Runtime Complexity

CS 331: Data Structures and Algorithms
Michael Lee <lee@iit.edu>
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 \( n \), find a function \( T(n) \) 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

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

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

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
```

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

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

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

?’s will vary based on initial “sortedness”
... useful to contemplate worst case scenario
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
```
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 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 \[ 1 + 2 + \cdots + (n - 1) = \sum_{t=1}^{n-1} t = \frac{(n - 1)n}{2} \]
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
        1, 2, ..., (n - 1)
        1, 2, ..., (n - 1)
        1, 2, ..., (n - 1)
        0
        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
```

\[\sum_{t=1}^{n-1} t\]

\[\sum_{t=1}^{n-1} t\]

\[0\]

\[0\]
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
$T(n) = \frac{3}{2}n^2 - \frac{n}{2} - 1$

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

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

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
3.1 Asymptotic notation

(b)

(\text{for}) \ 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_1 \cdot g(n) \ and \ c_2 \cdot g(n), \ inclusive.

(c)

(\text{for}) \ 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 \ below \ c \cdot g(n).

(\text{for}) \ 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 \ c \cdot g(n).

A function \ f(n) \ belongs to the set \ \Theta(g(n)) \ if \ there \ exist \ positive \ constants \ c_1 \ and \ c_2 \ such \ that \ it \ can \ be \ "sandwiched" \ between \ c_1 \cdot g(n) \ and \ c_2 \cdot g(n), \ for \ sufficiently \ large \ n. \ Because \ \Theta(g(n)) \ is \ a \ set, \ we \ could \ write \ "f(n) \in \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 O(g(n)). \ For \ all \ values \ of \ n \ at \ and \ to \ the \ right \ of \ n_0, \ the \ value \ of \ f(n) \ lies \ at \ or \ above \ c_1 \cdot g(n) \ and \ at \ or \ below \ c_2 \cdot g(n). \ In \ other \ words, \ for \ all \ n \ \geq 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(n) \ 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)
\[ f(n) = \frac{3}{2} n^2 - \frac{n}{2} - 1 \]
\[ g(n) = \frac{3}{2} n^2 \]
technically, $f = O(g)$ does not imply a asymptotically tight bound
e.g., $n = O(n^2)$, but $n^2$ grows much faster than $n$
but in this class we will use big-O notation to signify asymptotically tight bounds.

i.e., there are constants $c_1$, $c_2$, $n_0$ such that:

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

(there’s another notation: $\Theta$ — big-theta — but we’ll avoid the formalism)
asymptotically tight bound: $g$ “sandwiches” $f$

(from Cormen, Leiserson, Riest, and Stein, Introduction to Algorithms)
E.g., binary search

```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
```

length $\Rightarrow N$

# iterations $= O(?)$

constant time
E.g., binary search

```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
```

length $\Rightarrow N$

# iterations = $O(\?)$

reduces search-space by $\frac{1}{2}$

worst-case: $x < \min(lst)$
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

E.g., binary search

length \Rightarrow N

# iterations \approx \# times we can divide length until = 1
E.g., binary search

```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
    return False
```

length = 1024

# iterations ≈ # times we can divide length until = 1

<table>
<thead>
<tr>
<th>Iterations</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements remaining</td>
<td>1024</td>
<td>512</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
length = \( N \)

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

\( \approx \log_2 N \)

= \( O(\log_2 N) \)

[ recall: \( \log_a x = \log_b x / \log_b a \) ]

= \( O(\log N) \)

1 = \( N / 2^x \)

\( 2^x = N \)

\( \log_2 2^x = \log_2 N \)

\( x = \log_2 N \)
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

E.g., binary search

length ⇒ N

# iterations = $O(\log N)$

binary-search($N$) = $O(\log N)$

constant time
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)
```

$= O(?)$
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.

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

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

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