Data Structures

3. Complexity Analysis

Comparing Algorithms

- 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 Algorithms

- 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

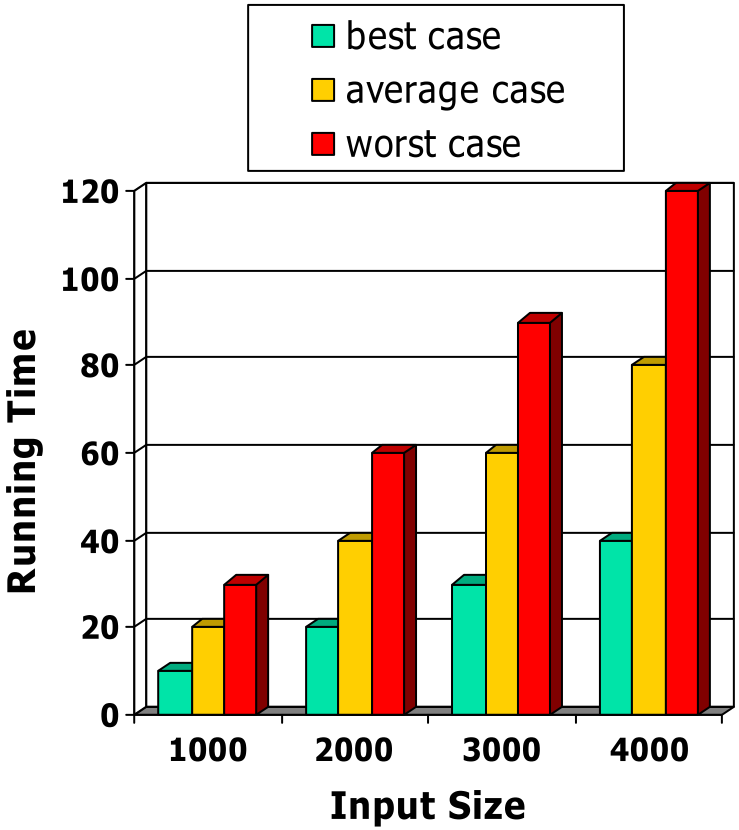
- Even on inputs of the same size, running time can be very different

```
int search(int arr[], int n, int x) {
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

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

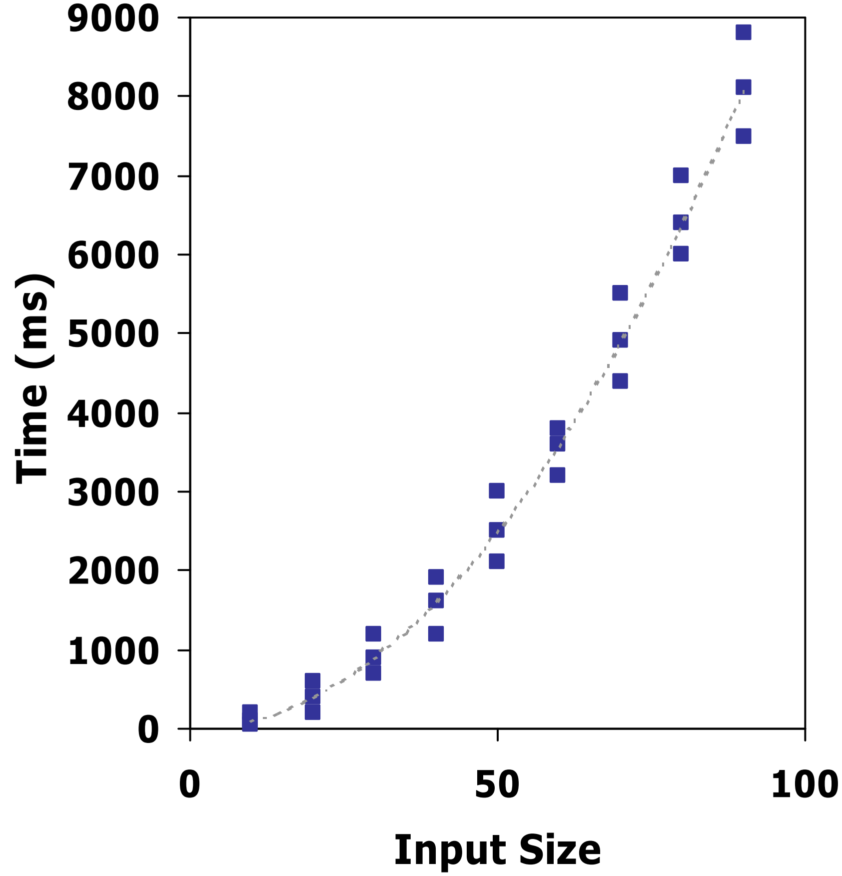
How To Calculate Running Time

- 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

Experimental evaluation of running time is very useful but

- 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 secs
  - Each elementary operation takes 1 steps

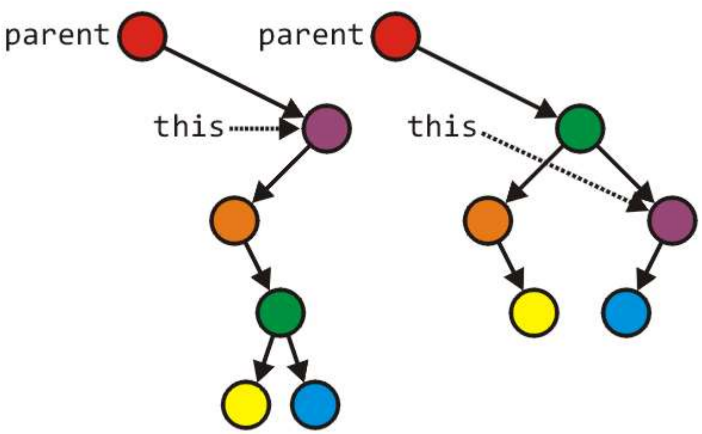
- Example of operations
  - Retrieving/storing variables from memory
  - Variable assignment   =
  - Integer operations   + - * / % ++ --
  - Logical operations   && || !
  - Bitwise operations   & | ^ ~
  - Relational operations   == != < <= => >
  - Memory allocation and deallocation   new delete

Analyzing an Algorithm – Blocks of Operations

- Each operation runs in a step of 1 time unit
- Therefore any fixed number of operations also run in 1 time step
  - `s1; s2; …. ; sk`
  - As long as number of operations k is constant

```
Tree_node *lrl = left->right->left;
Tree_node *lrr = left->right->right;
parent = left->right;
parent->left  = left;
parent->right = this;
left->right = lrl;
left = lrr;
```

```
// Swap variables a and b
int tmp = a;
a = b;
b = tmp;
```



Analyzing an Algorithm

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

int SumArray(int A[], int n){
   int s=0;                  1

   for (int i=0; i< n; i++)
   2                  3        4
     s = s + A[i];
   5           6       7
   return s;
}                   8
```

- Operations 1,2,8 are executed once
- Operations 3,4,5,6,7: Once per each iteration of for loop n iteration

- The complexity function of the algorithm is : $T(n) = 5n + 3$

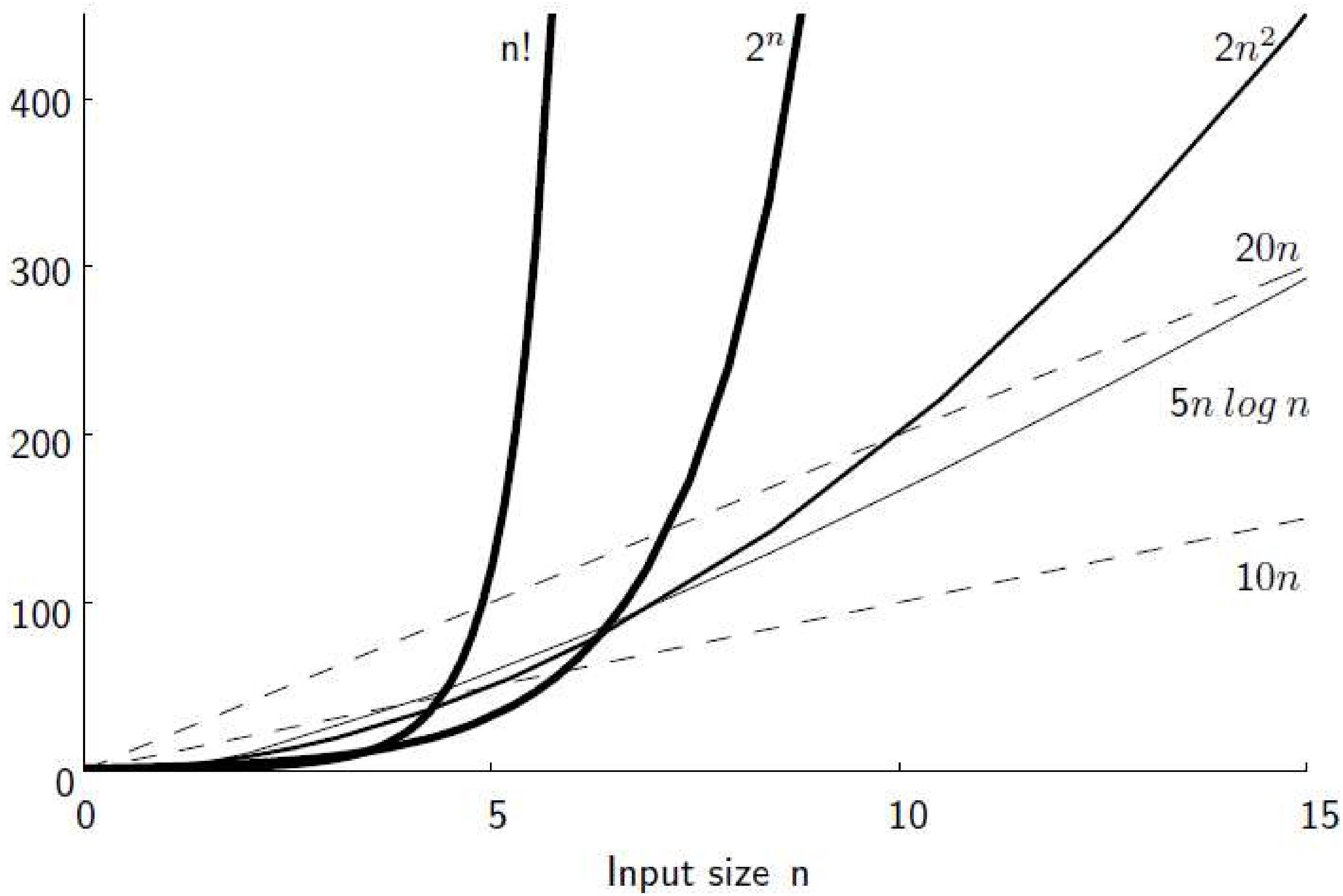
Analyzing an Algorithm – Growth Rate

- Estimated running time for different values of n:

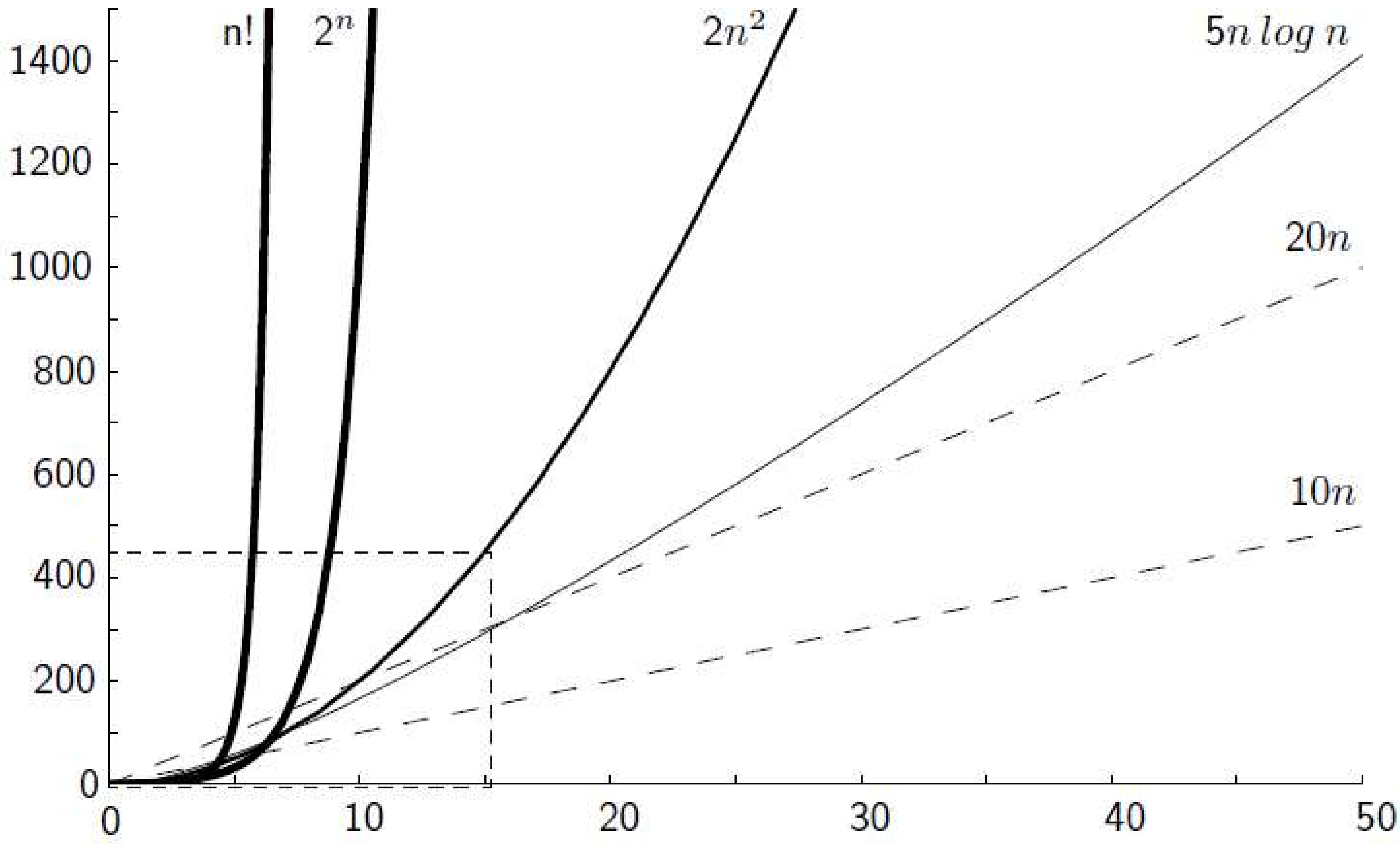
  - n = 10 => 53 steps
  - n = 100 => 503 steps
  - n = 1,000 => 5003 steps
  - n = 1,000,000 => 5,000,003 steps

- As n grows, number of steps T(n) grow in linear proportion to n

Growth Rate



Growth Rate



n!
2^n
$2n^2$
$5n\ log\ n$
$20n$
$10n$
1400
1200
1000
800
600
400
200
0
0
10
20
30
40
50

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`

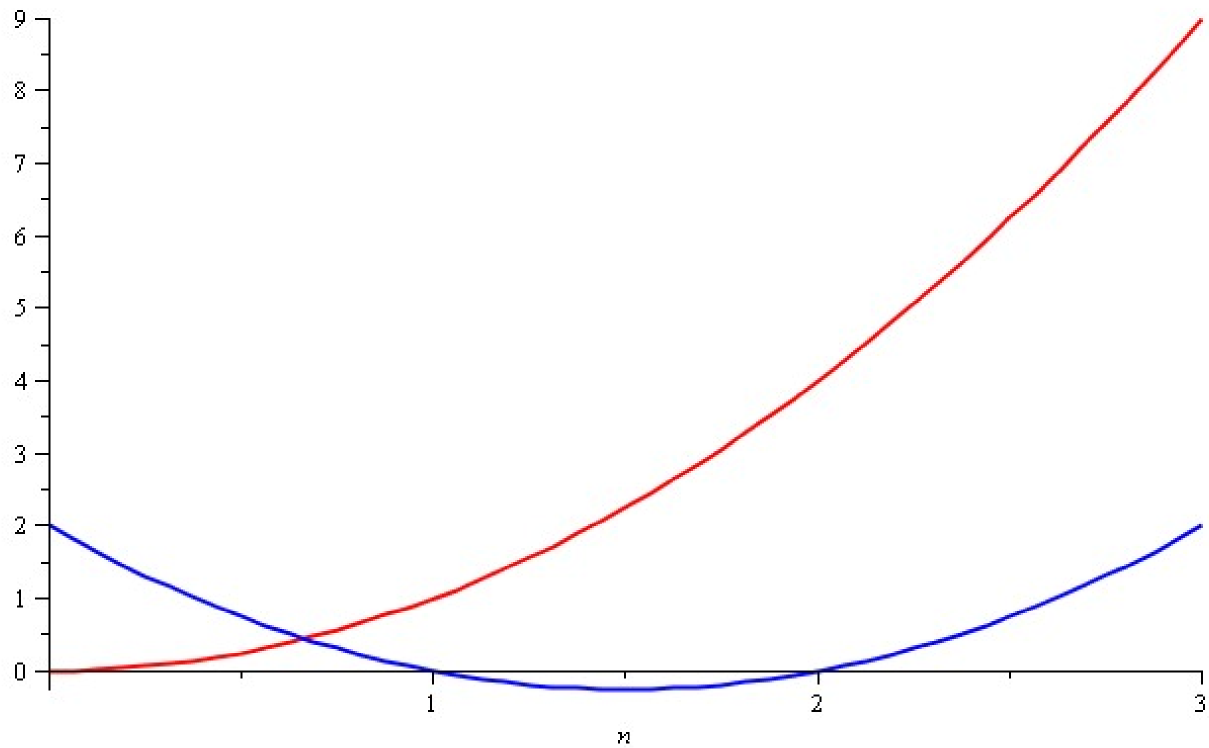
Constant Factors

- 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

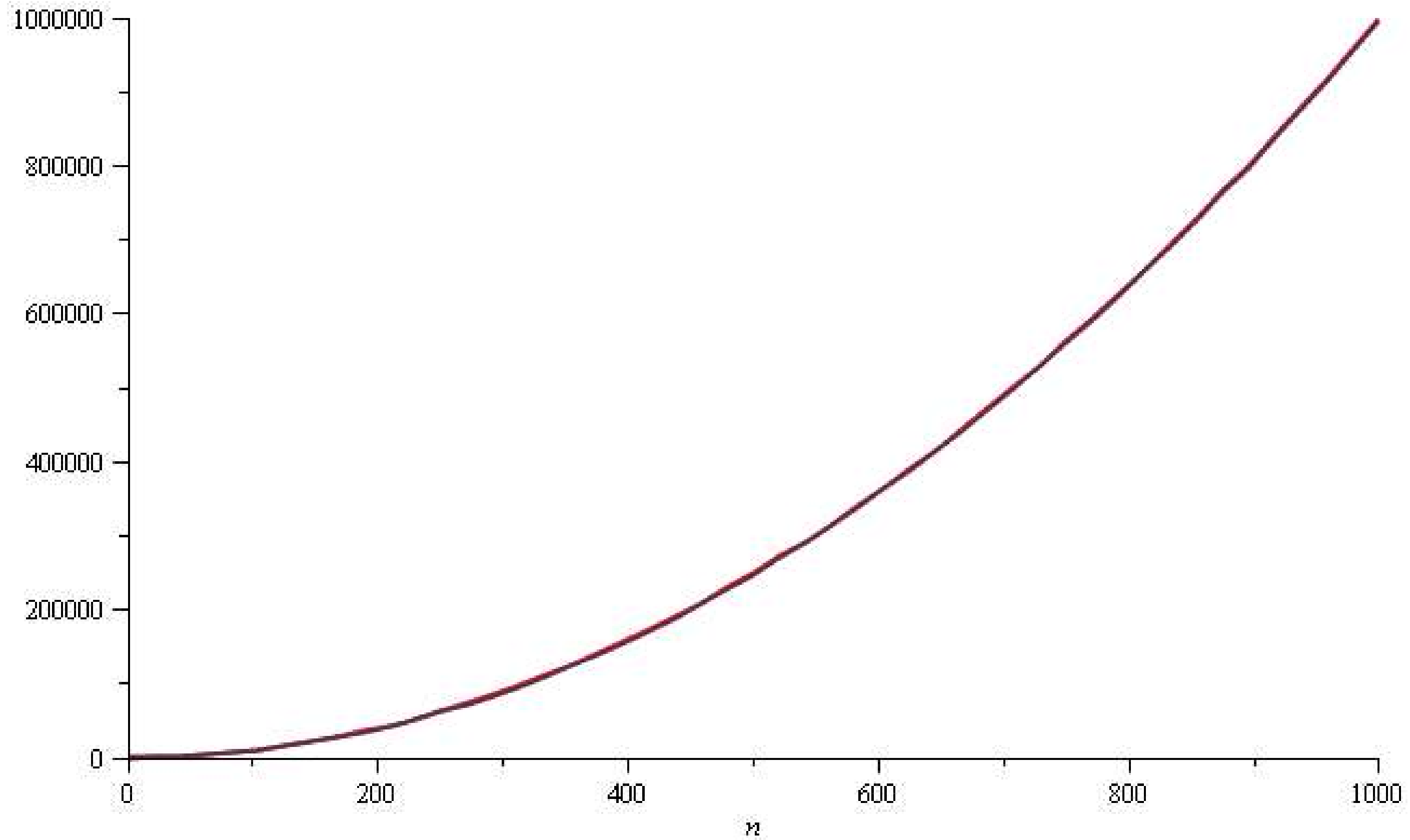
Growth Rate – Example

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



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) = 1 000 000`
  - `g(1000) =   997 002`

- But the relative difference is very small

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

  - The difference goes to zero as $n \to \infty$

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 $c > 0$ & $n_0 \geq 1$ 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 $c > 0$ & $n_0 \geq 1$ 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

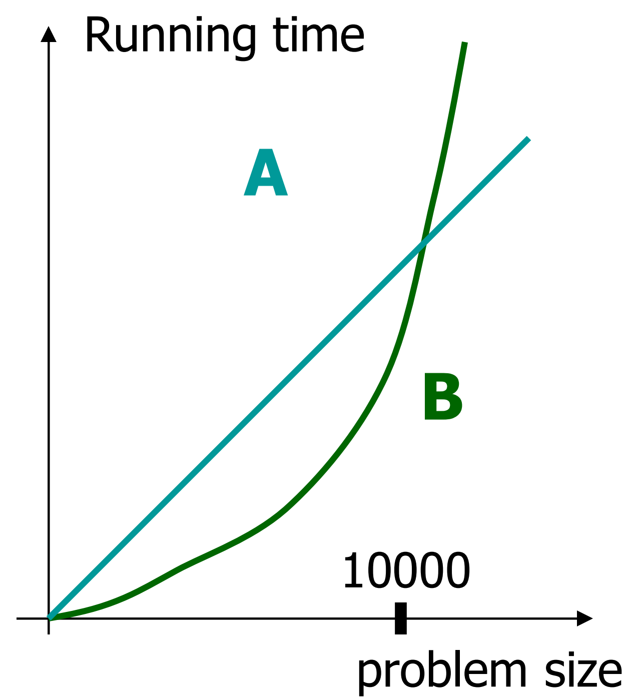
- Loop index depends on outer loop index
  - ```
    for(j=0;j<=n;j++)
        for(k=0;k<j;k++){
            s;
        }
    ```
  - Inner loop executed `0, 1, 2, 3, …., n` times

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

  - $O(n^2)$
    ```

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 3n2
- 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



Any Question So Far?