CS111
Introduction to Computer Science

Fall 2015

- Efficiency of algorithms
- Searching and array
  - Sequential or Linear Search
  - Binary Search
- Asymptotic Complexity

How To Compare Algorithms?

- There are a plethora of algorithms to perform the same task. Which one to choose?
  – The one that runs faster?
  – The one that requires less space?

- How to know which algorithm performs better, meaning more efficient?

Efficiency of Algorithms

- Efficiency is a measure of speed and space consumption
- Speed – time complexity
  - also called running or execution time
  - denoted by the letter *O* (big oh)
- Space consumption – space complexity
  - memory usage
  - also represented using big *O*
  - always *less than or equal to* the time requirement

Efficiency of Algorithms

- How to measure?
  - time depends on computer architecture, language, compiler, programmer
- To measure running time count the number of operations that are most often executed
- But the time taken by an algorithm grows with the size of the input (e.g. insert the last element in an array)
  - use the <u>input size</u> to describe the running time

Efficiency of Algorithms

- Input size depends on the problem being studied
  - for searching in an array of size *n*
  - in a graph of *v* vertices and *e* edges

- Running time of an algorithm on a particular input is the number of primitive operations executed
  - for searching, the primitive operation is *compare*

Efficiency of Algorithms

1. Identify the basic operations in an algorithm
2. Determine the running time of an algorithm by counting its basic operations
3. Express the running time of an algorithm as a function of the input size
4. Derive the big *O* order of the running time

Sequential Search

- Searching an array (ordered or unordered)
  - How to find a target value?
    - check each element in sequence

	3	5	12	2	56	32	8	14		
Array index	0	1	2	3	4	5	6	7	8	9

```
void SequencialSearch (int[] array, int n, int target) {
    for (int i = 0; i < n; i++) {
        if (array[i] == target) {
            System.out.println(target + “ found”);
            return;
    }
    System.out.println(target + “ not found”);
}
```

Done *n* times

Basic Operation

Running time
f(n) = n

Sequential Search: Efficiency Analysis

- Check each element in sequence to find the target

	3	5	12	2	56	32	8	14		
Array index	0	1	2	3	4	5	6	7	8	9

- Scenarios
  - Best case
    - which number is fastest to find (requires the least number of comparisons)?
  - Worst case
    - which number is the longest to find (requires the largest number of comparisons)?
  - Average case
    - average all possibilities (takes into account the probability of a possibility)

Sequential Search: Efficiency Analysis

- Check each element in sequence to find the target

3	5	12	2	56	32	8	14			
0	1	2	3	4	5	6	7	8	9	Array index

- Best case
  - target = 3 (first array element)
  - always 1 comparison, no matter how big is *n*
- Worst case
  - target = 14 (last array element)
  - 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8
  - always takes n comparisons

Input size (n)	Number of Basic Operations	
	Best	Worst
8	1	8
100000	1	100000

Running time
f(n) = n

Asymptotic Complexity

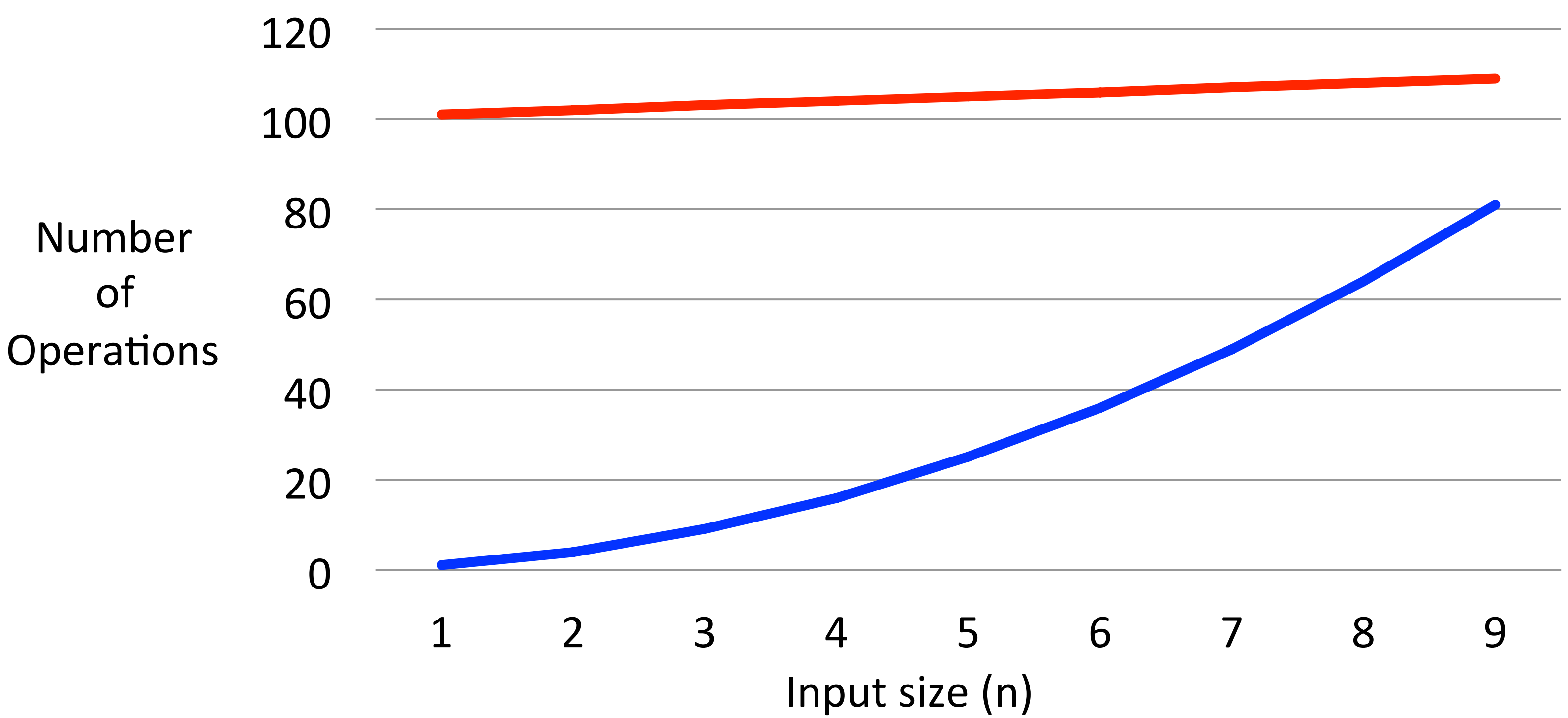
- Asymptotic analysis is a method of describing a limiting behavior

- The O notation asymptotically bounds a function
  - estimate the complexity in asymptotic sense, i.e. estimate the complexity of a function for arbitrarily large inputs

Asymptotic Complexity

- Which function is bigger?

  $f(n) = n^2$ or $g(n) = n + 100$

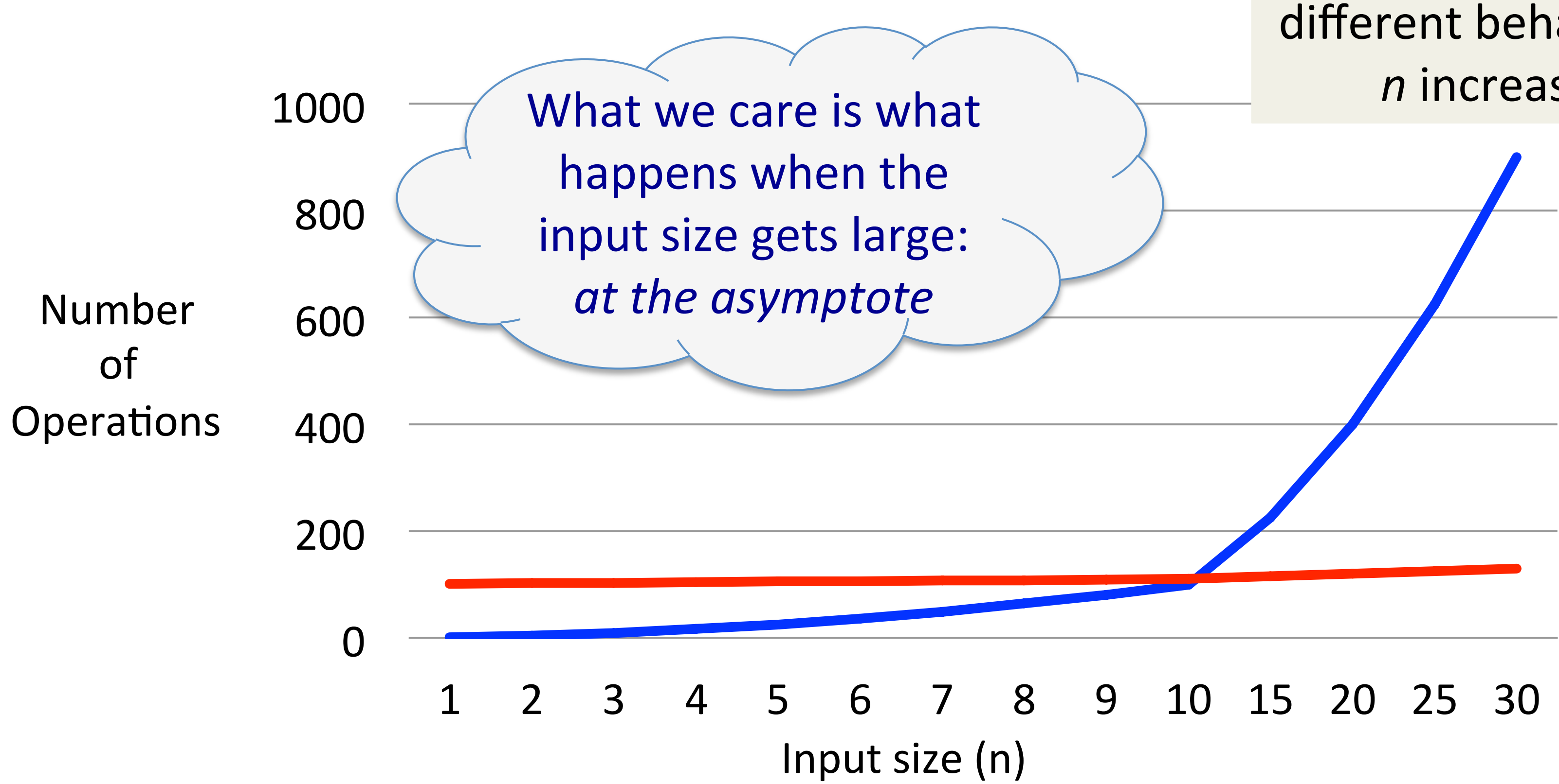


Asymptotic Complexity

- Which function is bigger?

$f(n) = n^2$ or $g(n) = n + 100$

These functions have different behavior as *n* increases

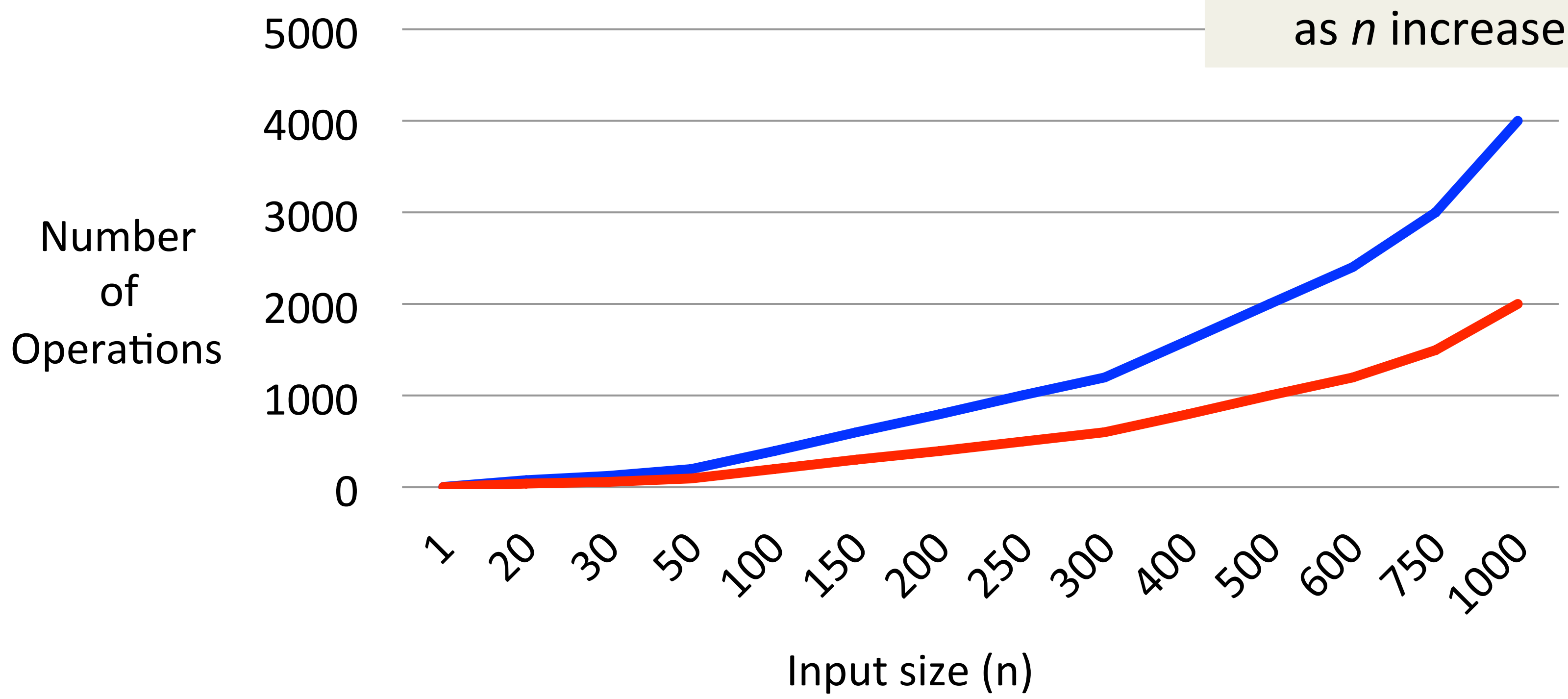


Asymptotic Complexity

- Which function is bigger?

  f(n) = 4*n or g(n) = 2*n

Both functions have the same behavior as *n* increases



Big O

- $O(f(n))$ is a group of functions that
  - behave like $f(n)$ as n gets large
  - ignoring constant multiples
- $O(n)$ includes
  - n, $n*20$, $n+17$, $5*n+\log(n)$
- $O(n^2)$ includes
  - n^2, $n^2+20*n-99$, $n^2-n^{1/2}$

1. Express the running time as a function of the input size
2. Simplify by keeping only the fastest growing term
   - Drop constants
   - Drop insignificant terms

Rules for Big O

- k is in O(1) for any constant k

  783 is in O(1)
- f+g is in max(O(f), O(g))

  n+1 is in max(O(n),O(1)) = O(n)
- k*f = O(f)

  $4*n^3$ is in $O(n^3)$
- $O(n^A) < O(n^B)$ if A < B

  $O(n^3) < O(n^4)$
- O(polynomial) is in O(highest exponent term)

  $5 n^4 + 44 n^2 + 55n + 12$ is in $O(n^4)$

Order-of-growth classification

description	**order of growth**	**typical code**	**description**	**example**
constant	1	`a = b + c;`	*statement*	*add two numbers*
logarithmic	log N	[see search algorithms]	*divide in half*	*binary search*

linear	N	`double max = a[0];`
		`for (int i=1; i<N; i++)`
		`  if (a[i] > max)`
		`    max = a[i];`
			loop	*find the maximum*

logarithmic	N log N	[will learn later]	*divide and conquer*	*mergesort*

Order-of-growth classification

description	order of growth	typical code	description	example

quadratic	N^2	for(int i=0; i<N; i++)
		for(int j=i+1; j<N; j++)
		if(a[i]+a[j] == 0)
		cnt++
			double loop	*check all pairs*

cubic	N^3	for(int i=0; i<N; i++)
		for(int j=i+1; j<N; j++)
		for(int k=j+1; k<N; k++)
		if(a[i]+a[j]+a[k] == 0)
		cnt++
			triple loop	*check all triples*

exponential	2^N	[will learn later in CS]	*exhaustive search*	*check all subsets*

Sequential Search: Efficiency Analysis

- Check each element in sequence to find the target

3	5	12	2	56	32	8	14			
0	1	2	3	4	5	6	7	8	9	Array index

- Best case: O(1) constant time
  - always 1 comparison, no matter how big is *n*
- Worst case: O(n) linear time
  - always takes n comparisons

Input size (n)	**Number of comparisons**	
	Best	Worst
10	1	10
100000	1	100000

Binary Search

- Searching an ordered array
  - How to find a target value?
    - one test can rule out a whole region of the array

3	5	12	22	56	62	85	94	95	99	
0	1	2	3	4	5	6	7	8	9	Array index
↑ **Left**				↑ **Middle**					↑ **Right**	

- How to find the middle point of the array?

  middle = (left + right) / 2 (integer division)

Binary Search

- Searching an ordered array
  - How to find 12?

3	5	12	22	56	62	85	94	95	99	
0	1	2	3	4	5	6	7	8	9	Array index

Left	Right	Middle
0	9	4

12 == 56? no   12 < 56? yes

Binary Search

- Searching an ordered array
  - How to find 12?

3	5	12	22	56	62	85	94	95	99
0	1	2	3	4	5	6	7	8	9

Array index

Left	Right	Middle
0	9	4
0	3	1

12 == 56? no   12 < 56? yes

12 == 5? no   12 < 5? no

Binary Search

- Searching an ordered array
  - How to find 12?

3	5	12	22	56	62	85	94	95	99	
0	1	2	3	4	5	6	7	8	9	Array index

Left	Right	Middle
0	9	4
0	3	1
2	3	2

12 == 56? no   12 < 56? yes

12 == 5? no   12 < 5? no

12 == 12? yes   **Found it!**

Binary Search: Efficiency Analysis

```
void BinarySearch (int[] array, int n, int target) {

    int l = 0, r = n - 1;
    while (l <= r) {
        int m = (l+r)/2;
        if (array[m] == target) {
            System.out.println(target + " found");
            return;
        }
        if (target < array[m]) {
            r = m - 1;
        } else {
            l = m + 1;
        }
    }
    System.out.println(target + " not found");
}
```

Executed *y* times where n = 2^y
Recall the log function: $\log_a b=c$ is equivalent to $b = a^c$ Therefore, loop executed log n time

Basic Operations

Running time
f(n) = 2* log (n)

Binary Search: Efficiency Analysis

- Check middle element to rule out one half of the array

3	5	12	22	56	62	85	94	95	99	
0	1	2	3	4	5	6	7	8	9	Array index

  – Best case: O(1)
    - always 1 comparison, no matter how big is *n*

  – Worst case: O (log n)
    - always takes log n comparisons

Running time
f(n) = 2* log (n)

Input size (n)	**Number of comparisons**	
	Best	Worst
8	1	6
4096	1	24

Efficiency Analysis

- Linear Search: O(n)
- Binary Search: O(log n)