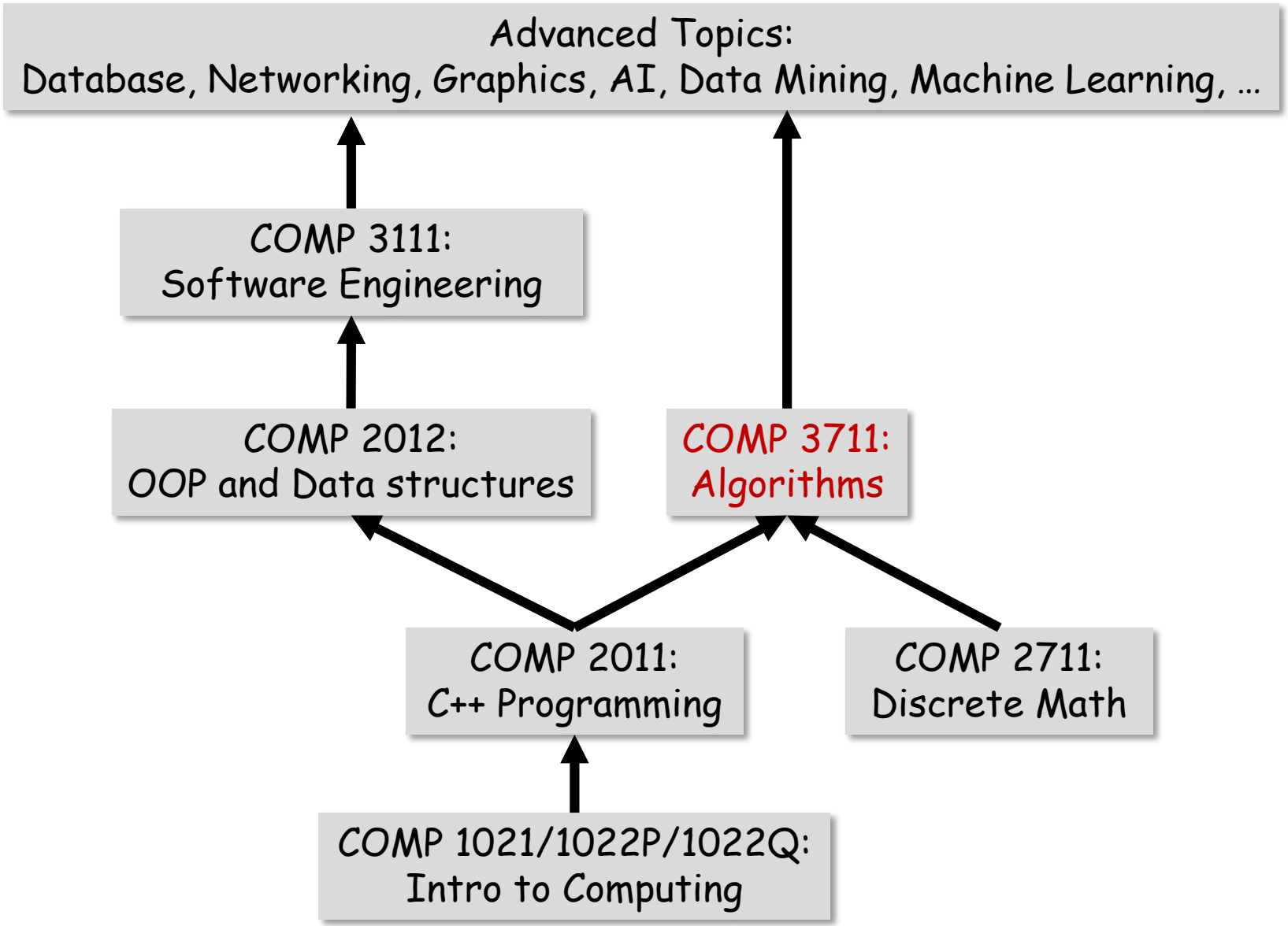
COMP 3711 Design and Analysis of Algorithms

Lecture 1: Introduction

Version of January 21, 2019

Where are we?


Advanced Topics:
Database, Networking, Graphics, AI, Data Mining, Machine Learning, …
COMP 3111:
Software Engineering
COMP 2012:
OOP and Data structures
COMP 3711:
Algorithms
COMP 2011:
C++ Programming
COMP 2711:
Discrete Math
COMP 1021/1022P/1022Q:
Intro to Computing

What is an Algorithm?

Definition:

An algorithm is a recipe for doing something.

An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions.

Comes from al-Khwārizmi, the name of a 9th century Persian mathematician, astronomer, geographer, and scholar

This is NOT an algorithm

Problem: How to pass COMP 3711.

```
don't go to lectures
copy other students' homework
give best shot at the exam
if score > cutoff then
    say "yippee!"
else
    beg for good grade
```

- Contains ambiguous instructions
- Not (hopefully) correct

This IS an Algorithm: Adding Two Numbers

Input: Two numbers x and y (potentially very long), each consisting of n digits: $x = \overline{x_n x_{n-1} \dots x_1}, y = \overline{y_n y_{n-1} \dots y_1}$

Output: A number $z = \overline{z_{n+1} z_n \dots z_1}$, such that $z = x + y$.

$c \leftarrow 0$
for $i \leftarrow 1$ **to** n
    $z_i \leftarrow x_i + y_i + c$
    if $z_i \geq 10$ **then** $c \leftarrow 1,\ z_i \leftarrow z_i - 10$
        else $c \leftarrow 0$
$z_{n+1} \leftarrow c$

```
  529501233
+612345678
----------
 1141846911
```

You've been running algorithms all your life!

The Sorting Problem

Input: An array $A[1 \ldots n]$ of elements

[4 8 2 7 5 6 9 3]

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

[2 3 4 5 6 7 8 9]

Sorting is a very important (sub)routine that comes up all the time.

- One of the early **Killer-Apps** of Computing
  - In the 1960s', computer manufacturers estimated that more than 25% of their machines' running times were spent on sorting
- Even now, sorting is a common application AND a standard subroutine used in solutions to many other problems
- The Sorting Problem is also a good testbed to test out algorithmic techniques

The First Algorithm: Selection Sort

Input: An array $A[1 \ldots n]$ of elements

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

(*) n-1 passes
In pass i, find smallest item in $A[i \ldots n]$
and place it in $A[i]$

Intuition

1st Pass: {1 5 8 6 7 2 } → (1 {5 8 6 7 2})

2nd Pass: (1 {5 8 6 7 2}) → (1 2 {5 8 6 7})

3rd Pass: (1 2 {5 8 6 7}) → (1 2 5 {8 6 7})

4th Pass: (1 2 5 {8 6 7}) → (1 2 5 6 {8 7})

5th Pass: (1 2 5 6 {8 7}) → (1 2 5 6 7 {8}) → (1 2 5 6 7 8)

The First Algorithm: Selection Sort

Input: An array $A[1 \ldots n]$ of elements

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

(*) n-1 passes
In pass i, find smallest item in $A[i \ldots n]$ and swap it with $A[i]$

```
Selection-Sort(A):
for i ← 1 to n − 1
    for j ← i + 1 to n
        if A[i] > A[j] then
            swap A[i] and A[j]
```

Implementation

1st Pass: (5 1 8 6 7 2) → (1 5 8 6 7 2) → (1 5 8 6 7 2) → (1 5 8 6 7 2) → (1 5 8 6 7 2) → (1 5 8 6 7 2)

2nd Pass: (1 5 8 6 7 2) → (1 5 8 6 7 2) → (1 5 8 6 7 2) → (1 5 8 6 7 2) → (1 2 8 6 7 5)

3rd Pass: (1 2 8 6 7 5) → (1 2 6 8 7 5) → (1 2 6 8 7 5) → (1 2 5 8 7 6)

4th Pass: (1 2 5 8 7 6) → (1 2 5 7 8 6) → (1 2 5 6 8 7)

5th Pass: (1 2 5 6 8 7) → (1 2 5 6 7 8) → (1 2 5 6 7 8)

Correctness of Selection Sort

Claim: When Selection sort terminates, the array is sorted.

Proof: By induction on n.

When n = 1 the algorithm is obviously correct,

Assume that the algorithm sorts every array of size n-1 correctly

Now consider what the algorithm does on $A[1 \ldots n]$

1. It firsts puts the smallest item in $A[1]$

2. It then runs Selection sort on $A[2 \ldots n]$
   - By induction, this sorts the items in $A[2 \ldots n]$

3. Since $A[1]$ is smaller than every item in $A[2 \ldots n]$, all the items in $A[1 \ldots n]$ are now sorted.

Running Time of Selection Sort

What is meant by running "time"?

Seconds?

Total Operations Performed?

Memory Accesses?

```
Selection-Sort(A):
1. for i ← 1 to n
2.     for j ← i + 1 to n
3.         if A[i] > A[j] then
4.             swap A[i] and A[j]
```

We will count number of comparisons, i.e., # of times $A[i] > A[j]$ is run.
This "dominates" all of the other operations.

For any fixed i, line 3 calls $n - i$ values of j. Total # comparisons is

$$\sum_{i=1}^{n-1}(n-i) = \sum_{k=1}^{n-1} k = \frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$$

Alternatively, note that line 3 is run exactly once for every possible (i,j) pair with $1 \le i < j \le n$. This is exactly # of ways to choose a pair out of n items which is

$$\binom{n}{2} = \frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$$

The Second Sorting Algorithm: Insertion Sort

Input: An array $A[1 \ldots n]$ of elements

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

(*) Create Sorted $A[1 \ldots i]$ from already sorted $A[1 \ldots i-1]$ by ***inserting*** $A[i]$ into proper location.

Intuition

Insertion is performed by successively swapping $A[i]$ with items to its left until an item smaller than $A[i]$ is found.

i=2: (5 1 8 6 3 2) → (1 5 8 6 3 2)

i=3: (1 5 8 6 3 2) → (1 5 8 6 3 2)

i=4: (1 5 8 6 3 2) → (1 5 6 8 3 2)

i=5: (1 5 6 8 3 2) → (1 3 5 6 8 2)

i=6: (1 3 5 6 8 2) → (1 2 3 5 6 8)

The Second Sorting Algorithm: Insertion Sort

Input: An array $A[1 \ldots n]$ of elements

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

(*) Create Sorted $A[1 \ldots i]$ from already sorted $A[1 \ldots i-1]$ by ***inserting*** $A[i]$ into proper location.

Insertion is performed by successively swapping $A[i]$ with items to its left until an item smaller than $A[i]$ is found.

```
Insertion-Sort(A):
for i ← 2 to n do
    j ← i − 1
    while j ≥ 1 and A[j] > A[j + 1] do
        swap A[j] and A[j + 1]
        j ← j − 1
```

i=2: (5 1 8 6 3 2) → (1 5 8 6 3 2) → (1 5 8 6 3 2)

i=3: (1 5 8 6 3 2) → (1 5 8 6 3 2)

i=4: (1 5 8 6 3 2) → (1 5 6 8 3 2) → (1 5 6 8 3 2)

i=5: (1 5 6 8 3 2) → (1 5 6 3 8 2) → (1 5 3 6 8 2) → (1 3 5 6 8 2) → (1 3 5 6 8 2)

i=6: (1 3 5 6 8 2) → (1 3 5 6 2 8) → (1 3 5 2 6 8) → (1 3 2 5 6 8) → (1 2 3 5 6 8)
→ (1 2 3 5 6 8)

The Second Sorting Algorithm: Insertion Sort

Input: An array $A[1 \ldots n]$ of elements

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

(*) Create Sorted $A[1 \ldots i]$ from already sorted $A[1 \ldots i-1]$ by ***inserting*** $A[i]$ into proper location.

Insertion is performed by successively swapping $A[i]$ with items to its left until an item smaller than $A[i]$ is found.

```
Insertion-Sort(A):
for i ← 2 to n do
    j ← i − 1
    while j ≥ 1 and A[j] > A[j + 1] do
        swap A[j] and A[j + 1]
        j ← j − 1
```

Correctness: After step i, items in A[1..i] are in proper order.
i 'th iteration puts key=A[i] in proper place.

Running Time of Insertion Sort

```
Insertion-Sort(A):
1. for i ← 2 to n do
2.     j ← i − 1
3.     while j ≥ 1 and A[j] > A[j + 1] do
4.         swap A[j] and A[j + 1]
5.         j ← j − 1
```

Line 3 is the dominant part.

# of times line 3 is run is at most

$$\sum_{i=2}^{n}(i-1) = \sum_{k=1}^{n-1} k = \frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$$

So, Insertion Sort is never worse than Selection sort which always uses $\frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$.

This "worst case" of every item being compared to every other item occurs, for example (why?) when the array is given in reverse sorted order, e.g., (6 5 4 3 2 1).

If original array is sorted, e.g., (1 2 3 4 5 6), then no item is ever moved.
Each item is only compared to its left neighbor, so line 3 is only run $\boldsymbol{(n-1)}$ times.

How should we describe the running time of Insertion Sort? $\frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$? $\boldsymbol{(n-1)}$?

Is Insertion Sort Better than Selection Sort? What does *Better* even mean?

Wild-Guess Sort

Input: An array $A[1 \ldots n]$ of elements

Output: Array $A[1 \ldots n]$ of elements in sorted order (ascending)

Wild-Guess-Sort(A):
$\pi \leftarrow [4, 7, 1, 3, 8, 11, 5, \ldots]$
check if $A[\pi[i]] \leq A[\pi[i+1]]$ **for all** $i = 1, 2, \ldots, n-1$
if yes, output A **according to** π **and terminate**
else Insertion-Sort(A)

Q: Is wild-guess sort faster than insertion sort?

A: Yes, when the input exactly agrees with the guess.

A: But for all other inputs, it is slower.

How to evaluate an algorithm / compare two algorithms?

What to measure?

- **Memory** (space complexity)
  - total space
  - working space (excluding the space for holding inputs)
- **Running time** (time complexity)

How to measure?

- **Empirical** – depends on actual implementation, hardware, etc.
- **Analytical** – **depends only on the algorithms, focus of this course**

Comparing two algorithms is not (usually) a simple matter!

- Depends on the input size n
- Even for same n, different inputs can lead to different running times
- Calculating exact # of instructions executed is very difficult/tedious

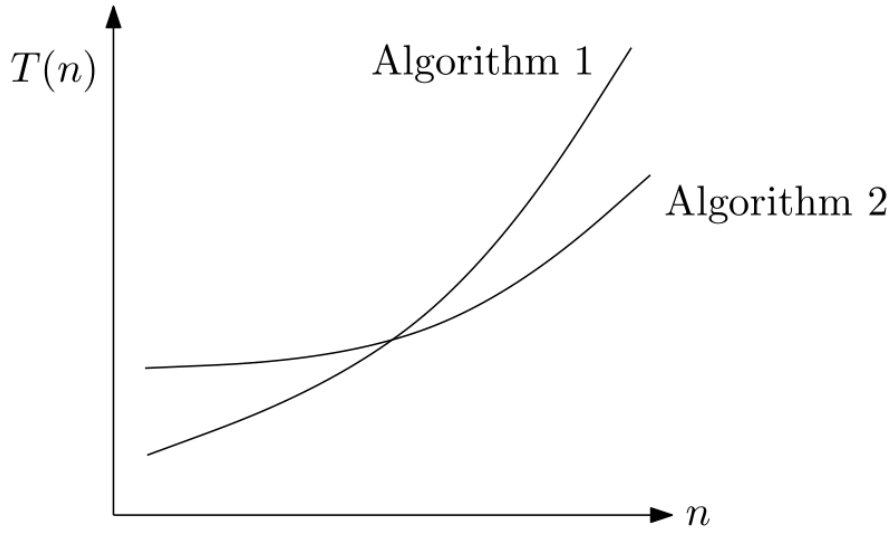
Very rarely (and difficult) can we draw conclusions like

"Insertion sort is always better than Selection Sort"

Measuring Running Time

We will

- measure running time as the number of machine instructions
- measure running time as a function of input size: $T(n)$
- use worst-case analysis
- use asymptotic notation



Which algorithm is better for large n?

- For Algorithm 1, $T(n) = 3n^3 + 6n^2 - 4n + 17 = \Theta(n^3)$
- For Algorithm 2, $T(n) = 7n^2 - 8n + 20 = \Theta(n^2)$
- Clearly, Algorithm 2 is better **for large enough n**

Best-Case Analysis

Best case: An instance for a given size n that results in the fastest possible running time.

Example (insertion sort): Input already sorted

```
Insertion-Sort(A):
for i ← 2 to n do
    j ← i − 1
    while j ≥ 1 and A[j] > A[j + 1] do
        swap A[j] and A[j + 1]
        j ← j − 1
```

"key" is only compared to element to its immediate left, so

$$T(n) = n - 1 = \Theta(n).$$

Worst-Case Analysis

Worst case: An instance for a given size n that results in the slowest possible running time.

Example (insertion sort): Input inversely sorted

```
Insertion-Sort(A):
for i ← 2 to n do
    j ← i − 1
    while j ≥ 1 and A[j] > A[j + 1] do
        swap A[j] and A[j + 1]
        j ← j − 1
```

"key" is compared to every element preceding it, so

$$T(n) = \Theta\left(\sum_{i=2}^{n}(i-1)\right) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2).$$

Average-Case Analysis

Average case: Running time averaged over all possible instances for the given size, assuming some probability distribution on the instances.

Example (insertion sort): assuming that each of the $n!$ permutations of the n numbers is equally likely

Rigorous analysis is complicated, but intuitively, "key" is compared, on average to half the items preceding it, so

$$T(n) = \Theta\left(\sum_{i=2}^{n} \frac{i-1}{2}\right) = \Theta\left(\frac{n(n-1)}{4}\right) = \Theta(n^2).$$

Three Kinds of Analyses

Best case: Clearly useless

Worst case: Commonly used

Gives running time guarantee ***independent of actual input***

- Fair comparison among different algorithms
- Not perfect: For some problems, worst-case input never occurs in real life; some algorithms with bad worst-case running time actually work very well in practice (e.g. the simplex algorithm for linear programming)
- **Worst-case analysis is the default (used in this class)**

Average case: Sometimes used

- Needs to assume some input distribution:

  but real-world inputs are seldom uniformly random!
- Analysis is complicated
- Will see two examples later

Asymptotic Notation: Quick Revision

Upper bounds. $T(n) = O(f(n))$

if exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $T(n) \leq c \cdot f(n)$.

Equivalent definition: $\lim_{n \to \infty} \frac{T(n)}{f(n)} < \infty$.

Lower bounds. $T(n) = \Omega(f(n))$

if exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $T(n) \geq c \cdot f(n)$.

Equivalent definition: $\lim_{n \to \infty} \frac{T(n)}{f(n)} > 0$.

Tight bounds. $T(n) = \Theta(f(n))$

if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

Note: Here "=" means "is", not equal.

More mathematically correct expression should be $T(n) \in O(f(n))$.

Example: $T(n) = 32n^2 + 17n - 32$.

- $T(n)$ is $O(n^2), O(n^3), \Omega(n^2), \Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n), \Omega(n^3), \Theta(n)$, or $\Theta(n^3)$.

Asymptotic notation: Quick Revision 2

Knowing/understanding this notation is essential to taking this class. Students are expected to have learned it in COMP2711. If you have forgotten the material or have not taken COMP2711 please review the revision slides on the tutorial web page.

- $9999^{9999^{9999}} = \Theta(1)$
- 2^{10n} is not $O(2^n)$

- $\sum_{i=1}^{n} i \leq n \cdot n = O(n^2)$, $\sum_{i=1}^{n} i \geq \frac{n}{2} \cdot \frac{n}{2} = \Omega(n^2)$ => $\boxed{\sum_{i=1}^{n} i = \Theta(n^2)}$
- $\sum_{i=1}^{n} i^2 \leq n^2 \cdot n = O(n^3)$, $\sum_{i=1}^{n} i^2 \geq \left(\frac{n}{2}\right)^2 \cdot \frac{n}{2} = \Omega(n^3)$ => $\boxed{\sum_{i=1}^{n} i^2 = \Theta(n^3)}$
- $\boxed{\sum_{i=1}^{n} c^i = \frac{c^n - 1}{c-1} = \begin{cases} \Theta(c^n), c > 1 \\ \Theta(n), c = 1 \\ \Theta(1), c < 1 \end{cases}}$ (geometric series)
- $\boxed{\Theta(f(n) + g(n)) = \Theta\big(\max\big(f(n), g(n)\big)\big)}$

Asymptotic notation: Revision with more examples (II)

- $\log_{10} n = \frac{\log_2 n}{\log_2 10} = \Theta(\log_2 n) = \Theta(\log n)$
- $\log^{10} n = O(n^{0.1}),\ n^{100} = O(2^n),\ \ \text{loglog}\, n = O(\log n)$
- $n \log n = O\left(\frac{n^2}{\log n}\right)$
- $n^{0.1} + \log^{10} n = \Theta(n^{0.1})$

- Harmonic Series: $\boxed{\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)}$

- Simple case of Stirling's fomula
  - $\log(n!) = \log(n) + \log(n-1) + \cdots + \log 1 = O(n \log n)$
  - $\log(n!) \geq \log(n) + \log(n-1) + \cdots + \log\left(\frac{n}{2}\right) = \Omega(n \log n)$
  - => $\boxed{\log(n!) = \Theta(n\ \log n)}$

More on Worst-Case Analysis

What does each of these statements mean? (*n is the instance size)*

The algorithm's worst case running time is $O(f(n))$

- On all inputs of size n, the algorithm's running time is $O(f(n))$

The algorithm's worst case running time is $\Omega(f(n))$

- For every large enough n there exists at least one input of size n
  for which the running time of the algorithm is $\geq c \cdot f(n)$

More on Worst-Case Analysis (Expanded)

What does each of these statements mean? (*n is the instance size*)

The algorithm's worst case running time is $O(f(n))$

- On all inputs of size n, the algorithm's running time is $O(f(n))$

Further expanded:
There exist constants $c > 0, n_0 \geq 0$, such that for any $n \geq n_0$ and all inputs of size n, the algorithm's running time is $\leq c \cdot f(n)$.
- Implication 1: No need to really find the worst input.
- Implication 2: No need to consider input of size smaller than a constant n_0.

The algorithm's worst case running time is $\Omega(f(n))$

- For every large enough n there exists at least one input of size n for which the running time of the algorithm is $\geq c \cdot f(n)$

Further expanded:
There exist const $c > 0, n_0 \geq 0$, s.t. for any $n \geq n_0$, there exists some input of size n on which the algorithm's running time is $\geq c \cdot f(n)$.
- Mainly used to show that the big-Oh analysis is tight (i.e., the best possible upper bound); often not required.

More on Worst-Case Analysis

Example: Insertion Sort (*n is the instance size*)

We saw that, on all inputs of size n, Insertion Sort runs in $\leq \frac{n(n-1)}{2}$ *time.*

On some inputs, e.g., if the items are in reverse order, it ***requires*** $\frac{n(n-1)}{2}$ *time, while on others, e.g., already sorted data, it only takes* n *time*

⇒ Insertion sort runs in $O(n^2)$ time (upper bound)
Insertion sort runs in $\Omega(n^2)$ time (lower bound)

⇒ Insertion sort runs in $\theta(n^2)$ time

Simple theoretical analysis is not the end of the story

Example: Selection sort, insertion sort, and wild-guess sort all have worst-case running time $\Theta(n^2)$. How to distinguish between them?

Loses information due to

- Asymptotic notation suppresses multiplicative constants
- Worst-case analysis ignores non worst-case inputs, which could be "normal" input.

But, theoretical analysis provides first guidelines

- Useful when you don't know what inputs to expect
- An $\Theta(n \log n)$ algorithm is always better than an $\Theta(n^2)$ algorithm, for inputs large enough (there are some exceptions, though)
  - Will see several $\Theta(n \log n)$ sorting algorithms later

When algorithms have the same theoretical running time differentiate via a

- Closer examination of hidden constants
- Careful analysis of typical expected inputs
- Other factors such as cache efficiency, parallelization are important
- Empirical comparison

Writing down algorithms in pseudocode

- How should we describe algorithms
  - Implementable code is too hard to read
    - Also, "most" lines of code, while important in practice, are not important algorithmically and would just hide the main ideas
  - Natural language is usually not descriptive enough to concisely describe algorithmic ideas

- Will often use "pseudocode"
  - Tries to get across main ideas clearly. Models programming syntax but also uses natural language.
  - Not formally defined
  - Ad Hoc "rules" on next page

Insertion-Sort(A):

for $j \leftarrow 2$ **to** n **do**
    $i \leftarrow j - 1$
    while $i \geq 1$ **and** $A[i] > A[i+1]$ **do**
        swap $A[i]$ **and** $A[i+1]$
        $i \leftarrow i - 1$

Wild-Guess-Sort(A):

$\pi \leftarrow [4, 7, 1, 3, 8, 11, 5, \dots]$
check if $A[\pi[i]] \leq A[\pi[i+1]]$ **for all** $i = 1, 2, \dots, n-1$
if yes, output A **according to** π **and terminate**
else Insertion-Sort(A)

Writing down algorithms in pseudocode

- Use standard keywords (`if/then/else, while, for, repeat/until, return`) and notation: variable ← value, Array[index], function(arguments), etc.
- Indent everything carefully and consistently; may also use **{ }** for clarity.

- Use **standard** mathematical notation. For example,
  write i = i+1 instead of i++
  write $x \cdot y$ instead of $x * y$; write $x \bmod y$ instead of $x \,\%\, y$
  write $\sqrt{x}$ instead of $sqrt(x)$; write a^b instead of $power(a, b)$
  use $=$ for equality test instead of $==$

- Use data structures as black boxes. If data structure is new, define its functionality first; then describe how to implement each operation.
- Use standard/learned algorithms (e.g. sorting) as black boxes
- Use functions to decompose complex algorithms.

- Use language when it's clearer or simpler (e.g., if A is an array, you may write "$x \leftarrow$ the maximum element in A").