Lecture 1 - Introduction

AIAA 5037 Advanced Algorithms and Data Structures

Ying Sun, AI Thrust

Outline

- Course Information
- Algorithm
- Data Structure
- Insertion Sort
- Complexity Analysis

Course Information

Course Overview

1. Introduction
2. Elementary Data Structure
3. Basic Algorithm Design and NP-Completeness
4. Divide and Conquer
5. Trees
6. Dynamic Programming
7. Greedy
8. Graph Algorithms
9. …

Course Goals

- Master the methodology of algorithm design and analysis
- Master typical fundamental and advanced algorithms and data structures
- Demonstrate ability to design algorithms to solve various practical problems

Teaching Styles

- Convey basic ideas and give detailed analysis
- Provide example algorithms and show implementation details
- Practice on programming problems

Prerequisites

- Proficient in at least one programming language
- Basic mathematical knowledge

Common Questions

1. Suitable for beginners?
    - Designed for students who have not taken algorithm courses. We teach basic algorithm design ideas from scratch.
2. Difference from under-graduate course / How is it advanced?
    - More contents and introduce each part faster
    - More focus on the theoretical analysis of algorithms. For example, in greedy and dynamic programming algorithms, we'll spend more time proving their correctness.
    - Each section will cover more advanced algorithms compared to undergraduate studies. For instance, in the dynamic programming segment, our example algorithms include dynamic programming on trees.

Grading

Final Exam	50%
Assignments	30%
Course Participation	20%

Assignments:

1. Programing on OJ
2. Writing assignments on Canvas

Examination format:

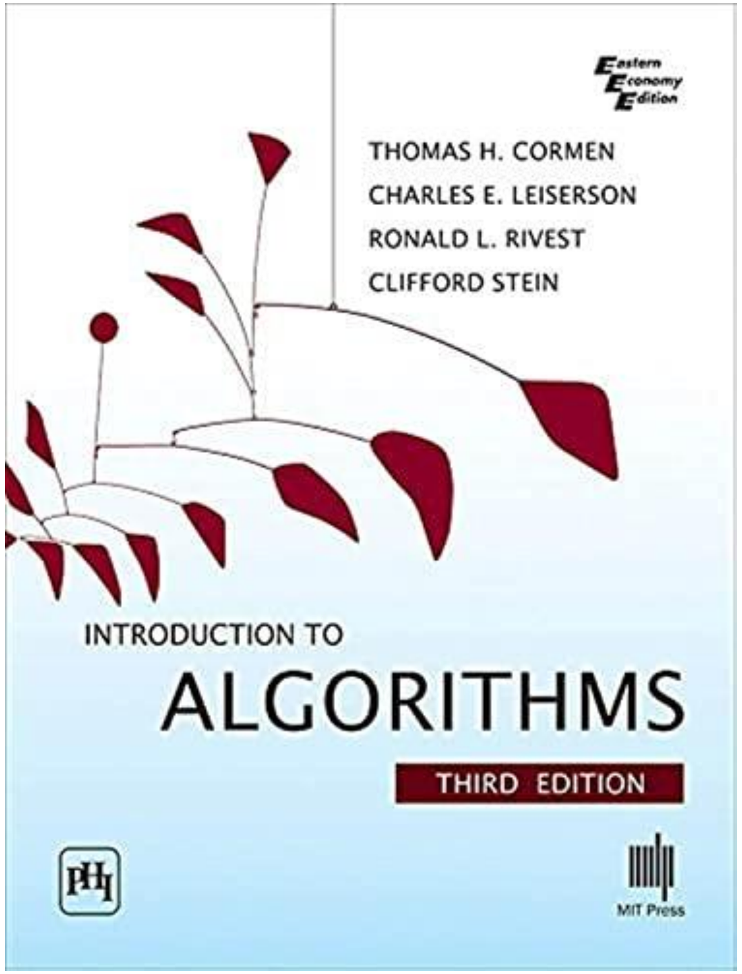
1. Written test on algorithm design and analysis

Canvas

- https://canvas.hkust-gz.edu.cn/
- Organize course materials, assignments, and notifications
- Do not send me message through Canvas

Reference Materials

Introduction to Algorithms, Third Edition

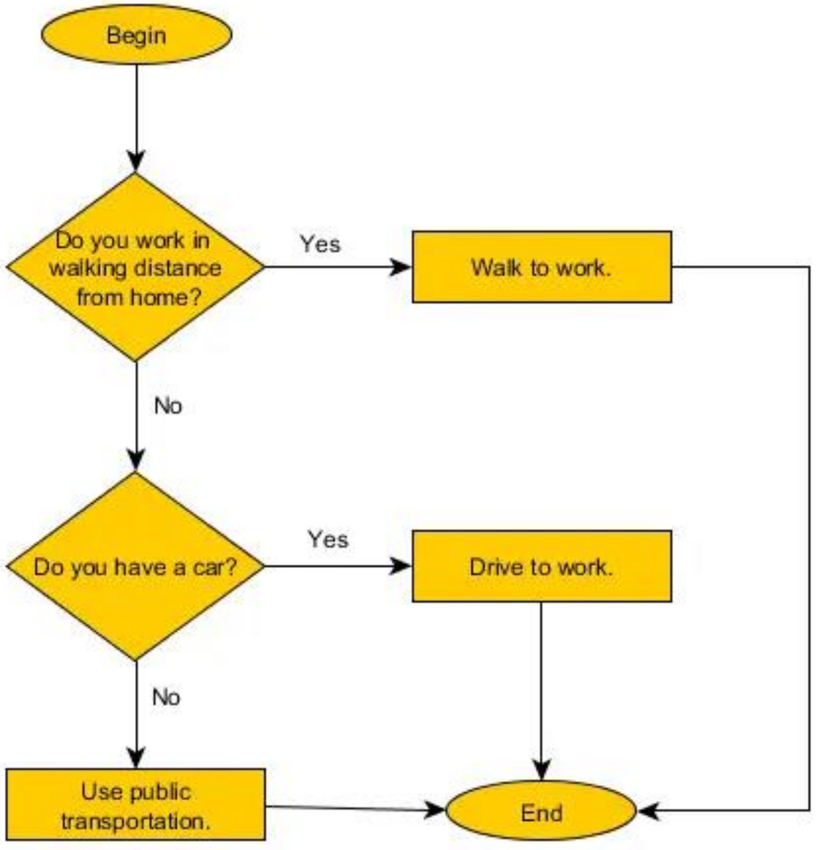


Algorithm

What is Algorithm

An ***algorithm*** is a finite sequence of instructions for performing a task

Put an elephant into a refrigerator



Decide how to go to work

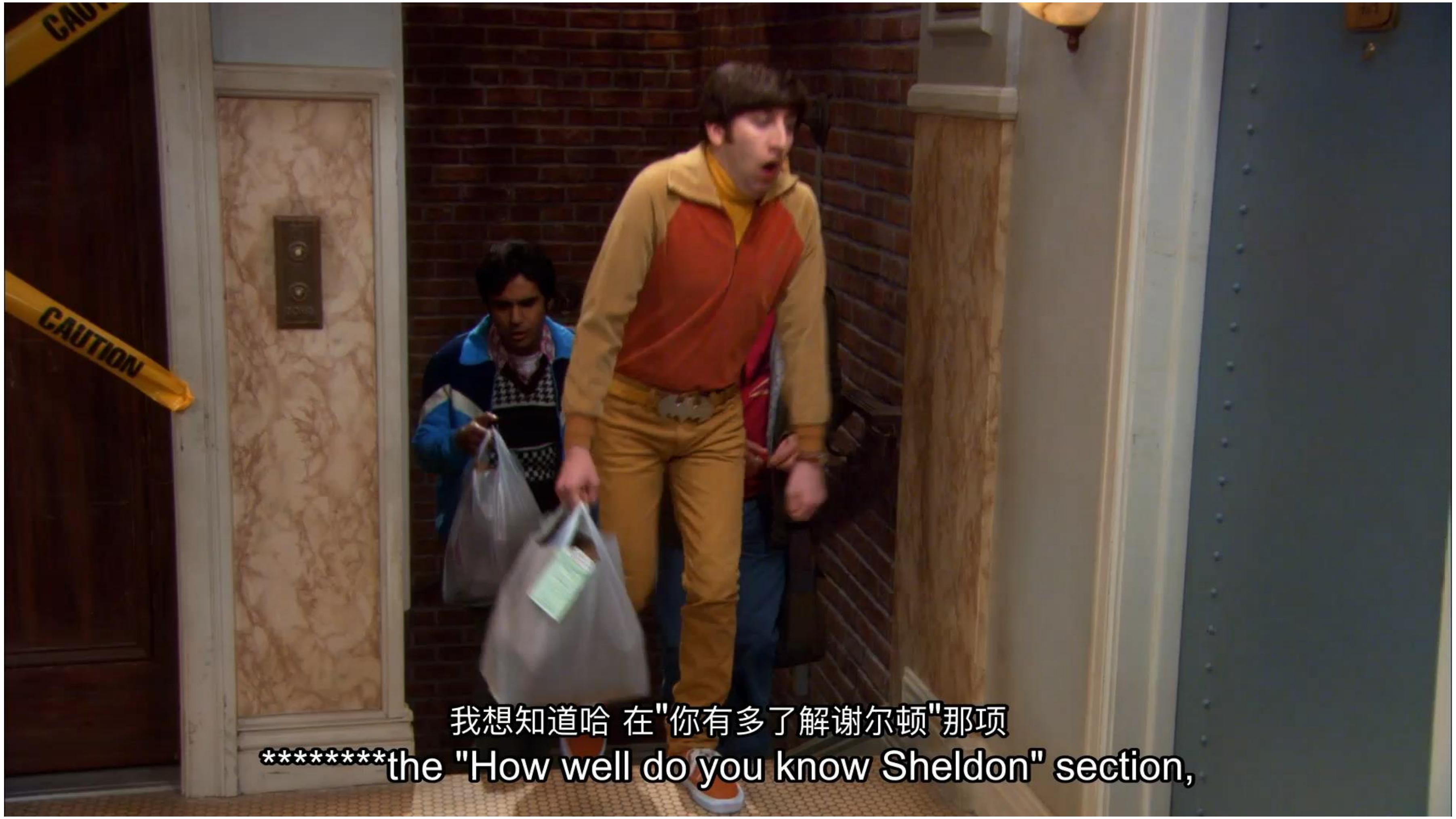
What is Algorithm

Recipes are life examples for algorithms

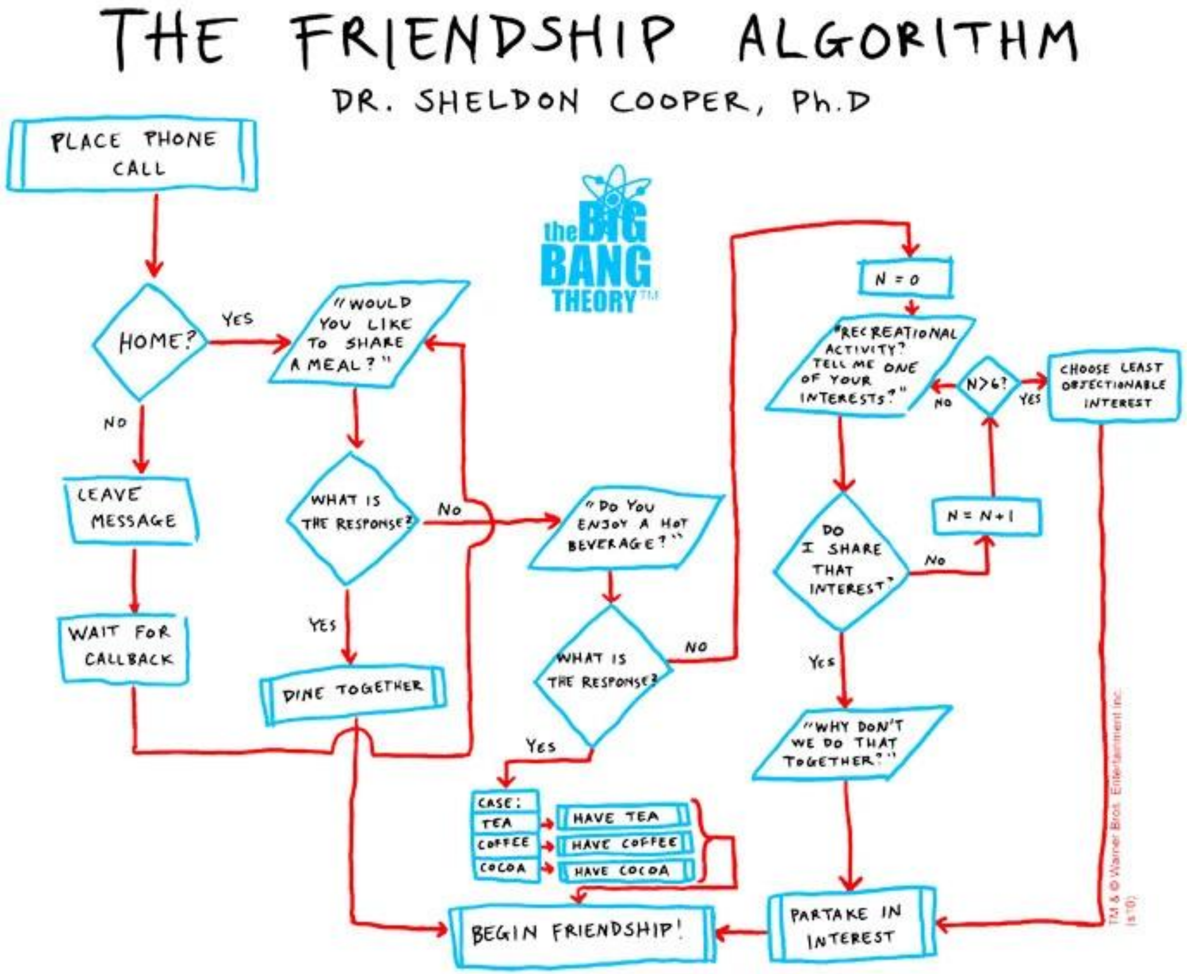
Input: *1 cup of milk, 2 tablespoons of sugar, 1 cup of flour, 3 large eggs, 1 pinch of salt, and oil for the pan*
Output: *a stack of pancakes on a plate*

1. *Mix flour, eggs, sugar, and salt with an egg beater until the mixture is homogeneous.*
2. *Slowly add the milk while stirring the mixture.*
3. *Heat a pan with oil.*
4. ***Repeat** the following steps:*
    1. *Take a ladle of batter and pour into the pan.*
    2. *Fry pancake on one side.*
    3. *Flip pancake.*
    4. *Fry pancake on other side.*
    5. *Take pancake out of the pan.*
    6. *Put pancake onto plate.*
5. ***Until** bowl is empty.*
6. ***Return** plate with pancakes.*

What is Algorithm

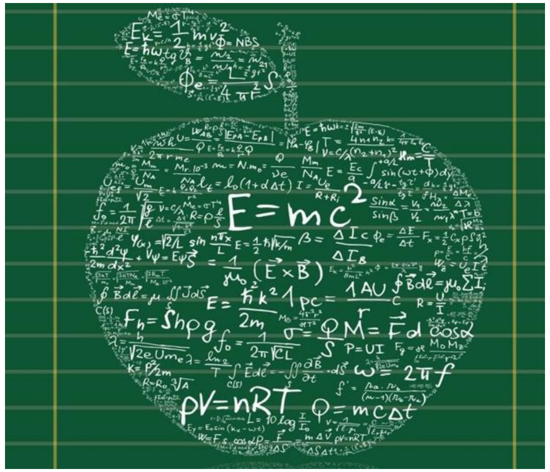


What is Algorithm



Algorithm is Everywhere

Computer Science



Mathematics

Artificial Intelligence

Data Science

Why learning algorithms

- The soul of programming, improve your program in various ways
  - ✓Higher Efficiency
  - ✓Lower Resource Usage
  - ✓Higher Scalability

Why learning algorithms

- The wisdom of life: algorithms are not restricted to coding

  How do you look up a word in a dictionary?

  - Check one by one from the first page —— Brute Force
  - Summarize words in a hierarchical table ——Data Structure + Indexing
  - Randomly open a page and figure out if you should look up the word before it or after it —— Binary search

AI Is No Special

Consider a generic problem formulation

$$\arg_{x} \max f(x)$$

$$\text{s.t. } \{\mathrm{c}(x) = True | c \in C\}$$

Various properties: Continuous/Discrete space, domain-specific constraints, (non-)parametric, (non-)differentiable, (non-)convex, …

- Select 5 students in a group that performs the best as a basketball team
- Buy the most candies with limited money
- Find the largest common divisor of any two number
- Find the shortest path on a given graph
- Model training: find parameters that minimize a model's empirical loss on a given dataset

AI Is No Special

$$\arg_{\boldsymbol{x}} \max f(x)$$

$$\text{s.t. } \{\mathrm{c}(x) = True | c \in C\}$$

Many interesting ways to solve different problems, not just gradient descent

- Brute Force, Divide and Conquer, Greedy, Dynamic Programming, Numerical Optimization Algorithms, Randomized Algorithms, Evolutionary Algorithms, Heuristic Search Algorithms, …

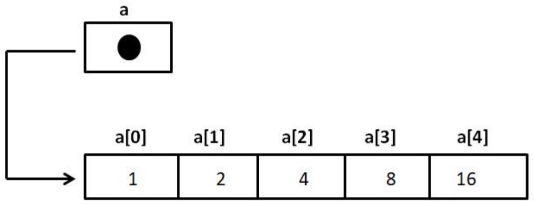
Data Structure

What is Data Structure

- Data structure: a systematic way to organize data for using data meaningfully and efficiently
- Example: how to store a list of integers (e.g., 100)?
  - Create 100 variables? —— Redundant Storage, No representation on the relationship among the integers
  - Use array ——Less redundant metadata storage, easy retrieval

Separate containers
No order, not related

In one container, ordered, related

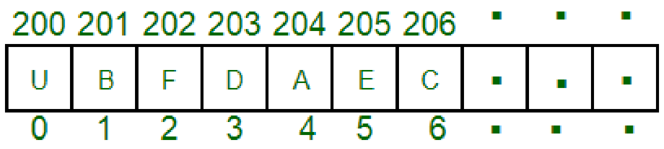


Memory Representation of Array

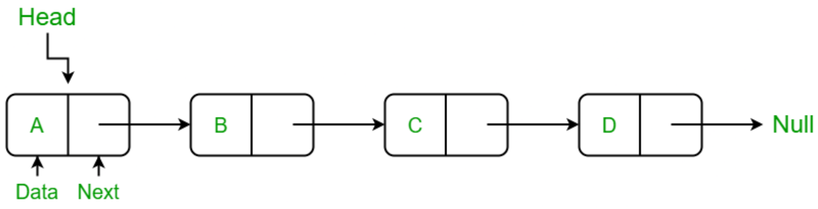
What is Data Structure

Various kinds of data structures

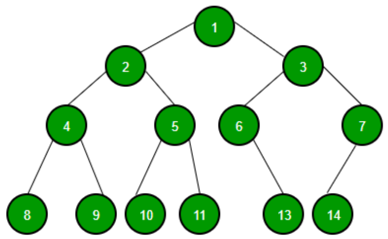
- Linear Lists & Trees & Graphs
- Basic & Hybrid
- Similar architecture but with different logic & operations



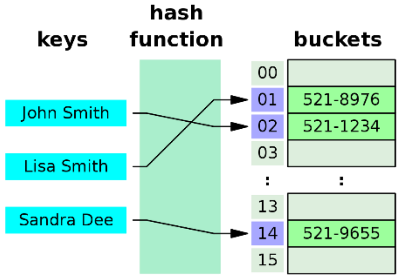
Array



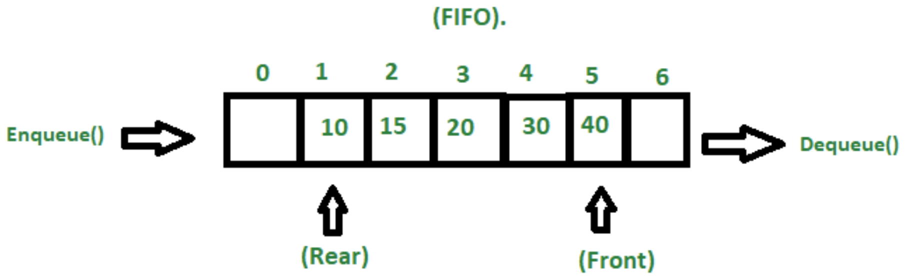
Linked list



Trees



Hash Tables

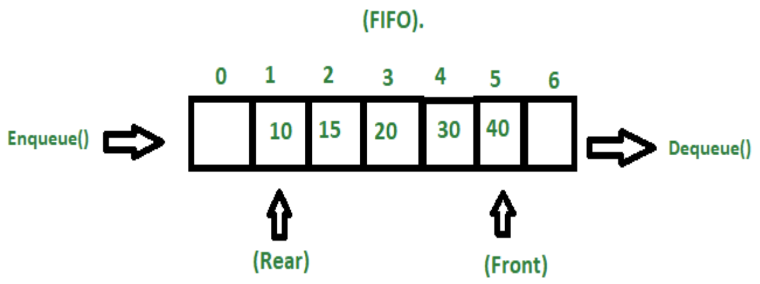


Queue & Stack

Abstract Data Types (ADT) and Data Structure

Closely related concepts referring to different aspects

- ADT: logical form, user's view, abstract, interface of data structure
- Data Structure: physical form, implementer's view, concrete
- Example: queue, commonly used in job scheduling
    - A data structure, also an ADT
    - Support insertion and popping elements following the FIFO (First-In, First-Out) policy
    - Can be implemented with array or linked list



Data Structure and Algorithm

- Efficient data structures are key to designing efficient algorithms
  - Examples:
    - ➢ Sequence modification and query
    - ➢ Sparse Matrix Multiplication
- Algorithms support efficient implementation of data structures
  - Example: adding new elements to a heap
- Our focus: (1) principles and implementations of data structures instead of just the APIs; (2) efficient algorithm design with data structures

Insertion Sort

Sorting Problem

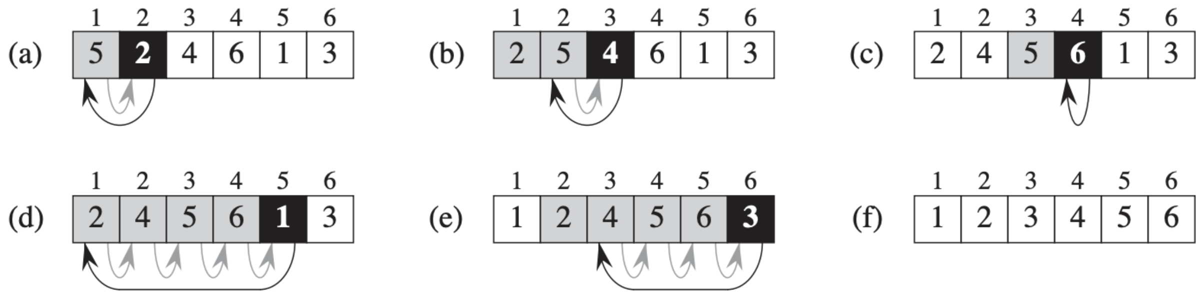
Sorting: The process of rearranging a sequence of elements according to a certain order or criterion. For numbers, this is typically in non-decreasing (or non-increasing) order

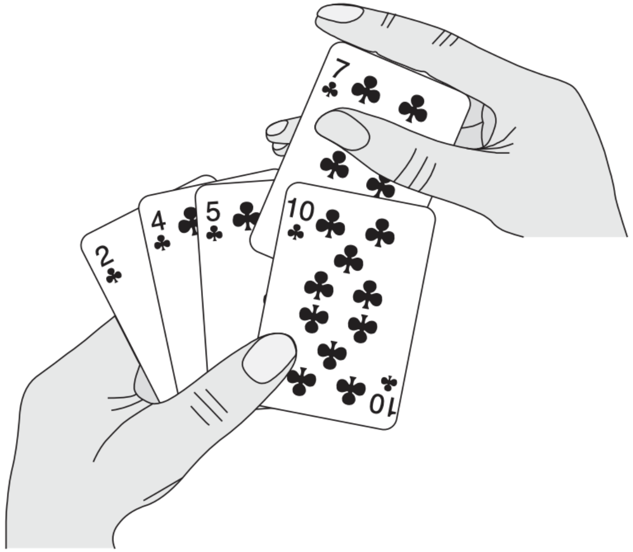
Input: A sequence of n numbers $< a_1, a_2, a_3, \dots, a_n >$

Output: A reordered sequence $< a_{k_1}, a_{k_2}, a_{k_3}, \dots, a_{k_n} >$ of the input sequence such that $a_{k_1} \leq a_{k_2} \leq a_{k_3} \dots \leq a_{k_n}$

Insertion sort

- Sort the array from left to right, expand new element to the sorted array
- New element placed at the correct position in the sorted part





Implementation

Pseudocode: a high-level, informal representation of an algorithm

- Designed for humans, not machines
- Not bound by strict syntax of any specific programming language
- Can be concise or detailed; natural language permissible for succinctly explaining complex algorithms

```
INSERTION-SORT(A)
for j = 1 to A.length - 1
    currentItem = A[j]
    i = j – 1
    while i >= 0 and A[i] > currentItem
        A[i+1] = A[i]
        i = i - 1
    A[i + 1] = currentItem
```

Move elements larger than current item to its right

```
quicksort (array){
   if (array.length > 1){
      choose a pivot;
      while (there are items left in array){
         if (item < pivot)
            put item into subarray1;
         else
            put item into subarray2;
      }
      quicksort(subarray1);
      quicksort(subarray2);
   }
}
```

Time Complexity

- Measurement of the amount of computation to run an algorithm
- Counting the number of elementary operations (independent of problem size)
  - e.g., assignment, comparison
  - Not elementary: e.g., (1) Copy an input array (list.copy), (2) scan an array (list.find)
- Elementary operation is regarded taking constant time

```
for i in range(n):
    print('2023')
```

print n lines

T(n) = n

```
for i in range(n):
    for j in range(n):
        print('2023')
```

print n^2 lines

T(n) = n^2

Time Complexity of Insertion Sort

Supposing the length is n

```
INSERTION-SORT(A)
```

`for j = 1 to A.length - 1`	$n-1$ iterations
`    currentItem = A[j]`	Constant
`    i = j - 1`	Constant
`    while i >= 0 and A[i] > currentItem`	Right shift the values before j that are larger than it: m_j iterations
`        A[i+1] = A[i]`	Constant
`        i = i - 1`	Constant
`    A[i + 1] = currentItem`	Constant

T(n) = $(m_1 + m_2 + \cdots + m_{n-1}) \times 4 + 3 \times (n-1)$

Time Complexity of Insertion Sort

- Supposing the length is n, $T(n) = (m_1 + m_2 + \cdots + m_{n-1}) \times 4 + 3 \times (n-1)$
- m_j depends on the input: the number of A_i (i < j) larger than A_j
- $(m_1 + m_2 + \cdots + m_{n-1}) = \sum_{j=1}^{n-1} \sum_{i=0}^{j-1} \{A_i > A_j\}$ - the number of inversions
- $T(n) = 4\ \#inversions\ +\ 3(n-1)$

- Worst case: in reversed order

10	9	8	7	6	5	4	3	2	1

$m_i = \text{i}$

$T(n) = (1 + 2 + \cdots + n - 1) \times 4 + 3(n-1) = 2n^2 + \text{n} - 3$

- Best case: already sorted

1	2	3	4	5	6	7	8	9	10

$m_i = 0$

$T(n) = 3(n-1)$

- Average case (average on all inputs of size n)?

Time Complexity of Insertion Sort

Average case (on all inputs of size n):

$$T(n) = E[4\ \#inversions\ +\ 3(n-1)]$$

$$= 4E[\ \#inversions] +\ 3(n-1)$$

$$= 4\sum_{j=1}^{n-1}\sum_{i=0}^{j-1} E[\{A_i > A_j\}] +\ 3(n-1)$$

$$= 4\sum_{j=1}^{n-1}\sum_{i=0}^{j-1} p(A_i > A_j) +\ 3(n-1)$$

Assuming a uniform distribution for order, for any pair (i, j), there's an equal probability that $a_i > a_j$ and $a_i < a_j$

Therefore, $T(n) = 4\sum_{j=1}^{n-1}\sum_{i=0}^{j-1} 0.5 + 3(n-1) = n^2 + 2n - 3$

Time Complexity of Insertion Sort

Worst case:

10	9	8	7	6	5	4	3	2	1

$T(n) = 2n^2 + n - 3$

Best case:

1	2	3	4	5	6	7	8	9	10

$T(n) = 3n - 3$

Average case: $T(n) = n^2 + 2n - 3$

Where is the familiar big O notation?

Space Complexity

- Measurement of the amount of memory an algorithm needs
- Estimated by counting the number of allocated space for storing values
- Size of specific datatype is regarded as constant
- Same notation system as time complexity

```
a = []
for i in range(n):
    a.append(i)
```

$T(n) = n$

```
for i in range(n):
    print(i)
```

$T(n) = constant$

Space Complexity of Insertion Sort

- **Auxiliary Space Complexity**: *constant, only for intermediate variables i, j, currentItem*
- **Total Space Complexity**: *n,* required by the input array A

```
INSERTION-SORT(A)
for j = 1 to A.length - 1
    currentItem = A[j]
    i = j – 1
    while i >= 0 and A[i] > currentItem
        A[i+1] = A[i]
        i = i - 1
    A[i + 1] = currentItem
```

Rate of Growth

Rate of Growth / Order of Growth

How to compare complexity of algorithms?

- e.g., $T_1(n) = 10log^3n + n^2$ and $T_2(n) = 3n^2logn + n$

Straightforward comparison for the changing problem size

- Whose running time grows slower with problem size
- How: compare running time as n tends towards infinity

$$\lim_{n\to\infty} \frac{T_1(n)}{T_2(n)} \in \{\infty, \text{constant}, 0\}$$

- ∞: $T_1(n)$ grows faster than $T_2(n)$
- constant: $T_1(n)$ and $T_2(n)$ grow at a similar rate
- 0: $T_1(n)$ grows slower than $T_2(n)$

Rate of Growth / Order of Growth

$$\lim_{n\to\infty}\frac{T_1(n)}{T_2(n)} \in \{\infty, \text{constant}, 0\}$$

∞: $T_1(n)$ grows faster than $T_2(n)$, constant: $T_1(n)$ and $T_2(n)$ grow at a similar rate, 0: $T_1(n)$ grows slower than $T_2(n)$

- Only leading term is significant
- Coefficient of terms do not influence the result

Θ notation for concise complexity representation: only leading term, without coefficients

- Example (worst-case time complexity of insertion sort): $T(n) = 2n^2 + n - 3 = \Theta(n^2)$
- Easy comparison: $\Theta(1) < \Theta(\log n) < \Theta(n) < \Theta(n^2) < \Theta(2^n)$ ...
- What is $O(n)$?

Rate of Growth / Order of Growth

Back to the problem: $T_1(n) = log^3 n + n^2$ and $T_2(n) = n^2 logn + n$

- $T_1(n) = n^2 + log^3 n = \Theta(n^2)$
- $T_2(n) = n^2 logn + n = \Theta(n^2 logn)$
- Algorithm 1 is more efficient

What is $O(n)$?

Asymptotic Bounds ($\Theta, O, \Omega, o, \omega$)

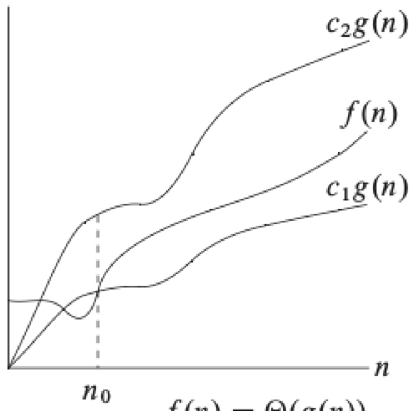
Big Theta Notation (Θ) - asymptotically tight bound

When we say $f(n) = \Theta(g(n))$, it means that $g(n)$ serves as both an upper and a lower bound for $f(n)$, up to constant factors.

- For a $f(n)$, if there exist positive constants c_1, c_2, and n_0 such that:

$$\forall n \geq n_0, \qquad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

- Then $f(n) = \Theta(g(n))$



$f(n) = \Theta(g(n))$

Asymptotic Bounds (Θ, O, Ω, o, ω)

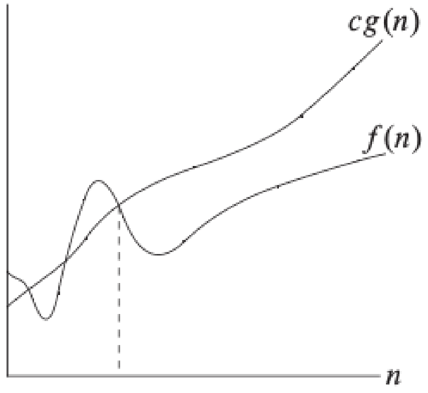
Big O Notation (O) - asymptotically upper bound

When we say $f(n) = O(g(n))$, it means that $g(n)$ serves as an upper bound for $f(n)$, up to constant factors

- For a $f(n)$, if there exist positive constants c, and n_0 such that:

$$\forall n \geq n_0, \qquad 0 \leq f(n) \leq cg(n)$$

- Then $f(n) = O(g(n))$



$f(n) = O(g(n))$

Common, because we usually compare upper bound

Asymptotic Bounds ($\Theta, O, \Omega, o, \omega$)

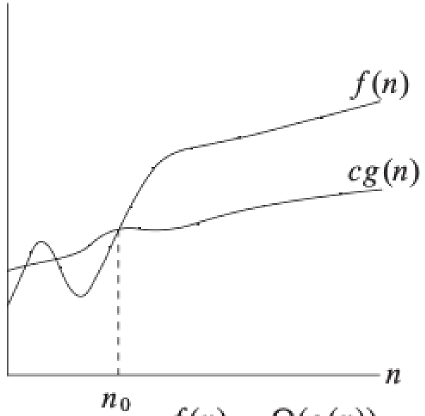
Big Omega Notation (Ω) - asymptotically lower bound

When we say $f(n) = \Omega(g(n))$, it means that $g(n)$ serves as a lower bound for $f(n)$, up to constant factors

- For a $f(n)$, if there exist positive constants c, and n_0 such that:

$$\forall n \geq n_0, \qquad 0 \leq cg(n) \leq f(n)$$

- Then $f(n) = \Omega(g(n))$



$f(n) = \Omega(g(n))$

Asymptotic Bounds (Θ, O, Ω, o, ω)

Small O Notation (o) - stronger assertions than the "Big O"

- When we say $f(n) = o(g(n))$, it means that *f*(*n*) grows strictly slower than *g*(*n*), i.e.,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

- $f(n) = o(g(n))$ implies $f(n) = O(g(n))$, but the converse is not necessarily true.

Small Omega Notation (ω) - stronger assertions than the "Big Omega"

- When we say $f(n) = \omega(g(n))$, it means that *f*(*n*) grows strictly faster than *g*(*n*), i.e.,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

- $f(n) = \omega(g(n))$ implies $f(n) = \Omega(g(n))$, but the converse is not necessarily true.

Asymptotic Bounds ($\Theta, O, \Omega, o, \omega$)

Notation	Rough Meaning
Θ	asymptotically tight bound (same order)
O	asymptotically upper bound
Ω	asymptotically lower bound
o	Asymptotically upper bound but not tight
ω	Asymptotically lower bound but not tight

Summary

- Course Information
- Algorithm
- Data Structure
- Insertion Sort
- Complexity Analysis

群聊：AIAA5037-24Fall

Assignment

http://10.108.6.129/

- Will be posted soon
- Please register an account using your university email address and your real name (e.g., "San Zhang (张三)") as the nickname.

Thank you!

AIAA 5037 Advanced Algorithms and Data Structures