CS302 - Data Structures
using C++

Topic: Algorithm Efficiency

Kostas Alexis

What is a Good Solution?

- A program incurs a real and tangible cost
    - Computing time
    - Memory required
    - Difficulties encountered by users
    - Consequences of incorrect actions by the program

What is a Good Solution?

- A program incurs a real and tangible cost
  - Computing time
  - Memory required
  - Difficulties encountered by users
  - Consequences of incorrect actions by the program
- A solution is good if …
  - The total cost incurs …
  - Overall all phases of its life … is minimal

What is a Good Solution?

- Important elements of the solution
  - Good structure
  - Good documentation
  - Efficiency

What is a Good Solution?

- Important elements of the solution
  - Good structure
  - Good documentation
  - Efficiency
- Be concerned with efficiency when
  - Developing underlying algorithm
  - Choice of objects and design of interaction between these objects

Measuring Efficiency of Algorithms

- Important because
  - Choice of algorithm has significant impact
- Examples
  - Responsive word processors
  - Internet search engines
  - Real-time guidance systems
  - Autonomous cars

Measuring Efficiency of Algorithms

- Analysis of algorithms
    - The area of computer science that provides tools for contrasting efficiency of different algorithms
    - Comparison of algorithms should focus on significant differences in efficiency
    - We consider comparisons of **algorithms**, not programs

Measuring Efficiency of Algorithms

- Difficulties with comparing programs (instead of algorithms)
    - How are the algorithms coded
    - What computer will be used
    - What data should the program use
- Algorithms analysis should be independent of
    - Specific implementations, computers, and data

Measuring Efficiency of Algorithms

- Even a simple program can be "inefficient"
- What is an efficient algorithm?
  - Algorithms take time to execute
  - Algorithms need storage for data and variables
- Complexity
  - Time and storage requirements of an algorithm
- Analysis of algorithms
  - Measuring of the complexity of an algorithm
- Types of complexity
  - Time complexity
    - Speed (number of operations)
  - Space complexity
    - Storage (memory or disk)
  - Inverse relationship
    - Faster algorithms can require more space
    - Reducing storage can increase execution time

Measuring Complexity

- Measuring Complexity
  - Express complexity in **problem size**
  - Number of items processed by algorithm
  - Usually represented by **n**
- Cannot compute actual time for an algorithm
  - Compute **growth-rate function**
    - Simple function that is directly proportional to the algorithms time requirement
  - Gives common basis for comparison

Measuring Complexity

- Measuring Complexity
    - Express complexity in **problem size**
    - Number of items processed by algorithm
    - Usually represented by **n**
- Cannot compute actual time for an algorithm
    - Compute **growth-rate function**
        - Simple function that is directly proportional to the algorithms time requirement
    - Gives common basis for comparison

- Comparing algorithms should be independent of
    - Specific Implementation
        - How are the algorithms coded
    - Computer
        - What computer is used
    - Data
        - The data should the program uses

Growth-Rate Functions

- Find the sum of the first **n** positive integers

```
1 + 2 + 3 + … + n-1 + n
for an integer n > 0
```

Growth-Rate Functions

- Find the sum of the first **n** positive integers

```
1 + 2 + 3 + … + n-1 + n
 for an integer n > 0
```

Algorithm A

```
sum = 0
for i = 1 to n
   sum = sum + I
```

Growth-Rate Functions

- Find the sum of the first **n** positive integers

```
1 + 2 + 3 + … + n-1 + n
for an integer n > 0
```

Algorithm A

```
sum = 0
for i = 1 to n
   sum = sum + I
```

Algorithm B

```
sum = 0
for i = 1 to n
   for j = 1 to i
      sum = sum + 1
```

Growth-Rate Functions

- Find the sum of the first **n** positive integers

```
1 + 2 + 3 + … + n-1 + n
for an integer n > 0
```

Algorithm A

```
sum = 0
for i = 1 to n
   sum = sum + I
```

Algorithm B

```
sum = 0
for i = 1 to n
   for j = 1 to i
      sum = sum + 1
```

Algorithm C

```
sum = n*(n+1)/2
```

Growth-Rate Functions

- Find the sum of the first **n** positive integers

```
1 + 2 + 3 + … + n-1 + n
for an integer n > 0
```

Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + I
```

Algorithm B

```
sum = 0
for i = 1 to n
    for j = 1 to i
        sum = sum + 1
```

Algorithm C

```
sum = n*(n+1)/2
```

Algorithm B takes noticeably longer

Growth-Rate Functions

- Find the sum of the first **n** positive integers
- Finding the growth-rate function
    - **Count basic operations**

```
1 + 2 + 3 + … + n-1 + n
for an integer n > 0
```

Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + I
```

Algorithm B

```
sum = 0
for i = 1 to n
    for j = 1 to i
        sum = sum + 1
```

Algorithm C

```
sum = n*(n+1)/2
```

Algorithm B takes noticeably longer

Growth-Rate Functions

- Find the sum of the first **n** positive integers
- Finding the growth-rate function
    - **Count basic operations**

```
1 + 2 + 3 + … + n-1 + n
for an integer n > 0
```

Algorithm A

```
sum = 0
for i = 1 to n
   sum = sum + I
```

Algorithm B

```
sum = 0
for i = 1 to n
   for j = 1 to i
      sum = sum + 1
```

Algorithm C

```
sum = n*(n+1)/2
```

Algorithm B takes noticeably longer

Count “action” statements

- Statements directly related to accomplishing goal – Additions, Multiplications, Comparisons, Moves
- Ignore “bookkeeping” statements

Growth-Rate Functions

```
1 + 2 + 3 + … + n-1 + n
  for an integer n > 0
```

- Find the sum of the first **n** positive integers
- Finding the growth-rate function
  - **Count basic operations**

Algorithm A

```
sum = 0
for i = 1 to n
   sum = sum + I
```

Algorithm B

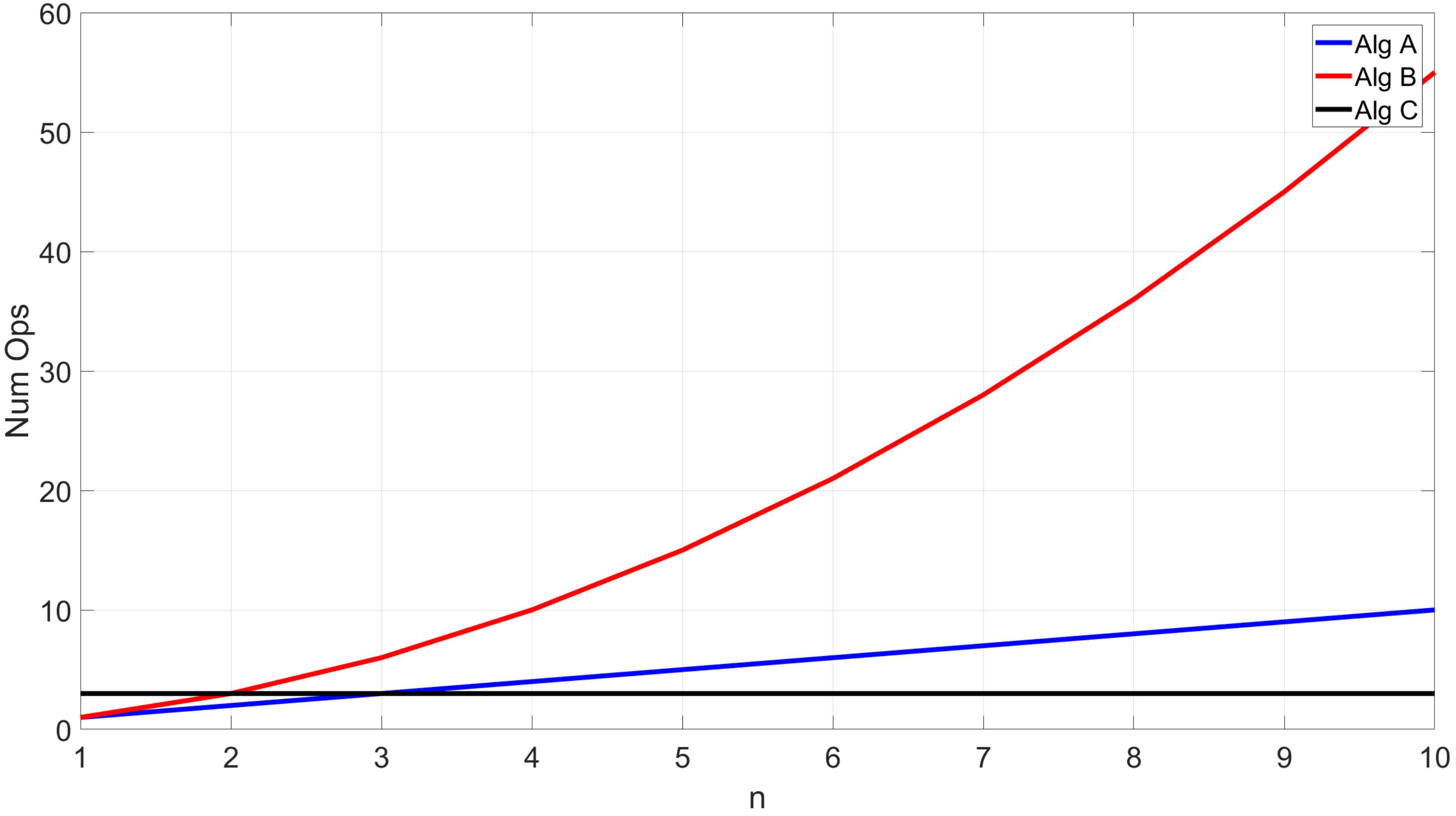
```
sum = 0
for i = 1 to n
   for j = 1 to i
      sum = sum + 1
```

Algorithm C

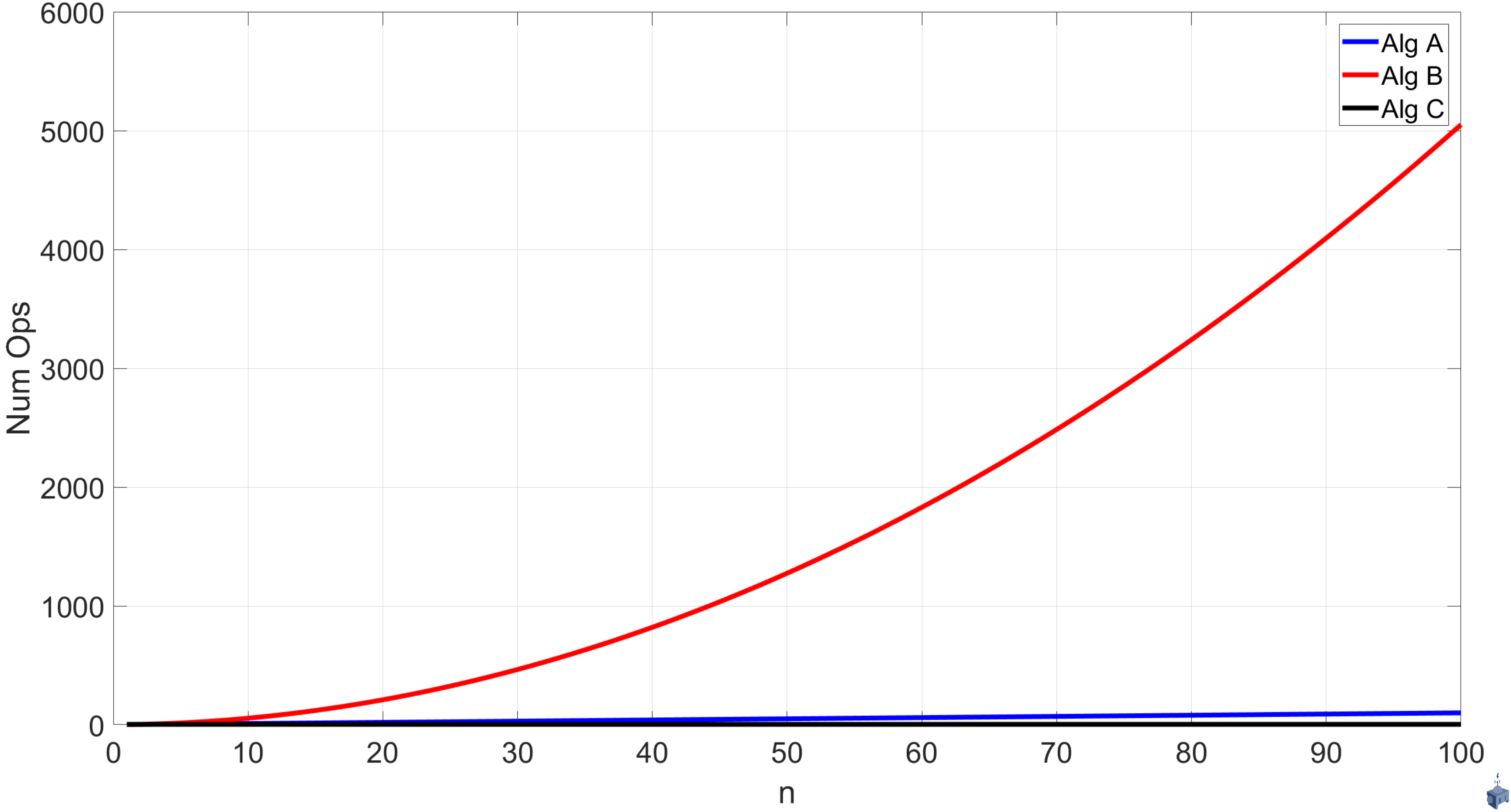
```
sum = n*(n+1)/2
```

	Algorithm A	Algorithm B	Algorithm C
Additions	**n**	**n(n+1)/2**	**1**
Multiplications	**0**	**0**	**1**
Divisions	**0**	**0**	**1**
Total Operations	**n**	**$(n^2+n)/2$**	**3**

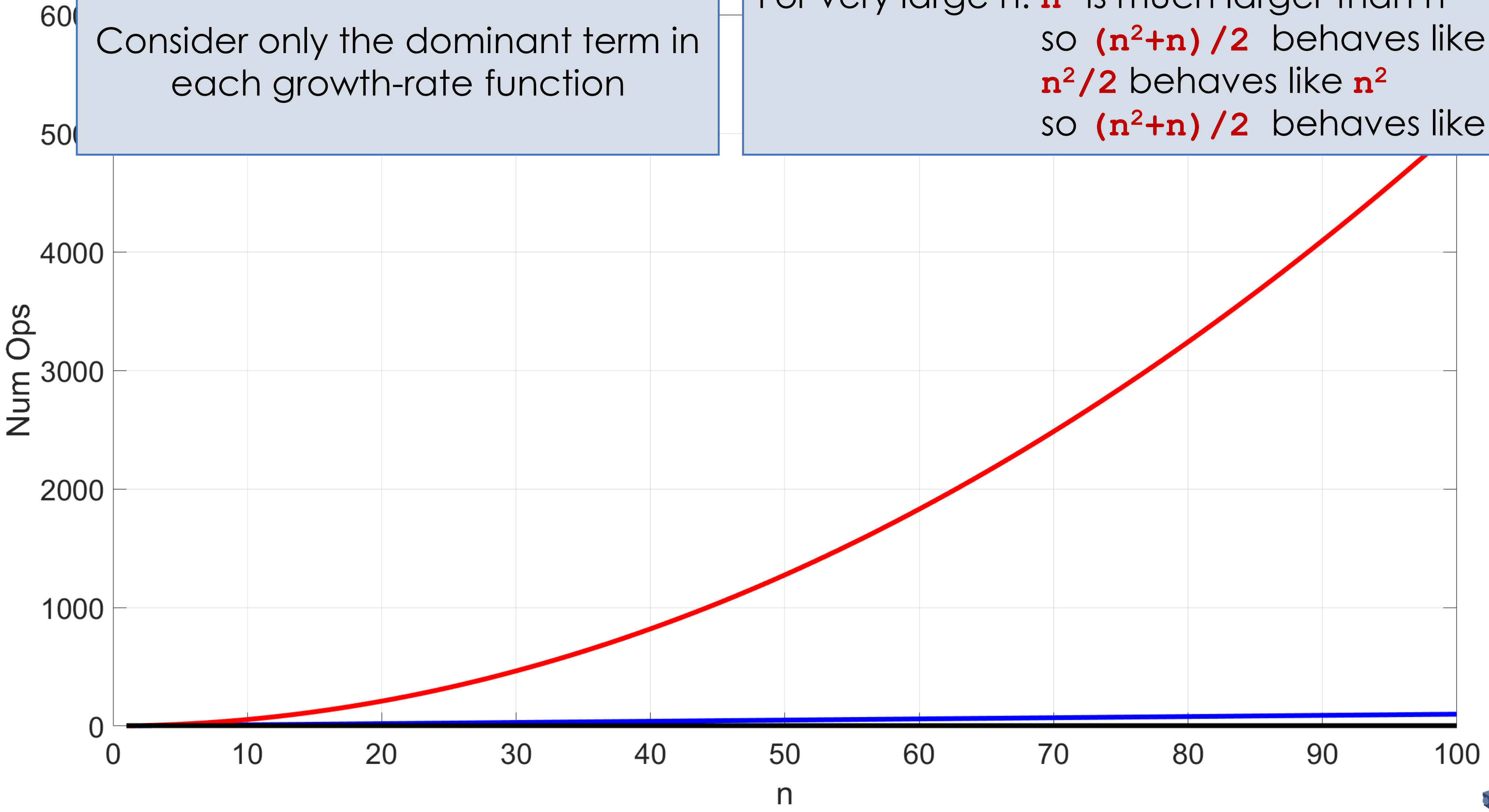
Growth-Rate Functions



Growth-Rate Functions



Growth-Rate Functions



Consider only the dominant term in each growth-rate function

For very large n: n^2 is much larger than n
so $(n^2+n)/2$ behaves like $n^2/2$
$n^2/2$ behaves like n^2
so $(n^2+n)/2$ behaves like n^2

Growth-Rate Functions

- Number of operations required as a function of **n**

n	log(logn)	logn	$\log_2 n$	n	nlogn	n^2	n^3	2^n	n!
10	2	3	11	10	33	10^3	10^2	1024	3528800
10^2	3	7	44	100	664	10^6	10^4	$1.2677*10^{30}$	$9.33*10^{157}$
10^3	3	10	99	1000	9966	10^9	10^6	$10.71*10^{300}$	*
10^4	4	13	177	10000	132877	10^{12}	10^8	*	*
10^5	4	17	276	1000000	1600964	10^{16}	10^{10}	*	*
10^6	4	20	397	1000000	19931569	10^{18}	10^{12}	*	*

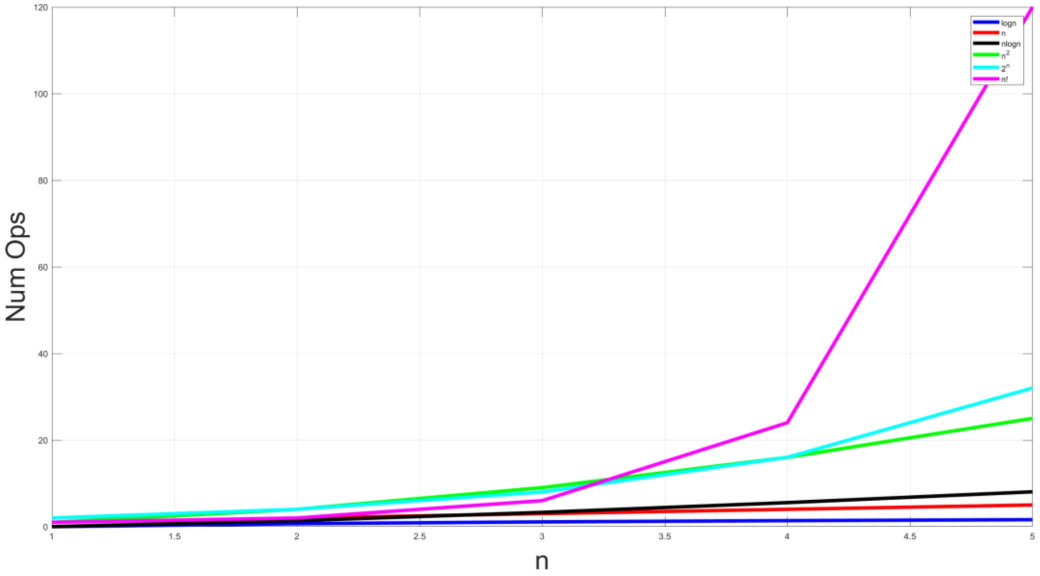
Growth-Rate Functions

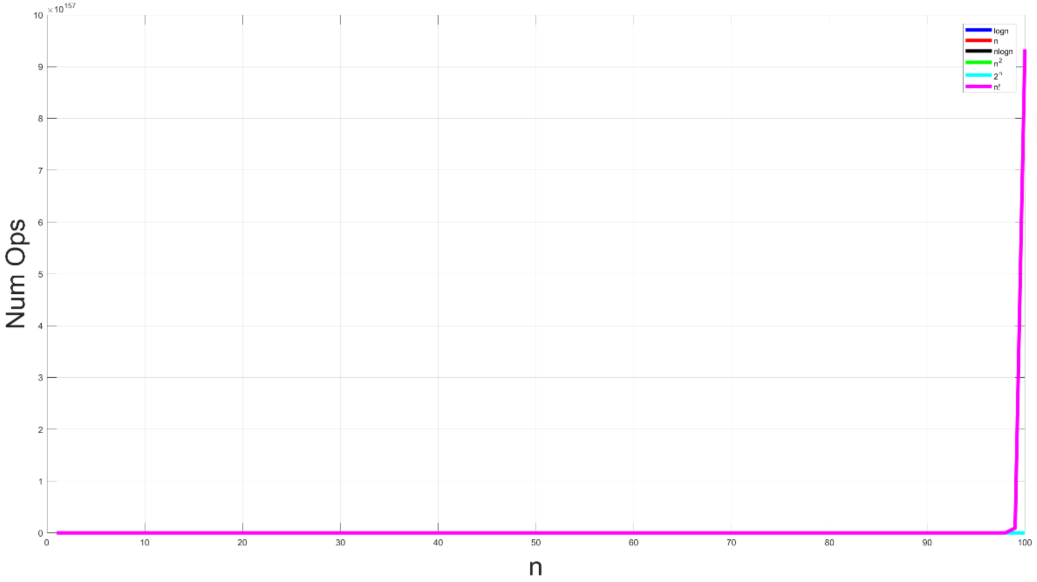
- Number of operations required as a function of **`n`**

n	**log(logn)**	**logn**	**log$_2$n**	**n**	**nlogn**	**n^2**	**n^3**	**2^n**	**n!**
10	2	3	11	10	33	10^{3}	10^{2}	1024	3528800
10^{2}	3	7	44	100	664	10^{6}	10^{4}	1.2677*10$_{3}$ 0	$9.33*10^{157}$
10^{3}	3	10	99	1000	9966	10^{9}	10^{6}	$10.71*10^{300}$	*
10^{4}	4	13	177	10000	132877	10^{12}	10^{8}	*	*
10^{5}	4	17	276	1000000	1600964	10^{16}	10^{10}	*	*
10^{6}	4	20	397	1000000	19931569	10^{18}	10^{12}	*	*

Growth-Rate Functions

- Number of operations required as a function of **`n`**





Growth-Rate Functions

- Representing an Algorithm's complexity
  - Big O notation
  - Order of at most n
  - Order of at most n^2
  - Order of at most 1

Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + I
```

Algorithm B

```
sum = 0
for i = 1 to n
    for j = 1 to i
        sum = sum + 1
```

Algorithm C

```
sum = n*(n+1)/2
```

	Algorithm A	Algorithm B	Algorithm C
Total Operations	**n**	**$(n^2+n)/2$**	**3**
Big O Notation	**O(n)**	**$O(n^2)$**	**O(1)**

Growth-Rate Functions

- Effect of **doubling** the problem size on an algorithm's time requirement

Growth-Rate Function for Size n Problems	Growth-Rate Function for Size 2n Problems	Effect on Time Requirement
1	1	None
logn	1 + logn	Negligible
n	2n	Doubles
nlong	2n logn + 2n	Doubles then add 2n
n^2	$(2n)^2$	Quadruples
n^3	$(2n)^3$	Multiplies by 8
^{2n}	2^{2n}	Squares

Analysis and Big O Notation

- Algorithm A is said to be order f(n)
  - Denoted as O(f(n))
  - Function f(n), called algorithm’s growth rate function
  - Notation with capital O denotes order
- Algorithm A of order denoted O(f(n))
  - Constants k and n_0 exist such that
  - A requires no more than $kf(n)$ time units
  - For problem of size $n \geq n_0$

Analysis and Big O Notation

- Order of growth of some common functions

$$O(1) < O(log_2 n) < O(n) < Onlog_2 n) < O(n^2) < O(n^3) < O(2^n)$$

Analysis and Big O Notation

- Worst-case analysis
  - Worst case analysis usually considered
  - Easier to calculate, thus more common
- Average-case analysis
  - More difficult to perform
  - Must determine relative probabilities of encountering problems of a given size

Keeping your Perspective

- ADT used makes a difference
  - Array-based getEntry is O(1)
  - Link-based getEntry is O(n)
- Choosing implementation of ADT
  - Consider how frequently certain operations will occur
  - Seldom used but critical operations must also be efficient

Analysis and Big O Notation

- If problem size is always small
    - Possible to ignore algorithm's efficiency
- Weight trade-offs between
    - Algorithm's time and memory requirements
- Compare algorithms for style and efficiency

Note: Efficiency of Searching Algorithms

- Sequential search
  - Worst case: O(n)
  - Average case: O(n)
  - Best case: O(1)
- Binary search
  - Worst case: $O(\log_2 n)$
  - At the same time, maintaining array in sorted order requires overhead cost which can be substantial

Thank you