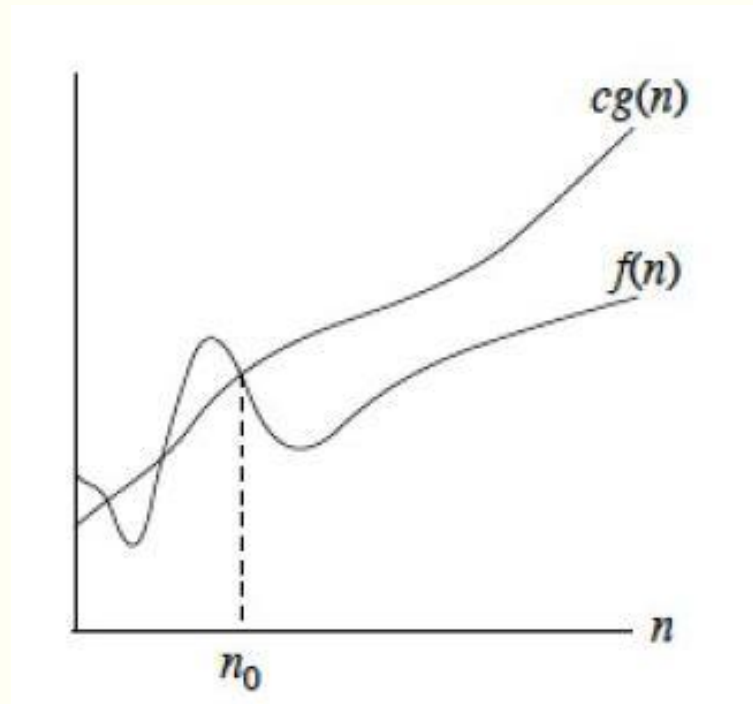
- A function $f(n)$ is said to be in $O(g(n))$ if there exist constants c and n_0 such that:

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$

Definition of Big-O

- A function $f(n)$ is said to be in $O(g(n))$ if there exist constants c and n_0 such that:

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$



Definition of Big-O

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$

- Typically, $f(n)$ is the running time of an algorithm, in your favorite units, implementation, and machine. This can be a rather complicated function.

Definition of Big-O

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$

- Typically, $f(n)$ is the running time of an algorithm, in your favorite units, implementation, and machine. This can be a rather complicated function.
- In algorithmic analysis, we try to find a $g(n)$ that is simple, and such that $f(n) \in O(g(n))$

Definition of Big-O

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$

- The Big-Oh notation greatly simplifies the analysis task, by:
 1. Ignoring constant factors. How is this achieved?
 - By the C in the definition. We are free to choose ANY constant C we want, to make the formula work.
 - Thus, Big-Oh notation is independent of programming language, compiler, machine performance, and so on...

Definition of Big-O

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$

- The Big-Oh notation greatly simplifies the analysis task, by:
 2. Ignoring behavior for small inputs. How is this achieved?
 - By the N_0 in the implementation. If a finite number of values are not compatible with the formula, just ignore them.
 - Thus, big-Oh notation focuses on asymptotic behavior.

Definition of Big-O

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$

- The Big-Oh notation greatly simplifies the analysis task, by:
 3. Allowing us to describe complex running time behaviors of complex algorithms with simple functions, such as N , $\log N$, N^2 , 2^N , and so on.
 - Such simple functions are sufficient for answering many important questions, **once you get used to Big-Oh notation.**

Is Big-Oh Notation Always Enough?

- NO! Big-Oh notation does not always tell us which of two algorithms is preferable.

Is Big-Oh Notation Always Enough?

- NO! Big-Oh notation does not always tell us which of two algorithms is preferable.
 - Example 1: if we know that the algorithm will only be applied to relatively small N , we may prefer a running time of N^2 over $N\log(N)$.
 - Example 2: even constant factors can be important. For many applications, we strongly prefer a running time of $3N$ over $1500N$.

Is Big-Oh Notation Always Enough?

- NO! Big-Oh notation does not always tell us which of two algorithms is preferable.
 - Example 1: if we know that the algorithm will only be applied to relatively small N , we may prefer a running time of N^2 over $N\log(N)$.
 - Example 2: even constant factors can be important. For many applications, we strongly prefer a running time of $3N$ over $1500N$.
- Big-Oh notation is not meant to tell us everything about running time.
- But, Big-Oh notation tells us a lot, and is often much easier to compute than actual running times.

Using Big-O

- We have mathematically proven that:

$$f_1(n) = C_{op} * n^2 + C_l * (n + 1)$$

- How we prove that $f_1(n) \in O(n^2)$?

Using Big-O

- We have mathematically proven that:

$$f_1(n) = C_{op} * n^2 + C_l * (n + 1)$$

- How we prove that $f_1(n) \in O(n^2)$?
- This is where the Big-Oh definition comes into play. We can find an n_0 such that, for all $n > n_0$:

$$f_1(n) \leq (C_{op} + 1) * n^2$$

- If you don't believe this, do the calculations for practice

Using Big-O

- Another way to show correctness: as n goes to infinity, what is the limit of $f(n) / n^2$?
 - $C_{\text{op}} > 0$
 - This shows that the non-quadratic terms become negligible as n gets larger

Using Big-O

- Another way to show correctness: as n goes to infinity, what is the limit of $f(n) / n^2$?
 - $C_{op} > 0$
 - This shows that the non-quadratic terms become negligible as n gets larger

Let $f(n)$ and $g(n)$ be two real function then, $f(n) = O(g(n))$ is equivalent to

$$\exists c \in \mathbb{R} : \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$$

where, $c \geq 0$

(Here c is finite or **can be zero**)

Properties of Big Oh:

- For all constants $c > 0$ and $a > 1$, and monotonically increasing functions $f(n)$:

$$f(n)^c \text{ is } O(a^{f(n)})$$

- For example:
 - n^c is $O(a^n)$
 - n^{256} is $O(1.0001^n)$
 - $(\log_a n)^c$ is $O(a^{\log_a n})$, which is $O(n)$.

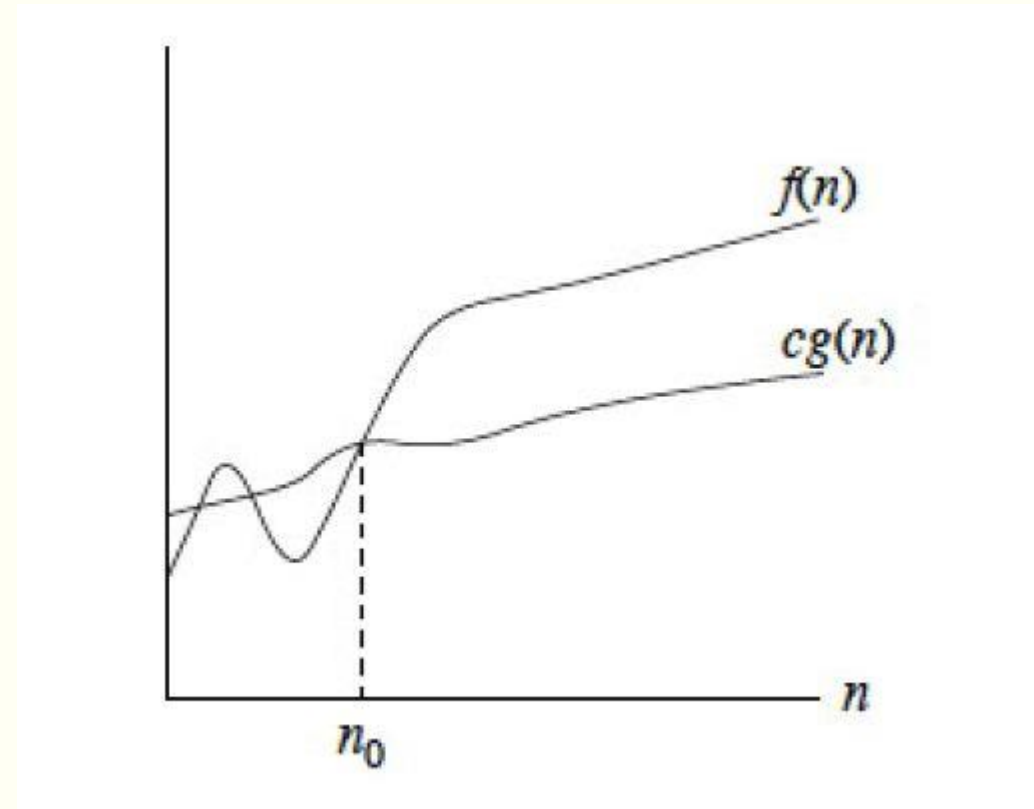
More Properties of Big Oh:

- O notation is additive (rule of Sums)
If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then:
 - $f_1(n) + f_2(n)$ is $O(\max\{g_1(n), g_2(n)\})$
- O notation is multiplicative (rule of Products)
If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then:
 - $f_1(n) * f_2(n)$ is $O(g_1(n) * g_2(n))$
- O notation is transitive
If $f(n)$ is $O(g(n))$, and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

Omega (Ω) Notation

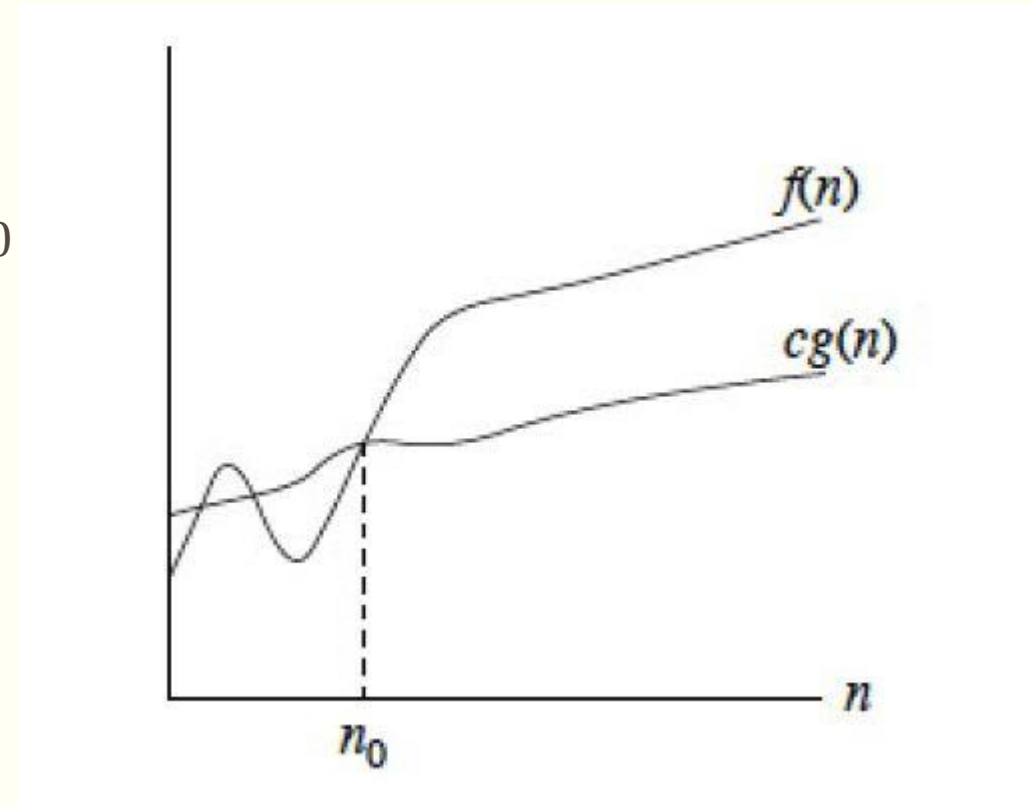
$$f(n) \in \Omega(n)$$

If $f(n)$ is bounded below
by some constant
multiple of $g(n)$
for all large n



Omega example

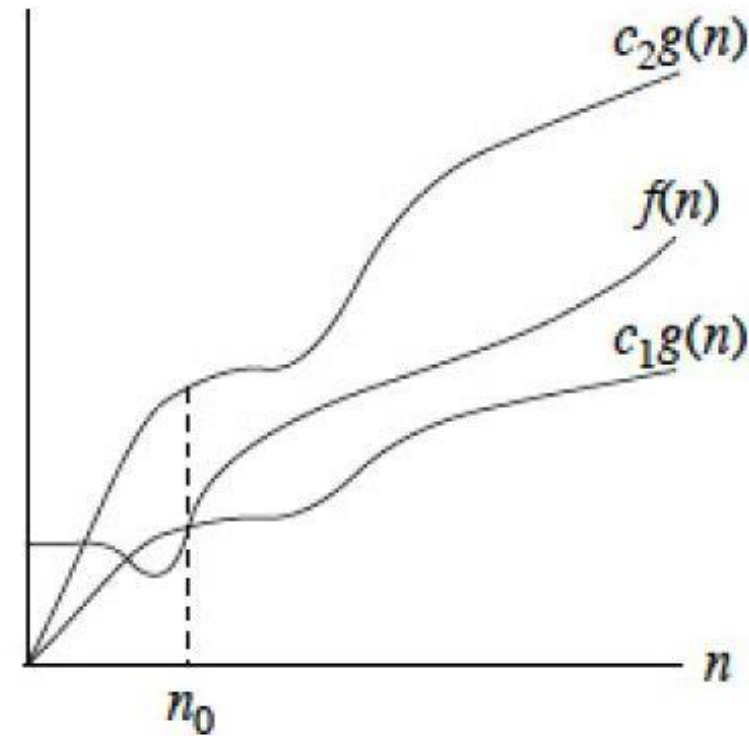
- $n*(n-1)/2 \in \Omega(n^2)$
- $f(n) \geq c * g(n)$, for all $n \geq n_0$



Theta (Θ) Notation

$$f(n) \in \Theta(n)$$

If $f(n)$ is bounded both above and below by some constant multiple of $g(n)$ for all large n

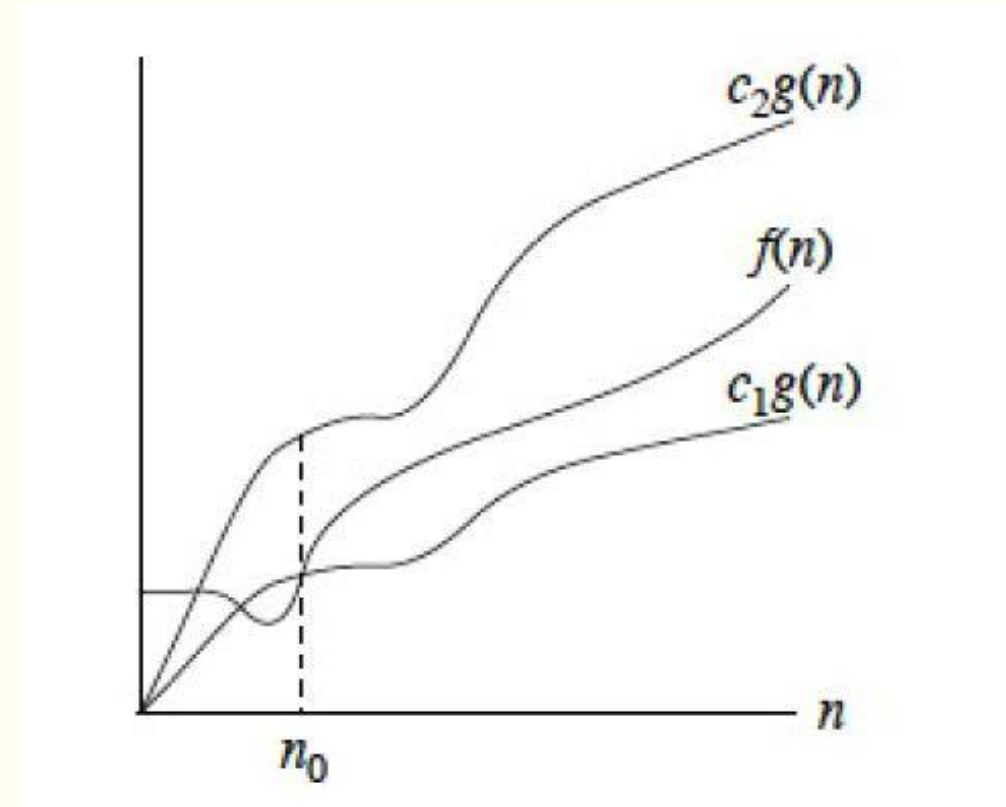


Omega (Ω) and Theta (Θ) Notations

- If $f(n) \in O(g(n))$, then we also say that $g(n) \in \Omega(f(n))$.
- If $f(n) \in O(g(n))$ **and** $f(n) \in \Omega(g(n))$, then we say that $f(n) \in \Theta(g(n))$.
- The Theta notation is clearly stricter than the Big-Oh notation:
 - We can say that $n^2 \in O(n^{100})$.
 - We **cannot** say that $n^2 \in \Theta(n^{100})$.
 - We can say that $2^{n+1} \in \Theta(2^n)$

Theta example

- $2^{n+1} \in \Theta(2^n)$
 - $f(n) \geq c_1 * g(n)$, for all $n \geq n_0$
- AND**
- $f(n) \leq c_2 * g(n)$, for all $n \geq n_0$



Using Limits for Comparing Order of Growth

- $\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = 0$, implies that $t(n)$ has a smaller order of growth than $g(n)$
- $\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = c, c > 0$, implies that $t(n)$ has the same order of growth as $g(n)$
- $\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \infty$, implies that $t(n)$ has a larger order of growth than $g(n)$
- Note that the first two cases mean that: $t(n) \in O(g(n))$
- Note that the last two cases mean that: $t(n) \in \Omega(g(n))$
- Note that the second case means that: $t(n) \in \Theta(g(n))$

Using Limits - Comments

- The previous formulas relating limits to big-Oh notation show once again that big-Oh notation ignores:
 - constants
 - behavior for small values of N .
- How do we see that?
 - In the previous formulas, it is sufficient that the limit is equal to a constant. **The value of the constant does not matter.**
 - In the previous formulas, only **the limit at infinity** matters. This means that we can ignore behavior up to any finite value, if we need to.