DATA STRUCTURES AND ALGORITHMS

Lecture 3: Complexity Analysis

Lecturer: Mohsin Abbas

National University of Modern Languages, Islamabad

COMPARING ALGORITHM

- Given two or more algorithms to solve the same problem, how do we select the best one?
- Some criteria for selecting an algorithm:
  - Is it easy to implement, understand, modify?
  - How long does it take to run it to completion?
  - How much of computer memory does it use?
- Software engineering is primarily concerned with the first criteria.
- In this course we are interested in the second and third criteria.

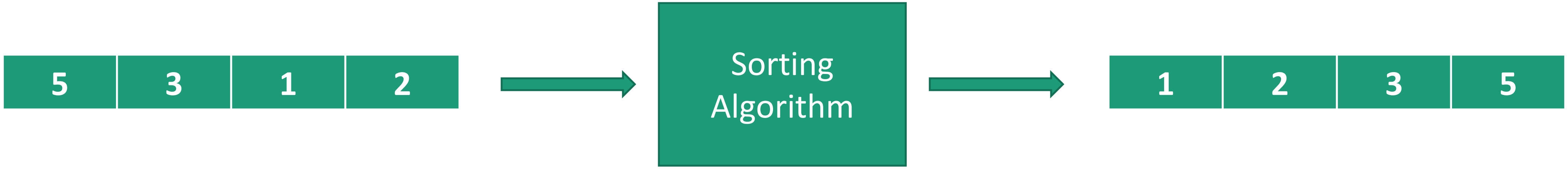
COMPARING ALGORITHM

- Time complexity:
  - The amount of time that an algorithm needs to run to completion
  - Better algorithm is the one which runs faster
    - Has smaller time complexity
- Space complexity:
  - The amount of memory an algorithm needs to run

- In this lecture, we will focus on analysis of time complexity

HOW TO CALCULATE RUNNING TIME

- Most algorithms transform input objects into output objects



- The running time of an algorithm typically grows with input size
  - Idea: analyze running time as a function of input size

HOW TO CALCULATE RUNNING TIME

- Most important factor affecting running time is usually the size of the input

```
int find_max( int array[], int n ) {
    int max = array[0];
    for ( int i = 1; i < n ; i++ ) {
        if ( array[i] > max ) {
            max = array[i];
        }
    }
    return max;
}
```

- Regardless of the size ***n*** of an array the *cost will always be same*.
    - Every element in the array is checked one time

HOW TO CALCULATE RUNNING TIME

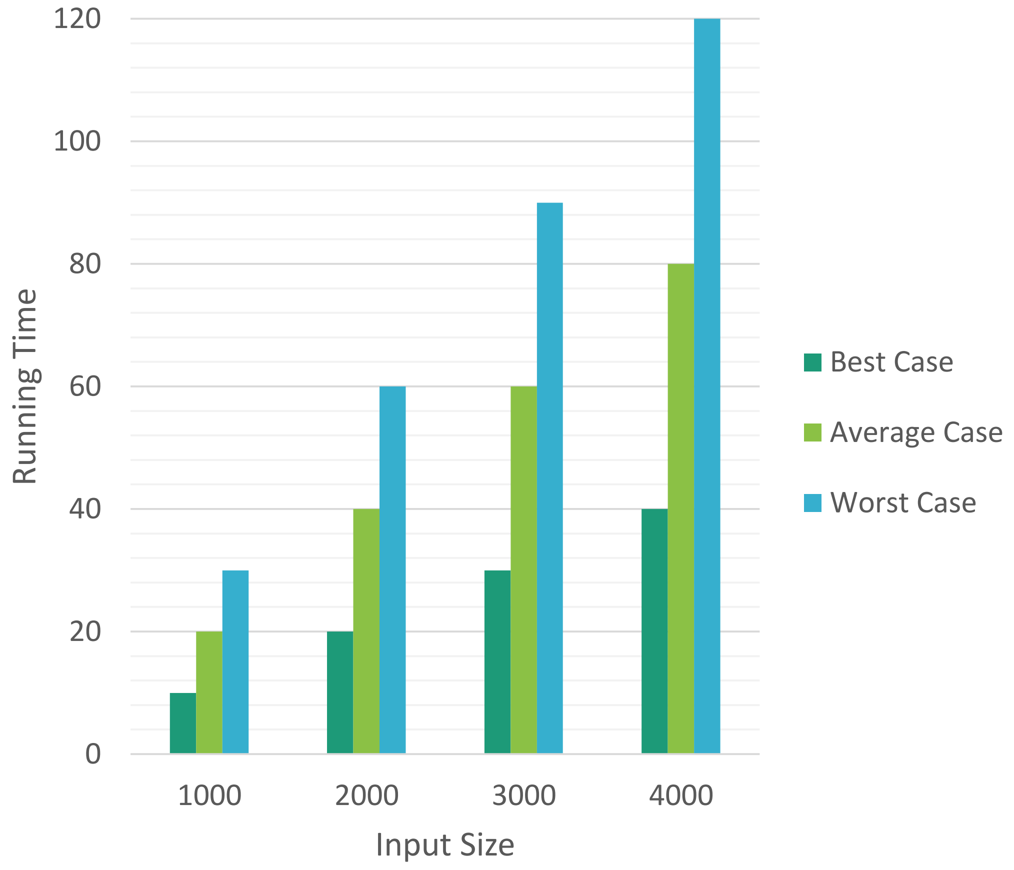
- Even on inputs of the same size, running time can be very different

```
int search( int array[], int n, int x ) {
    int loc = 0;
    for ( int i = 0; i < n; i++ ) {
        if ( array[i] == x ) {
            loc = i;
        }
    }
    return loc;
}
```

- Idea: Analyze running time for different cases.
    - Best case
    - Worst case
    - Average case

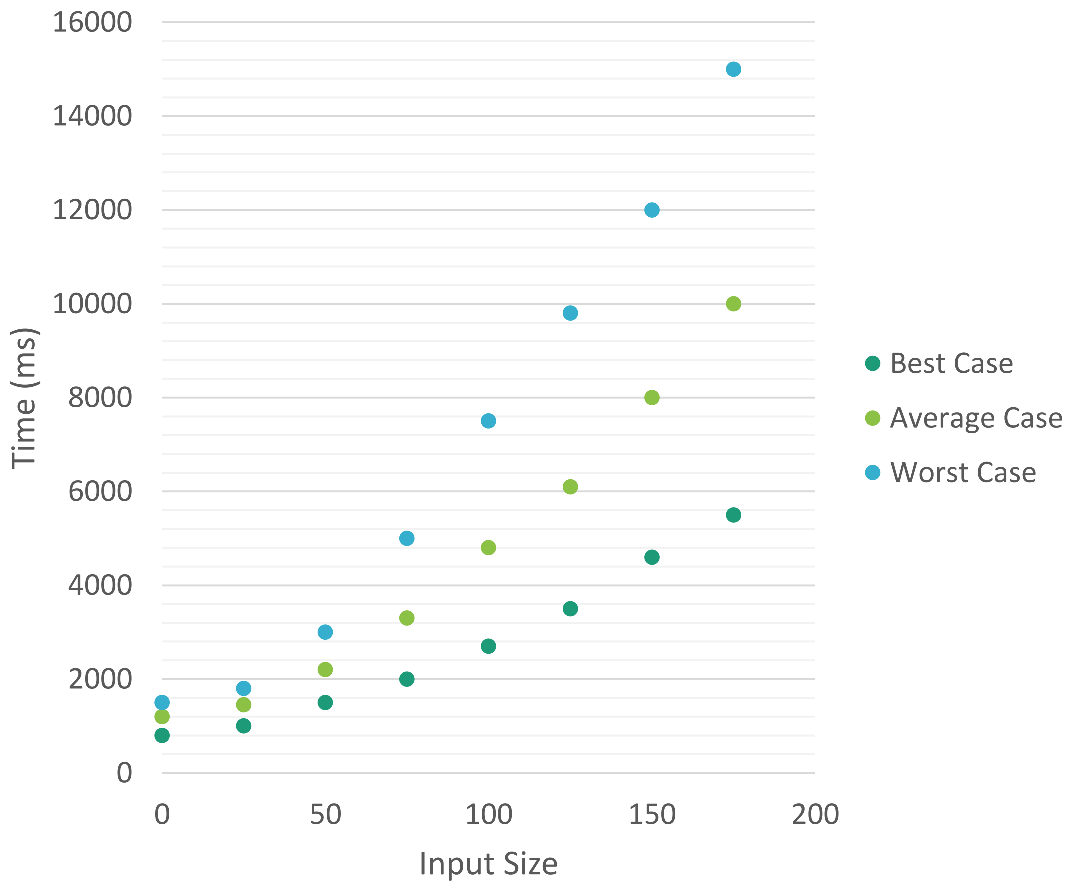
HOW TO CALCULATE RUNNING TIME

- Best case running time is usually not very useful
- Average case time is very useful but often hard to determine
- Worst case running time is easier to analyze
    - Crucial for real-time applications such as games, finance and robotics



EXPERIMENTAL EVALUATIONS OF RUNNING TIMES

- Write a program implementing the algorithm
- Run the program with inputs of varying size
- Use clock methods to get an accurate measure of the actual running time
- Plot the results



LIMITATIONS OF EXPERIMENTS

<u>Experimental evaluation of running time is very useful but</u>

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment
- In order to compare two algorithms, the same hardware and software environments must be used

THEORETICAL ANALYSIS OF RUNNING TIME

- Uses a pseudo-code description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size ***n***
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

ANALYZING AN ALGORITHM – OPERATIONS

- Each machine instruction is executed in a fixed number of cycles
  - We may assume each operation requires a fixed number of cycles
- Idea: Use abstract machine that uses steps of time instead of sec’s
  - Each elementary operation takes 1 steps
- Example of operations:
  - Retrieving/storing variables from memory
  - Variable assignment **=**
  - Integer operations **+ - * / % ++ --**
  - Logical operations **&& || !**
  - Bitwise operations **& | ^ ~**
  - Relational operations **== != < <= => >**
  - Memory allocation and de-allocation **new, delete**

ANALYZING AN ALGORITHM

```
int sumArray( int arr[], int n ) {
    int sum = 0;

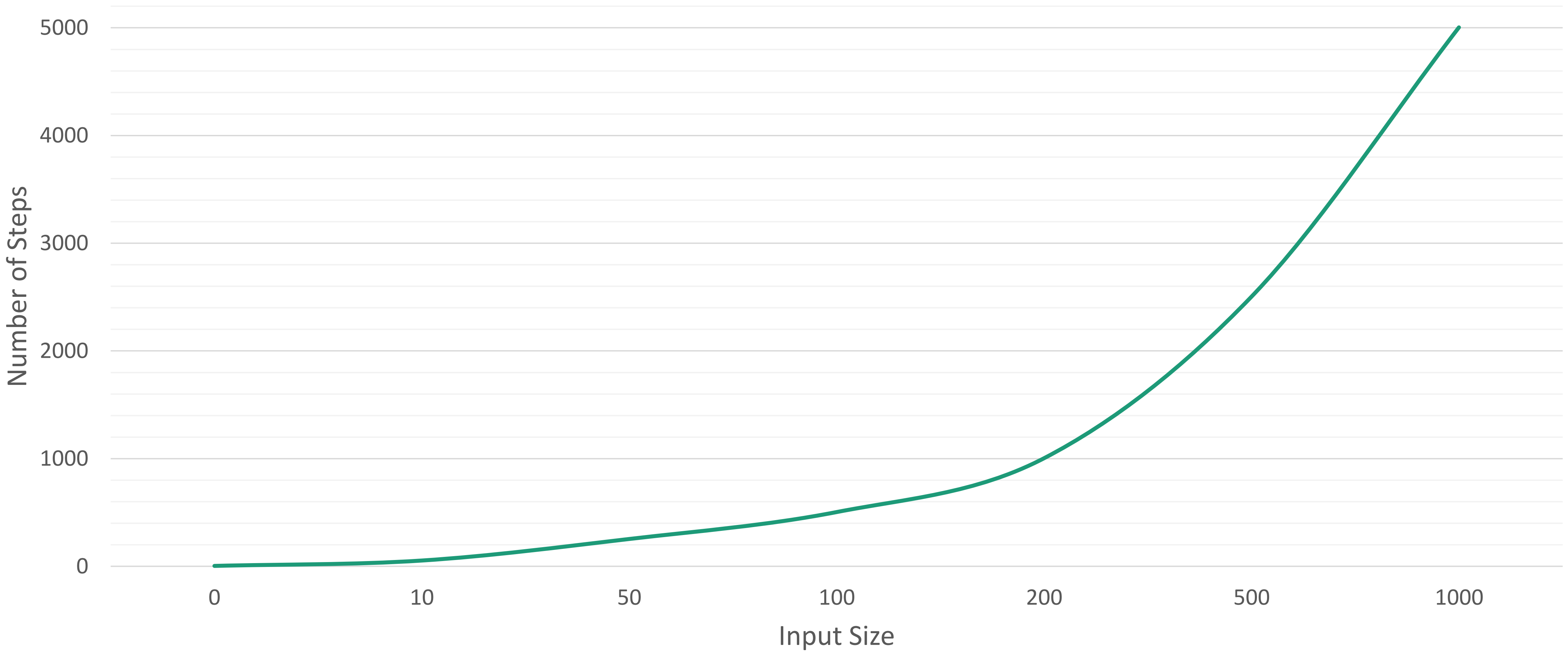
    for ( int i = 0; i < n; i++ ) {
        sum = sum + arr[i];
    }
    return sum;
}
```

- Operations 1,2,8 are executed once
- Operations 4,5,6,7: Once per each iteration of for loop n iteration
- Operation 3 is executed n+1 times
- The complexity function of the algorithm is : $T(n) = 5n + 4$

ANALYZING AN ALGORITHM – GROWTH RATE

- Estimated running time for different values of n:
  - n = 10 => 54 steps
  - n = 100 => 504 steps
  - n = 1000 => 5004 steps
  - n = 1,000,000 => 5,000,004 steps
- As n grows, number of steps T(n) grow in linear proportion to n

GROWTH RATE



5000
4000
3000
2000
1000
0
Number of Steps
0
10
50
100
200
500
1000
Input Size

EXAMPLE

```
int search( int array[], int n, int x ) {
    int loc = 0;
    for ( int i = 0; i < n; i++ ) {
        if ( array[i] == x ) {
            loc = i;
        }
    }
    return loc;
}
```

```
int binary_search( int arr[], int left, int
right, int num ) {
    while( left <= right ) {
        int mid = left + ( right – left ) / 2;

        if ( arr[mid] == num ) {
            return mid;
        }

        if ( arr[mid] < num ) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }
    return -1;
}
```

GROWTH RATE

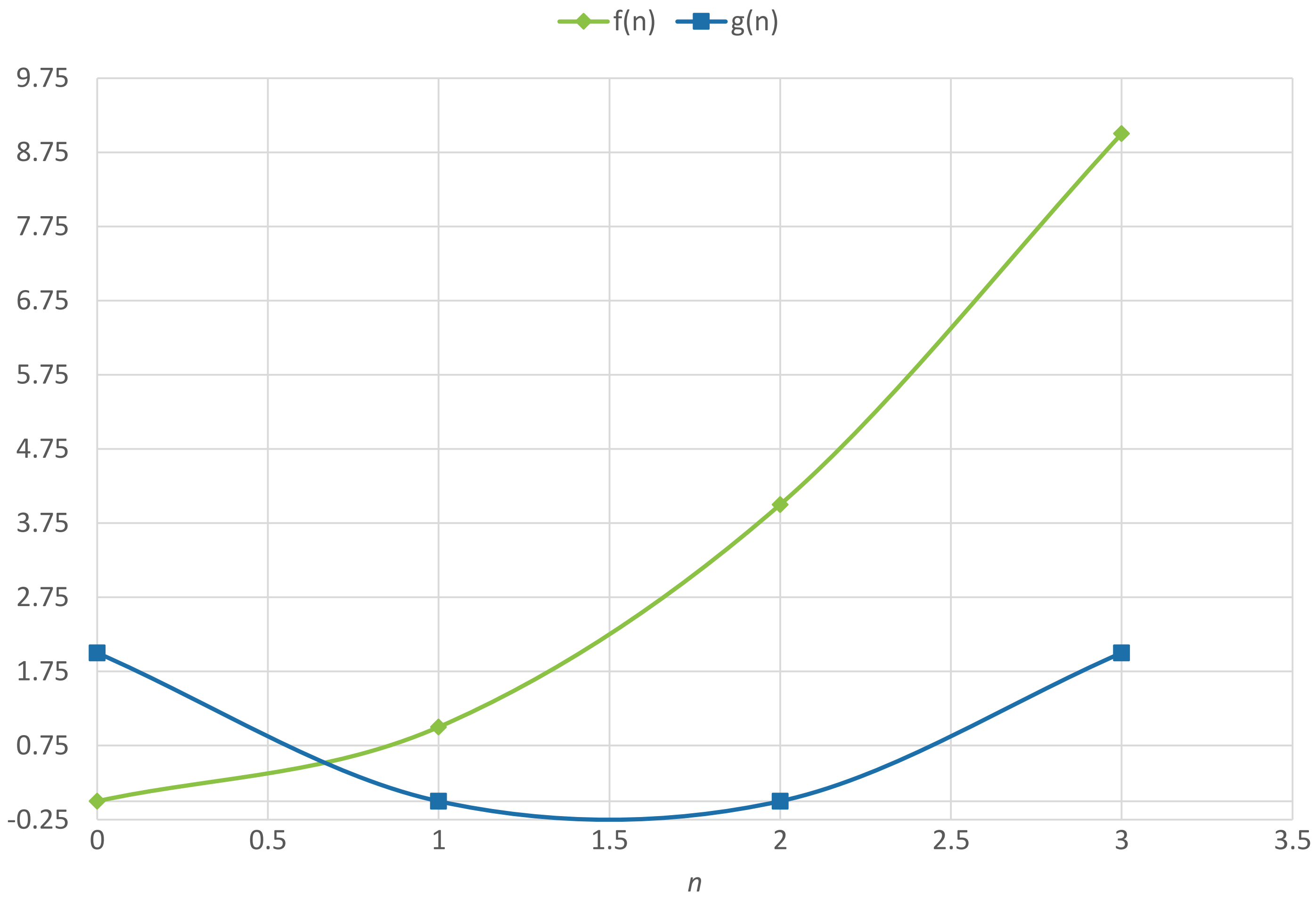
- Changing the hardware/software environment
  - Affects T(n) by a constant factor, but
  - Does not alter the growth rate of T(n)
- Thus we focus on the big-picture which is the growth rate of an algorithm
- The linear growth rate of the running time T(n) is an intrinsic property of algorithm `sumArray`

CONSTANT FACTORS

- The growth rate is not affected by
  - Constant factors or
  - Lower-order terms
- Examples:
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function

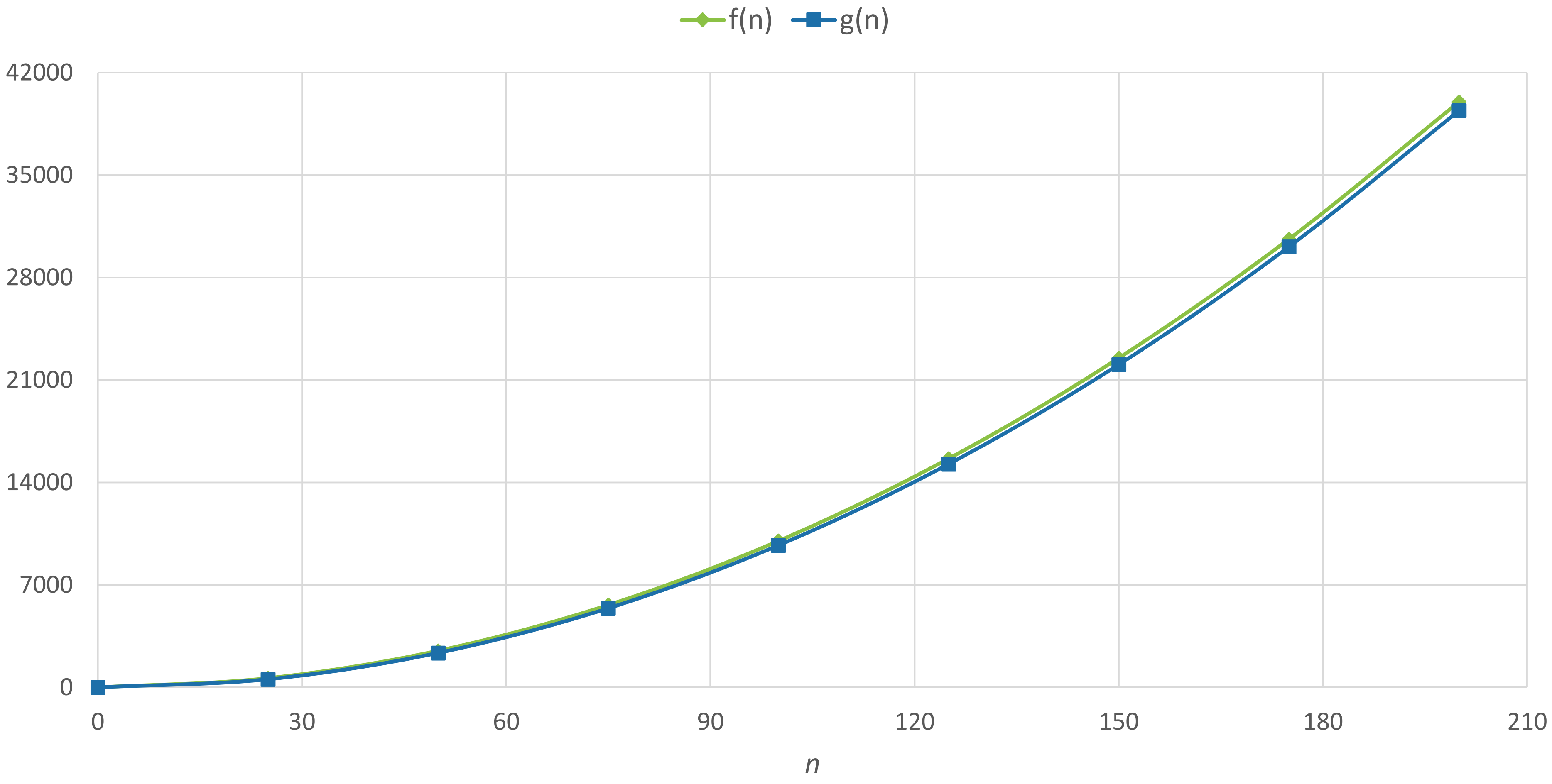
GROWTH RATE – EXAMPLE

- Consider the two functions:
  - $f(n) = n^2$
  - $g(n) = n^2 - 3n + 2$
- Around n = 0, they look very different
  - $f(0) = 0^2 = 0$
  - $g(0) = 0^2 - 3(0) + 2 = 2$



GROWTH RATE – EXAMPLE

- Yet on the range n = [0, 1000], f(n) and g(n) are (relatively) indistinguishable



GROWTH RATE – EXAMPLE

- The absolute difference is large, for example,
  - `f(1000) = 1000`^{2} `= 1,000,000`
  - `g(1000) = 1000`^{2} `– 3(1000) + 2 = 997,002`
- But the relative difference is very small

$$\left|\frac{f(1000) - g(1000)}{f(1000)}\right| = 0.002998 < 0.3\%$$

- The difference goes to zero as n → ∞

CONSTANT FACTORS

- The growth rate is not affected by
  - Constant factors or
  - Lower-order terms
- Examples:
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function
- How do we get rid of the constant factors to focus on the essential part of the running time?
  - Asymptotic Analysis

UPPER BOUND – BIG-OH NOTATION

- Indicates the upper or highest growth rate that the algorithm can have
  - Ignore constant factors and lower order terms
  - Focus on main components of a function which affect its growth

Definition: Given functions $f(n)$ and $g(n)$

- We say that $f(n)$ is $O(g(n))$
- If there are positive constants c and n_0 such that
  - $f(n) \le cg(n)$ for $n \ge n_0$

BIG-OH NOTATION – EXAMPLES

- $7n-2$ is $O(n)$
  - Need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c.n$ for $n \geq n_0$
  - True for $c = 7$ and $n_0 = 1$
- $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - Need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c.n^3$ for $n \geq n_0$
  - True for $c = 4$ and $n_0 = 21$
- $3 \log n + 5$ is $O(\log n)$
  - Need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$
  - True for $c = 8$ and $n_0 = 2$

ANALYZING AN ALGORITHM

- Simple Assignment
  - `a = b`
  - $O(1)$
- Simple Loops
  - `for(i=0; i<n; i++) { s; }`
  - $O(n)$
- Nested Loops
  - ```
    for(i=0; i<n; i++)
      for(j=0; j<n; j++) { s; }
    ```
  - $O(n^2)$
    ```

ANALYZING AN ALGORITHM

- Loop index doesn't vary linearly

```
h = 1;
while ( h <= n ) {
    S;
    h = 2 * h;
}
```

  - h takes values 1, 2, 4, … until it exceeds n
  - There are $1 + \log_2 n$ iterations
  - $O(\log_2 n)$

ANALYZING AN ALGORITHM

- Loop index depends on outer loop index

```
for( j = 0; j <= n; j++ ) {
    for( k = 0; k < j; k++ ) {
        S;
    }
}
```

  - Inner loop executed 0, 1, 2, 3, …., n times

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

  - $O(n^2)$

RELATIVES OF BIG-OH

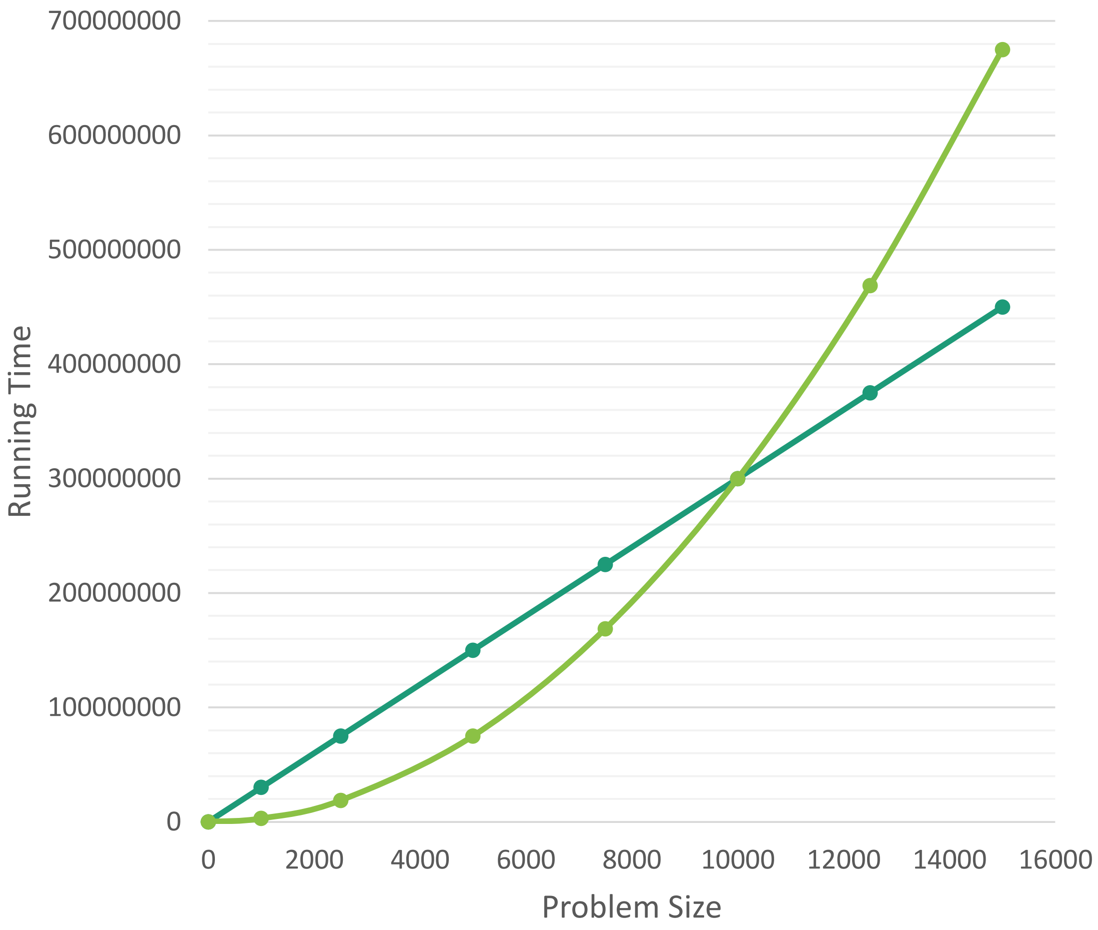
- Big-Omega
  - $f(n)$ is $\Omega(g(n))$
  - If there is a constant $c > 0$ and an integer constant $n_0 \geq 1$
  - Such that $f(n) \geq c.g(n)$ for $n \geq n_0$
- Big-Theta
  - $f(n)$ is $\Theta(g(n))$
  - If there are constants $c_1 > 0$ and $c_2 > 0$ and an integer constant $n_0 \geq 1$
  - Such that $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for $n \geq n_0$

INTUITION FOR ASYMPTOTIC NOTATION

- Big-Omega
  - $f(n)$ is $O(g(n))$ – if $f(n)$ is asymptotically less than or equal to $g(n)$
- Big-Omega
  - $f(n)$ is $\Omega(g(n))$ – if $f(n)$ is asymptotically greater than or equal to $g(n)$
  - Note: $f(n)$ is $\Omega(g(n))$ if and only if $g(n)$ is $O(f(n))$
- Big-Theta
  - $f(n)$ is $\Theta(g(n))$ – if $f(n)$ is asymptotically equal to $g(n)$
  - Note: $f(n)$ is $\Theta(g(n))$ if and only if
    - $g(n)$ is $O(f(n))$ and
    - $f(n)$ is $O(g(n))$

FINAL NOTES

- Even though in this course we focus on the asymptotic growth using big-Oh notation, practitioners do care about constant factors occasionally.
- Suppose we have 2 algorithms:
  - Algorithm A has running time 30000n
  - Algorithm B has running time $3n^2$
- Asymptotically, algorithm A is better than algorithm B.
- However, if the problem size you deal with is always less than 10000, then the quadratic one is faster.



CONCLUSION

- In this lecture we have studied:
  - Complexity of Algorithm
  - Different ways to measure Complexity
  - Analyzing an Algorithm
  - Growth Rate

Question?