CISC 235: Topic 1

Complexity of Iterative Algorithms

Outline

- Complexity Basics
- Big-Oh Notation
- Big-Ω and Big-θ Notation
- Summations
- Limitations of Big-Oh Analysis

Complexity

Complexity is the study of how the time and space to execute an algorithm vary with problem size.

The purpose is to compare alternative algorithms to solve a problem to determine which is “best”.

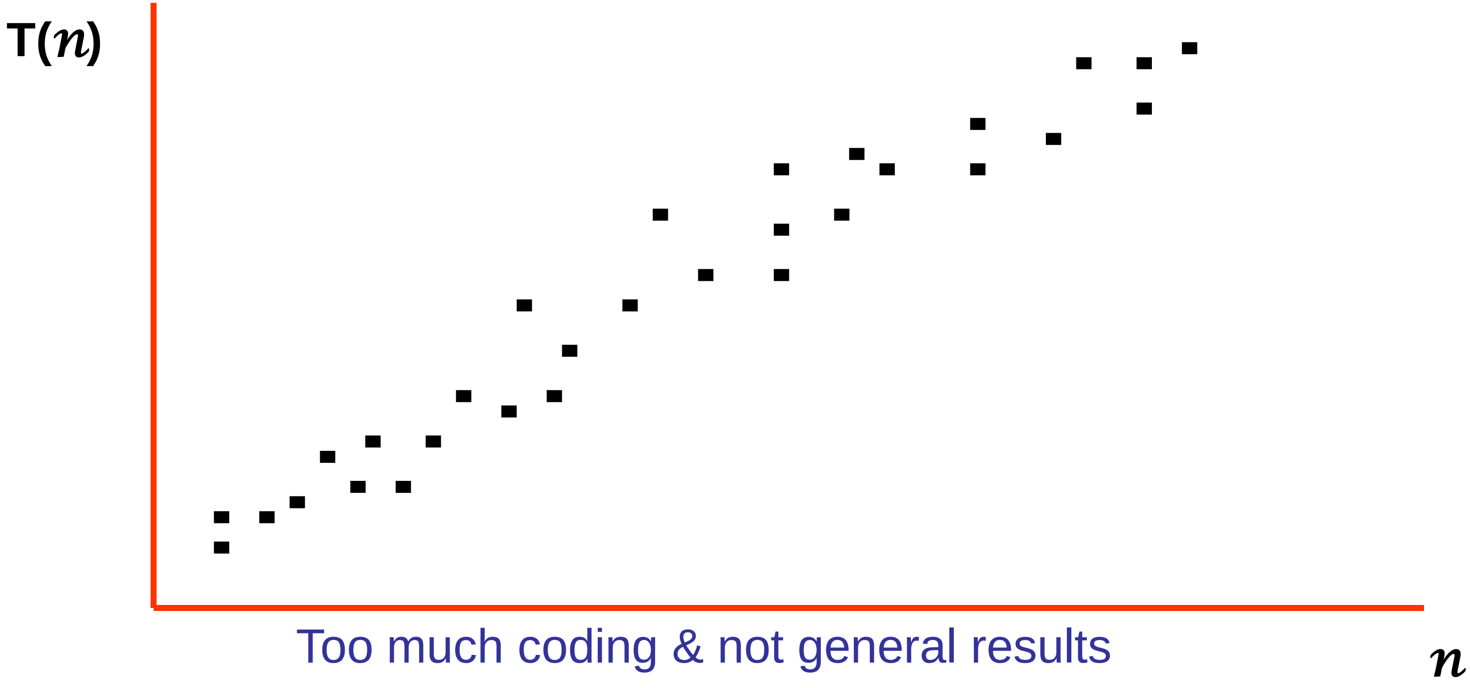
Time Complexity:

Let $\mathrm{T}(n)$ denote the time to execute an algorithm with input size n

How can we measure $\mathrm{T}(n)$?

Experimental Study

Implement the alternative algorithms and then time them with various benchmarks, measuring running time with a method like Java's System.currentTimeMillis( )



Too much coding & not general results

Mathematical Analysis

Analyze alternative algorithms mathematically prior to coding

- Define the amount of time taken by an algorithm to be a function of the size of the input data:

  $$T(n) = ?$$

- Count the key instructions that are executed to obtain the value of the function in terms of the size of the input data
- Compare algorithms by comparing how fast their running time functions grow as the input size increases

Finding an Algorithm's Running Time Function

Count the *key instructions* that are executed to obtain the running time in terms of the *size of the input data*.

Important Decisions:

- What is the measure of the size of input?
- Which are the key instructions?

Example: Find smallest value in array A of length n

```
int small = 0;
for ( int j = 0; j < n; j++ )
        if ( A[j] < A[small] )
                small = j;
```

Counting Assignments:

Line 1: 1

Line 2: n + 1

Line 3: 0

Line 4: best case: 0

worst case: n

So, $T(n) = 2n + 2$

Example: Find smallest value in array A of length n

```
int small = 0;
for ( int j = 0; j < n; j++ )
        if ( A[j] < A[small] )
                small = j;
```

Substitute constants
a & b to reduce analysis time:

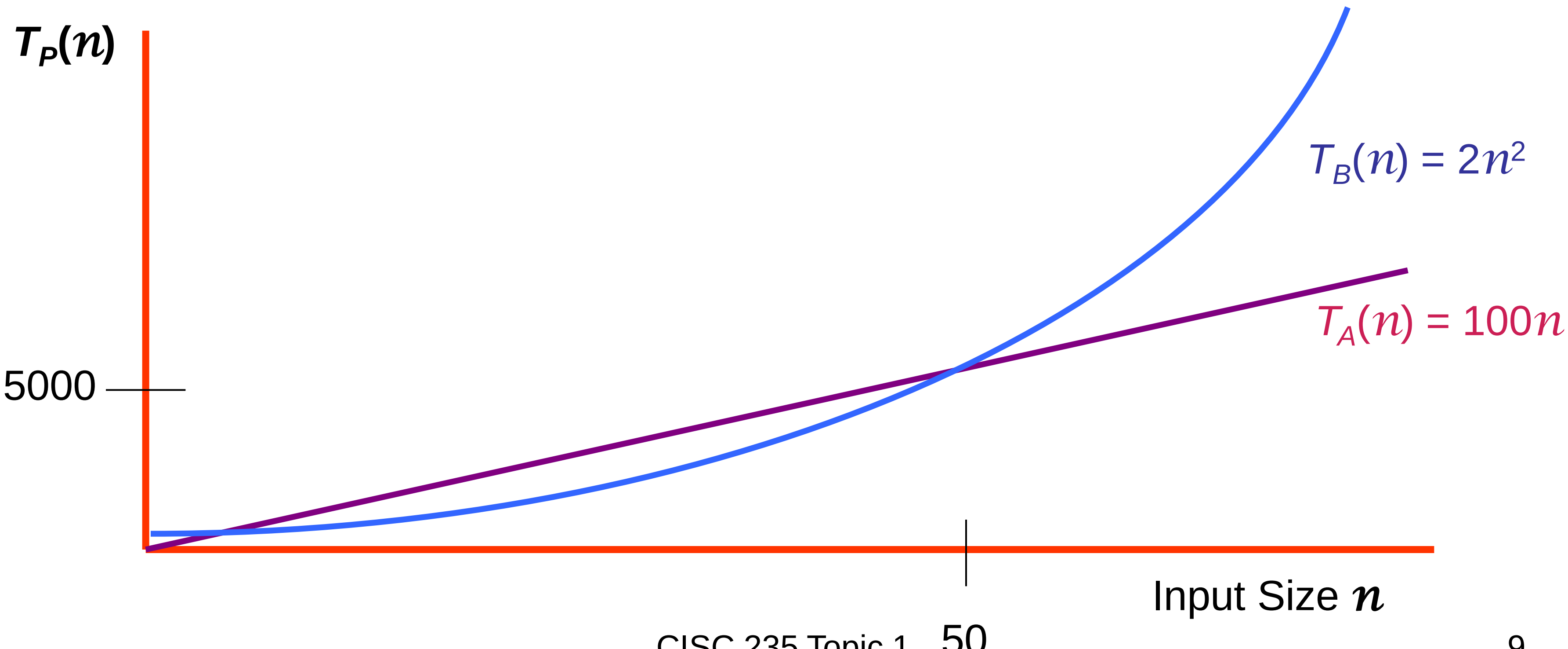
Line 1: b (constant time for everything outside loop)
Line 2: n (variable number of times through loop)
Lines 2, 3, 4: a (constant time for everything inside loop, including loop test & increment)

So, $T(n) = an + b$

For large input sizes, constant terms are insignificant

Program *A* with running time $T_A(n) = 100n$

Program *B* with running time $T_B(n) = 2n^2$



Big-Oh Notation

Purpose: Establish a relative ordering among functions by comparing their relative rates of growth

Example: $f(x) = 4x + 3$

Big-Oh Notation: $f(x) \in O(x)$

or $f(x)$ is $O(x)$

Definition of Big-Oh Notation

$f(x)$ is $O(g(x))$ if two constants C and k can be found such that:

for all $x > k$, $f(x) \le Cg(x)$

Note: Big-Oh is an upper bound

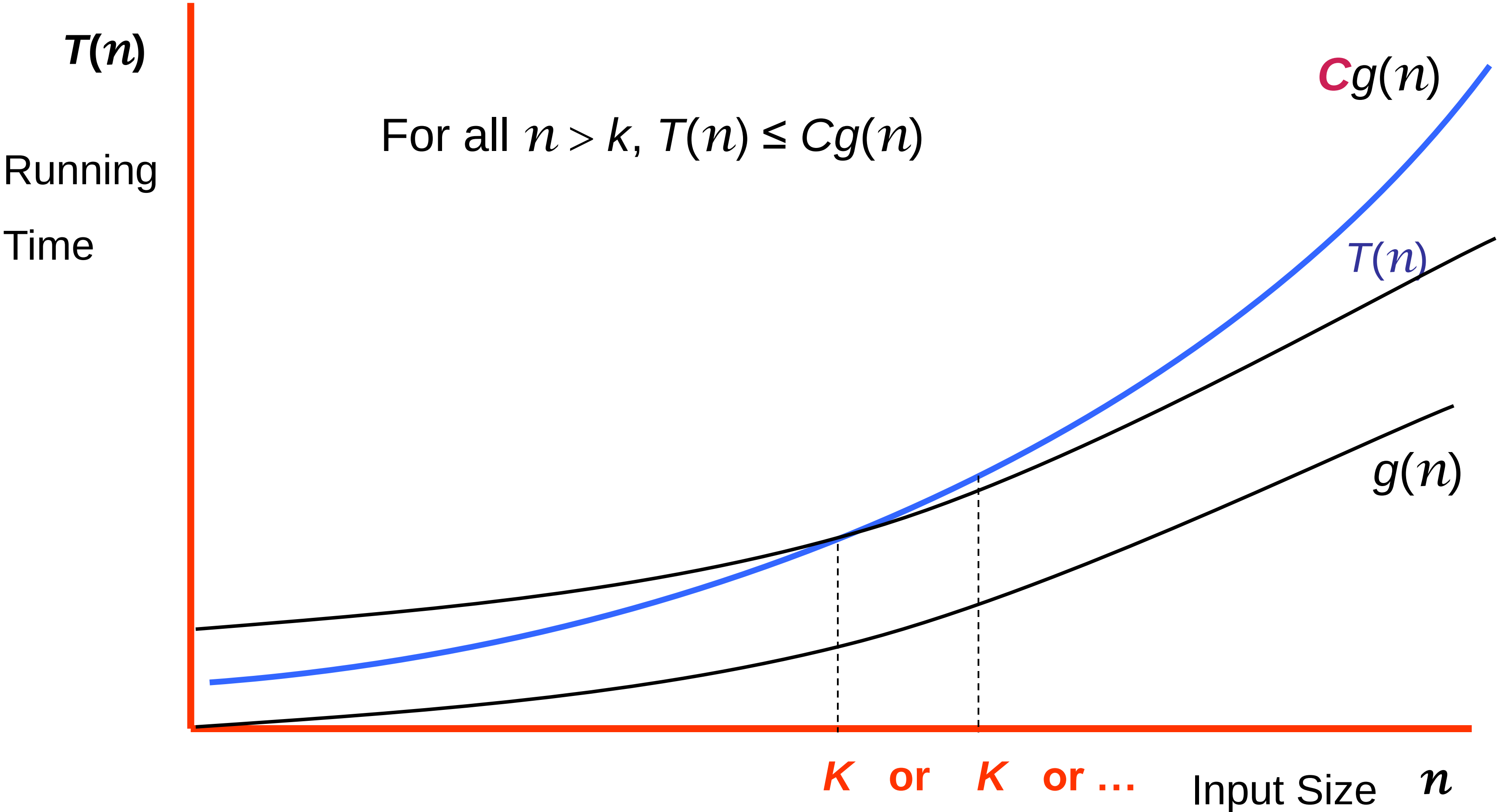
Meaning of Big-Oh Notation

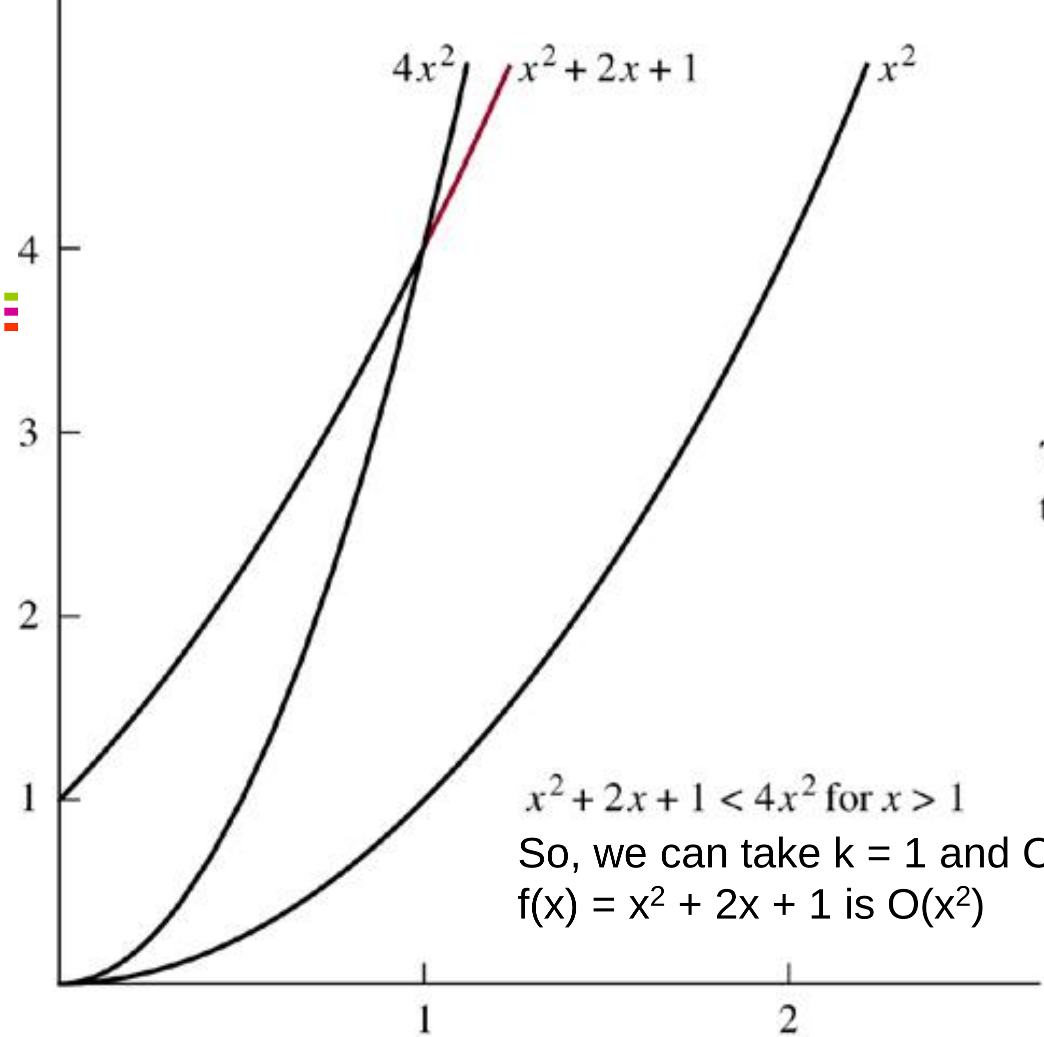
- The function $f(x)$ is one of the set of functions that has an order of magnitude growth rate $\leq$ the growth rate of $g(x)$

OR

- $f(x)$ is at most a constant times $g(x)$, except possibly for some small values of x

Graph of Big-Oh for a Program





Show that

$f(x) = x^2 + 2x + 1$

is $O(x^2)$

The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in color.

If $x > 1$, then $x^2 + 2x + 1 < x^2 + 2x^2 + x^2 = 4x^2$

If $x > 2$, then $x^2 + 2x + 1 < x^2 + x^2 + x^2 = 3x^2$

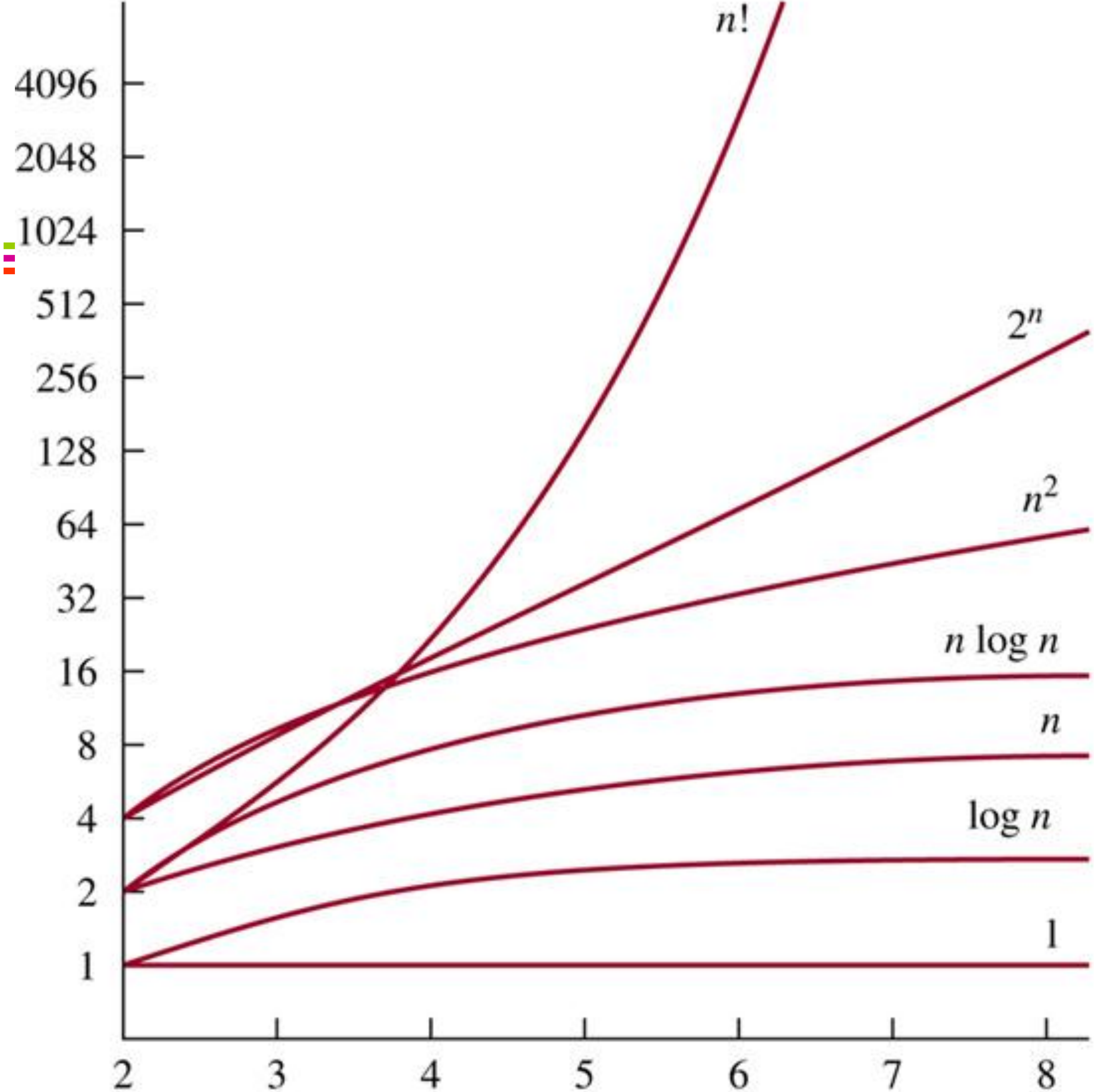
Rules for Big-Oh

- We want to find the closest upper bound
  - if $f(x) = 100x$,
    we choose $f(x)$ is $O(x)$,
    not $f(x)$ is $O(x^2)$
- Never include constants or lower-order terms within a Big-Oh (so also don't include the base of a logarithm)
  - if $f(x) = 2x^2 + x$
    we choose $f(x)$ is $O(x^2)$,
    not $f(x)$ is $O(2x^2)$ and not $f(x)$ is $O(x^2 + x)$

Order of Magnitude Growth Rates

Function	Descriptor	Big-Oh
c	Constant	$O(1)$
$\log n$	Logarithmic	$O(\log n)$
n	Linear	$O(n)$
$n \log n$	n log n	$O(n \log n)$
n^2	Quadratic	$O(n^2)$
n^3	Cubic	$O(n^3)$
n^k	Polynomial	$O(n^k)$
2^n	Exponential	$O(2^n)$
$n!$	Factorial	$O(n!)$

Order of Magnitude Growth Rates



Change to Running Times When Input Size n is Doubled?

Function	$n = 100$	$n = 200$	Change
c	c	c + ____	
$\log n$	~5	~5 + ____	
n	100	100 * ____	
n^2	100^2	100^2 * ____	
n^3	100^3	100^3 * ____	
2^n	$\sim 100^{15}$	$\sim 100^{15}$ * ____	

Growth of Combinations of Functions

If $f_1(x)$ is O($g_1(x)$) and $f_2(x)$ is O($g_2(x)$),
Then

$$(f_1 + f_2)(x) \text{ is } O(\max(g_1(x), g_2(x)))$$

and

$$(f_1 f_2)(x) \text{ is } O(g_1(x) g_2(x))$$

Deriving Big-Oh of Functions

$f(x)$	Big-Oh
$3x^2 + 5$	$O(x^2)$
$2x^3 - x^2 - 6$	$O(x^3)$
$\log_2 x + x$	$O(x)$
$(5 + \log_2 x)(3x - 3)$	$O(x \log x)$
$4(2^x - x^3)$	$O(2^x)$
$4c - 6d + 17a$	$O(1)$

Big-Omega (Big-Ω) Notation

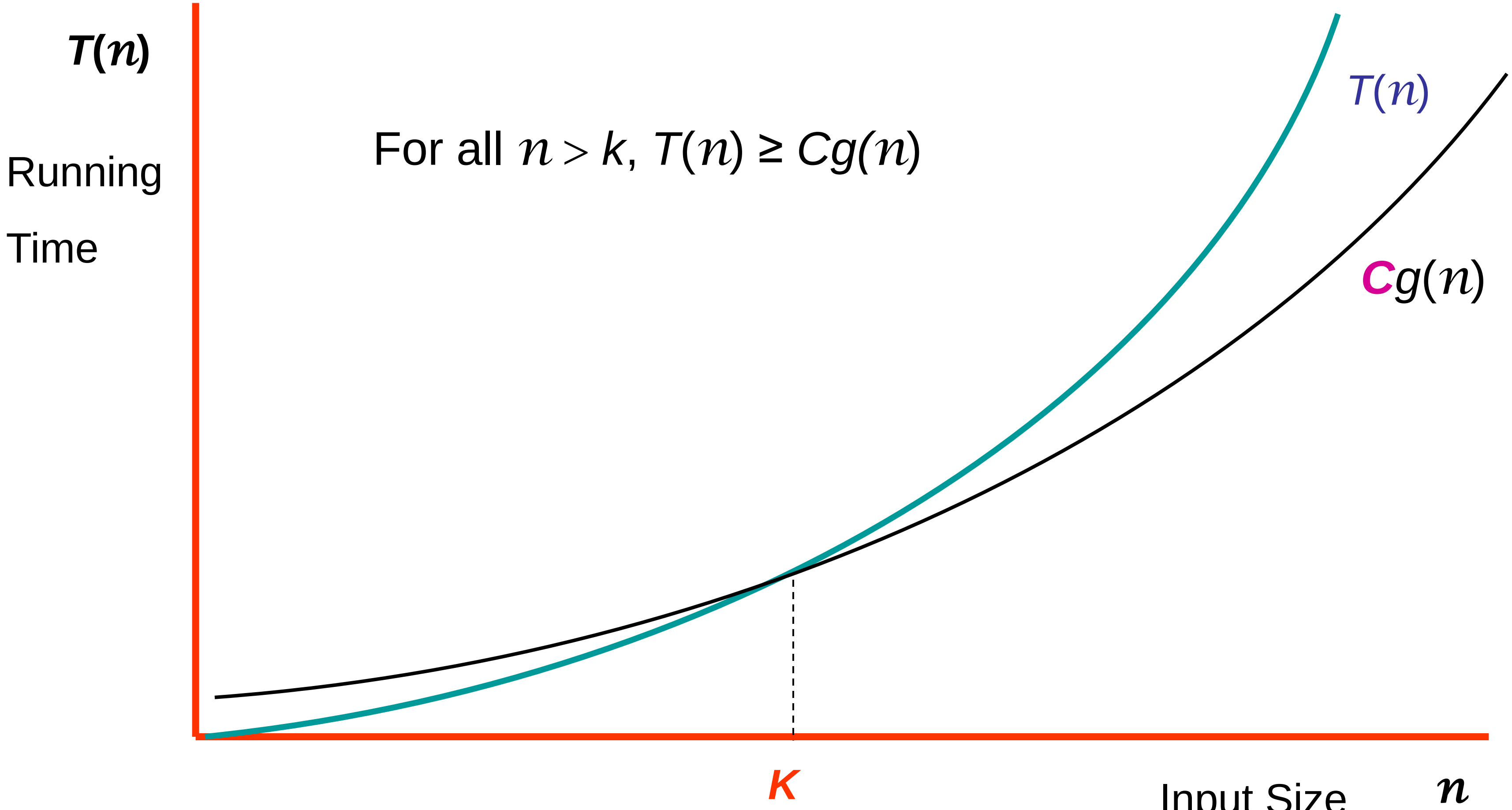
Expresses a lower-bound of a function

$f(x)$ is $\Omega(g(x))$ if $f(x)$ is at least a constant times $g(x)$, except possibly for some small values of x.

Formally:

$f(x)$ is $\Omega(g(x))$ if two constants C and k can be found such that for all $x > k$, $f(x) \geq Cg(x)$

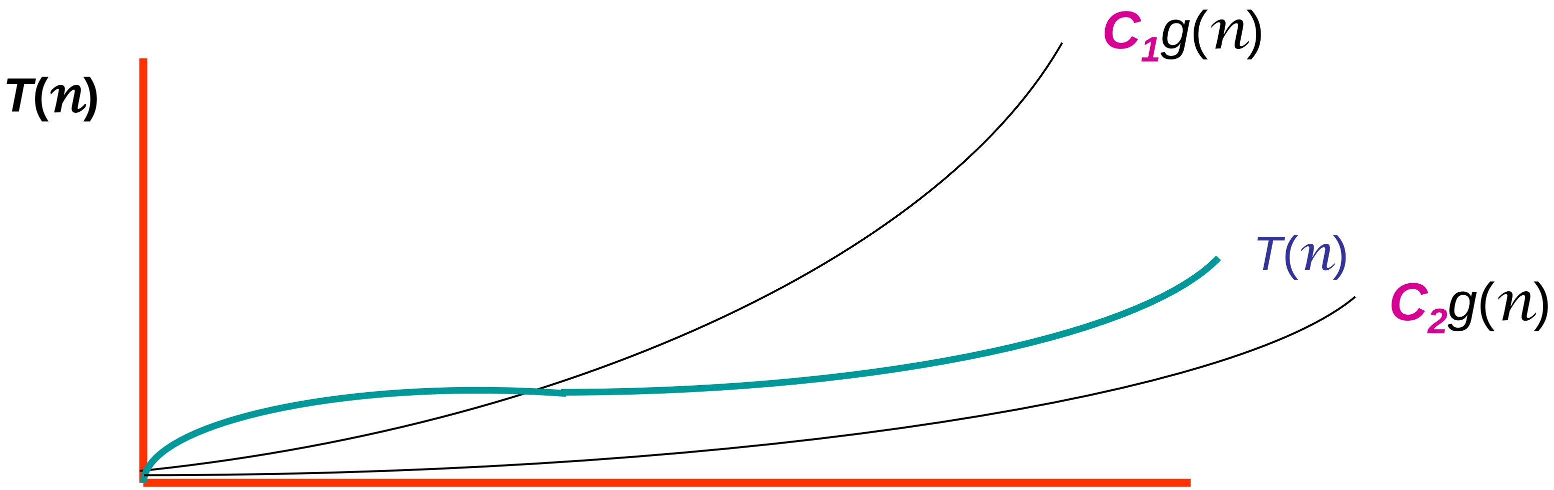
Graph of Big-Ω for a Program



Big-Theta (Big-Θ) Notation

Expresses a tight bound of a function

$f(x)$ is $\Theta(g(x))$ if $f(x)$ is both $O(g(x))$ and $\Omega(g(x))$



Example

Input: An n+1 element array A of coefficients and x, a number

Output: pVal, the value of the polynomial

$$A[0] + A[1]x + A[2]x^2 + \ldots + A[n]x^n$$

Algorithm 1

```
pVal = A[0]
for (int i = 1; i <= n; i++)
        pVal = pVal + A[i] * Math.pow(x,i);
```

What is the Big-Oh analysis of this algorithm?

Algorithm 2

```
pVal = A[0]
for ( int i = 1; i <= n; i++ )
{
    powX = 1
    for ( int j = 0; j < i; j++ )
            powX = x * powX
    pVal = pVal + A[i] * powX
}
```

What is the Big-Oh analysis of this algorithm?

Algorithm 3

Horner's method:

To evaluate $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$, use:

$P(x) = a_0 + (x\,(a_1 + x(a_2 + \ldots + x(a_{n-1} + xa_n) \ldots)))$

```
pVal = x * A[n]
for (int i = n -1; i > 0; i - -)
     pVal = x * ( A[i] + pVal )
pVal = pVal + A[0]
```

What is the Big-Oh analysis of this algorithm?

Example: Selection Sort

```
for ( int i = 0; i < n-1; i++ )
{
        int small = i;
        for ( int j = i + 1; j < n; j++ )
              if ( A[j] < A[small] )
                    small = j;
        int temp = A[small];
        A[small] = A[i];
        A[i] = temp;
}
```

Summation Notation

$$\sum_{i=m}^{n} a_i$$

Represents $a_m + a_{m+1} + \ldots + a_n$

TABLE 2 Some Useful Summation Formulae.

Sum	*Closed Form*
$\sum_{k=0}^{n} ar^k \; (r \neq 0)$	$\frac{ar^{n+1} - a}{r - 1}, r \neq 1$
$\sum_{k=1}^{n} k$	$\frac{n(n+1)}{2}$
$\sum_{k=1}^{n} k^2$	$\frac{n(n+1)(2n+1)}{6}$
$\sum_{k=1}^{n} k^3$	$\frac{n^2(n+1)^2}{4}$
$\sum_{k=0}^{\infty} x^k, \lvert x \rvert < 1$	$\frac{1}{1-x}$
$\sum_{k=1}^{\infty}, kx^{k-1}, \lvert x \rvert < 1$	$\frac{1}{(1-x)^2}$

Summation Manipulations

1. Take out constant terms:

$$\sum_{k=1}^{n} k/2 = ½ \sum_{k=1}^{n} k$$

2. Decompose large terms:

$$\sum_{k=1}^{n} n-k+k^2 = \sum_{k=1}^{n} n - \sum_{k=1}^{n} k + \sum_{k=1}^{n} k^2$$

Summation Manipulations

3. Partial Sums:

$$\sum_{k=j+1}^{n} k = \sum_{k=1}^{n} k - \sum_{k=1}^{j} k$$

4. Practice:

$$\sum_{j=i}^{n} kj + a = \underline{\hspace{10cm}}$$

Number of Terms in a Summation

Upper Bound – Lower Bound + 1

$$\sum_{k=2}^{n} 1 = 1 + 1 + \ldots + 1 = n - 2 + 1 = n - 1$$

$$\sum_{k=j+1}^{n-1} 1 = 1 + 1 + \ldots + 1 = (n-1) - (j+1) + 1$$

$$= n - j - 1$$

Note: This is equivalent to calculating the number of iterations in a for loop

Calculating Arithmetic Summations

((First Term + Last Term) * Number of Terms) / 2

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = (1+n) * n/2 = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n-2} n - i - 1 = (n-1) + (n-2) + \ldots + 1$$

$$= ((n-1) + 1) * (n-1))/2$$

$$= \frac{n(n-1)}{2}$$

Example: Max Subsequence Sum

```
int maxSum = 0;
int besti = 0;   int bestj = 0;
for ( int i = 0; i < n; i++ )
        for ( int j = i; j < n; j++ )
        {       int thisSum = 0;
                for ( int k = i; k <= j; k++ )
                        thisSum = thisSum + A[k];
                if ( thisSum > maxSum )
                {       maxSum = thisSum;
                        besti = i; bestj = j;
                }
         }
```

Analyzing Complexity of Lists

Operation	Sorted Array	Sorted Linked List	Unsorted Array	Unsorted Linked List
Search(L, x)				
Insert(L, x)				
Delete(L, x)				

Limitations of Big-Oh Analysis

- Not appropriate for small amounts of input
  - Use the simplest algorithm
- Large constants can affect which algorithm is more efficient
  - e.g., 2nlogn versus 1000n
- Assumption that all basic operations take 1 time unit is faulty
  - e.g., memory access versus disk access
    (1000s of time more)
- Big-Oh can give serious over-estimate
  - e.g., loop inside an if statement that seldom executes