Runtime Complexity

Science

Computer Science

CS 331: Data Structures and Algorithms

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

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

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

But measured runtime is very sensitive to:

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

And measured runtime doesn’t always help us see *long-term / big picture trends*

Reframing the problem:

Given an algorithm that takes *input size* ***n***, we want a function ***T***(***n***) that describes the *running time* of the algorithm *input size* might be the *number of items* in the input (e.g., as in a list), or the *magnitude of the input value* (e.g., for numeric input).

an algorithm may also be dependent on the size of *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)
```

running time is based on # *of primitive operations* (e.g., statements, computations) carried out by the algorithm.

ideally, machine independent!

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

cost	times
c_1	1
c_2	$n-1$
c_3	$n-1$
c_4	1

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

Messy! Per-instruction costs obscure the "big picture" runtime function.

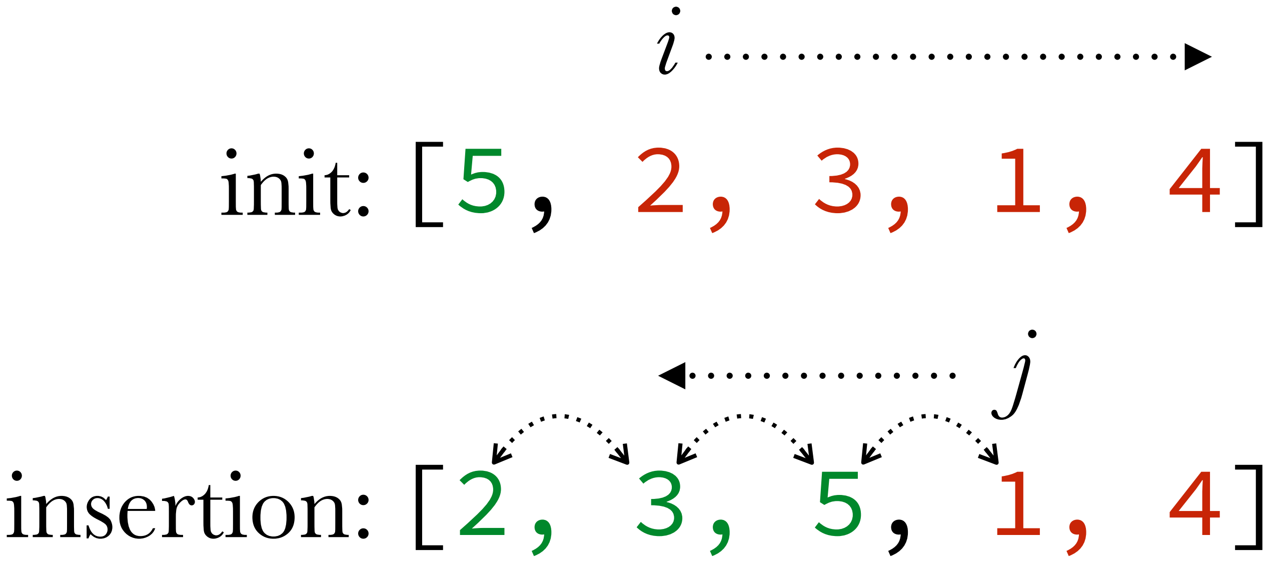
```
def factorial(n):                            times
    prod = 1 ............................... 1
    for k in range(2, n+1): ........ n − 1
        prod *= k .................. n − 1
    return prod ...................... 1
```

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

Simplification #1: ignore actual cost of each line of code.
Runtime is *linear* w.r.t. input size.

Next: a sort algorithm — *insertion* sort

Inspiration: sorting a hand of cards



```
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
```

```
                                                          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):                                                   times
    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  ...........................................  ?
```

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

```
                                                                 times
def insertion_sort(lst):
    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 — this is our default analysis hereafter unless otherwise noted

Review (or crash course) on *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 (find average) (6/2=3)
- multiplying by num of values (3×5=15)

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

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

```
                                                                times
def insertion_sort(lst):
    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
```

```
def insertion_sort(lst):                                                      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  .................................................... $0$
```

```
def insertion_sort(lst):                                                  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
```

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

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

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

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

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

Simplification #3: *ignore constant coefficients*

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

... we conclude that insertion sort has a *worst-case runtime complexity* of n^2

we write: $T(n) = O(n^2)$

read: “is big-O of”

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

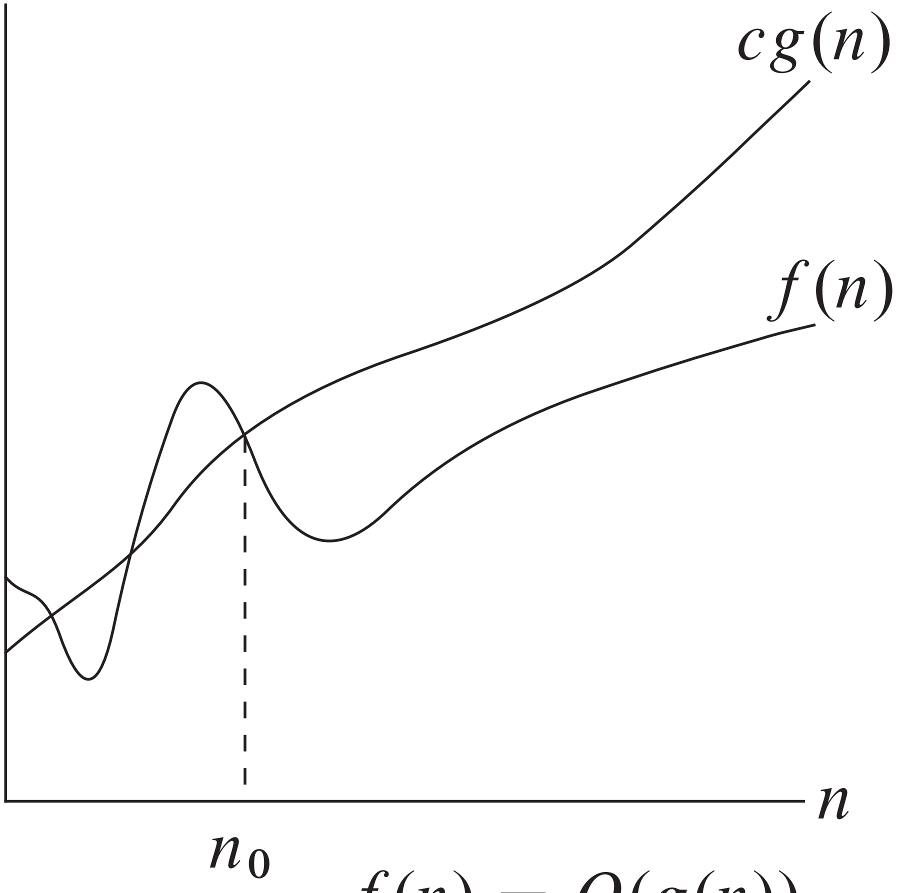
means that there exists constants c, n_0

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

for all $n \geq n_0$

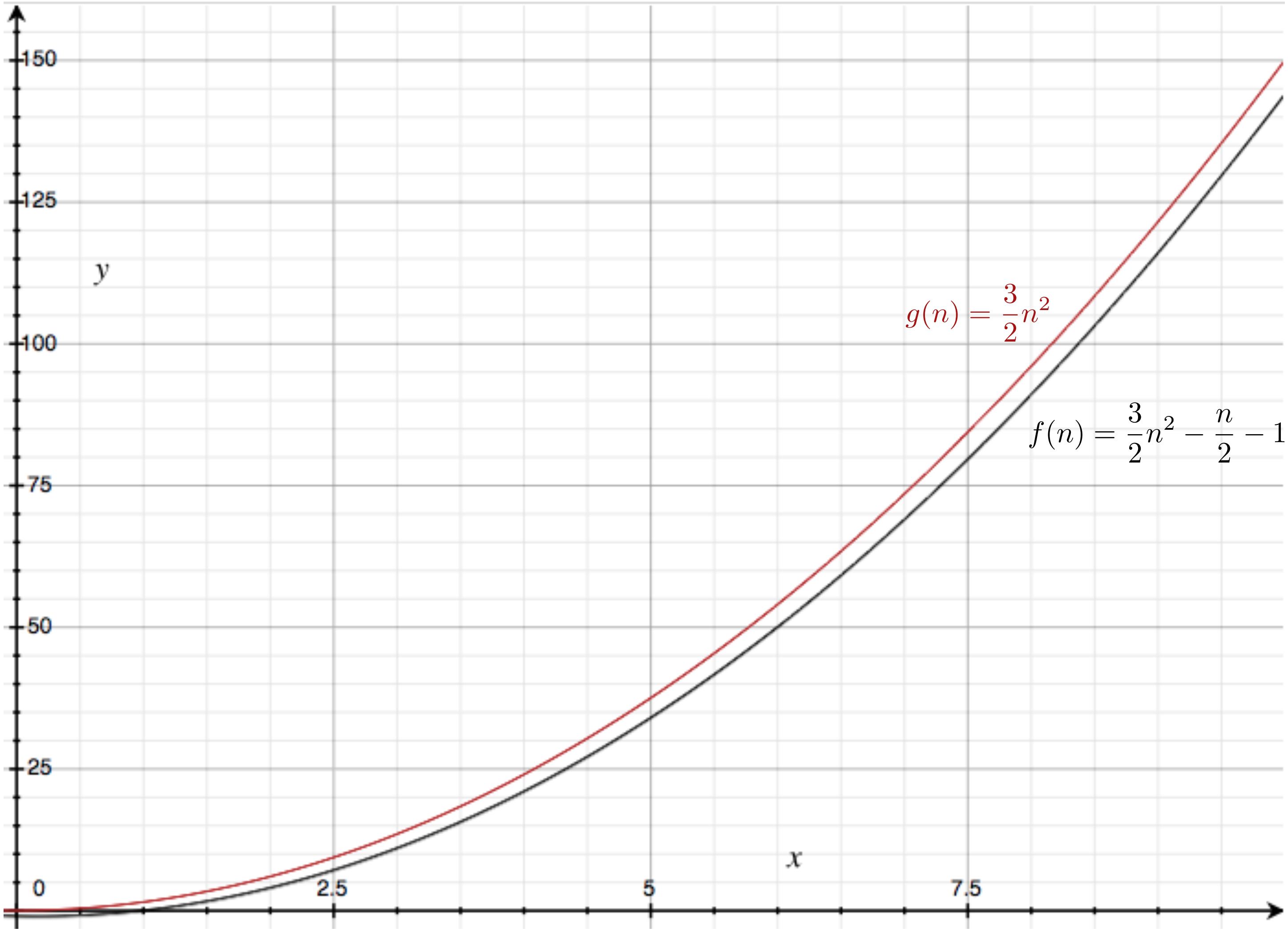
i.e., $f(n) = O(g(n))$

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)


y
x
150
125
100
75
50
25
0
2.5
5
7.5
$g(n) = \frac{3}{2}n^2$
$f(n) = \frac{3}{2}n^2 - \frac{n}{2} - 1$

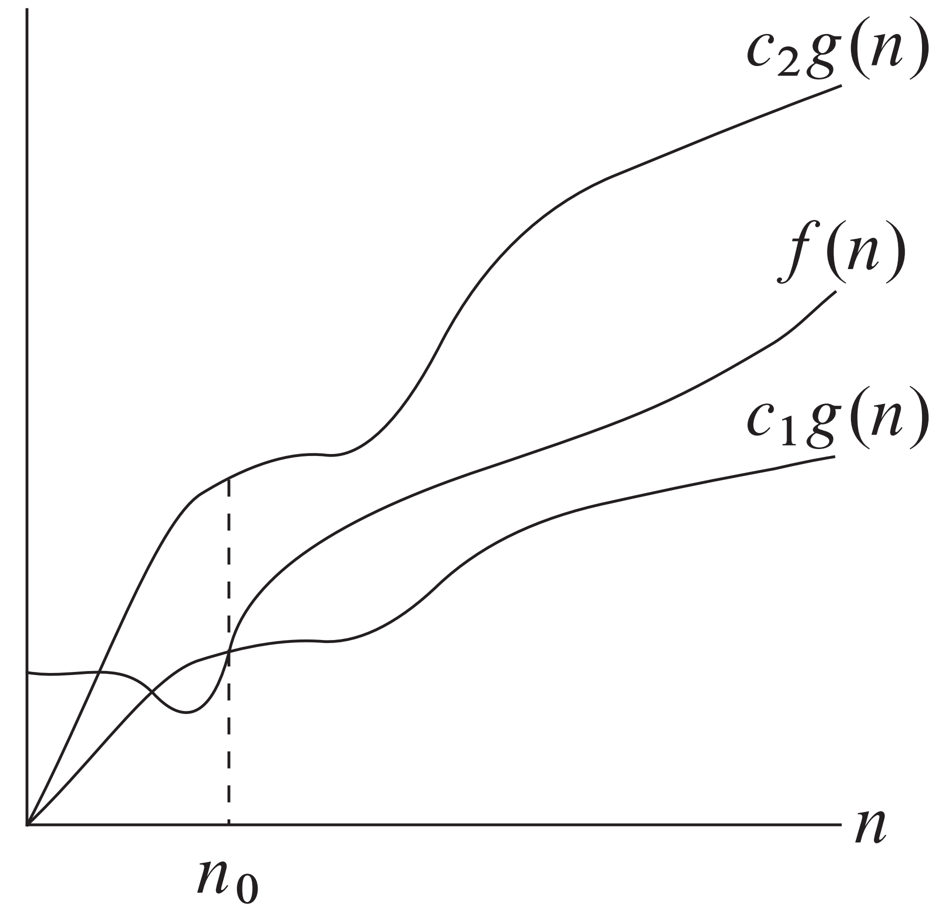
technically, $f = O(g)$ does not imply a asymptotically *tight bound*

e.g., $n = O(n^2)$ is true, but there is no constant c such that cn^2 will approximate the growth of n, as n gets large but in this class we *will* use big-O notation to signify asymptotically tight bounds

i.e., there are constants c_1, c_2 such that:

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \text{for } n \geq n_0$$

(there's another notation: Θ — big-theta — but we're avoiding the formalism) *asymptotically tight bound: g* “sandwiches” *f*



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

So far, we've seen:

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

```
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)
```

$$= O(?)$$

```
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.

$$= O(?)$$

```
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
```

$$= O(?)$$

```
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
```

$$= O(?)$$

```
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

```
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
```

$$= O(dim^3)$$

$$\begin{array}{r}\text{using “brute force” to}\\ \text{crack an } n\text{-bit password}\end{array} = O(?)$$

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

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 (coming!)
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