CSc 120
Introduction to Computer Programming II

Adapted from slides by Dr. Saumya Debray

08: Efficiency and Complexity
EFFICIENCY MATTERS
reasoning about performance
Reasoning about efficiency

Consider two different programs that sum the integers from 1 to n.

**def sumv1(n):**
```python
def sumv1(n):
    num = 0
    for i in range(1, n+1):
        num += i
    return num
```

**def sumv2(n):**
```python
def sumv2(n):
    num = (n*(n+1)) / 2
    return num
```
Reasoning about efficiency

How would we compare them to see which is "better"?

```python
def sumv1(n):
    num = 0
    for i in range(1, n+1):
        num += i
    return num

def sumv2(n):
    num = (n*(n+1)) / 2
    return num
```
Reasoning about efficiency

How would we compare them to see which is "better"? Ideas?

def sumv1(n):
    num = 0
    for i in range(1,n+1):
        num += i
    return num

def sumv2(n):
    num = (n*(n+1)) / 2
    return num
Reasoning about efficiency

• We could compare the difference in running times:
  - Download $\text{sumv1}(n)$
    - [http://www2.cs.arizona.edu/classes/cs120/spring19/NOTES/sumv1.py](http://www2.cs.arizona.edu/classes/cs120/spring19/NOTES/sumv1.py)
    - run this for these values of $n$: 10,000, 100,000, 1,000,000
  - Download $\text{sumv2}(n)$
    - [http://www2.cs.arizona.edu/classes/cs120/spring19/NOTES/sumv2.py](http://www2.cs.arizona.edu/classes/cs120/spring19/NOTES/sumv2.py)
    - run this for these values of $n$: 10,000, 100,000, 1,000,000
Reasoning about efficiency

• Observations on $\text{sumv1}(n)$ vs $\text{sumv2}(n)$:
  - For $\text{sumv1}$, as we increase $n$, the running time increases
    • increases in proportion to $n$
  - For $\text{sumv2}$, as we increase $n$, the running time stays the same

• We noticed this by running the programs

• But this depends on many external factors
Reasoning about efficiency

• The time taken for a program to run can depend on:
  – processor properties that have nothing to do with the program (e.g., CPU speed, amount of memory)
  – what other programs are running (i.e., system load)
  – which inputs we use (some inputs may be worse than others)

• We would like to compare different algorithms:
  – without requiring that we implement them both first
  – focusing on running time (not memory usage)
  – abstracting away processor-specific details
  – considering all possible inputs
Reasoning about efficiency

• Algorithms vs. programs
  
  – Algorithm:
    o a step-by-step list of instructions for solving a problem
  
  – Program:
    o an algorithm that been implemented in a given language

• We would like to compare different algorithms *abstractly*
Comparing algorithms

• Search for a word `my_word` in a dictionary (a book)
• A dictionary is sorted
  – Algo 1 (search from the beginning):
    start at the first word in the dictionary
    if the word is not `my_word`, then go to the next word
    continue in sequence until `my_word` is found
  – Algo 2:
    start at the middle of the dictionary
    if `my_word` is greater than the word in the middle,
      start with the middle word and continue from there to the end
    if `my_word` is less than the word in the middle,
      start with the middle word and continue from there to the beginning
EXERCISE

• Which is better, Algo 1 (search from the beginning) or Algo 2 (search from the middle)? What is the reason?

• Which ever algo you chose, is there ever a scenario where the other algo is better?

• When considering which is better, what measure are we using?
Comparing algorithms

• Call comparison a *primitive* operation
  – an abstract unit of computation

• We want to characterize an algorithm in terms of how many primitive operations are performed
  – best case and worst case

• We want to express this in terms of the size of the data (or size of its input)
Primitive operations

• Abstract units of computation
  – convenient for reasoning about algorithms
  – approximates typical hardware-level operations

• Includes:
  – assigning a value to a variable
  – looking up the value of a variable
  – doing a single arithmetic operation
  – comparing two numbers
  – accessing a single element of a Python list by index
  – calling a function
  – returning from a function
Primitive ops and running time

• A primitive operation typically corresponds to a small constant number of machine instructions

• No. of primitive operations executed
  \( \propto \) no. of machine instructions executed
  \( \propto \) actual running time
Example

Code

def lookup(str_, list_):
    for i in range(len(list_)):
        if str_ == list_[i]:
            return i
    return -1

Primitive operations

- \text{len}(\text{list}_) : 1
- \text{range}( ) : 1
- \text{in} : 1
- \text{for} : 2
- \text{list}[i] : 1
- str_ : 1
- == : 1
- if : 1

Each iteration: 9 primitive ops
Primitive ops and running time

• We consider how a function's running time depends on the size of its input
  – *which input do we consider?*
Best case vs. worst case inputs

# lookup(str_, list_): returns the index where str_ occurs in list_

def lookup(str_, list_):
    for i in range(len(list_)):
        if str_ == list_[i]:
            return i
    return -1

• Best-case scenario?:

• Worst-case scenario?:
Best case vs. worst case inputs

# lookup(str_, list_): returns the index where str_ occurs in list_

```python
def lookup(str_, list_):
    for i in range(len(list_>):
        if str_ == list_[i]:
            return i
    return -1
```

• **Best-case scenario:** str_ == list_[0]  # first element
  – loop does not have to iterate over list_ at all
  – running time does not depend on length of list_
  – does not reflect typical behavior of the algorithm
Best case vs. worst case inputs

```python
# lookup(str_, list_): returns the index where str_ occurs in list_
def lookup(str_, list_):
    for i in range(len(list_)):
        if str_ == list_[i]:
            return i
    return -1
```

- **Worst-case scenario**: `str_ == list_[-1]`  
  # last element
  - loop iterates through list_
  - running time is proportional to the length of list_
  - captures the behavior of the algorithm better
Best case vs. worst case inputs

# lookup(str_, list_): returns the index where str_ occurs in list_

def lookup(str_, list_):
    for i in range(len(list_)):
        if str_ == list_[i]:
            return i
    return -1

- In reality, we get something in between
  - but "average-case" is difficult to characterize precisely
What about “average case”?
Worst-case complexity

• Considers worst-case inputs
• Describes the running time of an algorithm as a function of the size of its input ("time complexity")
• Focuses on the rate at which the running time grows as the input gets large
• Typically gives a better characterization of an algorithm's performance

• This approach can also be applied to the amount of memory used by an algorithm ("space complexity")
Example

Code

def lookup(str_, list_):
    for i in range(len(list_)):
        if str_ == list_[i]:
            return i
    return -1

Primitive operations

```
len(list_) : 1
range() : 1
in : 1
for : 2
list_[i] : 1
str_ : 1
== : 1
if : 1
```

each iteration: 9 primitive ops
def lookup(str_, list_):
    for i in range(len(list_)):
        if str_ == list_[i]:
            return i
    return -1

Total primitive ops executed:
  1 iteration: 9 ops
∴ n iterations: 9n ops
+ return at the end: 1 op

∴ total worst-case running time for a list of length n = 9n + 1
# What is the total worst-case running time of the following code fragment expressed in terms of n?

```python
for i in range(n):
k = 2 + 2
```
# What is the total worst-case running time of the following code fragment expressed in terms of n?

```python
a = 5
b = 10
for i in range(n):
    x = i * b
for j in range(n):
    z += b
```
asymptotic complexity
Asymptotic complexity

• In the worst-case, lookup(str_, list_) executes 9n + 1 primitive operations given a list of length n

• To translate this to running time:
  – suppose each primitive operation takes k time units
  – then worst-case running time is (9n + 1)k

• But k depends on specifics of the computer, e.g.:

<table>
<thead>
<tr>
<th>Processor speed</th>
<th>k</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>slow</td>
<td>20</td>
<td>180n + 20</td>
</tr>
<tr>
<td>medium</td>
<td>10</td>
<td>90n + 10</td>
</tr>
<tr>
<td>fast</td>
<td>3</td>
<td>27n + 3</td>
</tr>
</tbody>
</table>
Asymptotic complexity

worst case running time = $A n + B$

depends on how the algorithm processes data

depends on processor-specific characteristics
Asymptotic complexity

• For algorithm analysis, we focus on how the running time grows as a function of the input size $n$
  – usually, we do not look at the exact worst case running time
  – it's enough to know proportionalities

• E.g., for the lookup() function:
  – executes $9n + 1$ primitive operations given a list of length $n$
  – we say only that its running time is "proportional to the input length $n"
Example

Code

def list_positions(list1, list2):
    positions = []
    for value in list1:
        idx = lookup(value, list2)
        positions.append(idx)
    return positions
def list_positions(list1, list2):
    positions = []
    for value in list1:
        idx = lookup(value, list2)
        positions.append(idx)
    return positions

Worst case behavior:
primitive operations = n(9n + 5) + 2 = 9n^2 + 5n + 2
running time = k(9n^2 + 5n + 2)
Example

Code

def list_positions(list1, list2):
    positions = []
    for value in list1:
        idx = lookup(value, list2)
        positions.append(idx)
    return positions

Worst case: $9n^2 + 5n + 2$

As $n$ grows, the $9n^2$ term grows faster than $5n+2$

$\Rightarrow$ for large $n$, the $n^2$ term dominates

$\Rightarrow$ running time depends primarily on $n^2$
Example

As \( n \) grows larger, the \( n^2 \) term dominates \( \Rightarrow \) the contribution of the other terms becomes insignificant
Example 2: $2x^2 + 15x + 10$
Example 3: $x^3 + 100x^2 + 100x + 100$
Growth rates

• As input size grows, the fastest-growing term dominates the others
  – the contribution of the smaller terms becomes negligible
  – it suffices to consider only the highest degree (i.e., fastest growing) term

• For algorithm analysis purposes, the constant factors are not useful
  – they usually reflect implementation-specific features
  – to compare different algorithms, we focus only on proportionality
  ⇒ ignore constant coefficients
Comparing algorithms

Growth rate $\propto n$

```python
def lookup(str_, list_):
    for i in range(len(list_)):
        if str_ == list_[i]:
            return i
    return -1
```

def list_positions(list1, list2):
    positions = []
    for value in list1:
        idx = lookup(value, list2)
        positions.append(idx)
    return positions
```

Growth rate $\propto n^2$
Summary so far

• Want to characterize algorithm efficiency such that:
  – does not depend on processor specifics
  – accounts for all possible inputs

⇒ count primitive operations
⇒ consider worst-case running time

• We specify the running time as a function of the size of the input
  – consider proportionality, ignore constant coefficients
  – consider only the dominant term
    – e.g., $9n^2 + 5n + 2 \approx n^2$
big-O notation
Big-O notation

• Big-O formalizes this intuitive idea:
  – consider only the dominant term
    o e.g., $9n^2 + 5n + 2 \approx n^2$
  – allows us to say,
    "the algorithm runs in time proportional to $n^2"
Big-O notation

Intuition:

When we say... ...we mean
"f(n) is O(g(n))" "f is growing at most as fast as g"

"big-O notation"
**Definition**: Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers.

Then, $f(n)$ is $O(g(n))$ if there is a real constant $c$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c g(n) \quad \text{for all } n > n_0$$
**Big-O notation**

\[ f(n) \text{ is } \mathcal{O}(g(n)) \text{ if there is a real constant } c \text{ and an integer constant } n_0 \geq 1 \text{ such that } f(n) \leq c \cdot g(n) \text{ for all } n > n_0 \]

“Once the input gets big enough, \( c g(n) \) is (always) larger than \( f(n) \)”
Big-O notation: properties

• If \( g(n) \) is growing faster than \( f(n) \):
  - \( f(n) \) is \( O(g(n)) \)
  - \( g(n) \) is not \( O(f(n)) \)

• If \( f(n) = a_0 + a_1 n + \ldots + a_k n^k \), then:
  \[
  f(n) = O(n^k)
  \]
  - i.e., coefficients and lower-order terms can be ignored
Big-O notation

Growth rate $\propto n$

$O(n)$

def lookup(str_, list_):
    for i in range(len(list_)):
        if str_ == list_[i]:
            return i
    return -1

Growth rate $\propto n^2$

$O(n^2)$

def list_positions(list1, list2):
    positions = []
    for value in list1:
        idx = lookup(value, list2)
        positions.append(idx)
    return positions
Some common growth-rate curves

Running time

Input size

$O(n^3)$

$O(n^2)$

$O(n \log(n))$

$O(n)$

$O(\log n)$
Computing big-O complexities

Given the code:

\[
\begin{align*}
\text{line}_1 & \quad \ldots \quad O(f_1(n)) \\
\text{line}_2 & \quad \ldots \quad O(f_2(n)) \\
\ldots & \\
\text{line}_k & \quad \ldots \quad O(f_k(n))
\end{align*}
\]

The overall complexity is

\[
O(\text{max}(f_1(n), f_2(n), \ldots, f_k(n)))
\]

Given the code:

\[
\begin{align*}
\text{loop} & \quad \ldots \quad O(f_1(n)) \text{ iterations} \\
\text{line}_1 & \quad \ldots \quad O(f_2(n))
\end{align*}
\]

The overall complexity is

\[
O( f_1(n) \times f_2(n) )
\]
using big-O notation
Using big-O notation

<table>
<thead>
<tr>
<th>Code</th>
<th>Big-O complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>str_== list_[i]</code></td>
<td>O(1)</td>
</tr>
</tbody>
</table>

O(1)
Using big-O notation

<table>
<thead>
<tr>
<th>Code</th>
<th>Big-O complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if str_ == list_[i]:</code></td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>return i</code></td>
<td></td>
</tr>
</tbody>
</table>

O(1)
Using big-O notation

<table>
<thead>
<tr>
<th>Code</th>
<th>Big-O complexity</th>
</tr>
</thead>
</table>
| for i in range(len(list_)):  
  if str_ == list_[i]:  
    return i | O(n) |

O(n) (worst-case)  
(n = length of the list)  

O(1)
Using big-O notation

<table>
<thead>
<tr>
<th>Code</th>
<th>Big-O complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>def lookup(str__, list__):</td>
<td>O(n)</td>
</tr>
<tr>
<td>for i in range(len(list__)):</td>
<td></td>
</tr>
<tr>
<td>if str__ == list__[i]:</td>
<td></td>
</tr>
<tr>
<td>return i</td>
<td></td>
</tr>
<tr>
<td>return -1</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

- Time complexity: $O(n)$ for the loop iteration.
- Additional constant time operations: $O(1)$.
Using big-O notation

Code

```python
def list_positions(list1, list2):
    positions = []
    for value in list1:
        idx = lookup(value, list2)
        positions.append(idx)
    return positions
```

Big-O complexity

- $O(n^2)$ (worst-case)
  - $n$ = length of list2
- $O(n)$ (worst-case)
  - $n$ = length of list1

- $O(n)$ (worst-case)
  - $n$ = length of list1
Using big-O notation

Code

```python
def list_positions(list1, list2):
    positions = []
    for value in list1:
        idx = lookup(value, list2)
        positions.append(idx)
    return positions
```

Big-O complexity

O(1)  
O(n^2)  
O(n^2)
Computing big-O complexities

Given the code:

```
line_1  ... O(f_1(n))
line_2  ... O(f_2(n))
...
line_k  ... O(f_k(n))
```

The overall complexity is

\[ O(\max(f_1(n), f_2(n), ..., f_k(n))) \]

Given the code

```
loop  ... O(f_1(n)) iterations
line_1  ... O(f_2(n))
```

The overall complexity is

\[ O( f_1(n) \times f_2(n) ) \]
EXERCISE

# my_rfind(mylist, elt) : find the distance from the # end of mylist where elt occurs, -1 if it does not

def my_rfind(mylist, elt):
    pos = len(mylist) − 1
    while pos >= 0:
        if mylist[pos] == elt:
            return pos
        pos -= 1
    return -1

Worst-case big-O complexity = ???
EXERCISE

# for each element of a list: find the biggest value
# between that element and the end of the list

def find_biggest_after(arglist):
    pos_list = []
    for idx0 in range(len(arglist)):
        biggest = arglist[idx0]
        for idx1 in range(idx0+1, len(arglist)):
            biggest = max(arglist[idx1], biggest)
        pos_list.append(biggest)
    return pos_list

Worst-case big-O complexity = ????
Input size vs. run time: max()
EXERCISE

# for each element of a list: find the biggest value
# between that element and the end of the list

def find_biggest_after(arglist):
    pos_list = []
    for idx0 in range(len(arglist)):
        biggest = max(arglist[idx0:]):  # library code
        pos_list.append(biggest)
    return pos_list

Worst-case big-O complexity = ???