Data Structures

Fall 2023

---

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

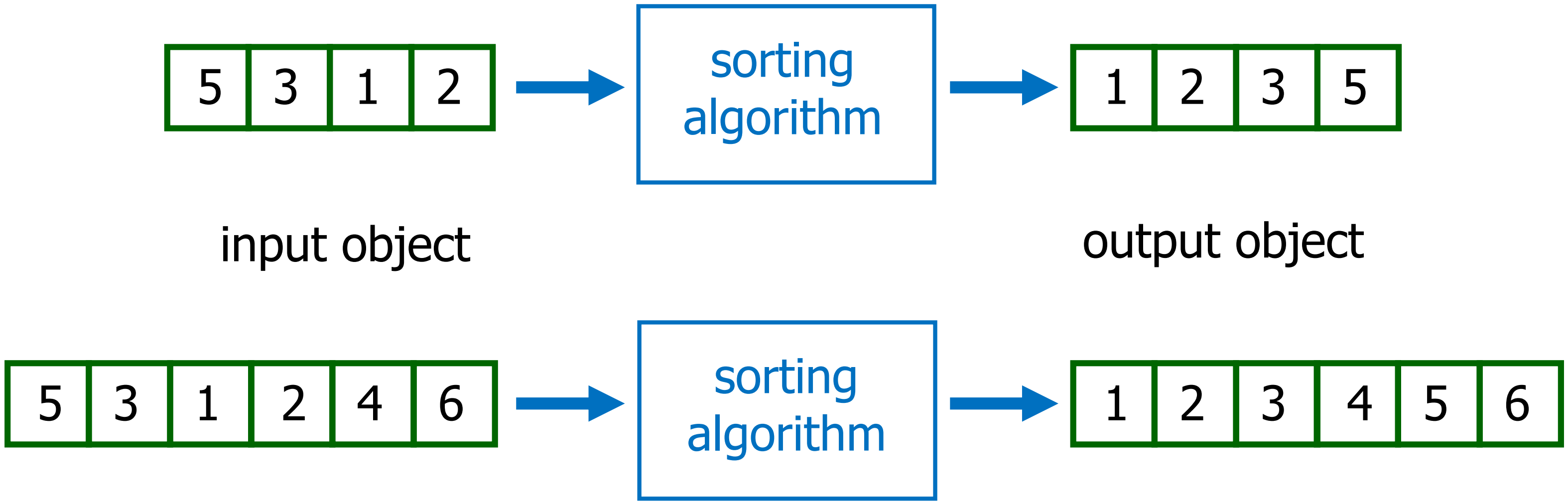
- In this course we are interested in the second and third criteria

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 time complexity 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;
}
```

- Example: Search for 1
  - Best case: Loop runs 1 times

1	2	3	4	5	6

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;
}
```

- Example: Search for 1
  - Worst case: Loop runs n times

6	5	4	3	2	1

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;
}
```

- Example: Search for 1
  - Average case: Loop runs between 1 and n times

3	2	1	4	5	6

How To Calculate Running Time

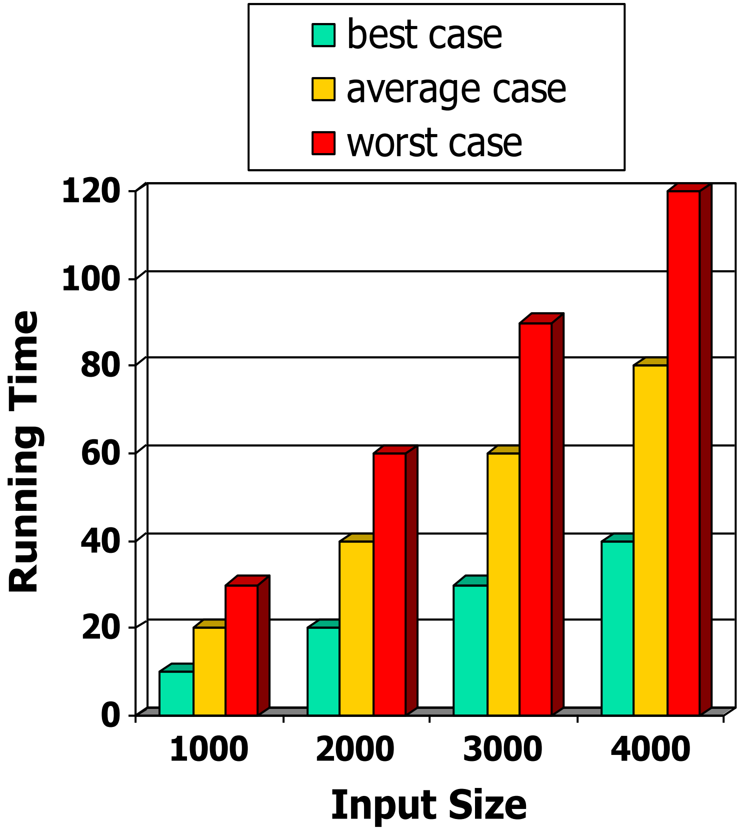
- Even on inputs of the same size, running time can be very different

```
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

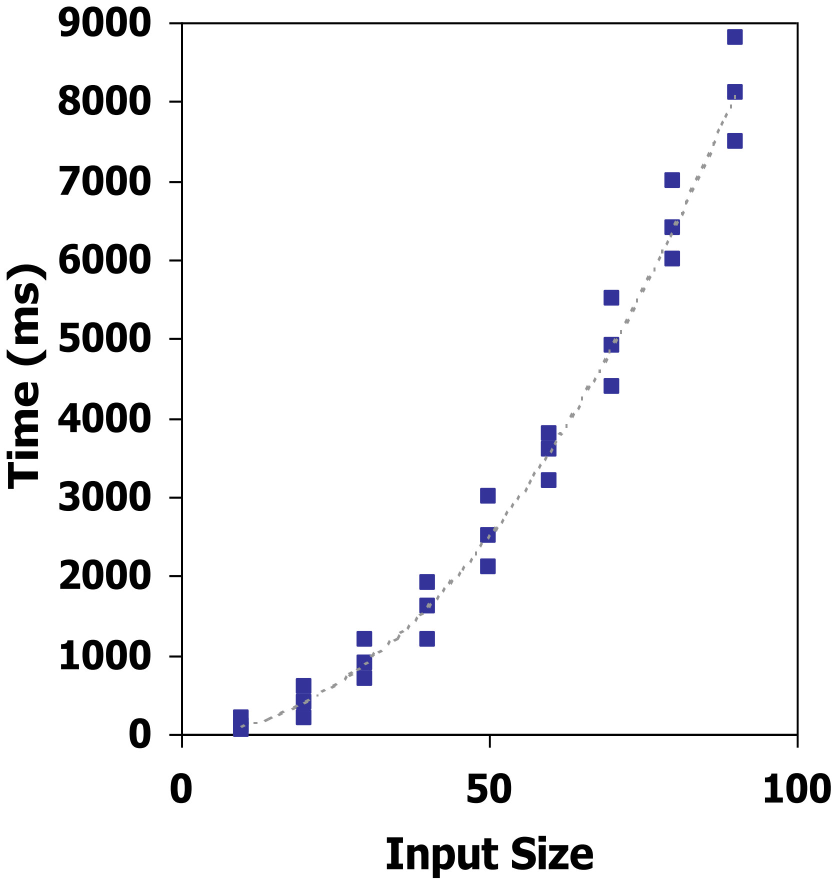
How To Calculate Running Time

- 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

```
// 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, and 8 are executed once
- Operations 4, 5, 6, and 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

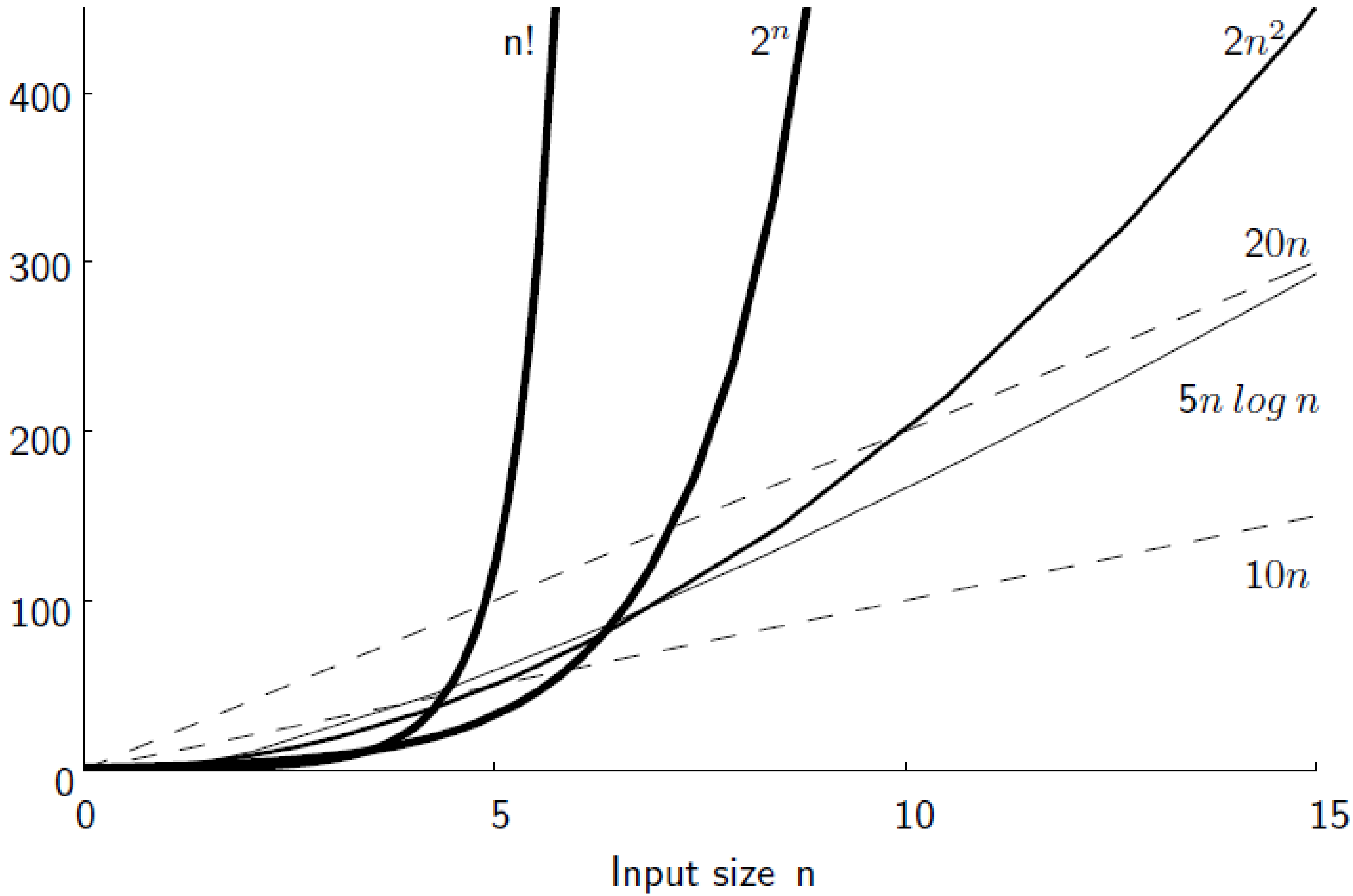
Analyzing an Algorithm – Growth Rate

- Estimated running time for different values of n:

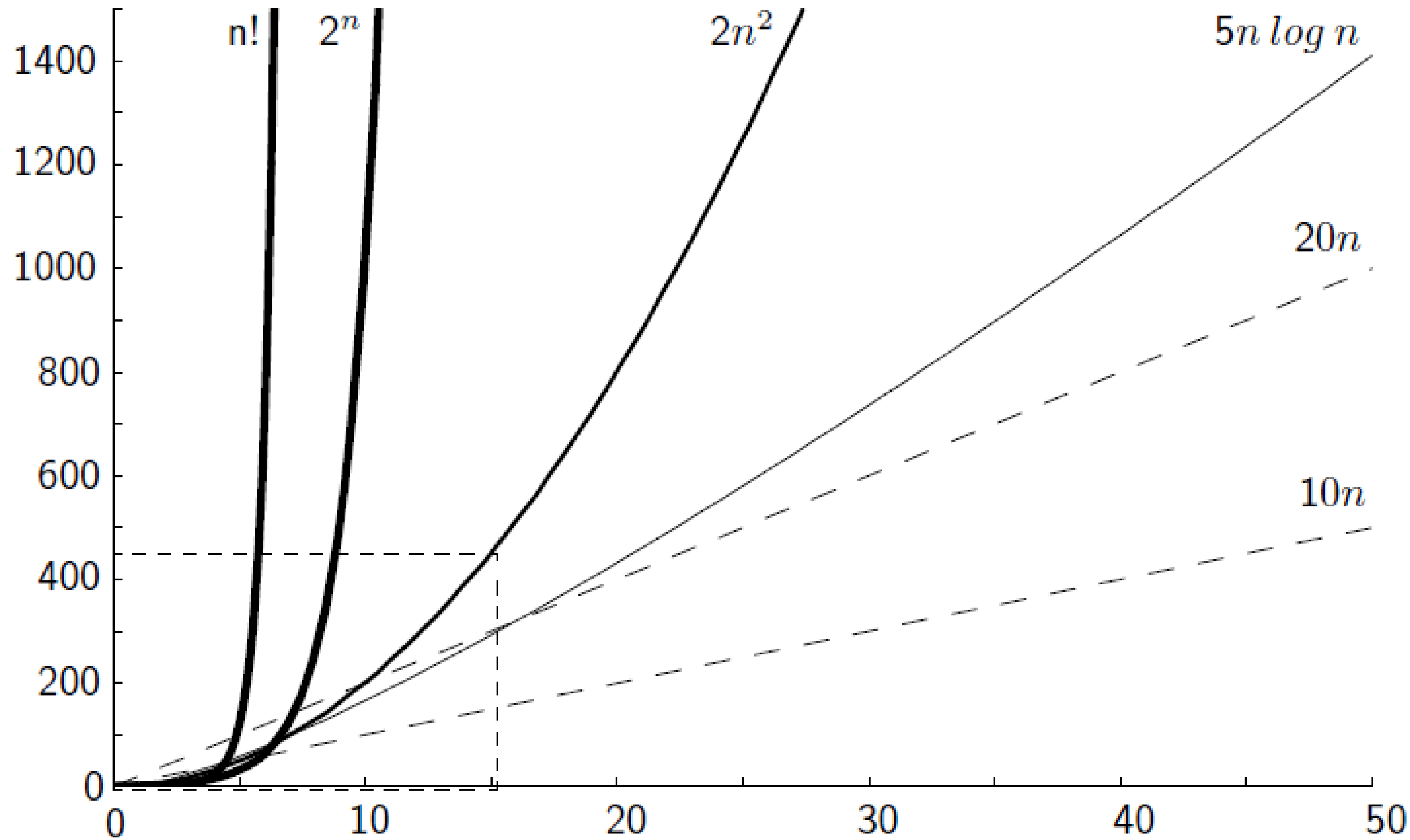
  - n = 10 => 54 steps
  - n = 100 => 504 steps
  - n = 1,000 => 5004 steps
  - n = 1,000,000 => 5,000,004 steps

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

Growth Rate



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`

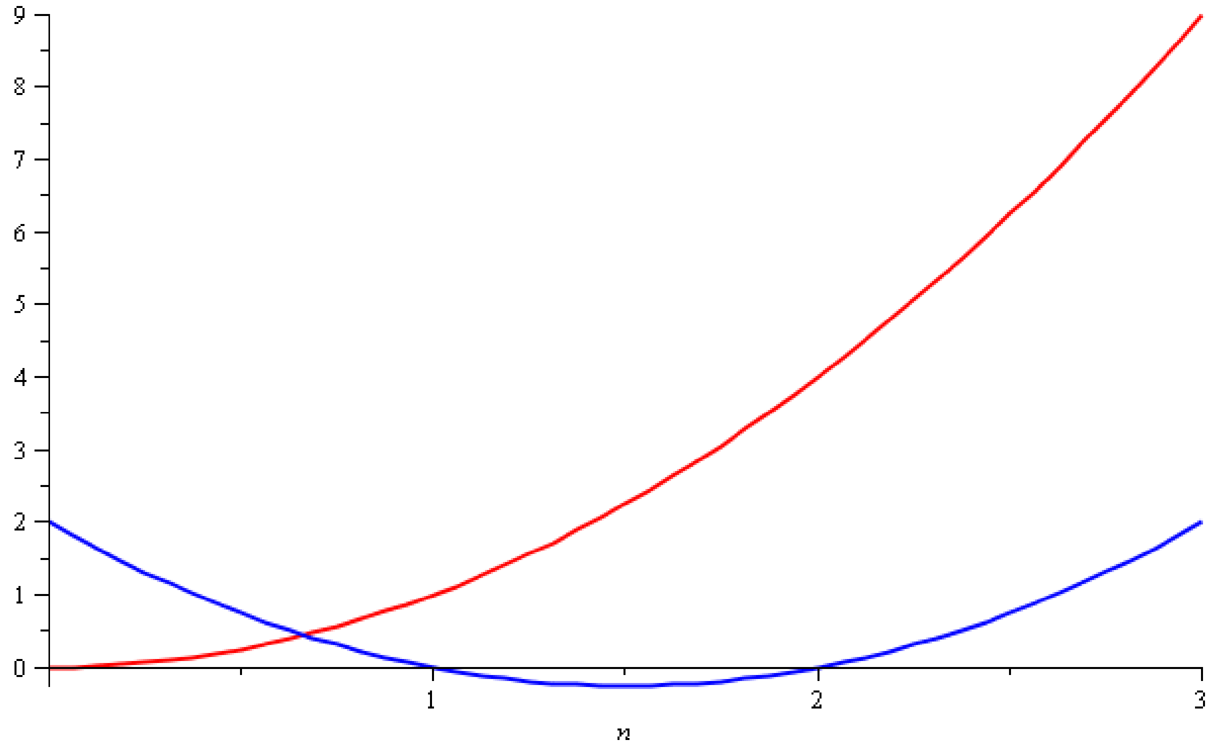
Constant Factors

- The growth rate is not affected by
  - Constant factors or
  - Lower-order terms

- Example:
  - $\mathrm{f}(n) = n^2$
  - $\mathrm{g}(n) = n^2 - 3n + 2$

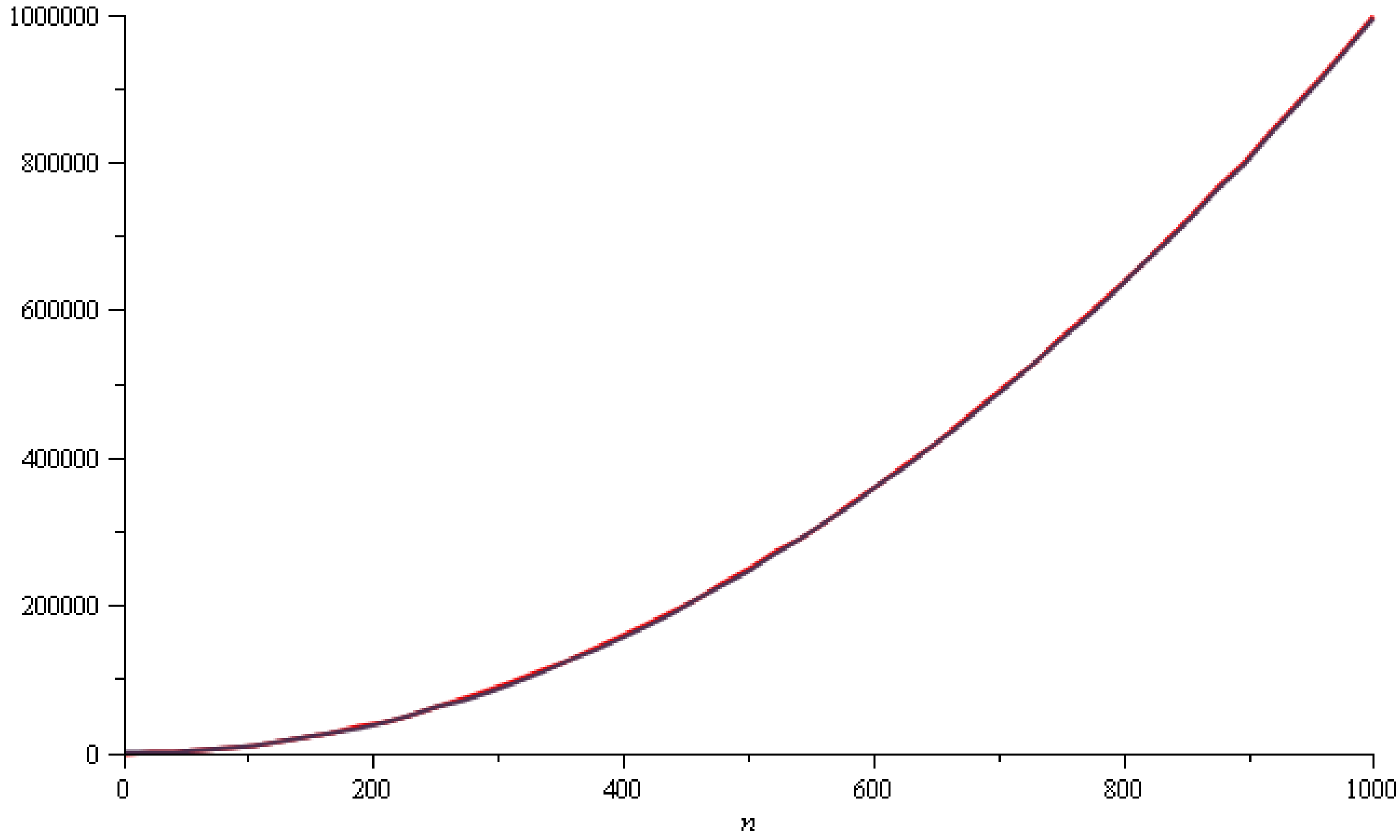
Growth Rate – Example

- 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

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

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

Constant Factors

- The growth rate is not affected by
  - Constant factors or
  - Lower-order terms

- Example:
  - $f(n) = n^2$
  - $g(n) = n^2 - 3n + 2$

For n = 1

% of running time due to n^2 = 1/(1+3+2)*100 = 16.66%

% of running time due to $3n$ = 3/(1+3+2)*100 = 50%

% of running time due to 2 = 2/(1+3+2)*100 = 33.33%

Constant Factors

n	n^2	3n	2
1	16.66%	50%	33.33%
10	75.75%	22.72%	1.515%
100	97.06%	2.912%	0.019%
1000	99.7%	0.299%	0.0001%

- How do we get rid of the constant factors to focus on the essential part of the running time?
  - Asymptotic Analysis

Upper Bound – Big-O 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

- Examples
  - $55 = O(1)$
  - $25c + 32k = O(1)$ // if c,k are constants
  - $5n + 6 = O(n)$
  - $n^2 - 3n + 2 = O(n^2)$
  - $7n + 2n\log(5n) = O(n\log n)$

Analyzing an Algorithm

- Simple Assignment
  - `a = b`
  - `O(1)                    // Constant time complexity`

- Simple loops
  - `for (i=0; i<n; i++) { s; }`
  - `O(n)                    // Linear time complexity`

- Nested loops
  - ```
    for (i=0; i<n; i++)
        for (j=0; j<n; j++) { s; }
    ```
  - $O(n^2)$ `                  // Quadratic time complexity`
    ```

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
  - $O(\log_2 n)$ // Logarithmic time complexity
    ```

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

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

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

Exercises in Time Complexity Analysis

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

$O(n + m)$

```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

$O(n^2)$

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

$O(n \log_2 n)$

Exercises in Time Complexity Analysis

```
int a = 0, i = N;
while (i > 0) {
     a += i;
     i /= 2;
}
```

$O(\log_2 n)$

```
for(int i=0;i<n;i++){
   i*=k;
}
```

$O(\log_k n)$

```
function(int n)
{
    if (n==1)
       return;
    for (int i=1; i<=n; i++)
    {


        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

$O(n)$

Big-O Notation: Mathematical Definition

- Most commonly used notation for specifying asymptotic complexity—i.e., for estimating the rate of function growth—is the **big-O** notation introduced in 1894 by Paul Bachmann

Definition

- Given two positive-valued functions f and g:

$$f(n) = O(g(n))$$

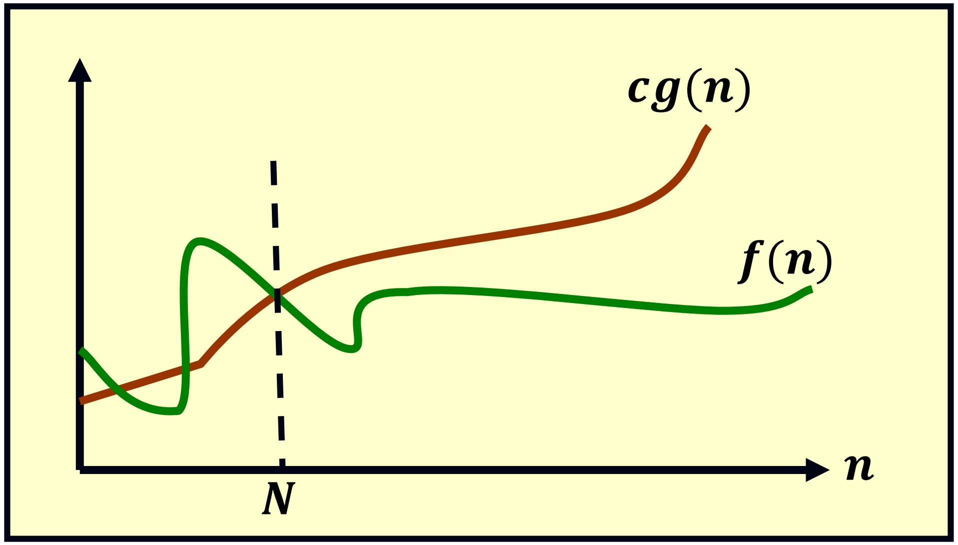
if there exist positive numbers c and N such that

$$f(n) \leq c \cdot g(n) \; for \; all \; n \geq N$$

- $\boldsymbol{f}$ is **big-O** of $\boldsymbol{g}$ if there is a positive number $\boldsymbol{c}$ such that $\boldsymbol{f}$ is not larger than $\boldsymbol{c \cdot g}$ for sufficiently large $\boldsymbol{n}$; that is, for all $\boldsymbol{n}$ larger than some number $\boldsymbol{N}$

Relationship b/w f and g

- The relationship between $\boldsymbol{f}$ and $\boldsymbol{g}$ can be expressed by stating either that $\boldsymbol{g(n)}$ is an upper bound on the value of $\boldsymbol{f(n)}$ or that, in the long run, $\boldsymbol{f}$ grows at most as fast as $\boldsymbol{g}$



$\boldsymbol{c \cdot g(n)}$ is an approximation to $\boldsymbol{f(n)}$, bounding from above

Calculating *c* and *g*

- Usually infinitely many pairs of $\boldsymbol{c}$ and $\boldsymbol{N}$ that can be given for the same pair of functions $\boldsymbol{f}$ and $\boldsymbol{g}$

$$f(n) = 2n^2 + 3n + 1 \qquad \leq cg(n) = cn^2 = O(n^2)$$

where $\boldsymbol{g}(\boldsymbol{n}) = \boldsymbol{n}^{\mathbf{2}}$, candidate values for $\boldsymbol{c}$ and $\boldsymbol{N}$ are

c	≥ 6	$\geq 3\frac{3}{4}$	$\geq 3\frac{1}{9}$	$\geq 2\frac{13}{16}$	$\geq 2\frac{16}{25}$	$\ldots$	$\rightarrow$	2
N	1	2	3	4	5	$\ldots$	$\rightarrow$	∞

Different values of c and N for function $f(n) = 2n^2 + 3n + 1 = O(n^2)$ calculated according to the definition of big-O

Calculating c and g

- We obtain these values by solving the inequality:

$$2n^2 + 3n + 1 \leq cn^2 \text{ for different } \boldsymbol{n}$$

- Because it is one inequality with two unknowns, different pairs of constants $\boldsymbol{c}$ and $\boldsymbol{N}$ for the same function $\boldsymbol{g} = \boldsymbol{n}^2$ can be determined

c	≥ 6	$\geq 3\frac{3}{4}$	$\geq 3\frac{1}{9}$	$\geq 2\frac{13}{16}$	$\geq 2\frac{16}{25}$	$\ldots$	$\rightarrow$	2
N	1	2	3	4	5	$\ldots$	$\rightarrow$	∞

Different values of c and N for function $f(n) = 2n^2 + 3n + 1 = O(n^2)$ calculated according to the definition of big-O

Calculating c and g

- To choose the best $\boldsymbol{c}$ and $\boldsymbol{N}$, it should be determined for which $\boldsymbol{N}$, a certain term in $\boldsymbol{f}$ becomes the largest and stays the largest
- In $\boldsymbol{f(n)}$, the only candidates for the largest term are $\boldsymbol{2n^2}$ and $\boldsymbol{3n}$; these terms can be compared using the inequality $\boldsymbol{2n^2 > 3n}$ that holds for $\boldsymbol{n > 1.5}$
- Thus, the chosen values are $\boldsymbol{N = 2}$ and $\boldsymbol{c \geq 3\frac{3}{4}}$

c	≥ 6	$\geq 3\frac{3}{4}$	$\geq 3\frac{1}{9}$	$\geq 2\frac{13}{16}$	$\geq 2\frac{16}{25}$	$\ldots$	$\rightarrow$	2
N	1	2	3	4	5	$\ldots$	$\rightarrow$	∞

Different values of c and N for function $f(n) = 2n^2 + 3n + 1 = O(n^2)$ calculated according to the definition of big-O

Practical Significance

- All of them are related to the same function $g(n) = n^2$ and to the same $f(n)$
- The point is that f and g grow at the same rate
- The definition states, however, that g is almost always greater than or equal to f if it is multiplied by a constant c where "almost always" means for all n not less than a constant N
- The crux of the matter is that the value of c depends on which N is chosen, and vice versa

Practical Significance

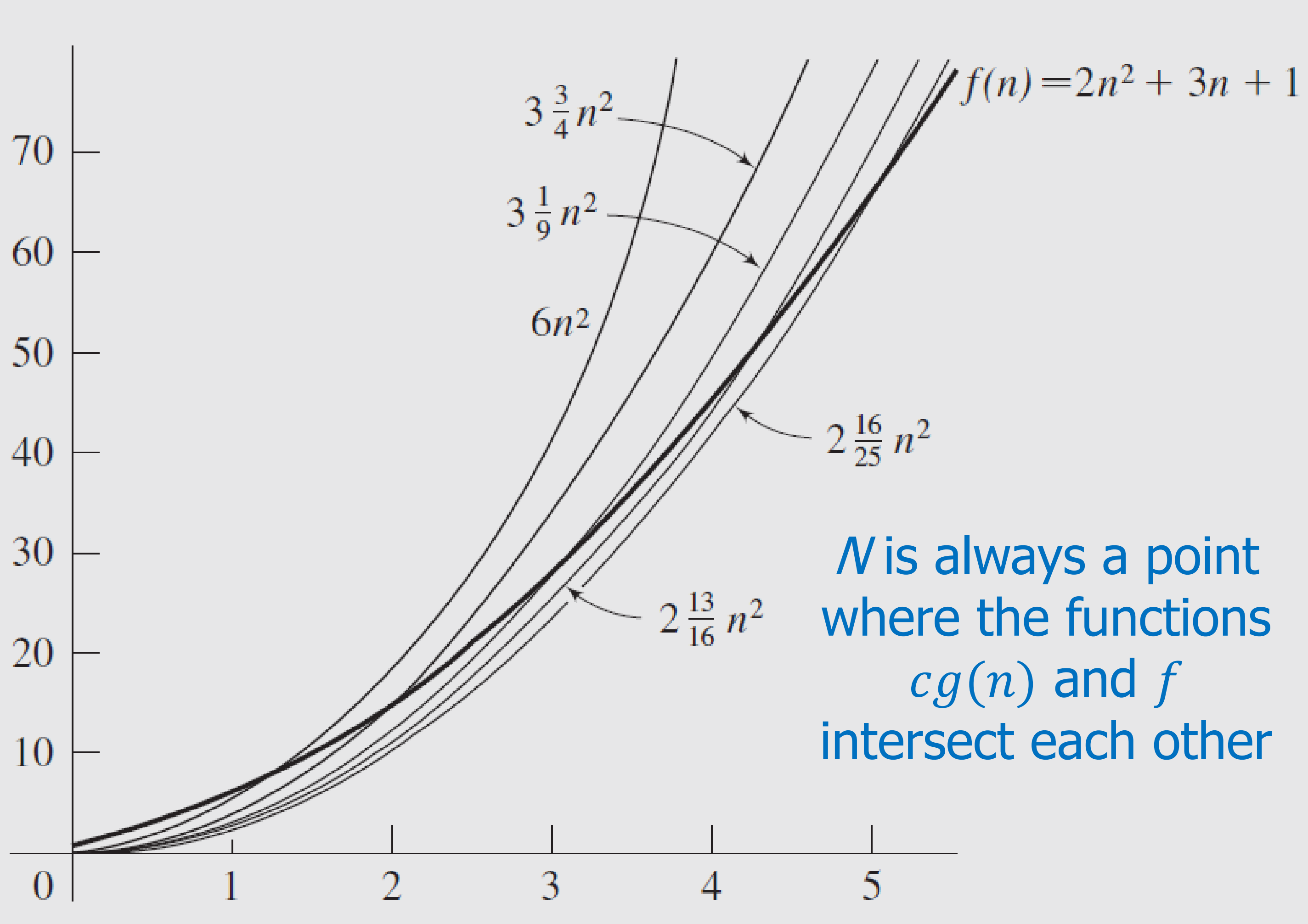
- E.g., if 1 is chosen as the value of $\boldsymbol{N}$—that is, if $\boldsymbol{g}$ is multiplied by $\boldsymbol{c}$ so that $\boldsymbol{c \cdot g(n)}$ will not be less than $\boldsymbol{f}$ right away—then $\boldsymbol{c}$ has to be equal to 6 or greater

- Or if $\boldsymbol{c \cdot g(n)}$ is greater than or equal to $\boldsymbol{f(n)}$ starting from $\boldsymbol{n = 2}$, then it is enough that $\boldsymbol{c}$ is equal to $\boldsymbol{3\frac{3}{4}}$

c	≥ 6	$\geq 3\frac{3}{4}$	$\geq 3\frac{1}{9}$	$\geq 2\frac{13}{16}$	$\geq 2\frac{16}{25}$	$\ldots$	$\rightarrow$	2
N	1	2	3	4	5	$\ldots$	$\rightarrow$	∞

Different values of c and N for function $f(n) = 2n^2 + 3n + 1 = O(n^2)$ calculated according to the definition of big-O

g(n) vs *c*



function *g* is plotted with different coefficients *c*

Best *g(n)*

- The inherent imprecision of the big-O notation goes even further, because there can be infinitely many functions $\boldsymbol{g}$ for a given function $\boldsymbol{f}$

- $\boldsymbol{f(n) = 2n^2 + 3n + 1}$ is **big-O** not only of $\boldsymbol{n^2}$, but also of $\boldsymbol{n^3, n^4, \dots, n^k}$ for any $\boldsymbol{k \geq 2}$
  - To avoid this, the smallest function $\boldsymbol{g}$ is chosen, i.e., $\boldsymbol{n^2}$ in this case

- Also, the approximation of $\boldsymbol{f}$ can go further
  - i.e., $\boldsymbol{f(n) = 2n^2 + 3n + 1}$ can be approximated as $\boldsymbol{f(n) = 2n^2 + O(n)}$
  - Or the function $\boldsymbol{n^2 + 100n + \log_{10} n + 1000}$ can be approximated as $\boldsymbol{n^2 + 100n + O(\log_{10} n)}$

Big-O Examples

- Prove that running time $\mathrm{T(n) = n^3 + 20n + 1}$ is $O(n^3)$

Proof:

- By the Big-O definition, $\mathrm{T(n)}$ is $O(n^3)$ if

$$\mathrm{T(n)} \leq c \cdot n^3 \text{ for some } n \geq N$$

- Check the condition: $\boldsymbol{n^3 + 20n + 1 \leq \mathbf{c} \cdot n^3}$

  or equivalently $\boldsymbol{1 + \frac{20}{n^2} + \frac{1}{n^3} \leq c}$

- Therefore, the Big-O condition holds for $\mathbf{n \geq N = 1}$ and $\mathbf{c \geq 22}$ (= 1 + 20 + 1)

- Larger values of $\mathbf{N}$ result in smaller factors $\mathbf{c}$ (e.g., for $\mathbf{N = 10}$, $\mathbf{c \geq 1.201}$ and so on) but in any case the above statement is valid

Big-O Examples

- Prove that running time $\mathrm{T(n) = n^3 + 20n + 1}$ is not $O(n^2)$

Proof:

- By the Big-O definition, $\mathrm{T(n)}$ is $O(n^2)$ if

  $$\mathrm{T(n)} \leq c \cdot n^2 \text{ for some } n \geq N$$

- Check the condition: $\boldsymbol{n^3 + 20n + 1 \leq \mathrm{c} \cdot n^2}$

  or equivalently $\boldsymbol{n + \frac{20}{n} + \frac{1}{n^2} \leq c}$

- Therefore, the Big-O condition cannot hold since the left side of the latter inequality is **growing infinitely**, i.e., there is no such constant factor $\boldsymbol{c}$

Big-O Examples

- Prove that running time $\mathrm{T(n) = n^3 + 20n + 1}$ is not $O(n^2)$

Conclusion:

- The left side of the inequality depends on the value of $\boldsymbol{n}$, and it is possible for the left side to be bounded by a constant $\boldsymbol{c}$ for certain ranges of $\boldsymbol{n}$. However, as $\boldsymbol{n}$ gets larger, the terms $\frac{\mathbf{20}}{\boldsymbol{n}}$ and $\frac{\mathbf{1}}{\boldsymbol{n}^{\mathbf{2}}}$ become smaller and approach zero. This means that there is a limit to how large $\boldsymbol{c}$ can be while still satisfying the inequality.
- So, conclusion is that for larger values of $\boldsymbol{n}$, the left side of the inequality becomes smaller due to the decreasing fractions, and there is no constant $\boldsymbol{c}$ that can satisfy the inequality for all $\boldsymbol{n}$.

Relatives of Big-O

- 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 $f(n) \geq c \cdot g(n)$ 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 $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation

Big-O – Upper Bound

- $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$

Big-Omega – Lower Bound

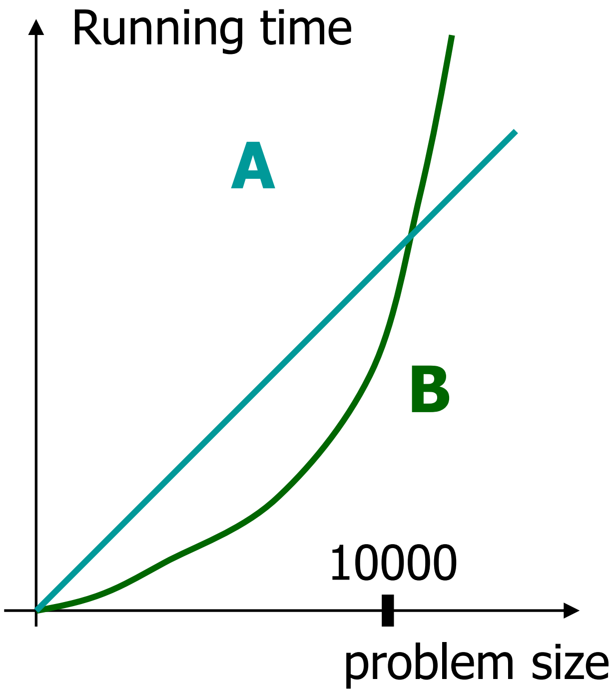
- $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically greater than or equal to $g(n)$
- **`Note:`** $f(n)$ is $\Omega(g(n))$ if and only if $g(n)$ is $O(f(n))$

Big-Theta – Exact Bound

- $f(n)$ 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 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?