UNIT 4C
Iteration: Scalability & Big O

Efficiency

• A computer program should be totally correct, but it should also
  – execute as quickly as possible (time-efficiency)
  – use memory wisely (storage-efficiency)

• How do we compare programs (or algorithms in general) with respect to execution time?
  – various computers run at different speeds due to different processors
  – compilers optimize code before execution
  – the same algorithm can be written differently depending on the programming paradigm
Counting Operations

- We measure time efficiency by counting the number of operations performed by the algorithm.
- But what is an operation?
  - assignment statements
  - comparisons
  - return statements
  - ...

Linear Search: Worst Case

```python
# let n = the length of list.
def search(list, key):
    index = 0
    while index < list.length:
        if list[index] == key:
            return index
        index += 1
    return nil

Total: 3n+3
```
Linear Search: Best Case

```python
# let n = the length of list.
def search(list, key):
    index = 0
    while index < list.length do
        if list[index] == key then
            return index
        end
        index = index + 1
    end
    return nil
end
```

Total: 4

Counting Operations

- How do we know that each operation we count takes the same amount of time? (We don’t.)
- So generally, we look at the process more abstractly and count whatever operation depends on the amount or size of the data we’re processing.
- For linear search, we would count the number of times we compare elements in the array to the key.
Linear Search: Worst Case Simplified

# let n = the length of list.
def search(list, key)
    index = 0
    while index < list.length do
        if list[index] == key then n
            return index
        end
        index = index + 1
    end
    return nil
end Total: n

Linear Search: Best Case Simplified

# let n = the length of list.
def search(list, key)
    index = 0
    while index < list.length do
        if list[index] == key then 1
            return index
        end
        index = index + 1
    end
    return nil
end Total: 1
Order of Complexity

• For very large \( n \), we express the number of operations as the (time) order of complexity.
• Order of complexity is often expressed using Big-O notation:

<table>
<thead>
<tr>
<th>Number of operations</th>
<th>Order of Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( 3n+3 )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( 2n+8 )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

Usually doesn’t matter what the constants are... we are only concerned about the highest power of \( n \).

O(n) (“Linear”)

![Graph showing O(n) (“Linear”) complexity]

Number of Operations

\( n \) (amount of data)
**O(n)**

For a linear algorithm, if you double the amount of data, the amount of work you do doubles (approximately).

**O(1) ("Constant-Time")**

For a constant-time algorithm, if you double the amount of data, the amount of work you do stays the same.
Linear Search

- Worst Case: \( O(n) \)
- Best Case: \( O(1) \)
- Average Case: ?

Insertion Sort: Worst Case

```python
# let n = the length of list.
def isort(list):
    a = list.clone n
    i = 1
    while i != a.length do
        move_left(a, i) n-1
        i = i + 1
    end
    return a
end
```
Insertion Sort: Worst Case

# let n = the length of list.
def move_left(a, i)
    x = a.slice!(i)
    j = i-1
    while j >= 0 && a[j] > x do
        j = j - 1
    end
    a.insert(j+1, x)
end

but how long do slice! and insert take?

move_left (alternate version)

# let n = the length of list.
def move_left(a, i)
    x = a[i]
    j = i-1
    while j >= 0 && a[j] > x do
        a[j+1] = a[j]
        j = j - 1
    end
    a[j+1] = x
end
Insertion Sort: Worst Case

• So the total number of operations is $n + (n-1 \text{ move}_\text{left}’s)$
• But each move_left performs $i+1$ operations, where $i$ varies from 1 to $n-1$:
• $n-1 \text{ move}_\text{left}’s = 2 + 3 + 4 + \ldots + n$
• Since $1 + 2 + \ldots + n = n(n+1)/2$,
  $n-1 \text{ move}_\text{left}’s = n(n+1)/2 - 1$
• The total number of operations is:
  $n + n(n+1)/2 - 1 = n + n^2/2 + n/2 - 1 = n^2/2 + 3n/2 - 1$

Order of Complexity

<table>
<thead>
<tr>
<th>Number of operations</th>
<th>Order of Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$n^2/2 + 3n/2 - 1$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$2n^2 + 7$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

Usually doesn't matter what the constants are... we are only concerned about the highest power of $n$.  

O(n^2) ("Quadratic")

Number of Operations

n^2

2n^2 + 7

n/2 + 3n/2 - 1

n (amount of data)

For a quadratic algorithm, if you double the amount of data, the amount of work you do quadruples (approximately).
<table>
<thead>
<tr>
<th>Insertion Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Worst Case: O(n^2)</td>
</tr>
<tr>
<td>• Best Case: ?</td>
</tr>
</tbody>
</table>

_We’ll compare these algorithms with others soon to see how scalable they really are based on their order of complexities._