Computer scientists talk like...

“The worst-case runtime of bubble-sort is in \( O(n^2) \).”

“I can sort it in \( n \log n \) time.”

“That’s too slow, make it linear-time.”

“That problem cannot be solved in polynomial time.”
Sorting Algorithms Comparison

- Bubble sort
- Merge sort

See demo at: http://www.sorting-algorithms.com/

Observations:

- **merge** is faster than **bubble**
- with larger input size, the advantage of **merge** over **bubble** becomes larger
## Runtime Observation

<table>
<thead>
<tr>
<th>Algorithm/Size</th>
<th>20 (s)</th>
<th>40 (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>8.6</td>
<td>38.0</td>
</tr>
<tr>
<td>Merge</td>
<td>5.0</td>
<td>11.2</td>
</tr>
</tbody>
</table>

When input size grows from 20 to 40…

- “runtime” of merge: roughly doubled
- “runtime” of bubble: roughly quadrupled
Runtime means?

✧ It does **NOT** mean how many seconds spent on running the algorithm.

✧ It means the **number of steps** taken by the algorithm.

Thus, the runtime is independent from the **hardware** where you run the algorithm; but, only depends on the algorithm itself.

You can run **bubble** on a super-computer, and run **merge** on a mechanical watch! That has nothing to do with the fact that **merge** is a faster sorting algorithm than **bubble**.
The runtime described in number of steps, as a function of \( n \) (size of input):

- Bubble: could be \( 0.5n^2 \) (steps)
- Merge: could be \( n \log n \) (steps)

But, we don’t really care about the number of steps...

<table>
<thead>
<tr>
<th>Algorithm/Size</th>
<th>20 (steps)</th>
<th>40 (steps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>200</td>
<td>800</td>
</tr>
<tr>
<td>Merge</td>
<td>120</td>
<td>295</td>
</tr>
</tbody>
</table>
Size n & step numbers?

We don’t care about the absolute number of steps, but we care about:

when input size doubles, the runtime quadruples.

In other words, $0.5n^2$ and $700n^2$ are no different!

What we really care is that:

✧ how the number of steps grows as the size of input increases.

Constant factors do NOT matter!
Constant factor, steps grow?

\[ T_1(n) = 0.5 n^2 \quad T_2(n) = 700 n^2 \]

\[
\frac{T_1(2n)}{T_1(n)} = \frac{0.5 (2n)^2}{0.5 n^2} = \frac{2n^2}{0.5n^2} = 4
\]

\[
\frac{T_2(2n)}{T_2(n)} = \frac{700 (2n)^2}{700 n^2} = \frac{2800 n^2}{700 n^2} = 4
\]

Constant factor does not matter, when it comes to growth!
Large input sizes

We care about algorithm design when the input size $n$ is very large.

- $n^2$ and $n^2 + n + 2$ are no different, because when $n$ is really large, $n+2$ is negligible compared to $n^2$

- Only the highest-order term matters!
Low-order terms

Low-order terms do not matter!

\[ T_1(n) = n^2 \quad T_2(n) = n^2 + n + 2 \]

\[ T_1(10000) = 100,000,000 \]
\[ T_2(10000) = 100,010,002 \]

\[ \text{difference} \approx 0.01\% \]
Summary of Runtime

Runtime evaluation:
→ we count the number of steps
→ constant factors don’t matter
→ only the highest-order term matters

Thus, the followings functions are of the same class:

\[ n^2 \quad 2n^2 + 3n \quad \frac{n^2}{165} + 1130n + 3.14159 \]

We call this: \( O(n^2) \)
Big-O Notation

O(n²) is an asymptotic notation

O( f(n) ) is the asymptotic upper-bound, which means that a set of functions grow no faster than f(n).

For example, when we say: \( 5n^2 + 3n + 1 \) is in \( O(n^2) \)

It means that:
\( 5n^2+3n+1 \) grows no faster than \( n^2 \), asymptotically
Asymptotic Notations

More notations to be introduced later:

- $O(f(n))$: the asymptotic upper-bound
- $\Omega(f(n))$: the asymptotic lower-bound
- $\Theta(f(n))$: the asymptotic tight-bound

Precise definitions of $O$, $\Omega$, and $\Theta$ to be given in next class
Asymptotic notations: abstraction

Asymptotic notations are a simplification of the “actual” runtime.

✧ It does not tell the whole story about how fast a program runs in reality.

In real-world applications, constant factor matters! hardware matters! implementation matters!

✧ This simplification makes possible the development of the whole theory of computational complexity.

IMPORTANT idea!
“Make everything as simple as possible, but not simpler.”
—Albert Einstein
Quick note

In CSC165, we use asymptotic notations such as $O(n^2)$, and sometimes, we use the exact forms, such as $3n^2 + 2n$ to be more precise. It depends on the problem requirements.
ANALYZE THE TIME COMPLEXITY OF A PROGRAM
def LS(A, x):
    """Return index i, x == A[i].
    Otherwise, return -1 """
    i = 0
    while i < len(A):
        if A[i] == x:
            return i
        i = i + 1
    return -1

What’s the runtime of this program?

Can’t say yet, it depends on the input (A, x).
def LS(A, x):
    """Return index i, x == A[i].
    Otherwise, return -1 """
    1. i = 0
    2. while i < len(A):
    3.     if A[i] == x:
    4.         return i
    5.     i = i + 1
    6. return -1

Count time complexity
(# of lines of code executed)

\[ t_{LS}([2, 4, 6, 8], 4) = 7 \]
**Linear Search**

```python
def LS(A, x):
    ''' Return index $i$, $x == A[i]$. Otherwise, return -1.'''
    i = 0
    while i < len(A):
        if A[i] == x:
            return i
        i = i + 1
    return -1
```

Count time complexity

$$t_{LS}([2, 4, 6, 8], 6) = 10$$
Linear Search

def LS(A, x):
    '''
    Return index i, x == A[i].
    Otherwise, return -1
    '''
    i = 0
    while i < len(A):
        if A[i] == x:
            return i
        i = i + 1
    return -1

What is the runtime of LS(A, x)?

if the first index where x is found is k
i.e., A[k] == x

t_{LS}(A, x) = 1 + 3(k+1) = 3k + 4

t_{LS}([2, 4, 6, 8], 6) = 10
def LS(A, x):
    """Return index i, x == A[i]. Otherwise, return -1 """
    i = 0
    while i < len(A):
        if A[i] == x:
            return i
        i = i + 1
    return -1

Count time complexity
LS([2, 4, 6, 8], 99)

"""Linear Search"""

$t_{LS}([2, 4, 6, 8], 99) = 15$
**Linear Search**

```python
def LS(A, x):
    """Return index i, x == A[i]. Otherwise, return -1"""
    1. i = 0
    2. while i < len(A):
    3.     if A[i] == x:
    4.         return i
    5.     i = i + 1
    6. return -1
```

What is the runtime of \( LS(A, x) \)?

If \( x \) is not in \( A \) at all, let \( n \) be the size of \( A \)

\[
t_{LS}(A, x) = 1 + 3n + 2 = 3n + 3
\]

\( t_{LS}([2, 4, 6, 8], 99) = 15 \)
Takeaway

✧ program runtime varies with inputs

✧ among inputs of a given size, there is a worst case in which the runtime is the longest
Worst-case time Complexity

$t_P(x)$: running time of program $P$ with input $x$

the worst-case time complexity of $P$
with input $x \in I$ of size $n$

$$W_P(n) = \max \{ t_P(x) \mid x \in I \land \text{size}(x) = n \}$$
Linear Search

```python
def LS(A, x):
    """Return index i, x == A[i]. Otherwise, return -1 """
    i = 0
    while i < len(A):
        if A[i] == x:
            return i
        i = i + 1
    return -1
```

What is the worst-case running time of $LS(A, x)$, given that $\text{len}(A) == n$?

$W_{LS}(n) = 1 + 3n + 2$

$= 3n + 3$

Worst-case: $x$ is not in $A$ at all!

$t_{LS}([2, 4, 6, 8], 99) = 15$
✧ **Worst-case**: performance in the worst situation, what we typically do in CSC165, and in CSC236

✧ **Best-case**: performance in the best situation, not very interesting, rarely studied

✧ **Average-case**: the expected performance under random inputs following certain probability distribution, will study in CSC263
Next class

➔ More on asymptotic notations & definitions
➔ Algorithm analysis