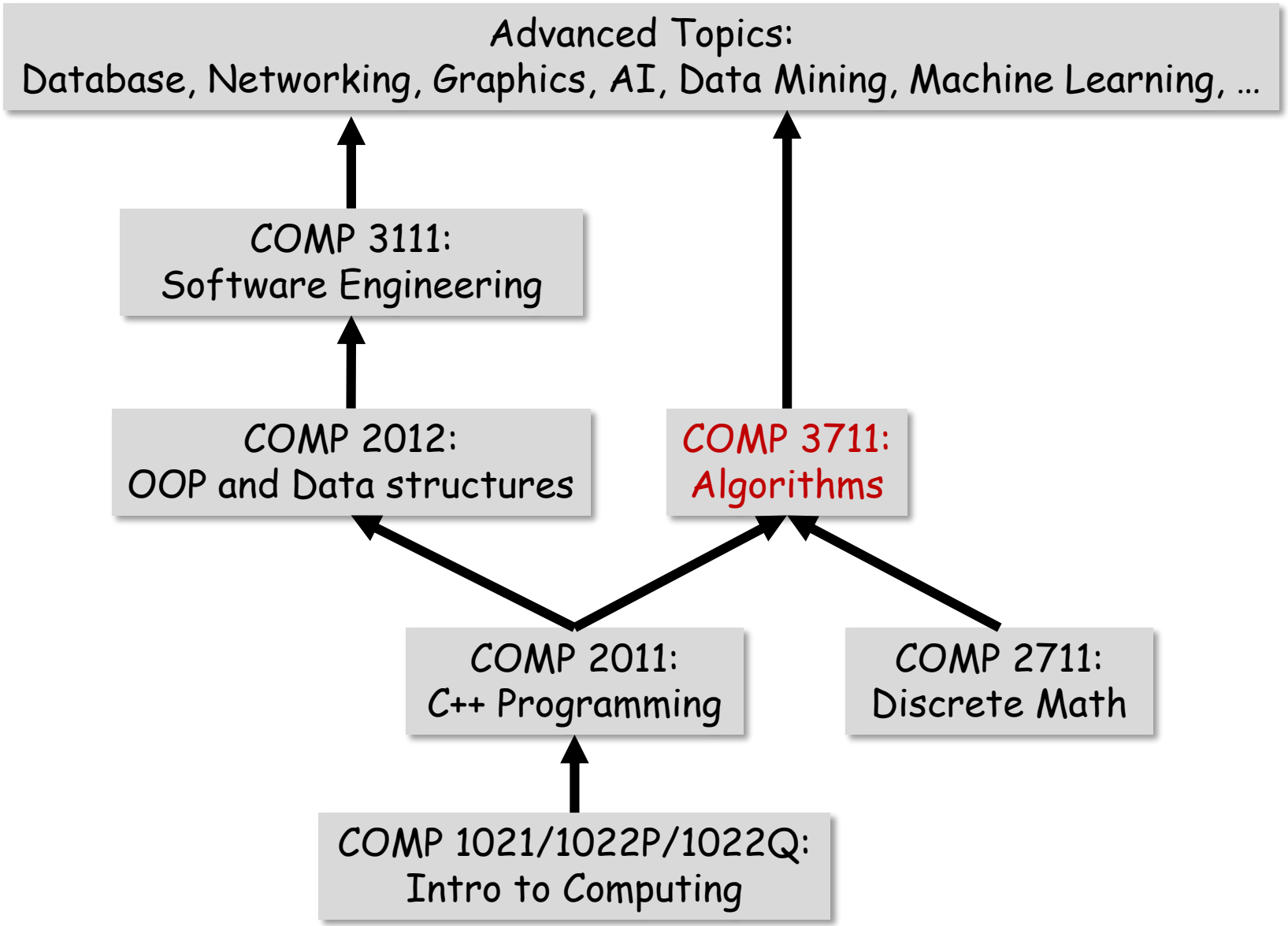
COMP 3711 Design and Analysis of Algorithms

Lecture 1: Introduction

So, where are we so far?



Course Organization

Lectures: Wed & Fri 3-4:20pm (L2), 4:30-5:50pm (L1).

Textbook:

- (Required) T. Cormen, C. Leiserson, R. Rivest, C. Stein. Introduction to Algorithms, Third Edition, MIT Press.
  - Library has e-version
- (Reference) Jon Kleinberg and Éva Tardos. Algorithm Design, Addison-Wesley.

Tutorials:

- Starting from week 2.
- All sections in the same week cover the same topics.

Grading:

- 4 Written assignments: 5% * 4 = 20%
- 4 Programming assignments: optional, each one carries 1% extra credit
- Midterm exam: 30%
- Final exam: 50%

What is an Algorithm?

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

Example: How to sing "5 little monkeys jumping on the bed"

```
for n ← 5 downto 1
    sing "n little monkeys jumping on the bed."
    if n = 1 then sing "He fell off and bumped his head."
             else sing "One fell off and bumped his head."
    sing "Mama called the doctor and the doctor said,"
    sing "No more monkeys jumping on the bed!"
```

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
----------
 1241846911
```

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
    cry and beg the instructor
```

- Contains ambiguous instructions
- Not (always) correct

The First Algorithm: Bubble Sort

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

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

```
Bubble-Sort(A):
repeat
    swapped ← false
    for i ← 1 to n − 1
        if A[i] > A[i + 1] then
            swap A[i] and A[i + 1]
            swapped ← true
until not swapped
```

1st Pass: (4 1 8 2 5) → (1 4 8 2 5) → (1 4 2 8 5) → (1 4 2 5 8)

2nd Pass: (1 4 2 5 8) → (1 2 4 5 8)

3rd Pass: No swaps, thus terminate

Correctness of bubble sort

Claim: When bubble sort terminates, the array must be sorted.

Proof: Trivial.

Claim: Bubble sort terminates after at most $n - 1$ passes.

Proof:

- After the 1st pass, the largest element must be at $A[n]$, and it will not be swapped any more.
- After the 2nd pass, the 2nd largest element must be at $A[n-1]$, and it will not be swapped any more.
- …
- After the $(n-1)$-th pass, the 2nd smallest element must be at $A[2]$, and the smallest element must be at $A[1]$.

The First (Reasonably) Good Sorting Algorithm

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

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

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

Correctness: Each iteration of the outer loop finds the right position to put "key".

Termination: Obvious

Insertion Sort: Example

1st iteration:

- (4 1 8 2 5) → (4 4 8 2 5) → (1 4 8 2 5)
- key = 1

2nd iteration:

- (1 4 8 2 5)
- key = 8

3rd iteration:

- (1 4 8 2 5) → (1 4 8 8 5) → (1 4 4 8 5) → (1 2 4 8 5)
- key = 2

4th iteration:

- (1 2 4 8 5) → (1 2 4 8 8) → (1 2 4 5 8)
- key = 5

Bubble Sort vs Insertion Sort

Memory

- Both operate directly on the input array A (in-place algorithm)
- Both require 3 extra variables (working memory)

Running time?

Rewriting the insertion sort algorithm:

Insertion-Sort(A):

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

Insertion-Sort-Using-Swaps(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** $A[i+1]$
        $i \leftarrow i - 1$

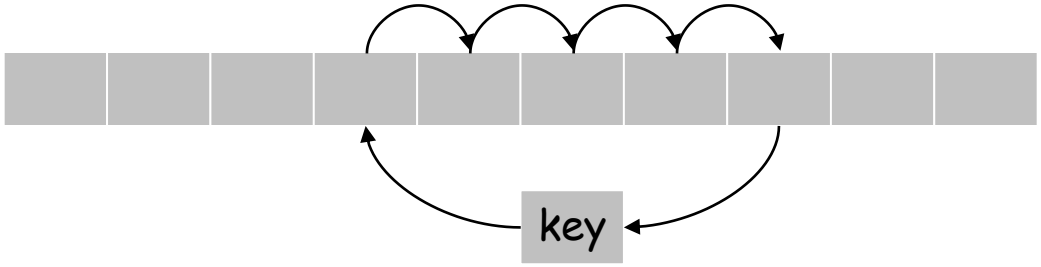
Bubble Sort vs Insertion Sort

Theorem: On any input, **`Insertion-Sort-Using-Swaps`** uses exactly the same number of swaps as bubble sort.

Proof: Later (relates to "inversion number").

Observations:

- Each swap is implemented as 3 assignments:
  - $x \leftarrow A[i]; A[i] \leftarrow A[j]; A[j] \leftarrow x$
- Insertion sort essentially implements a series of swaps more efficiently:



Conclusion: Insertion sort is always better than bubble sort.

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):
π ← [4, 7, 1, 3, 8, 11, 5, ...]
check if A[π[i]] ≤ A[π[i + 1]] for all i = 1, 2, ..., 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 no simple matter!

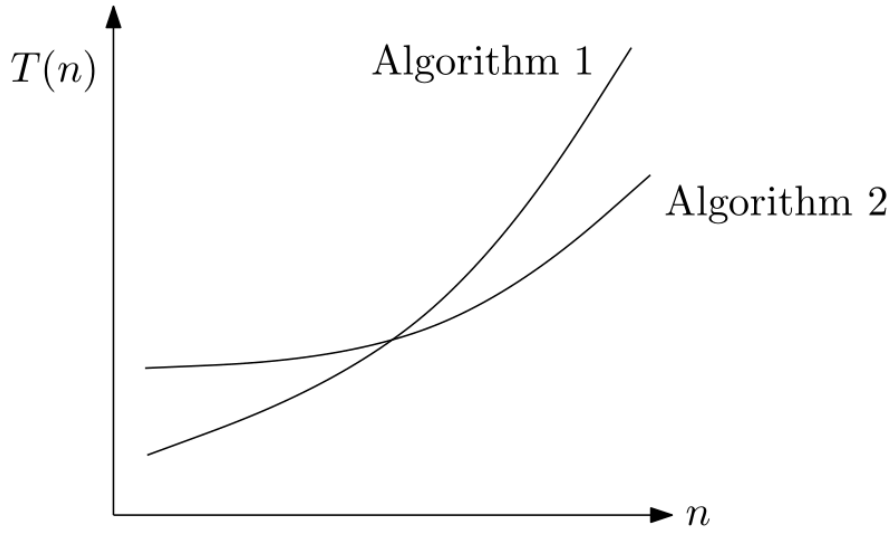
- Depends on the input, especially the input size n
- Depends on what operations to measure
- Calculating the exact number of instructions executed is very difficult or even impossible

Very rarely (and difficult) can we draw conclusions like "insertion sort is always better than bubble 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 asymptotic notation
- use worst-case analysis



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

Asymptotic Notation

Upper bounds. $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.

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

Lower bounds. $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.

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. The more mathematically correct way should be $T(n) \in O\big(f(n)\big)$.

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: More examples

- $\log_{10} n = \frac{\log_2 n}{\log_2 10} = \Theta(\log_2 n) = \Theta(\log n)$
- $9999^{9999^{9999}} = \Theta(1)$
- the number of 1's in the binary representation of $n = O(\log n), \Omega(1)$
- 2^{10n} is not $O(2^n)$
- $\sum_{i=1}^{n} i \leq n \cdot n = O(n^2), \sum_{i=1}^{n} i \geq \frac{n}{2} \cdot \frac{n}{2} = \Omega(n^2)$
- $\sum_{i=1}^{n} i^2 \leq n^2 \cdot n = O(n^3), \sum_{i=1}^{n} i \geq \left(\frac{n}{2}\right)^2 \cdot \frac{n}{2} = \Omega(n^3)$
- $\sum_{i=1}^{n} c^i = \frac{c^n - 1}{c - 1} = \begin{cases} \Theta(c^n), c > 1 \\ \Theta(n), c = 1 \\ \Theta(1), c < 1 \end{cases}$ (geometric series)
- $\log(n!) = \log(n) + \log(n-1) + \cdots + \log 1 = O(n \log n)$
  $\log(n!) \geq \log(n) + \log(n-1) + \cdots + \log(n/2) = \Omega(n \log n)$

Asymptotic notation: More examples

- $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$ (harmonic series, derivation on board)
- $\log^{10} n = O(n^{0.1}), n^{100} = O(2^n), \text{loglog}\, n = O(\log n)$
- $n \log n = O\left(\frac{n^2}{\log n}\right)$
- $n^{0.1} + \log^{10} n = \Theta(n^{0.1})$
- $f(n) + g(n) = \Theta\big(\max\big(f(n), g(n)\big)\big)$

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

```
for j ← 2 to n do
    key ← A[j]
    i ← j − 1
    while i ≥ 1 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```

"key" is compared to only the element right before it, so $T(n) = \Theta(n)$.

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

```
for j ← 2 to n do
    key ← A[j]
    i ← j − 1
    while i ≥ 1 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```

"key" is compared to every element before it, so $T(n) = \Theta(\sum_{i=2}^{n} i) = \Theta(n^2)$.

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 intuitive, "key" is compared to half of the elements before it on average, so

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

Three Kinds of Analyses

Best case: Clearly useless

Worst case: Commonly used, will also be used in this course

- Gives a running time guarantee no matter what the input is
- Fair comparison among different algorithms
- Not perfect: For some problems, the 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 will be the default

Average case: Used sometimes

- Need to assume some distribution: real-world inputs are seldom uniformly random!
- Analysis is complicated
- Will see one example later

More on Worst-Case Analysis

What does each of these statements mean?

The algorithm's running time is $O(n^2)$

- On any input, the algorithm's running time is $O(n^2)$
- Further expanded: There exist constants $c > 0, n_0 \geq 0$, such that for any $n \geq n_0$ and any input of size n, the algorithm's running time is $\leq c \cdot n^2$.
- 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 running time is $\Omega(n^2)$

- There exist constants $c > 0, n_0 \geq 0$, such that for any $n \geq n_0$, there exists some input of size n on which the algorithm's running time is $\geq c \cdot n^2$.
- Mainly used to show that the big-Oh analysis is tight (i.e., the best possible upper bound); often not required.

Theoretical analysis may not be accurate enough

Example: Bubble sort, insertion sort, and wild-guess sort all have running time $\Theta(n^2)$

Lose information due to

- Asymptotic notation suppresses constant difference
- Worst-case analysis ignores other inputs

But, theoretical analysis provides the first guideline

- Useful when you don't know what inputs you are to get
- 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

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

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 better clarity.
- Use standard mathematical notation. For example, write $x \cdot y$ instead of $x * y$ for multiplication; write $x \bmod y$ instead of $x \,\%\, y$ for remainder; write $\sqrt{x}$ instead of $sqrt(x)$ for square roots; write a^b instead of $power(a, b)$ for exponentiation; use $=$ for equality test.
- Use data structures as black boxes. If the 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").