EE2331 Data Structures and Algorithms

Introduction and a sorting algorithm

Today's agenda:

1) Introduce basic concepts of data structure and algorithm

2) Insertion sort

Two major topics

- Basic data structures
- Algorithms

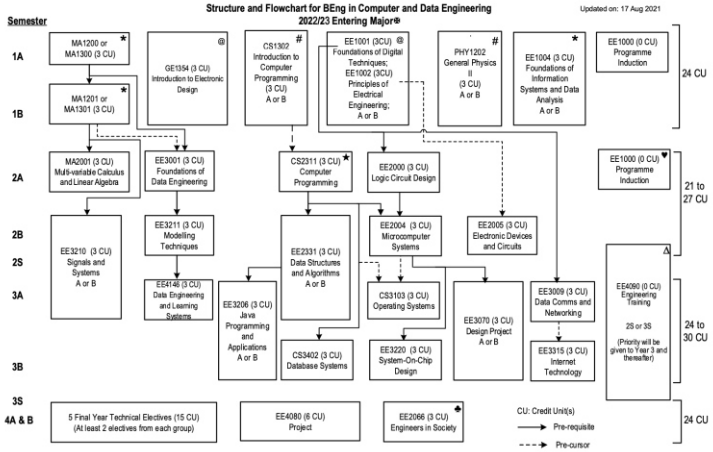
You are given as set of cards with students' names on them. How to **organize them** so that we can **easily** find a student, remove a student, add a student, sort the student by names?

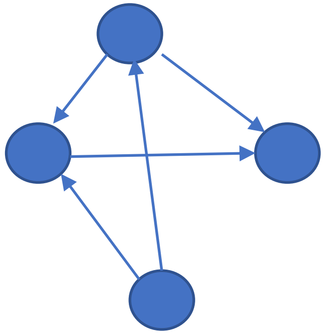
I have about 100 students in EE2331. How should I organize the test papers so that a student can find his/her test paper fast?

Data structures

- Data structure provides a way to organize data items. Each data structure has associated operations
  - What are the data structures in the following examples?
    - Example 1. MTR map

Example 2: flow chart of CDE



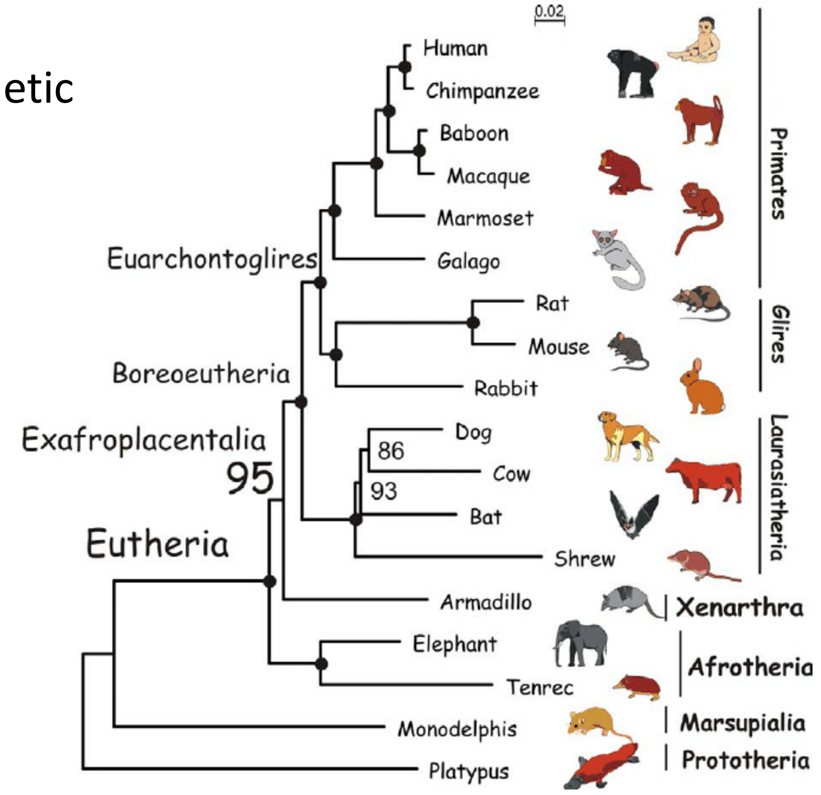


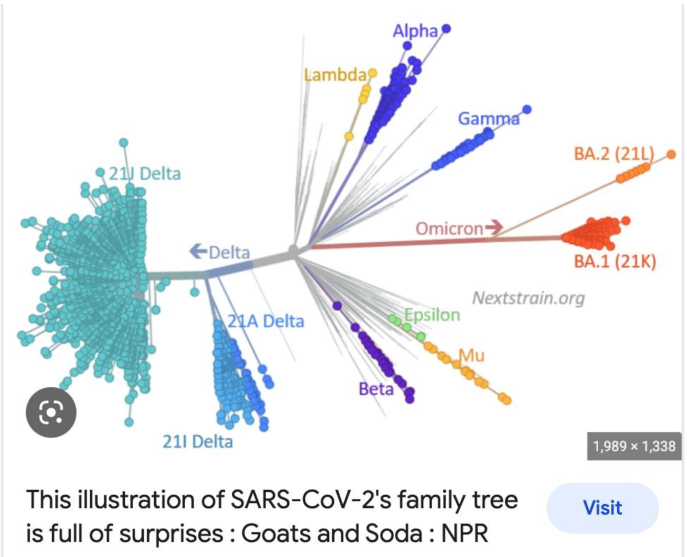
graph

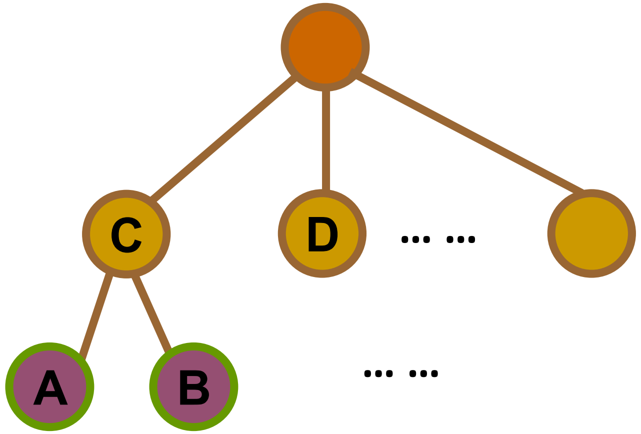
Data structures

- Data structure provides a way to organize data items. Each data structure has associated operations
  - What are the data structures in the following examples?

Example: Phylogenetic tree







Tree -> not reliable

Data structures

- Data structure provides a way to organize data items. Each data structure has associated operations
  - What are the data structures in the following examples?

Name	Age
Andy	5
Judy	6
Mathew	7
Raymond	5.5
Hayden	7.5

list

Andy	judy					

Question: How to you organize all the students' records at CityU? What data structure do you choose?

Algorithms

- "a bag of tricks", can be executed systematically by the computer

Input → algorithm → Output

Basic algorithmic techniques:

- **sorting, searching**

A way to think about computation

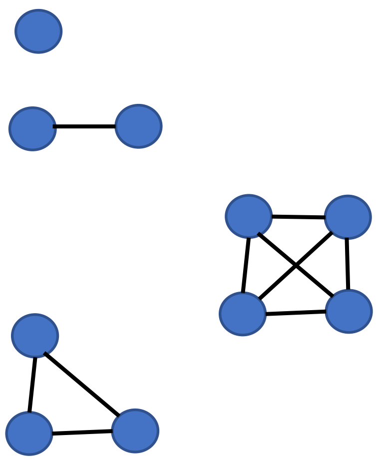
- What is a "good" algorithm?
- What does "fast/ faster" mean?

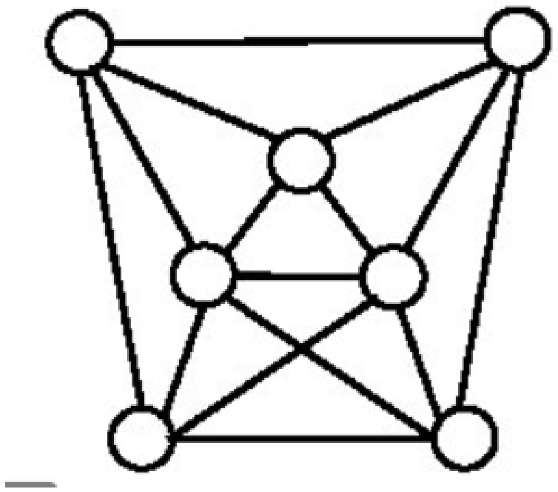
Algorithms

- Example problems.
  The **Subway Challenge** is a challenge in which participants must navigate the entire New York City Subway system in the shortest time possible.
  - Can you solve this on Hong Kong MTR?

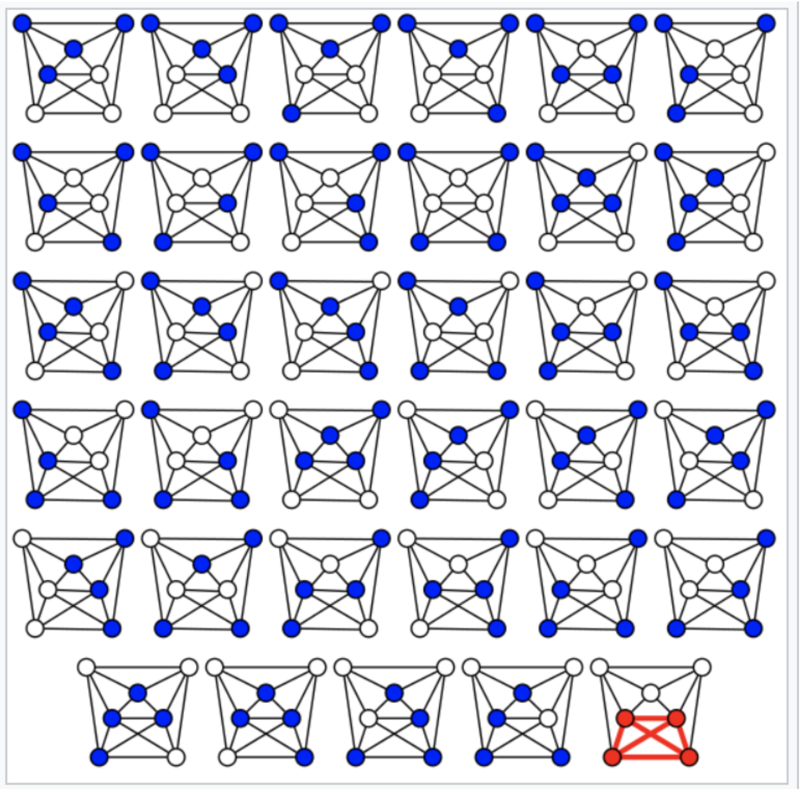
Algorithms

- Clique problem. What is a clique: subsets of vertices, all adjacent to each other, also called complete subgraphs in a graph
- **Maximum clique problem: identify the maximum clique in a graph. Can you come up with an algorithm?**





What is the maximum clique in this graph?



Brue force algorithm

Algorithms

- Analytical techniques
  - Asymptotic notation (next topic, stay tuned)

- **Summary**
  - Data structure & algorithms are practical and basic to computer science culture (more than just writing code)

First problem: Sorting

- Input: a sequence of n numbers

$$<a_1, a_2, \ldots, a_n>$$

- Output: a permutation (re-ordering)

$< a_1', a_2', \ldots, a_n'>$ of the input sequences,

s.t. $a_1' \leq a_2' \leq a_3' \leq \ldots \leq a_n'$

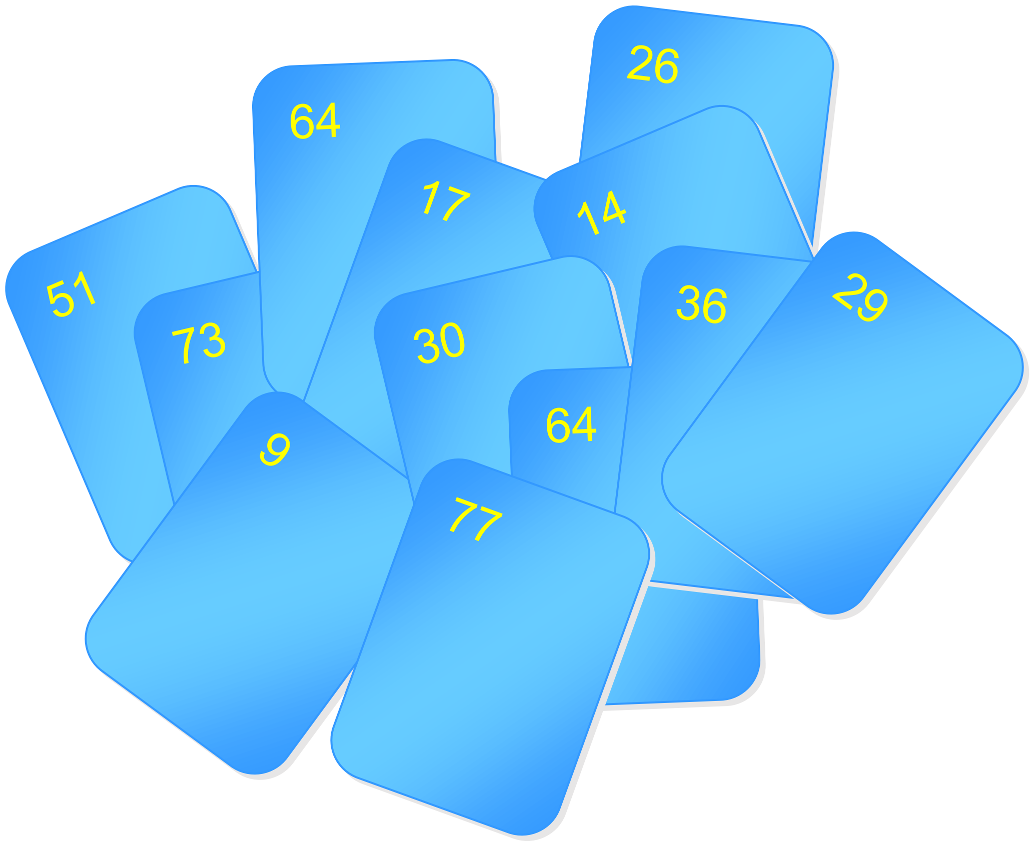
50	100	1	0		0	1	50	100
a_1	a_2	a_3	a_4	$\Rightarrow$	a_1'	a_2'	a_3'	a_4'

Question: can you think of some examples of sorting in real life?

(dictionary is a very good example)

If a list is sorted…

- How to find the largest number?
- How to find the smallest number?
- How to determine if an arbitrary number exists in the list?



Sorting

- To rearrange the order (ascending, descending, increasing, decreasing, non-decreasing) of data for **ease of searching**

- We will discuss various ways to sort a large amount of data and compare them by time/space efficiency.

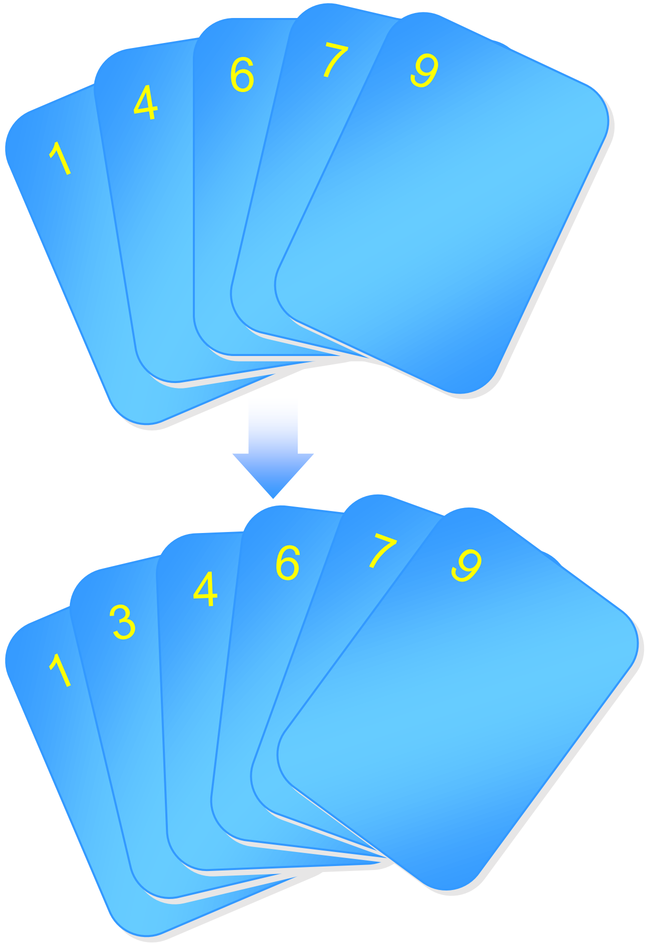
Insertion Sort

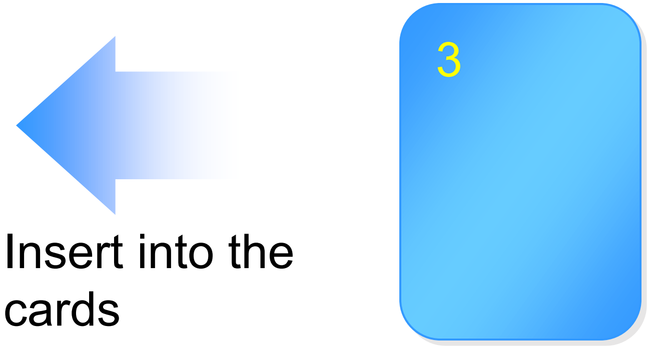
A basic sorting algorithm.

Basic operation: insert an element into a sorted list such that the final list is still sorted

Daily Life Example

- The idea of insertion is like playing cards





Insert into the cards

Any pre-condition?

Insertion Sort

- Insertion sort successively inserts a new element into a (sorted) sublist in each pass
- Initially 1st element may be thought of as a sorted sublist of only one element
- After each sorted-insertion, the sorted sublist's length grows by 1.
- Insertion sort makes use of the fact that elements in the sublist are already known to be in sorted order.

Insertion Sort Example

The unsorted list:

The 1st pass

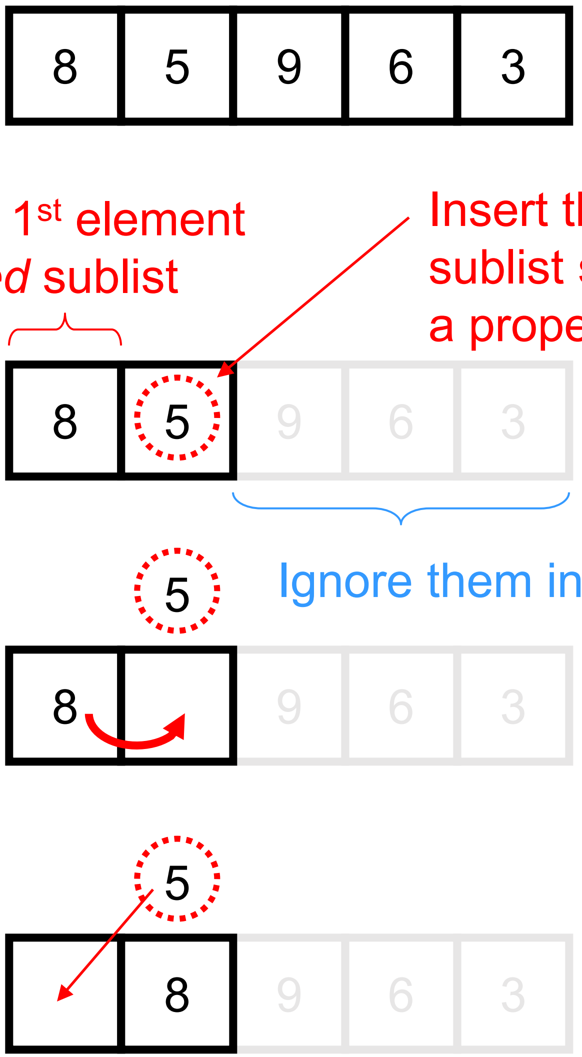
Consider the 1st element as a *sorted* sublist

Insert this element into the left sublist such that they maintain a proper order

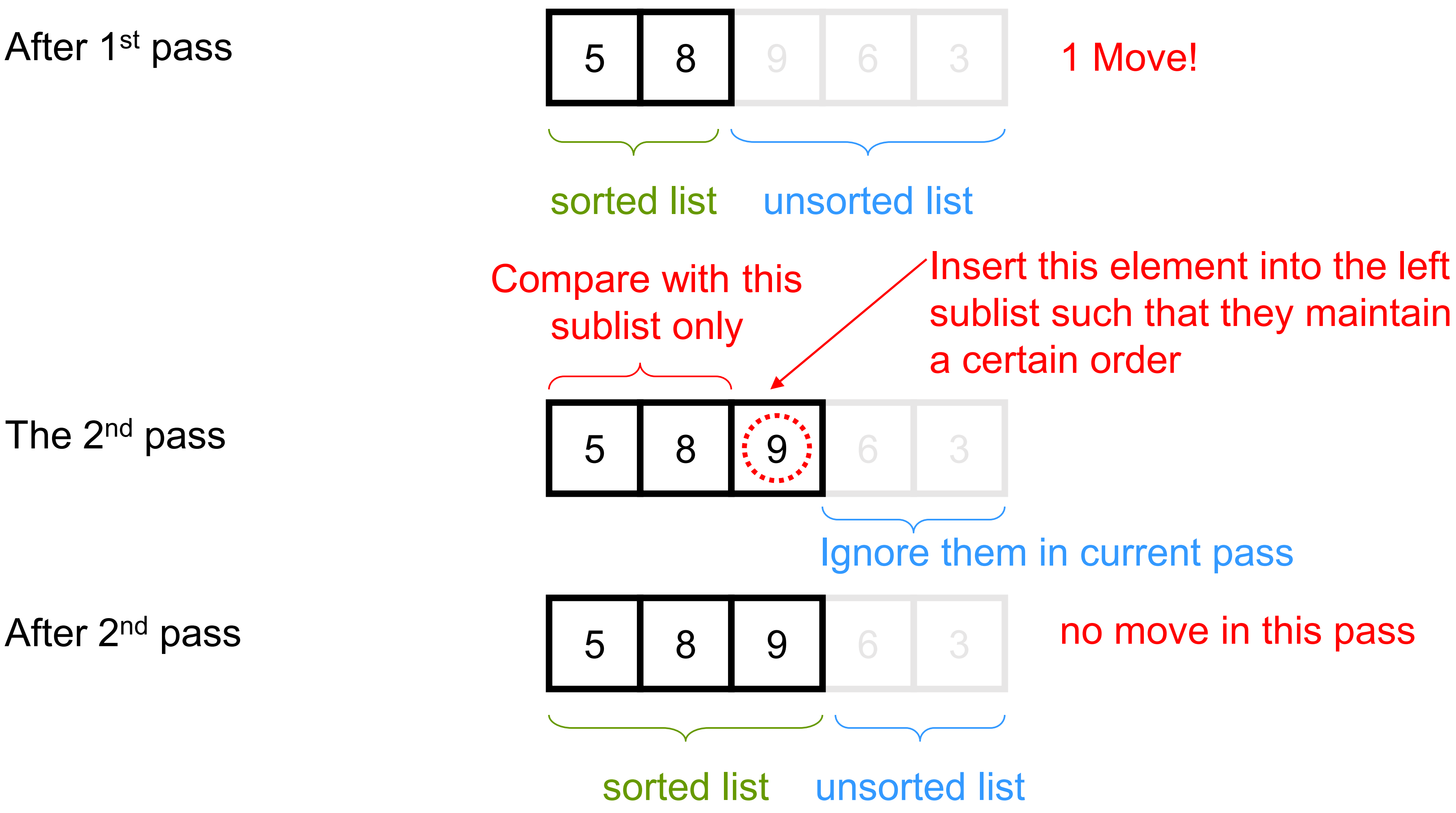
Ignore them in current pass

Pick up “5”. Move “8” to right

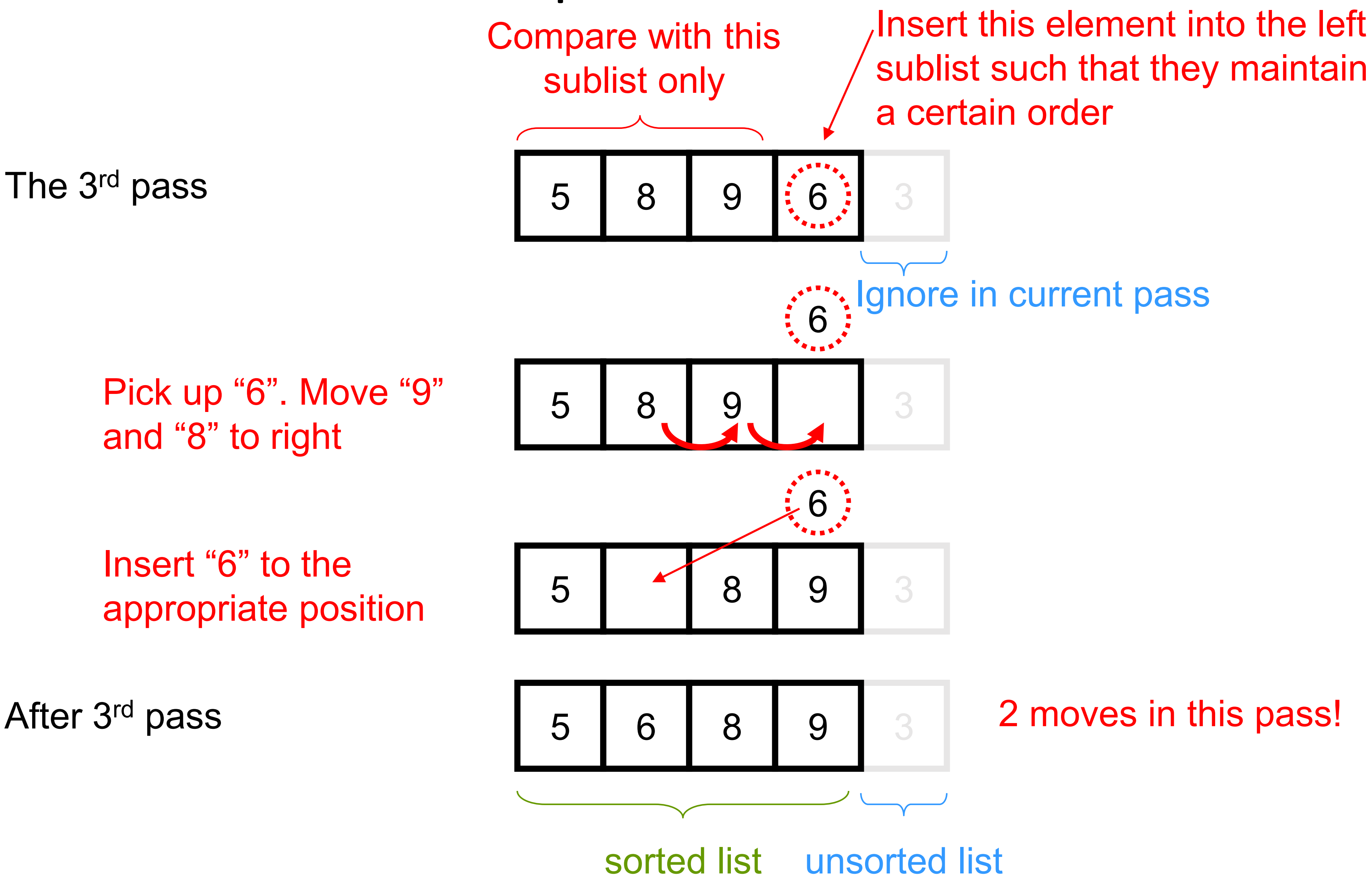
Insert “5” to the appropriate position



Insertion Sort Example



Insertion Sort Example



Insertion Sort Example

Compare with this sublist only

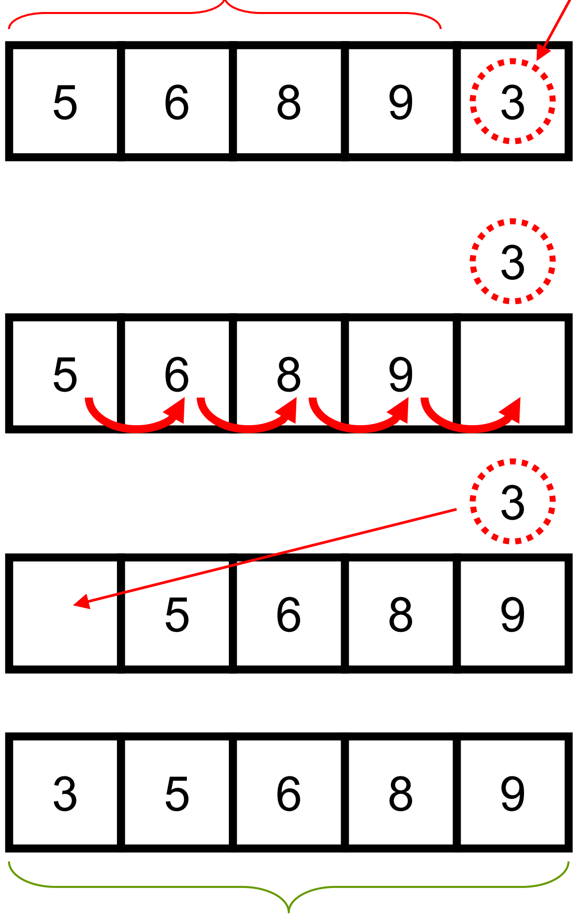
Insert this element into the left sublist such that they maintain a certain order

The 4th pass

Pick up “3”. Move “9”, “8”, “6” and “5” to right

Insert “3” to the appropriate position

After 4th pass



4 moves in this pass!

sorted list

Pseudo Code Review (before we introduce the pseudo code of Insertion sort)

- We need a language to express program development
  - English is too verbose and imprecise.
  - The target language, e.g. C/C++, requires too much details.
- Pseudo code resembles the target language in that
  - it is a sequence of steps (each step is precise and unambiguous)
  - it has similar control structure of C/C++
- Pseudo code is a kind of structured English for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax.

```
x = max{a, b, c}
```

Pseudo code

```
x = a;
if (b > x)  x = b;
if (c > x)  x = c;
```

C++ code

Insertion sort “detailed” pseudo code

- Basic operation: insert an element into a sorted list s.t. the final list is sorted.

```
Void INSERT_SORT(A)            // length [A] = n, A’s index starts with 0
for (int i = 1; i < n; i++) {
     int temp = data[i];
     // element (data[i]) to be inserted
     int j = i-1; //the last element in the sorted list
    while(j >= 0 && data[j] > temp){
          data[j+1] = data[j]; //movement operation
          j = j-- ;}
     data[j+1] = temp; }
```

https://yongdanielliang.github.io/animation/web/InsertionSortNew.html

Insertion sort animation game

How do you ”evaluate” an algorithm?

- E.g. there are other sorting algorithms too. Which of them is the “best”?
  - How to rank the algorithms?
  - Criteria
    - Correct?
    - **Fast?**
    - Low memory usage?

We will focus on running time analysis now

- Basic running time analysis examples

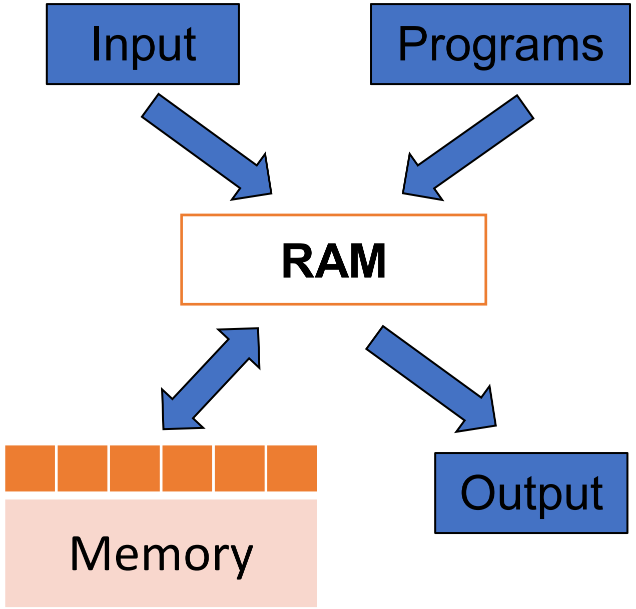
- Insertion sort’s running time

How to measure running time?

- Can we run two programs on computers and just report their actual running time?
  - Different hardware/OS can affect the running time
  - Different operators/people can run the program differently
- Thus, we must be able to measure the running time independent of the hardware, OS, and the users.
  - RAM model

RAM model

- Running time analysis using random-access machine (RAM) model
  - RAM: a generic one-processor instruction is executed one after another; no concurrent operations.



  - Each “simple” operation (+, *, -, /, ==, if, else, =(←)) takes exactly 1 step (most arithmetic operations)

Algorithms Analysis Example

- Find the sum of 1 + 2 + 3 + 4 + … + 998 + 999

- Method 1:
  - 1 + 2 = 3
  - 3 + 3 = 6
  - 6 + 4 = 10
  - …
  - 498,501 + 999 = 499,500

998 addition

~1,000x

- Method 2:
  - ((1 + 999) x 999) / 2
  - = 499,500

1 addition, 1 multiplication, 1 division

Algorithms Analysis Example

- Find the sum of 1 + 2 + 3 + 4 + … + 999,999

- Method 1:
  - 1 + 2 = 3
  - 3 + 3 = 6
  - 6 + 4 = 10
  - …
  - 498,998,500,001 + 999,999 = 499,999,500,000

999,998 addition!

~1,000,000x

- Method 2:
  - (1 + 999,999) x 999,999 / 2
  - = 499,999,500,000

Still 1 addition, 1 multiplication, 1 division! (independent of the input size)

Algorithms Analysis Example

Method 1:

```
int sumOfSeries(int n) {
    int i, sum = 0;
    for (i = 1; i < n; i++)
        sum += i;
    return sum;
}
```

n is the size of the elements. For example, if n=10, the sum is 1+2+3+…+9.

n - 1 addition

Which one is better?

Method 2:

```
int sumOfSeries(int n) {
    return (1 + n) * n / 2;
}
```

1 addition, 1 multiplication, 1 division

An Example Program

```
#include <iostream>

int main(int argc, char *argv[]) {

    int i, n, sum = 0;

    cin >> n;

    for(i = 0; i < n; i++)
        sum += i;

    return 0;
}
```

Constant time, C_1

Constant time, C_2

Variable time, depends on n
$= C_3 \times n$

Constant time, C_4

Total execution time
$= C_1 + C_2 + C_3 \times n + C_4$
$\approx C_3 \times n$ (if n is very large)

Analysis

- The exact value of C_i is not important, but the order of magnitude is important
- Usually C_i is a very small number
- $n * C_i$ would be a very significant number if n is a very large number
- e.g. suppose C_i is 1ms
  - If n is 1, execution time is 1ms
  - If n is 10, execution time is 10ms
  - If n is 1 million, execution time is 1,000s
- C_i is machine dependent
- To simplify our analysis, simply count how many times each instruction is executed in the algorithm.

Count the No. of Operations

```
int i, n, sum = 0;
```

This instruction being executed once

```
cin >> n;
```

This instruction being executed once

```
for(i = 0; i < n; i++)
    sum += i;
```

This block being executed *n* times

Total execution
$= 1 + 1 + n$
$= n + 2$
$\approx n$ (if n is very large)

Count the No. of Operations

Practice time: how many times is each code executed?

```
//Code A
sum += i;
```

This instruction being executed once

```
//Code B
for(i = 0; i < n; i++)
    sum += i;
```

This code being executed *n* times

```
//Code C
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
    sum += i * j;
```

This code being executed n^2 times!

Count the total number of operations of each statement

Practice time: how many times is each statement executed?

```
//Code A
sum += i;                      1 time in total
```

Total running time: 1

```
//Code B
for(i = 0; i < n; i++)         ~n times in total
      sum += i;                n times in total
```

Total running time: n+n

```
//Code C
for(i = 0; i < n; i++)         ~n times in total
    for(j = 0; j < n; j++)     ~n^2 times in total
      sum += i * j;            ~n^2 times in total
```

Total running time: $n+n^2+n^2$

Insertion sort analysis

- Basic operation: insert an element into a sorted list s.t. the final list is sorted.

t_i is the number of running time for the inner loop for a given *i* (*i* is the index in the outer loop)

INSERT_SORT(A) // length [A] = n	Cost	Times
`for (int i = 1; i < n; i++) {`	C_1	n
`    int temp = data[i];`	C_2	n
`        // element to be inserted`	0	
`    int j = i-1;`	C_4	n
`    while (j >= 0 && data[j] > temp){`	C_5	$\sum_{i=1 \; to \; n-1} t_i$
`        data[j+1] = data[j];`	C_6	$\sum_{i=1 \; to \; n-1} t_i$
`        j = j-- ;}`	C_7	$\sum_{i=1 \; to \; n-1} t_i$
`    data[j+1] = temp;`	C_8	n

Ci is the cost of the corresponding statement. Times: how many times each statement is executed

Insertion sort analysis

- Total running time (ignore all C_i)

  $T(n)$ =n + n + n + ($\sum_{i=1}^{n-1} t_i$) + ($\sum_{i=1}^{n-1} t_i$) + ($\sum_{i=1}^{n-1} t_i$) + n

- Minimum running time (best case: $t_i = 1$)

$T(n)$ =n + n + n + ($\sum_{i=1}^{n-1} 1$) + n
= n + n + n + $(n - 1)$ + n
= 5n - 1
= $a \cdot n + b$

Question: give me an example of the best-case input

Insertion sort analysis

- Maximum running time (worst case: $t_i =?$)

$$\sum_{i=1}^{n-1} t_i = \sum_{i=1}^{n-1} i = 1 + 2 + 3 + \cdots + n - 1 = \frac{n(n-1)}{2}$$

$T(n) =$n + n + n + $(\frac{n(n-1)}{2})$ + $(\frac{(n)(n-1)}{2})$ + $(\frac{(n)(n-1)}{2})$ + n

= $a'n^2 + b'n + c'$

Question: provide a worst-case input?

Insertion sort analysis

- Average case analysis ($t_i = \frac{i}{2}$)

$$\sum_{i=1}^{n-1} t_i = \sum_{i=1}^{n-1} \frac{i}{2}$$

Insertion sort analysis

- Running time analysis using random-access machine (RAM) model
  - RAM: a generic one-processor instruction is executed one after another; no concurrent operations
  - Each "simple" operation (+, *, -, /, ==, if, else, =(←)) takes exactly 1 step
  - Loops and subroutines are not simple operations but depends on the size of input data & the contents of a subroutine.

    "sort", "matrix multiplication", "length of an array"
  - Each memory access takes 1 step
  - Now, RAM model: $C_i = 1$

Complexity Analysis

- Space complexity: O(1)
- the temp. variable is used to **hold** the element that going to be inserted into the sublist

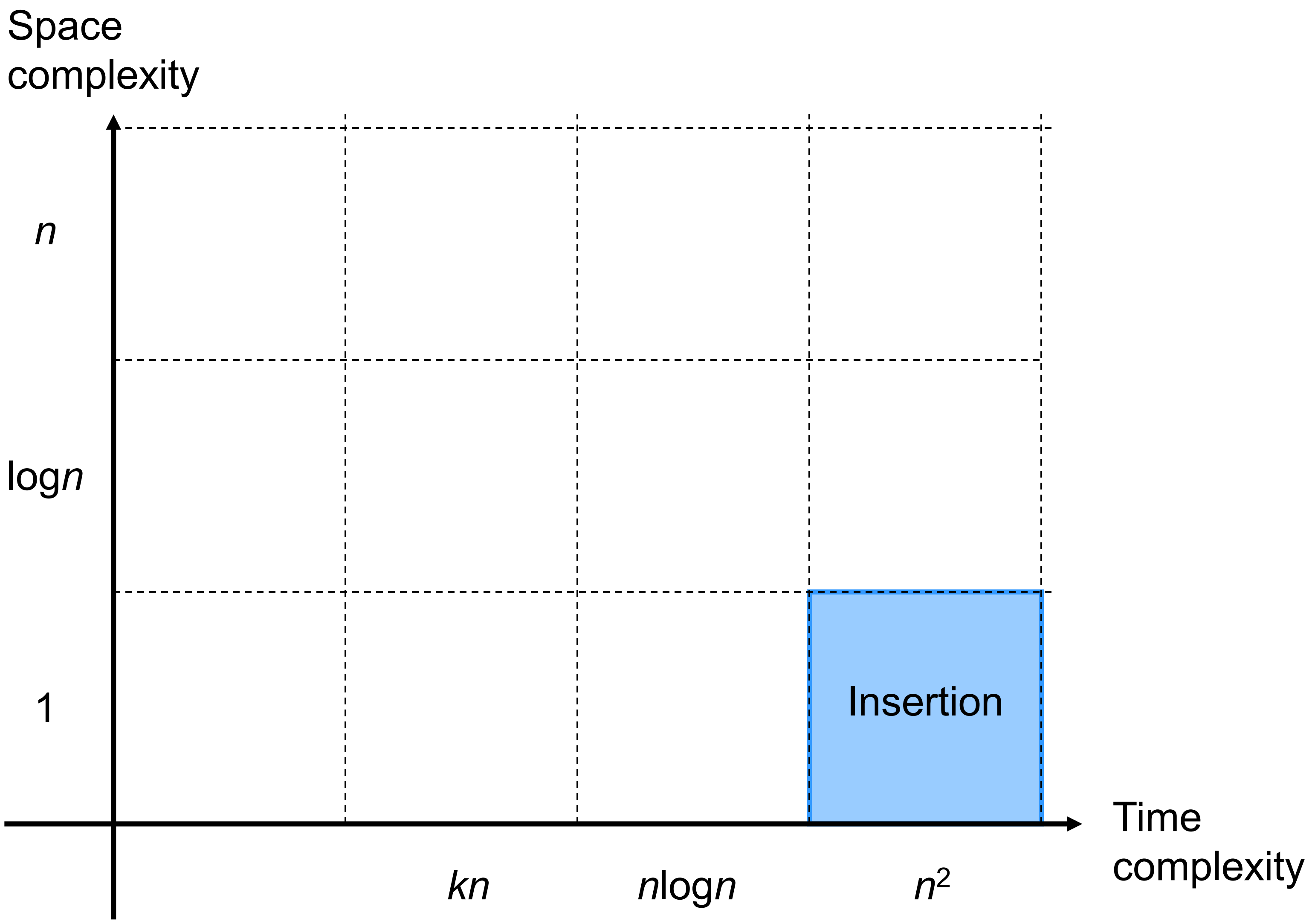
Big O notation will be introduced shortly. At this stage, think of big O as the fastest growing item in the running time equation (the dominate term/bottle neck)

Complexity Analysis

- The best case: O(n)
  - The list is already sorted; scan it once!
- The worst case: O(n^2)
  - *n-1* items to be inserted
  - At most *i* comparisons at *i-th* insertion
  - The total no. of comparisons = $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
- The average case: O(n^2)
  - Half the number of comparisons

- Because of the simplicity of insertion sort, it is the fastest sorting method when the number of elements *N* is small, e.g. $N < 10$.

Summary


Space
complexity
n
log$n$
1
Insertion
Time
complexity
kn
$n$log$n$
n^2

Analysis of Algorithms

- Often several different algorithms are available to solve the same problem. These algorithms may not run with same efficiency.
  - May be impractical for large input size
  - May run extremely slow for particular inputs
- We want to know the efficiency and complexity of algorithms so as to compare them and make a wise choice.
- The complexity growth rate is far more important than the actual execution time during analysis.

Running time analysis

- Worst-case and average-case analysis
  - 1. The longest running time for any input of size n: the worst case

    E.g., 5 3 2 1 0 for insertion sort
  - 2. The upper bound on the running time for any input
  - 3. The worst case happens often

    E.g., database search: fail to find a match
  - 4. The average case is often roughly as bad as the worst case

    E.g., insertion sort, roughly $\begin{cases} half\ elements \leq key \\ half\ elements > key \end{cases}$, $t_j = \frac{j}{2}$

Running time analysis

- Simplifications/ approximations

n	$\frac{3}{2}n^2$	$\frac{3}{2}n^2 + \frac{7}{2}n - 4$	% difference
10	150	181	17% ($\frac{181-150}{180}$)
50	3,750	3,921	4.4%
100	15,000	15,436	2.3%
500	375,000	376,746	0.5%

Highest-order term finally dominates the output

Asymptotic Complexity

- Asymptotic complexity is a way of expressing the main component of the cost of an algorithm.
- For example, when analyzing some algorithm, one might find that the time (or the number of steps) it takes to complete a problem of size n is given by

  $$T(n) = 4n^2 - 2n + 2$$

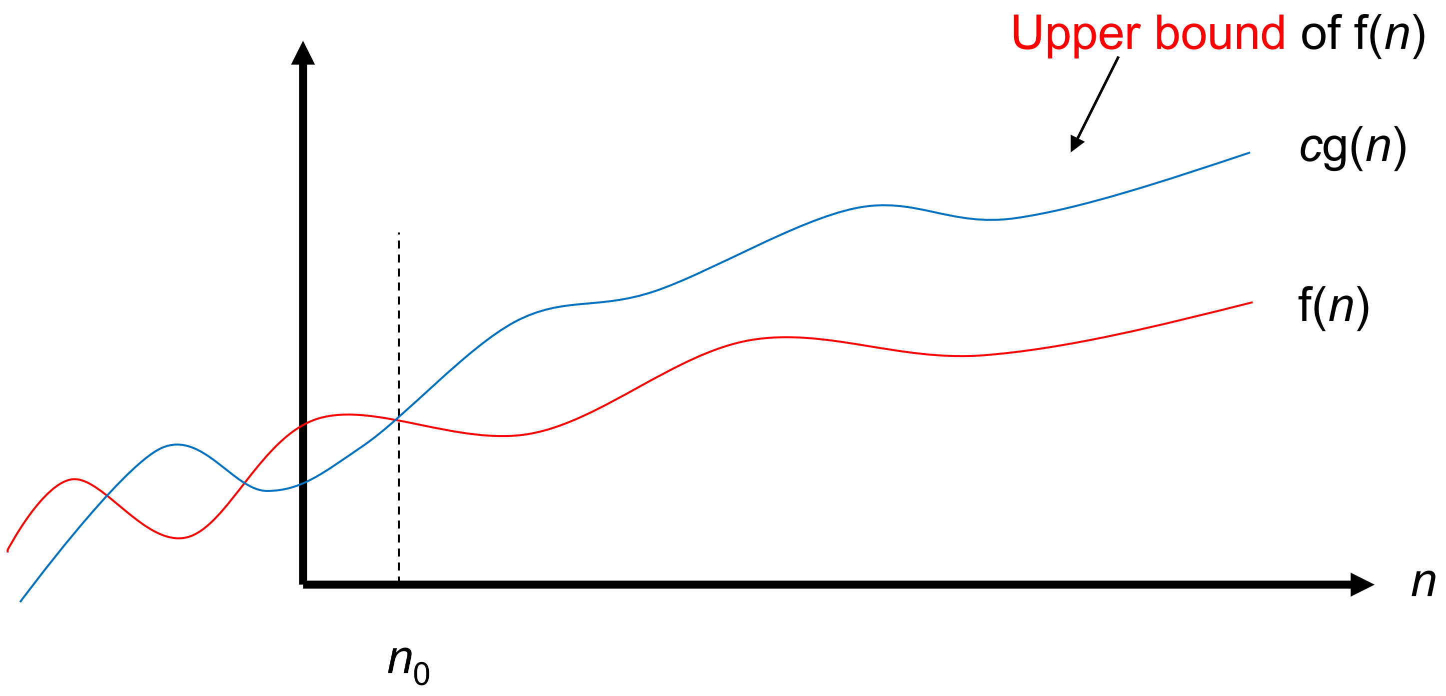
- If we ignore constants (which makes sense because those depend on the particular hardware the program is run on) and slower growing terms (i.e. 2n), we could say T(n) grows at the order of n^2 and write:

  $$T(n) = O(n^2)$$

- The letter O is used because the rate of growth of a function is also called its *Order*. Basically, it tells you how fast a function grows or declines.

Asymptotic Notation O

- Big-O notation defines an upper bound of an algorithm's running time.
- We say that a function f(n) is of the order of g(n), iff there exists constant $c > 0$ and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$ (this definition is not required)
- In other words, f(n) is at most a constant times of g(n) for sufficiently large of values of n
- Using Big-O notation: $f(n) = O(g(n))$



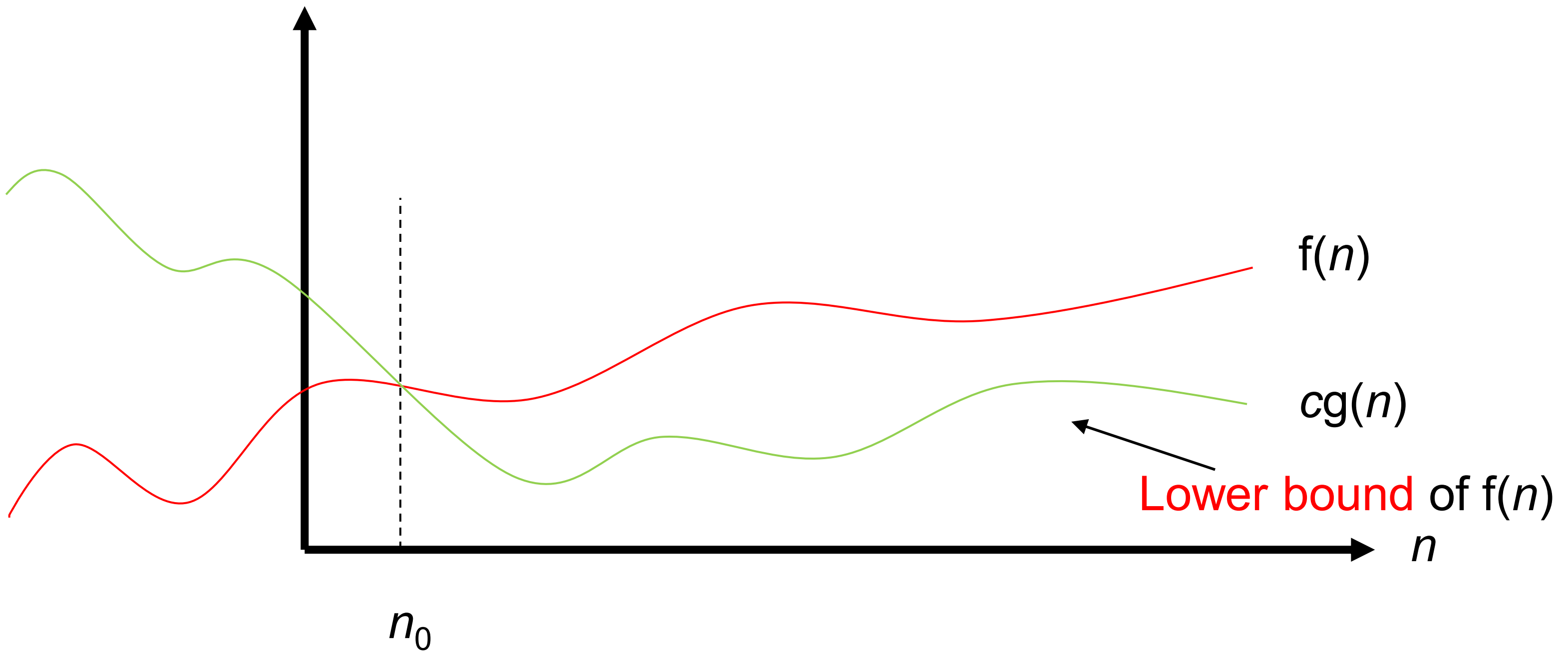
e.g. The time complexity of insertion sort is $O(n^2)$.

If $f(n)=5n^2+3$, we can say $f(n)=O(n^2)$

If $f(n)=100n+5$,
then $f(n)=O(n)$, $f(n)=O(100n)$, $f(n)=O(n^2)$, etc.

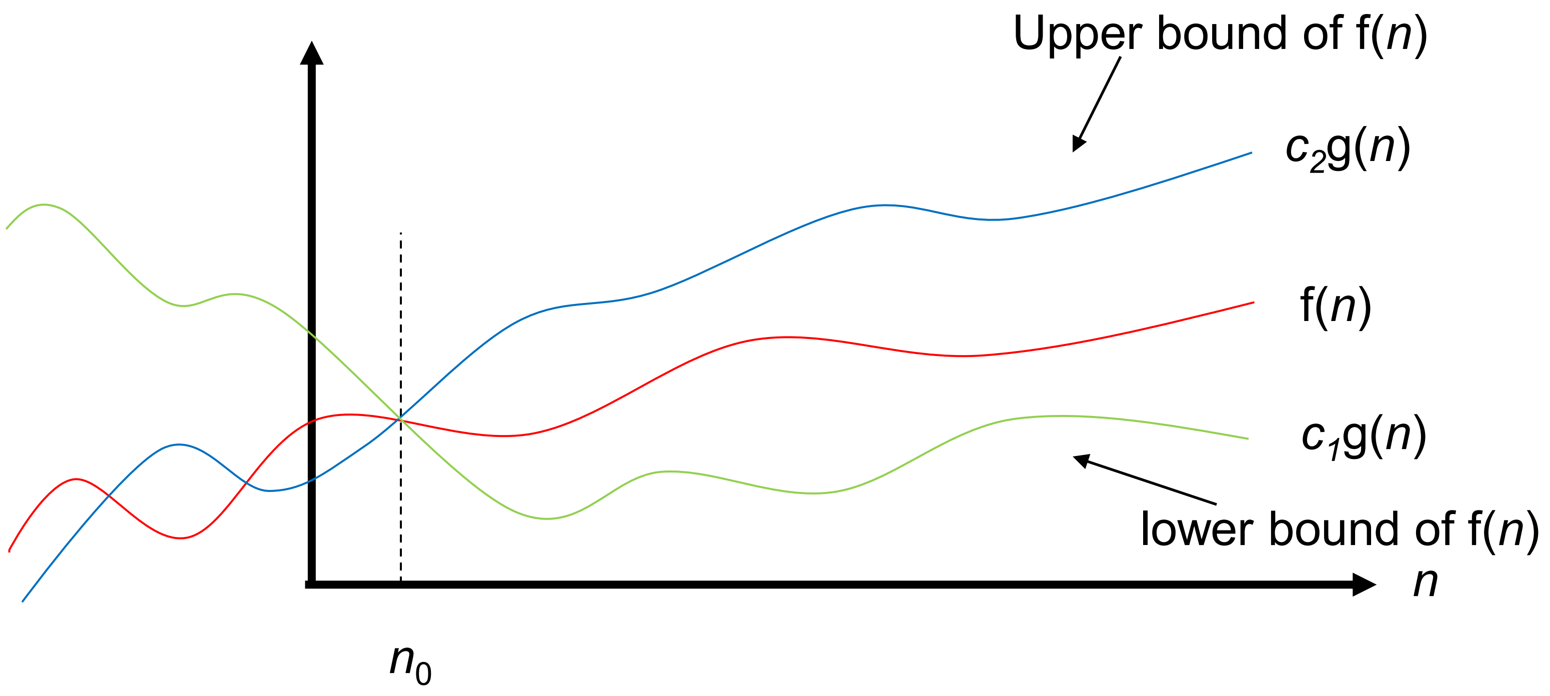
Asymptotic Notation Ω

- Big-Omega notation defines a lower bound of an algorithm's running time.
- f(n) = Ω(g(n)) iff there exists constant $c > 0$ and n_0 such that f(n) ≥ cg(n) for all $n \geq n_0$ (I won't test you how to prove this)



Asymptotic Notation Θ

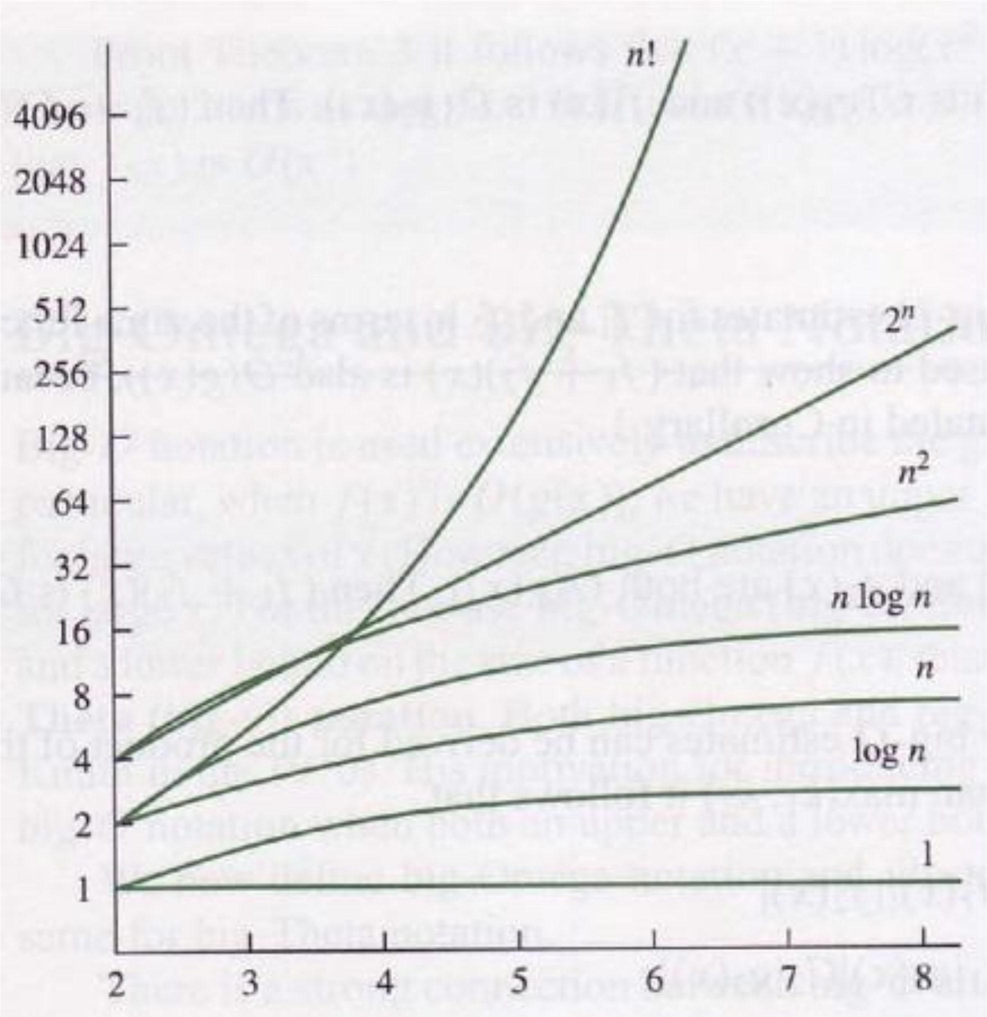
- Big-Theta notation defines an exact bound of an algorithm's running time.
- f(n) = Θ(g(n)) iff there exists constant $c_1 > 0$, $c_2 > 0$ and n_0 such that c_1g(n) ≤ f(n) ≤ c_2g(n) for all n, $n \geq n_0$ (the proof is not required)



Important Complexity Classes

- $O(1)$: Constant time
- $O(\log_2 n)$: Logarithmic time
- $O(n)$: Linear time
- $O(n\log_2 n)$: Log-linear time
- $O(n^2)$: Quadratic time
- $O(n^3)$: Cubic time
- $O(n^k)$: Polynomial time
- $O(2^n)$: Exponential time

Important Complexity Classes



Factorial time

Exponential time

Quadratic time

Log-linear time

Linear time

Logarithmic time

Constant time

Increasing complexity

Becomes not feasible when *n* grows larger

In-class exercise: write the time complexity of method 1 and method 2 using big O notation

Method 1:

```
int sumOfSeries(int m) {
    int i, sum = 0;
    for (i = 1; i < m; i++)
        sum += i²;
    return sum;
}
```

Method 2:

```
int sum(int n) {
    return (1 + n) * n / 2;
}
```