Algorithm Analysis

- Computation Resources
- Big-O Notation
- Time Complexity
  - Python List
  - Python Dictionary
Algorithms

- Algorithms are designed to solve problems.
  - generic, step-by-step list of instructions
- A problem can have multiple solutions.

*How do we determine which solution is the most efficient?*
Computing Resources

• space or memory
  • Typically dictated by the problem instance

• execution time (running time)
  • actual time required for the program to compute its result
  • the starting time and ending time wrt the system used
Example

• Problem: Computing the sum of the first $n$ integers.

• Observation:
  • Iterative solutions seem to be doing more work (repeated steps, longer)
  • The time required for iterative solution increase with the value of $n$
Problem

• Actual execution time depends on:
  • particular machine
  • program
  • time of day
  • compiler
  • programming language
  • ...
Example Algorithm

- Given a matrix of size \( n \times n \), compute the:
  - sum of each row of a matrix.
  - overall sum of the entire matrix.
Example Algorithm (v.1)

- How many addition operations?

\[
T(n) = 2n(n) = 2n^2
\]

\[
\text{rowSum} = \text{Array}(n)
\]

\[
\text{totalSum} = 0
\]

\[
\begin{align*}
\text{for } i \text{ in range( } n \text{ )} : \\
& \quad \text{rowSum}[i] = 0 \\
\text{for } j \text{ in range( } n \text{ )} : \\
& \quad \text{rowSum}[i] = \text{rowSum}[i] + \text{matrix}[i,j] \\
& \quad \text{totalSum} = \text{totalSum} + \text{matrix}[i,j]
\end{align*}
\]
Example Algorithm (v.2)

• How many additions are performed?

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

```python
rowSum = Array(n)
totalSum = 0

for i in range(n):
    rowSum[i] = 0

for j in range(n):
    for i in range(n):
        rowSum[i] = rowSum[i] + matrix[i,j]
    totalSum = totalSum + rowSum[i]
```
Compare the Results

- Number of additions: \( v_1: 2n^2 \quad v_2: n^2 + n \)
- Second version has fewer additions (\( n > 1 \))
  - Will execute faster than the first.
  - Difference will not be significant.
Growth Rates

- As $n$ increases, both algorithms increase at approx. the same rate:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$2n^2$</th>
<th>$n^2 + n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>200</td>
<td>110</td>
</tr>
<tr>
<td>100</td>
<td>20,000</td>
<td>10,100</td>
</tr>
<tr>
<td>1000</td>
<td>2,000,000</td>
<td>1,001,000</td>
</tr>
<tr>
<td>10,000</td>
<td>200,000,000</td>
<td>100,010,000</td>
</tr>
<tr>
<td>100,000</td>
<td>20,000,000,000</td>
<td>10,000,100,000</td>
</tr>
</tbody>
</table>
Big-O Notation

• No need to count precise number of steps
• Classify algorithms by order of magnitude
  • execution time
  • space requirements

Can approximate actual number of steps or actual storage in terms of variable-sized data sets.
Big-O Definition

- The order of magnitude: Big-O notation.
- Then, the algorithm has a time-complexity of or executes “on the order of” $f(n)$
  - We use the notation: $O(f(n))$
  - Big-O is intended for large values of $n$.

$f(n)$ indicates the rate of growth at which the run time increases as the input size increases.
Classes of Algorithms

• Many algorithms have a time-complexity selected from a common set of functions.

<table>
<thead>
<tr>
<th>$f()$</th>
<th>Common Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
</tr>
<tr>
<td>$\log n$</td>
<td>logarithmic</td>
</tr>
<tr>
<td>$n$</td>
<td>linear</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>log linear</td>
</tr>
<tr>
<td>$n^2$</td>
<td>quadratic</td>
</tr>
<tr>
<td>$n^3$</td>
<td>cubic</td>
</tr>
<tr>
<td>$a^n$</td>
<td>exponential</td>
</tr>
</tbody>
</table>
Classes of Algorithms
Different Cases

- Some algorithms have different run times for different sets of inputs of the same size.
  - best case
  - worst case
  - average case

- Typically be identified by:
  - event-controlled loop
  - conditional statement
Different Cases

```python
def findNeg( intSeq ):
    n = len( intSeq )
    for i in range( n ) :
        if intSeq[i] < 0 :
            return i
    return None

L = [ 72, 4, 90, 56, 12, 67, 43, 17, 2, 86, 33 ]
p = findNeg( L )

L = [ -12, 50, 4, 67, 39, 22, 43, 2, 17, 28 ]
p = findNeg( L )
```
Evaluating Python Code

- Basic operations only require constant time:
  - \( x = 5 \)
  - \( z = x + y \times 6 \)
  - \textbf{if} \( x > 0 \) \textbf{and} \( x < 100 \)

- What about function calls?
  \[ y = \text{ex1}(n) \]
Evaluation Focus on

• Repetition
• Selection statements
• Function and method calls
def ex1(n):
    count = 0
    for i in range(n):
        count += i
    return count
def ex2(n):
    count = 0
    for i in range(n):
        count += 1
    for j in range(n):
        count += 1
    return count
def ex3(n):
    count = 0
    for i in range(n):
        for j in range(n):
            count += 1
    return count
def ex4( n ):
    count = 0
    for i in range( n ):
        for j in range( 25 ):
            count += 1
    return count
Code Evaluation #5

```python
def ex5(n):
    count = 0
    for i in range(n):
        for j in range(i+1):
            count += 1
    return count
```
def ex6( n ):
    count = 0
    i = n
    while i >= 1:
        count += 1
        i = i // 2
    return count
def ex7( n ):
    count = 0
    for i in range( n ):
        count += ex6( n )
    return count
The Python List

- We used the list to implement many of our ADTs.
- Their efficiency depends on the efficiency of Python's list.
Python List: Traversal

- Iterates over the contiguous elements of the underlying array.

```python
# Sum the elements of a list.
sum = 0
for value in valueList:
    sum = sum + value

# Alternate version.
sum = 0
n = len(valueList)
for i in range(n):
    sum = sum + valueList[i]
```
Python List: Allocation

- Creating a non-empty list is not constant.

```python
temp = list()
listX = [0] * n
valueList = [4, 8, 20, 2, 15, 89, 60, 75]
```
Python List: Appending

- When space is available, the item is stored in the next slot.

What if the underlying array is full?
Python List: Expanding The List

- Assume the list contains \( n \) items

**Step 1:** create a new array, double the size.

**Step 2:** copy the items from original array to the new array.
Python List: Expanding The List

Step 3: replace the original array with the new array.

Original array:

```
pyList
4  12  2  34  17  50  18  64
```

New array:

```
pyList
4  12  2  34  17  50  18  64  \cdot  \cdot  \cdot  \cdot  \cdot  \cdot  \cdot  \cdot
```

Step 4: store value 6 in the next slot of the new array.

```
pyList
4  12  2  34  17  50  18  64  6  \cdot  \cdot  \cdot  \cdot  \cdot  \cdot  \cdot  \cdot
```
Python List: Inserting Items

- Some items have to be shifted to make room for the new item.

```python
pyList.insert( 3, 79 )
```
Python List: Extending

- Adds the contents of a source list to the end of the destination list.
Python List: Time-Complexities

<table>
<thead>
<tr>
<th>List Operation</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v = \text{list}() )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>( \text{len}(v) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>( v = [0] * n )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( v[i] = x )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>( v.\text{append}(x) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( v.\text{extend}(w) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( v.\text{insert}(x) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( v.\text{pop}() )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>traversal</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>
## Python Dictionaries

<table>
<thead>
<tr>
<th>Dictionary Operation</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>get item</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>set item</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>delete item</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>add item</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>contains (in)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>traversal</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>