Runtime Complexity

CS 331: Data Structures and Algorithms
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.
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!
```python
def factorial(n):
    prod = 1
    for k in range(2, n+1):
        prod *= k
    return prod
```

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod = 1</td>
<td>(c_1)</td>
<td>1</td>
</tr>
<tr>
<td>for k in range(2, n+1):</td>
<td>(c_2)</td>
<td>(n-1)</td>
</tr>
<tr>
<td>prod *= k</td>
<td>(c_3)</td>
<td>(n-1)</td>
</tr>
<tr>
<td>return prod</td>
<td>(c_4)</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ T(n) = c_1 + (n - 1)(c_2 + c_3) + c_4 \]

Messy! Per-instruction costs obscure the “big picture” runtime function.
```python
def factorial(n):
    prod = 1
    for k in range(2, n+1):
        prod *= k
    return prod
```

\[ T(n) = 2(n - 1) + 2 = 2n \]

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

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

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
def insertion_sort(lst):
    for i in range(1, len(lst)):  # $n - 1$ times
        for j in range(i, 0, -1):
            if lst[j] < lst[j-1]:  # ? times
                lst[j], lst[j-1] = lst[j-1], lst[j]
            else:  # ? times
                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)):  
        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!
**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
  *else*: ............................................. 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)
i.e., \( 1 + 2 + \cdots + n = \sum_{t=1}^{n} t = \frac{n(n + 1)}{2} \)

and \( 1 + 2 + \cdots + (n - 1) = \sum_{t=1}^{n-1} t = \frac{(n - 1)n}{2} \)
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)):
        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
\sum_{i=1}^{n-1} t
\sum_{i=1}^{n-1} t
\sum_{i=1}^{n-1} t
0
0

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
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

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

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
... we conclude that insertion sort has a \textit{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
3.1 Asymptotic notation

- **O**-notation gives an upper bound for a function to within a constant factor. We write $f(n) \in O(g(n))$ if there are positive constants $n_0$ and $c$ such that at and to the right of $n_0$, the value of $f(n)$ always lies on or below $cg(n)$.

- **\( \Theta \)**-notation bounds a function to within constant factors. We write $f(n) \in \Theta(g(n))$ if there exist positive constants $n_0$, $c_1$, and $c_2$ such that at and to the right of $n_0$, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$, inclusive.

- **\( \Omega \)**-notation gives a lower bound for a function to within a constant factor. We write $f(n) \in \Omega(g(n))$ if there are positive constants $n_0$ and $c$ such that at and to the right of $n_0$, the value of $f(n)$ always lies on or above $cg(n)$.

A function $f(n) \in \Theta(g(n))$ if there exist positive constants $c_1$ and $c_2$ such that it can be "sandwiched" between $c_1g(n)$ and $c_2g(n)$, for sufficiently large $n$. Because $\Theta(g(n))$ is a set, we could write "$f(n) \in 2 \Theta(g(n))$" to indicate that $f(n)$ is a member of $\Theta(g(n))$. Instead, we will usually write "$f(n) \in O(g(n))$" to express the same notion. You might be confused because we abuse equality in this way, but we shall see later in this section that doing so has its advantages.

Figure 3.1(a) gives an intuitive picture of functions $f(n)$ and $g(n)$, where $f(n) \in \Theta(g(n))$. For all values of $n$ at and to the right of $n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an asymptotically tight bound for $f(n)$.

The definition of $\Theta(g(n))$ requires that every member $f(n) \in \Theta(g(n))$ be asymptotically nonnegative, that is, that $f(n)$ be nonnegative whenever $n$ is sufficiently large. (An asymptotically positive function is one that is positive for all sufficiently large $n$.) Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty. We shall therefore assume that every function used within $\Theta$-notation is asymptotically nonnegative. This assumption holds for the other asymptotic notations defined in this chapter as well.

(from Cormen, Leiserson, Riest, and Stein, Introduction to Algorithms)
$g(n) = \frac{3}{2} n^2$

$f(n) = \frac{3}{2} n^2 - \frac{n}{2} - 1$
technically, \( f = \mathcal{O}(g) \) does not imply a asymptotically tight bound

e.g., \( n = \mathcal{O}(n^2) \) 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: $\Theta$ — big-theta — but we’re avoiding the formalism)
**asymptotically tight bound**: $g$ “sandwiches” $f$

(From Cormen, Leiserson, Riee, and Stein, Introduction to Algorithms)
So far, we've seen:

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

\[ \text{def quadratic_roots(a, b, c):} \]
\[ \text{discr} = \text{b}^{**2} - 4\text{a}\text{c} \]
\[ \text{if discr} < 0: \]
\[ \text{return None} \]
\[ \text{discr} = \text{math.sqrt(discr)} \]
\[ \text{return (-b+discr)/(2*a), (-b-discr)/(2*a)} \]

\[ = O(?) \]
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.

= \mathcal{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.

\[ T(n) = C = O(1) \]
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 × n)
def foo(n):
    for _ in range(n):
        for _ in range(n):
            for _ in range(n):
                pass

= \mathcal{O}(?)
\[
\text{def } \text{foo}(n):
    \text{for } _ \text{in } \text{range}(n):
        \text{for } _ \text{in } \text{range}(n):
            \text{for } _ \text{in } \text{range}(n):
                \text{pass}
\]

= \ 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

= \( O(dim^3) \)
using “brute force” to crack an $n$-bit password $= O(?)$
1 character (8 bits) \{(2^8 \text{ possible values})\}

\begin{align*}
&00000000 \\
&00000001 \\
&00000010 \\
&00000011 \\
&00000100 \\
&00000101 \\
&00000110 \\
&00000111 \\
&00001000 \\
&00001001 \\
&00001010 \\
&00001011 \\
&00001100 \\
&00001101 \\
&00001110 \\
&00001111 \\
&... \\
&11110010 \\
&11110011 \\
&11110100 \\
&11110101 \\
&11110110 \\
&11110111 \\
&11111000 \\
&11111001 \\
&11111010 \\
&11111011 \\
&11111100 \\
&11111101 \\
&11111110 \\
&11111111
\end{align*}

= \O(?)
using “brute force” to crack an $n$-bit password $= O(2^n)$
<table>
<thead>
<tr>
<th>Name</th>
<th>Class</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$O(1)$</td>
<td>Compute discriminant</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$O(\log n)$</td>
<td>Binary search</td>
</tr>
<tr>
<td>Linear</td>
<td>$O(n)$</td>
<td>Linear search</td>
</tr>
<tr>
<td>Linearithmic</td>
<td>$O(n \log n)$</td>
<td>Heap sort (coming!)</td>
</tr>
<tr>
<td>Quadratic</td>
<td>$O(n^2)$</td>
<td>Insertion sort</td>
</tr>
<tr>
<td>Cubic</td>
<td>$O(n^3)$</td>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>Polynomial</td>
<td>$O(n^c)$</td>
<td>Generally, $c$ nested loops over $n$ items</td>
</tr>
<tr>
<td>Exponential</td>
<td>$O(c^n)$</td>
<td>Brute forcing an $n$-bit password</td>
</tr>
<tr>
<td>Factorial</td>
<td>$O(n!)$</td>
<td>“Traveling salesman” problem</td>
</tr>
</tbody>
</table>

**Common order of growth classes**