Runtime Complexity

Science

Computer Science

CS 331: Data Structures and Algorithms

Michael Lee <lee@iit.edu>

ILLINOIS INSTITUTE OF TECHNOLOGY
College of Science

So far, our runtime analysis has been based on *empirical data*

— i.e., runtimes obtained from actually running our algorithms

This data is very sensitive to:

- platform (OS/compiler/interpreter)
- concurrent tasks
- implementation details (vs. high-level algorithm)

Also, doesn’t always help us see *long-term / big picture trends*

Reframing the problem:

Given an algorithm that takes *input size* $\boldsymbol{n}$, find a function $\boldsymbol{T}(\boldsymbol{n})$ that describes the *runtime* of the algorithm *input size* might be:

- the *magnitude of the input value* (e.g., for numeric input)
- the *number of items* in the input (e.g., as in a list)

An algorithm may also be dependent on *more than one input*.

```python
def sort(vals):
    # input size = len(vals)


def factorial(n):
    # input size = n


def gcd(m, n):
    # input size = (m, n)
```

fundamentally, runtime is determined by the *primitive operations* carried out during execution of the algorithm (in compiled code, by the interpreter, etc.)

E.g., factorial

		cost	times
`def factorial(n):`			
`    prod = 1`	....	c_1	1
`    for k in range(2, n+1):`	....	c_2	$n-1$
`        prod *= k`	....	c_3	$n-1$
`    return prod`	....	c_4	1

$$T(n) = c_1 + (n-1)(c_2 + c_3) + c_4$$

Messy! Per-instruction costs are machine specific, and obscure big picture runtime trends.

```python
def factorial(n):
    prod = 1 ...................................
    for k in range(2, n+1): .......
        prod *= k ........................
    return prod ...........................
```

	times
`prod = 1`	1
`for k in range(2, n+1):`	$n-1$
`prod *= k`	$n-1$
`return prod`	1

$$T(n) = 2(n-1) + 2 = 2n$$

Simplification #1: ignore actual cost of each line of code.

Easy to see that runtime is *linear* w.r.t. input size.

E.g., insertion sort

```python
def insertion_sort(lst):
    for i in range(1, len(lst)):
        for j in range(i, 0, -1):
            if lst[j] < lst[j-1]:
                lst[j], lst[j-1] = lst[j-1], lst[j]
            else:
                break
```

i

init: [5, 2, 3, 1, 4]

j

insertion: [2, 3, 5, 1, 4]

Code	*times*
`def insertion_sort(lst):`	
`    for i in range(1, len(lst)):`	$n-1$
`        for j in range(i, 0, -1):`	?
`            if lst[j] < lst[j-1]:`	?
`                lst[j], lst[j-1] = lst[j-1], lst[j]`	?
`            else:`	?
`                break`	?

?'s will vary based on initial "sortedness"

… useful to contemplate *worst case scenario*

```
def insertion_sort(lst):
    for i in range(1, len(lst)):  ...........................  n − 1
        for j in range(i, 0, -1):  ..........................  ?
            if lst[j] < lst[j-1]:  ..........................  ?
                lst[j], lst[j-1] = lst[j-1], lst[j]  ........  ?
            else:  ..........................................  ?
                break  ......................................  ?
```

times: $n-1$, ?, ?, ?, ?, ?

worst case arises when list values start out in *reverse order*!

```python
def insertion_sort(lst):
    for i in range(1, len(lst)):
        for j in range(i, 0, -1):
            if lst[j] < lst[j-1]:
                lst[j], lst[j-1] = lst[j-1], lst[j]
            else:
                break
```

Line	*times*
`for i in range(1, len(lst)):`	$n - 1$
`for j in range(i, 0, -1):`	$1, 2, ..., (n - 1)$
`if lst[j] < lst[j-1]:`	$1, 2, ..., (n - 1)$
`lst[j], lst[j-1] = lst[j-1], lst[j]`	$1, 2, ..., (n - 1)$
`else:`	0
`break`	0

worst case analysis is our default mode of analysis hereafter unless otherwise noted

Recall: *arithmetic series*

e.g., $1+2+3+4+5 = 15$

Sum can also be found by:

- adding first and last term ($1+5=6$)
- dividing by two (to find average) ($6/2=3$)
- multiplying by num of values ($3 \times 5=15$)

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

and $1+2+\cdots+(n-1)=\sum_{t=1}^{n-1} t=\frac{(n-1)n}{2}$

```python
def insertion_sort(lst):
    for i in range(1, len(lst)):
        for j in range(i, 0, -1):
            if lst[j] < lst[j-1]:
                lst[j], lst[j-1] = lst[j-1], lst[j]
            else:
                break
```

Line	*times*
`for i in range(1, len(lst)):`	$n-1$
`for j in range(i, 0, -1):`	$1, 2, ..., (n-1)$
`if lst[j] < lst[j-1]:`	$1, 2, ..., (n-1)$
`lst[j], lst[j-1] = lst[j-1], lst[j]`	$1, 2, ..., (n-1)$
`else:`	0
`break`	0

```python
def insertion_sort(lst):
    for i in range(1, len(lst)):
        for j in range(i, 0, -1):
            if lst[j] < lst[j-1]:
                lst[j], lst[j-1] = lst[j-1], lst[j]
            else:
                break
```

Line	*times*
`for i in range(1, len(lst)):`	$n-1$
`for j in range(i, 0, -1):`	$\sum_{t=1}^{n-1} t$
`if lst[j] < lst[j-1]:`	$\sum_{t=1}^{n-1} t$
`lst[j], lst[j-1] = lst[j-1], lst[j]`	$\sum_{t=1}^{n-1} t$
`else:`	0
`break`	0

```python
def insertion_sort(lst):
    for i in range(1, len(lst)):
        for j in range(i, 0, -1):
            if lst[j] < lst[j-1]:
                lst[j], lst[j-1] = lst[j-1], lst[j]
            else:
                break
```

Line	*times*
`for i in range(1, len(lst)):`	$n-1$
`for j in range(i, 0, -1):`	$(n-1)n/2$
`if lst[j] < lst[j-1]:`	$(n-1)n/2$
`lst[j], lst[j-1] = lst[j-1], lst[j]`	$(n-1)n/2$
`else:`	0
`break`	0

$$T(n) = (n-1) + \frac{3(n-1)n}{2}$$

$$= \frac{2n - 2 + 3n^2 - 3n}{2} = \frac{3}{2}n^2 - \frac{n}{2} - 1$$

$$T(n) = \frac{3}{2}n^2 - \frac{n}{2} - 1$$

i.e., runtime of insertion sort is a *quadratic function* of its input size.

$$T(n) = \frac{3}{2}n^2 - \frac{n}{2} - 1$$

Simplification #2: only consider *leading term*; i.e., with the *highest order of growth*

$$T(n) = \cancel{\frac{3}{2}} n^2 - \frac{n}{2} - 1$$

Simplification #3: *ignore constant coefficients*

$$T(n) = \frac{3}{2}n^2 - \frac{n}{2} - 1$$

we use the notation $T(n) = O(n^2)$ [read: $T(n)$ is big-oh of n^2]

to indicate that n^2 describes the *asymptotic worst-case runtime* behavior of the insertion sort algorithm, when run on input size n

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

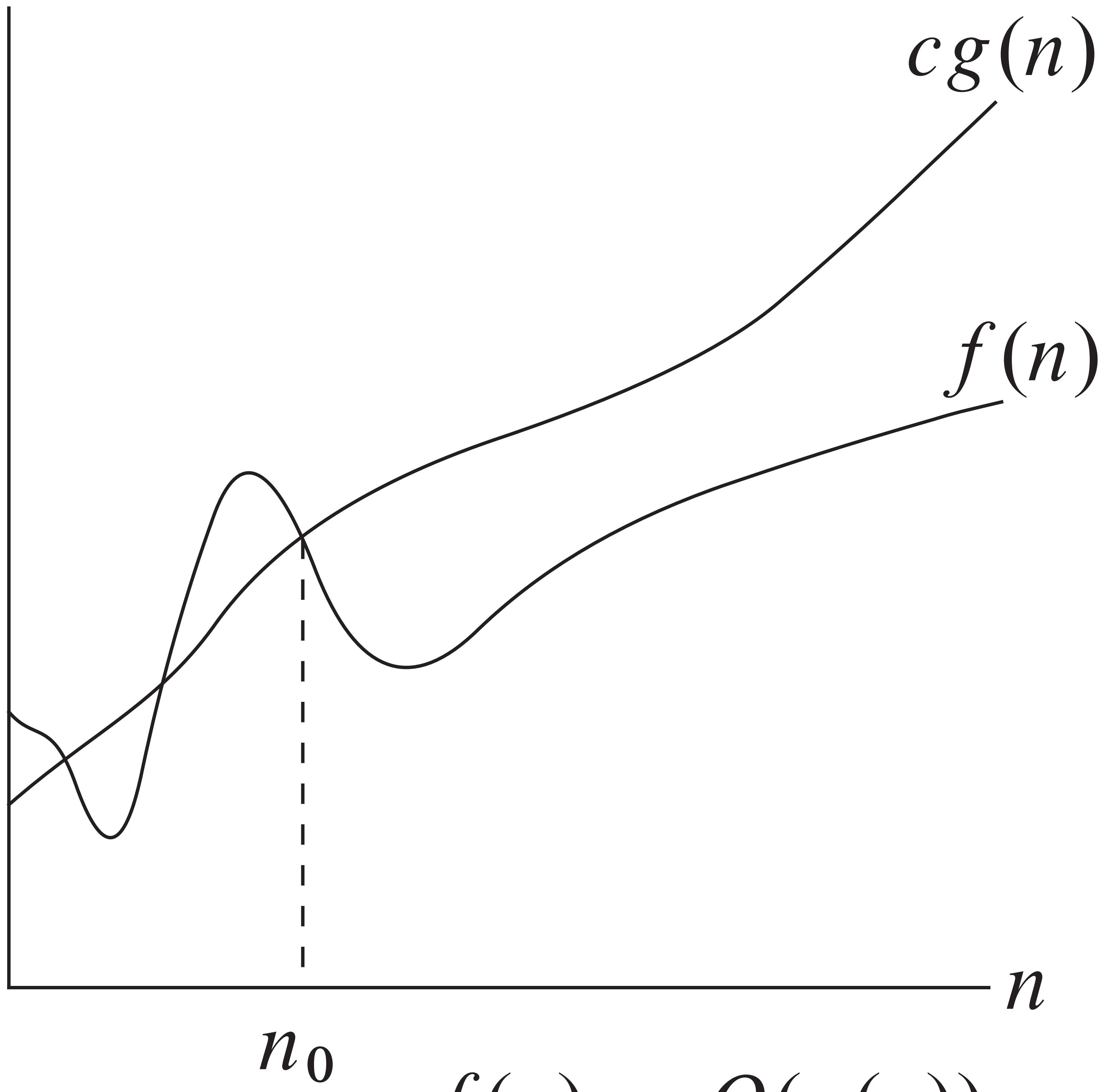
means that there exists constants c, n_0

such that $0 \leq f(n) \leq c \cdot g(n)$

for all $n \geq n_0$

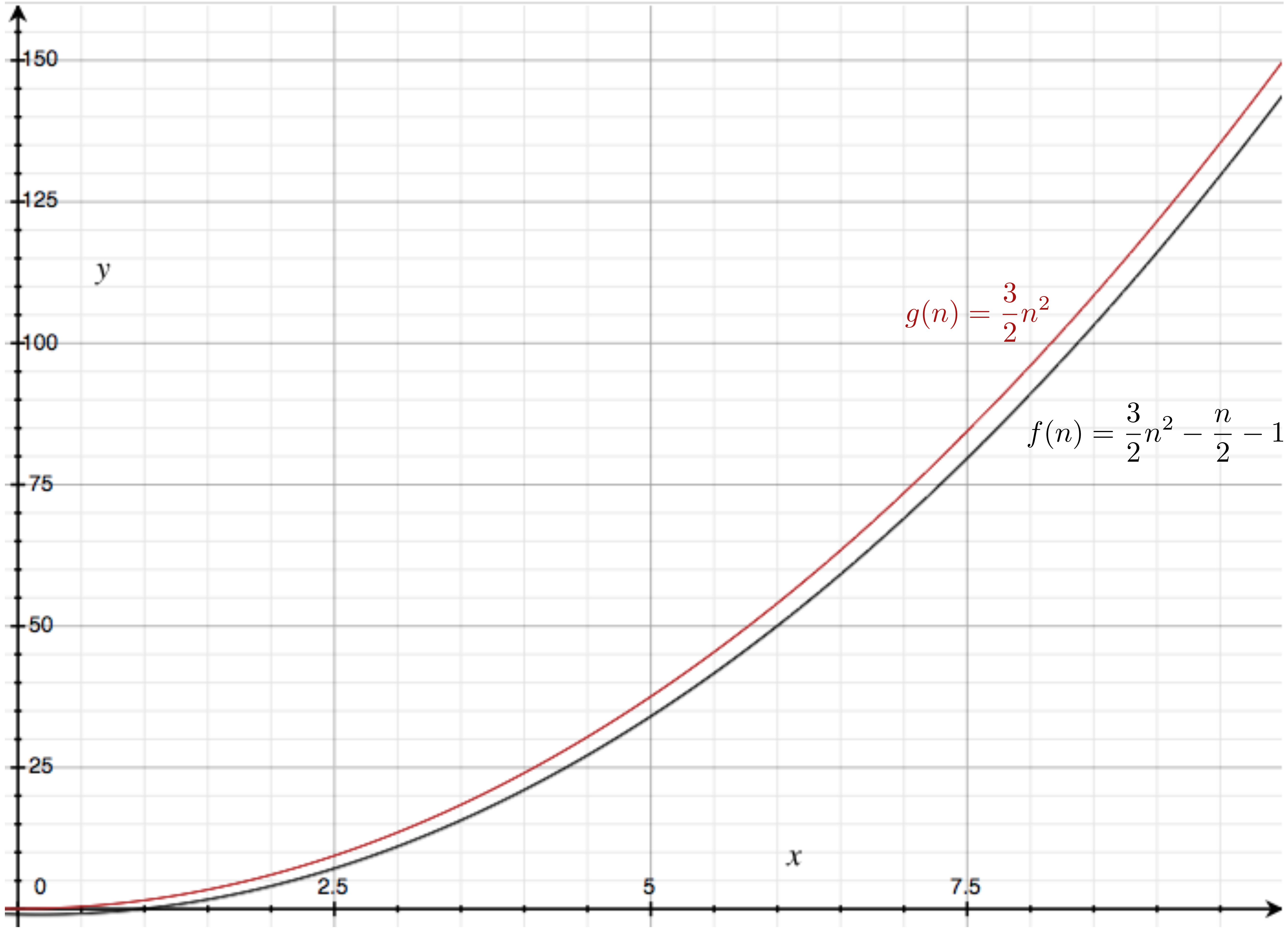
i.e., $f(n) = O(g(n))$

intuitively means that g (multiplied by a constant factor) sets an *upper bound* on f as n gets large — i.e., an *asymptotic bound*



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

(from Cormen, Leiserson, Riest, and Stein, Introduction to Algorithms)



ILLINOIS INSTITUTE OF TECHNOLOGY
College of Science

technically, $f = O(g)$ does not imply a *tight bound*

e.g., $n = O(n^2)$ is true, but there is no constant c such that $c \cdot n^2$ will approximate the growth of n, as n gets large

but we will generally try to find the tightest bounding function g

E.g., binary search

length $\Rightarrow N$

```python
def contains(lst, x):
    lo = 0
    hi = len(lst) - 1
    while lo <= hi:
        mid = (lo+hi) // 2
        if x < lst[mid]:
            hi = mid - 1
        elif x > lst[mid]:
            lo = mid + 1
        else:
            return True
    else:
        return False
```

iterations $= O(?)$

constant time

E.g., binary search

length ⇒ N

```
def contains(lst, x):
    lo = 0
    hi = len(lst) - 1
    while lo <= hi:
        mid = (lo+hi) // 2
        if x < lst[mid]:
            hi = mid - 1
        elif x > lst[mid]:
            lo = mid + 1
        else:
            return True
    else:
        return False
```

iterations = $O(?)$

reduces search-space by ½

worst-case: `x < min(lst)`

E.g., binary search

length $\Rightarrow N$

```
def contains(lst, x):
    lo = 0
    hi = len(lst) - 1
    while lo <= hi:
        mid = (lo+hi) // 2
        if x < lst[mid]:
            hi = mid - 1
        elif x > lst[mid]:
            lo = mid + 1
        else:
            return True
    else:
        return False
```

iterations $\approx$ # times we can divide length until = 1

E.g., binary search

length = 1024

```
def contains(lst, x):
    lo = 0
    hi = len(lst) - 1
    while lo <= hi:
        mid = (lo+hi) // 2
        if x < lst[mid]:
            hi = mid - 1
        elif x > lst[mid]:
            lo = mid + 1
```

iterations ≈ # times we can divide length until = 1

Iteration	0	1	2	3	4	5	6	7	8	9	10
Elements remaining	1024	512	256	128	64	32	16	8	4	2	1

$$\text{length} = N$$

$$\#\ \text{iterations} \approx \#\ \text{times we can divide length until} = 1$$

$$\approx \log_2 N$$

$$= O(\log_2 N)$$

$$[\ \text{recall: } \log_a x = \log_b x \ / \ \log_b a\ ]$$

$$= O(\log N)$$

$$1 = N \ / \ 2^x$$

$$2^x = N$$

$$\log_2 2^x = \log_2 N$$

$$x = \log_2 N$$

E.g., binary search

length $\Rightarrow N$

```python
def contains(lst, x):
    lo = 0
    hi = len(lst) - 1
    while lo <= hi:
        mid = (lo+hi) // 2
        if x < lst[mid]:
            hi = mid - 1
        elif x > lst[mid]:
            lo = mid + 1
        else:
            return True
    else:
        return False
```

iterations $= O(\log N)$

constant time

$\textbf{binary-search}(N) = O(\log N)$

So far:

- linear search = $O(n)$
- insertion sort = $O(n^2)$
- binary search = $O(\log\ n)$

```python
def quadratic_roots(a, b, c):
    discr = b**2 - 4*a*c
    if discr < 0:
        return None
    discr = math.sqrt(discr)
    return (-b+discr)/(2*a), (-b-discr)/(2*a)
```

$$= O(?)$$

```python
def quadratic_roots(a, b, c):
    discr = b**2 - 4*a*c
    if discr < 0:
        return None
    discr = math.sqrt(discr)
    return (-b+discr)/(2*a), (-b-discr)/(2*a)
```

Always a *fixed (constant) number* of LOC executed, regardless of input.

$$= O(?)$$

```python
def quadratic_roots(a, b, c):
    discr = b**2 - 4*a*c
    if discr < 0:
        return None
    discr = math.sqrt(discr)
    return (-b+discr)/(2*a), (-b-discr)/(2*a)
```

Always a *fixed (constant) number* of LOC executed, regardless of input.

$$T(n) = C = O(1)$$

```python
def foo(m, n):
    for _ in range(m):
        for _ in range(n):
            pass
```

$$= O(?)$$

```python
def foo(m, n):
    for _ in range(m):
        for _ in range(n):
            pass
```

$$= O(m \times n)$$

```python
def foo(n):
    for _ in range(n):
        for _ in range(n):
            for _ in range(n):
                pass
```

$$= O(?)$$

```python
def foo(n):
    for _ in range(n):
        for _ in range(n):
            for _ in range(n):
                pass
```

$$= O(n^3)$$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix}$$

$$c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + \cdots + a_{in}b_{nj}$$

i.e., for $n \times n$ input matrices, each result cell requires n multiplications

```python
def square_matrix_multiply(a, b):
    dim = len(a)
    c = [[0] * dim for _ in range(dim)]
    for row in range(dim):
        for col in range(dim):
            for i in range(dim):
                c[row][col] += a[row][i] * b[i][col]
    return c
```

$$= O(dim^3)$$

using “brute force” to crack an n-bit password $= O(?)$

$$\begin{array}{c}\text{1 character (8 bits)} \\ (2^8 \text{ possible values})\end{array} \left\{ \begin{array}{l} \texttt{00000000} \\ \texttt{00000001} \\ \texttt{00000010} \\ \texttt{00000011} \\ \texttt{00000100} \\ \texttt{00000101} \\ \texttt{00000110} \\ \texttt{00000111} \\ \texttt{00001000} \\ \texttt{00001001} \\ \texttt{00001010} \\ \texttt{00001011} \\ \texttt{00001100} \\ \texttt{00001101} \\ \texttt{00001110} \\ \texttt{...} \\ \texttt{11110010} \\ \texttt{11110011} \\ \texttt{11110100} \\ \texttt{11110101} \\ \texttt{11110110} \\ \texttt{11110111} \\ \texttt{11111000} \\ \texttt{11111001} \\ \texttt{11111010} \\ \texttt{11111011} \\ \texttt{11111100} \\ \texttt{11111101} \\ \texttt{11111110} \\ \texttt{11111111} \end{array} \right. = O(?)$$

using “brute force” to crack an n-bit password $= O(2^n)$

Name	Class	Example
Constant	$O(1)$	Compute discriminant
Logarithmic	$O(\log\ n)$	Binary search
Linear	$O(n)$	Linear search
Linearithmic	$O(n\ \log\ n)$	Heap sort
Quadratic	$O(n^2)$	Insertion sort
Cubic	$O(n^3)$	Matrix multiplication
Polynomial	$O(n^c)$	Generally, c nested loops over n items
Exponential	$O(c^n)$	Brute forcing an n-bit password
Factorial	$O(n!)$	"Traveling salesman" problem

Common order of growth classes

Input size	Orders of growth								
N	1	log N	N	N log N	N^2	N^10	2^N	N!	N^N
2	1	1	2	2	4	1,024	4	2	4
3	1	2	3	5	9	59,049	8	6	27
4	1	2	4	8	16	1,048,576	16	24	256
5	1	2	5	12	25	9,765,625	32	120	3,125
10	1	3	10	33	100	1.00E+10	1,024	3,628,800	1.00E+10
25	1	5	25	116	625	9.54E+13	33,554,432	1.55E+25	8.88E+34
50	1	6	50	282	2,500	9.77E+16	1.13E+15	3.04E+64	8.88E+84
75	1	6	75	467	5,625	5.63E+18	3.78E+22	2.48E+109	4.26E+140
100	1	7	100	664	10,000	1.00E+20	1.27E+30	9.33E+157	1.00E+200
200	1	8	200	1,529	40,000	1.02E+23	1.61E+60	7.88E+374	1.60E+460
500	1	9	500	4,483	250,000	9.77E+26	3.27E+150	1.22E+1134	3.05E+1349
1,000	1	10	1,000	9,966	1,000,000	1.00E+30	1.07E+301	4.02E+2567	1.00E+3000
10,000	1	13	10,000	132,877	100,000,000	1.00E+40	-	-	-
100,000	1	17	100,000	1,660,964	1E+10	1.00E+50	-	-	-
1,000,000	1	20	1,000,000	19,931,569	1E+12	1.00E+60	-	-	-