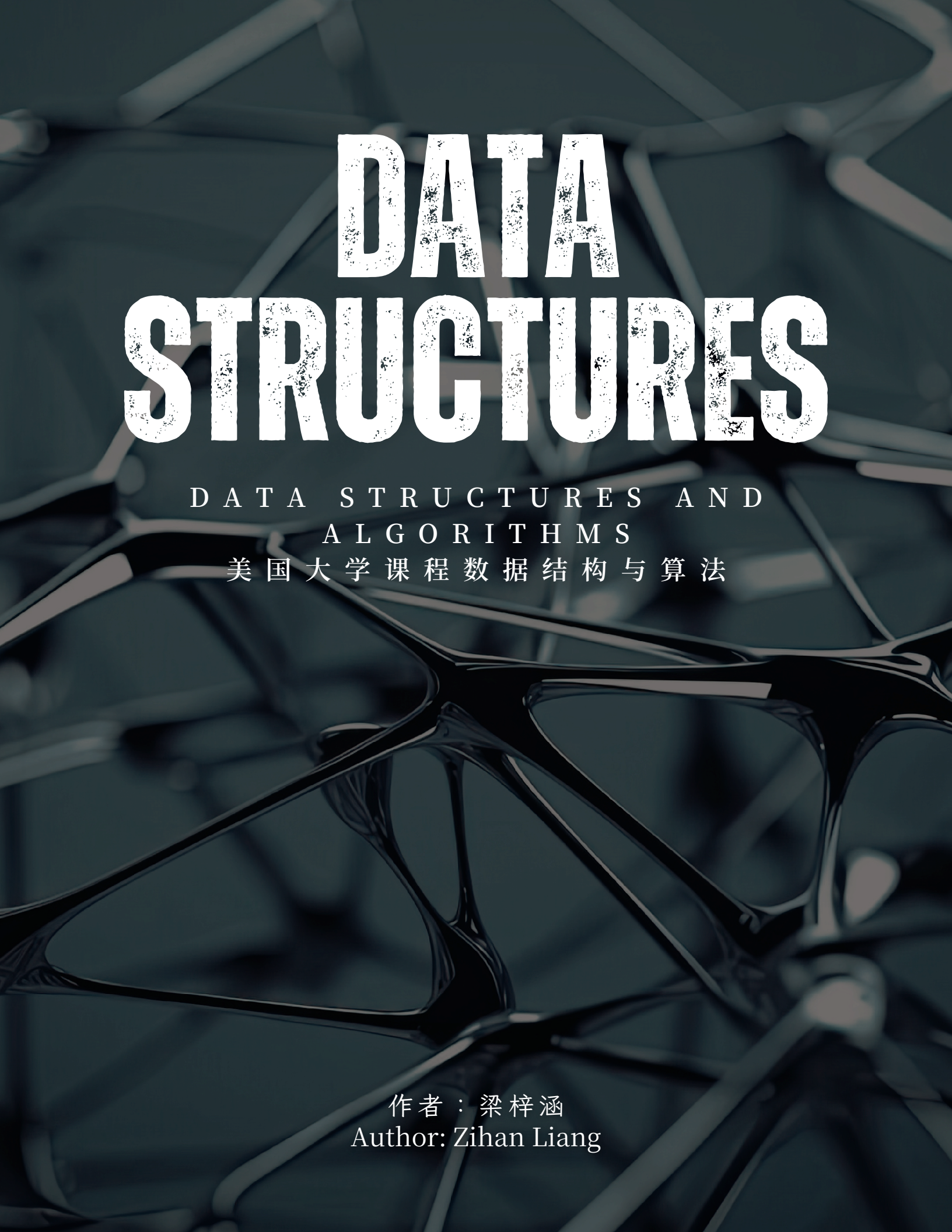


DATA STRUCTURES

DATA STRUCTURES AND ALGORITHMS

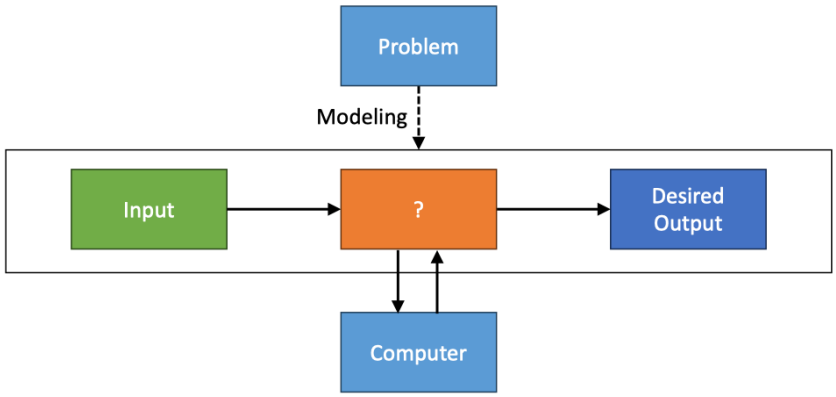
美国大学课程数据结构与算法

作者：梁梓涵
Author: Zihan Liang

Lecture 1 Course Overview

Problem Solving

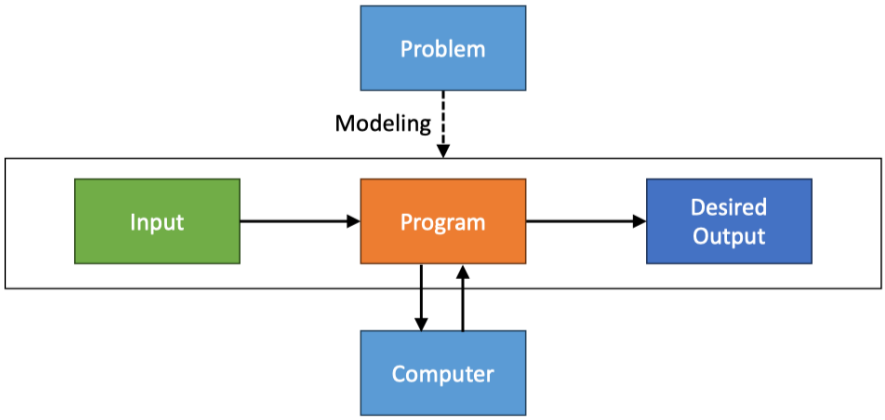
1. Identify a problem: How to get to the Emory Student Center?
2. Formally define the problem and solve the problem using computers



要从模糊的语言转化为“可以操作的模型”。用“Input → ? → Desired Results”来表示：

- Input：我们已知的信息，比如地图、当前位置；
- Desired Results：目标，比如到达 Student Center；
- ?：尚未确定的过程——这正是算法或程序要填补的部分。“Modeling（建模）”就是把问题抽象成输入输出关系的过程。
- “Computer”：我们不再只在纸上思考模型；我们要让机器执行这个“?”，即把建模的思路转化成计算机能理解的形式；此时，模型成为计算任务。

3. Implement a (specific) program for the problem



现在“?”变成了“Program”——也就是把想法实现为实际代码。程序接收输入，用计算机处理，生成输出。这是从理论到实践的桥梁。

4. The heart of the matter

a. 怎样表示和组织输入信息 How to represent/organize data (Data Structures)
b. 怎样设计"配方式"的逻辑步骤，让计算机得出期望输出 How to develop a logical procedure like a "recipe" (Algorithms)

Topics

1. Advanced Collections – binary search trees, balanced search trees, hash tables, priority queues (heaps)
2. Advanced sorting algorithms – heap sort, counting sort, bucket sort, radix sort (LSD)
3. Text processing – pattern matching, compression, tries
4. Dynamic programming – edit distance, longest common subsequence, knapsack problem
5. Greedy algorithms – coin-change problem, Huffman codes
6. Directed graphs – cycle detection, topological sort, strongly connected components, shortest paths (Dijkstra, Bellman-Ford, Floyd-Warshal), minimal spanning tree
7. Network flow – max-flow/min-cut, bipartite matching

Lecture 2 Basics

Problem and Instance

1. 模糊的问题: "How to get to the Emory Student Center?" 问题不够精确，因此计算机无法直接解决。
    a. 起点不明确 Origin: 不知道是从哪出发（MSC 还是当前位置）;
    b. 约束条件 Constraints: 例如交通方式（步行、开车）、障碍物、是否需要无障碍路线;
    c. 目标输出 Desired output: 最终要的是什么？是路线图？最短路径？时间最短的路径？

    现实世界的问题往往含糊不清，计算机需要形式化的输入和输出定义才能求解。
2. 具体化问题: "Give me a path from MSC to Emory Student Center on foot." 在这一页中，原本模糊的问题被形式化成一个具体实例 instance。这里添加了：
    a. 明确的起点（MSC, Math & Science Center）;
    b. 明确的目标（Emory Student Center）;
    c. 明确的约束条件（on foot）。
3. Problem 与 Instance 的定义
    a. A problem (in Computer Science): 一类任务的抽象描述，包含输入、输出的定义，但不包含求解步骤。
        i. Specifies an input
        ii. Computes a desired output (depending on the input)
        iii. Does not specify a method to go from input to output
    b. An Instance of a Problem: A specific case of the problem. 例如"从 MSC 到 Emory Student Center"就是最短路径问题的一个实例。

Algorithm

1. Algorithm 的定义: An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.
    a. 算法是一组清晰定义的 computational steps，它将 input 经过有限次操作转换成 output。
    b. 算法 = 一套明确、有限、有序的指令，用来完成某个任务。
2. Algorithm 的三大要求: An algorithm must be
    a. Specific: The number and types of inputs/outputs must be clearly defined.
    b. Unambiguous: Each instruction is precisely defined and executable. The sequence of instructions is unambiguous and well-ordered.

c. Finite: The notation of an algorithm must have a finite length.

3. Human language and pseudocode
   a. Problem: 对于任意自然数 n，计算 1+2+3+4+…+n。
   b. Algorithm (natural language)
      i. Initialize a variable to store the sum: Start by setting a variable, say sum, to 0. This will hold the running total of the sum as we go through each number.
      ii. Iterate through the numbers from 1 to n:
         - Begin a loop where you will process each integer from 1 to n.
         - On each iteration, take the current number and add it to sum.
      iii. Return the result: Once you have processed all the numbers from 1 to n, return the value of sum.
   c. Algorithm (pseudocode)

```
Algorithm SumTo(n)
    Input: n (the number of natural numbers to sum)
    Output: sum (the sum of the first n natural numbers)
    sum ← 0
    For i from 1 to n do
        sum ← sum + i
    Return sum
End Algorithm
```

   d. Why Pseudocode?
      i. Pseudocode is a high-level, human-readable description of an algorithm.
         - 它介于自然语言（像英语）和编程语言（如 Python、C、Java）之间；
         - 它描述的是逻辑步骤，而不是具体的语法细节。
      ii. Key Characteristics of Pseudocode:
         - Simplified Syntax: 用接近自然语言的方式写算法逻辑，但结构上又模仿编程语言。
         - Language-Independent: 伪代码不属于任何特定编程语言。写出的算法逻辑之后可以被翻译成任何语言（Java、Python、C++ 等）。
         - Clarity and Precision: 虽然伪代码不是正式语言，但必须写得足够清晰，让任何读者都能理解算法的逻辑。

4. How to Evaluate Algorithms?
   a. Properties of an algorithm: 好的 algorithm 要 effective、efficient、并且 reliable。

i. Generality —— 通用性
ii. Determinism —— 确定性
iii. Termination —— 终止性（或有限性）
iv. Correctness —— 正确性
v. Efficiency —— 效率

b. Generality: An algorithm solves a whole class of problems, not just a specific instance. A general algorithm can be applied to a wide variety of inputs or problem types, and it's designed to handle different scenarios without needing to be rewritten or specifically tailored each time. 它能应对不同情况，不需要每次重写。
    i. Sorting algorithms: 它们可以排序 any specific type of data（integers、strings、custom data types 等）。不论数据内容是什么，算法的逻辑（比较 + 交换）都不变。
    ii. Generality = 抽象性 + 可重用性（算法越通用，越具有理论与实际价值。）
c. Determinism: An algorithm is deterministic if it always produces the same output for the same input, with the underlying machine always passing through the same sequence of states. 对同样的输入，总是产生相同的输出，并且经历相同的中间步骤。
d. Termination: An algorithm is called terminating, if it always comes to an end after a finite number of steps. 算法必须在有限步数内结束，否则不能称为算法。
    i. Examples of non-terminating algorithms: 红绿灯、操作系统、扔一个骰子直到到 7…
    ii. Stochastic Termination: A non-terminating algorithm is said to stochastically terminate if the probability of termination converges to one. 有些算法虽然理论上不保证一定会结束，但从概率上来说，它几乎肯定会结束。比如：掷骰子直到掷出 6。理论上，你可能永远都掷不到 6（非常非常小的概率）；但实际上，掷的次数越多，掷出 6 的概率趋近于 1；所以这是一个 Stochastic Termination 算法。它几乎一定会停下来，只是我们无法预测在第几次。
e. Correctness
    i. Partial correctness: An algorithm is partially correct, if any returned result is correct, i.e., corresponds to the intended result. 如果算法返回了结果，那么结果是正确的。但算法可能永远不结束（即不一定终止）。例如一个计算过程可能死循环，但若停止了，输出没问题。

        - Note that an algorithm does not return any result if it does not terminate.
        - Partially correct algorithms may not terminate, but if they terminate, they return the correct result.
    ii. Total correctness: An algorithm is (totally) correct if it is partially correct and terminates. 部分正确性 + 终止性 = 完全正确性
f. Efficiency: Efficiency refers to the speed at which an algorithm executes and its resource consumption (time and space). The same task can be accomplished using different algorithms, but the difference in speed can be enormous.
    i. 给出一串数字让算法排序。可能的算法包括 Bubble Sort、Quick Sort、Merge Sort 等——它们的结果相同，但运行速度差异巨大。
    ii. 效率 = “在保证正确性的前提下，用最少的时间和资源完成任务”。

Data Structures

1. Data Structures 的定义: A data structure is a way to store and organize data in order to facilitate access and modifications. 也就是说，data structure 不仅仅是“存放数据的地方”，而是一种设计和组织数据的策略，让算法能够快速找到、插入、删除或更新信息。
    a. Functional View: Container objects with operations
    b. Data Structures 包含两部分信息: structure information（用于组织数据）和 actual data (payload)，例如，一个 node 既有指针结构，也存放了真实的数据值。
2. Data Structures 的分类
    a. Primitive Data Structures: 最基础的数据类型，例如 int、char、float 等。通常由编程语言直接提供。
    b. Non-Primitive Data Structures: 这类是由 Primitive Data Structures 构建的复杂结构，又分为两类
        i. Linear Structures: 数据按顺序排列，每个元素最多连接到一个前驱和一个后继。
            - Direct Access: 数组（Array）、多维数组（Multidimensional）、记录（Record）
            - Sequential Access: 链表（Linked List）、栈（Stack）、队列（Queue）
        ii. Non-Linear Structures: 数据元素之间不是简单的线性关系，而是层次或网络关系。

- Hierarchical: 树（Tree）、堆（Heap）、字典树（Trie）
- Unordered: 集合（Set）、图（Graph）、哈希表（Hash Table）

3. Dynamic Data Structures: 一种在 runtime 能够自动调整大小或形状的数据组织方式。它与静态数据结构（如固定大小的数组）相对。
    a. Motivation
        i. Array 的长度在 declaration 后通常是固定的，但有时候我们并不知道数据量的大小——比如用户输入多少条数据是不可预期的。
        ii. 希望结构能在运行中 grow/shrink，以节省内存或适应数据变化。
        iii. 理想情况：我们想要能自动扩展、几乎“无限长度”的数据结构（受限于内存）。
    b. Examples: Lists, Trees, Graphs
    c. Advantages
        i. Nodes are dynamically (at run-time) generated and connected.
        ii. Structures can grow/shrink.
        iii. Maximum size of a structure only limited by available memory. Doesn't have to be specified in advance.

Linked List

1. Linked List 基础结构

```
class Node<E> {
    E value;
    Node<E> next = null;
}
```

Linked List 中的 Node 包含两个部分

a. value（值 value）：存储数据。
b. next（指针 reference）：指向下一个节点。

```
[value | next] → [value | next] → null
```

Linked List 通过 reference 把多个节点连接成一条“链”。

2. Linked List 插入操作

插入节点 n2 到 n1 之后 `n1(1) → n2(2) → null`

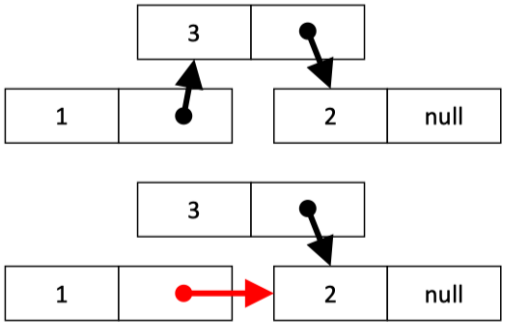
```
Node n1 = new Node(1);
Node n2 = new Node(2);

n2.next = n1.next;  // n2 连接到 n1 的下一个节点（当前为 null）
n1.next = n2;       // n1 连接到 n2
```

插入 n3 到 n1 后面，并且 n1 原本指向 n2 `n1(1) → n3(3) → n2(2) → null`

```
Node n3 = new Node(3);
```

```
n3.next = n1.next;   // n3 的 next 指向 n2
n1.next = n3;        // n1 的 next 改为指向 n3
```

3. Linked List 删除操作



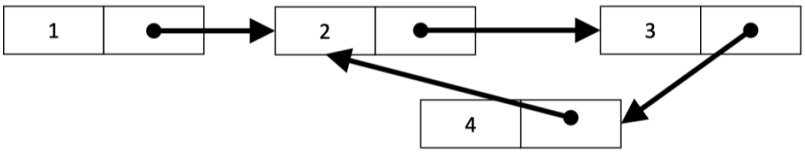

```
n1.next = n1.next.next;
```

让 n1 直接跳过 n3，指向 n3 的下一个节点（即 n2）。结果链表：

```
n1(1) → n2(2) → null
```

Java 使用垃圾回收机制 Garbage Collection: n3 不再被任何对象引用。系统会自动回收 n3 占用的内存。

4. Linked List 的潜在问题



这就产生了 循环引用（cycle）或错误链接 的情况。这种链表有以下几个问题:

   a. 无法确定链表的长度: 因为链表形成了环（例如 2→3→4→2…），遍历时会无限循环。
   b. 无法找到链表的末尾: 理论上链表末尾是指向 null 的节点，但这里循环了，没有节点指向 null。
   c. 无法安全地添加新节点: 因为“末尾”不明确，程序无法判断该把新节点接到哪里。

5. ArrayList vs. LinkedList in Java
   a. ArrayList Recall: ArrayList 是一种基于动态数组实现的 Java 数据结构，当元素数量超过当前容量时，它会自动扩容（通常翻倍），从而支持快速的随机访问和灵活的大小调整。在 ArrayList 里面存放的是一组 objects，而不是基本数据类型（primitive types，比如 int、double）。
   b. ArrayList vs. LinkedList in Java
      i. ArrayList
         - 底层使用 dynamic array 存储元素；

- 当数组装满时，会 自动扩容（通常是容量翻倍）；
- 访问任意元素很快（O(1) 时间），但在中间插入或删除较慢（因为要移动元素）。

ii. LinkedList

- 底层使用 doubly linked list；
- 每个节点包含：当前元素的值；指向 previous；指向 next；
- 插入或删除节点时很方便（只需修改指针），但访问中间元素较慢（必须顺序遍历）。

Stacks

1. Stacks 基础结构
   a. 核心思想: LIFO（Last In, First Out，后进先出）。意思是最后放进去的元素最先被取出。就像叠盘子，最后放的盘子最先拿。

```
interface Stack<E> {
     void push (E ele); // add an element on the top
     E pop(); // return and remove top element
     boolean isEmpty();
     }
```

   b. 结构特征: 只能访问“栈顶”（top）的元素。栈由一组对象组成。
2. Stack using Array
   a. 用一个 数组 存储 Stack 中的元素，并用 top 指针表示当前 Stack 顶。
   b. 主要逻辑:
      i. push()：把新元素放入数组中（top++）。
      ii. pop()：取出 top 元素（top--）。
      iii. isEmpty()：当 top == 0 时为空。
      iv. isFull()：当 top == stack.length 时表示溢出。
   c. 优点: 访问速度快。
   d. 缺点: 容量固定，满了必须重新分配数组。
3. Stack using List
   a. 用 Linked List 来实现 Stack 结构。

b. Stack 顶对应链表的头节点。
c. 主要逻辑:
    i. push()：新建一个节点，把它连在头部。
    ii. pop()：移除头节点并返回其值。
    iii. isEmpty()：当头节点为 null 时表示空。
d. 优点: 不需要预设容量，大小可动态变化。
e. 缺点: 比数组版本稍慢，占用更多内存。

Queues

1. Queues 基础结构
    a. 核心思想: FIFO（First-In-First-Out）先进先出，最早进入队列的元素最先被移除。

```
interface Queue<E> {
    void add(E ele);    // 从末尾添加元素
    E remove();         // 从开头移除元素
    boolean isEmpty(); // 判断是否为空
}
```

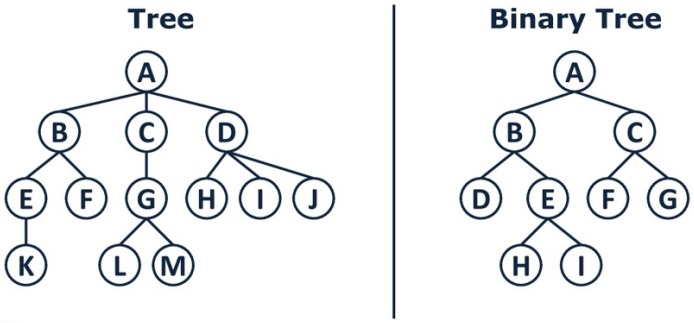
    b. 主要逻辑:
        i. 插入（add)：只能在队列的 末尾（end） 进行;
        ii. 移除（remove)：只能从队列的 开头（front） 进行。
2. Queue using List: 使用 Linked List 实现队列：
    a. 用两个指针：first（头）和 last（尾);
    b. add(E v)：
        i. 新建一个节点 ele;
        ii. 若队列为空 → first = ele;
        iii. 否则 → last.next = ele;
        iv. 最后更新 last = ele。
    c. remove()：
        i. 从 first 取值;
        ii. 移动 first = first.next;
        iii. 若 first == null，队列为空。
    d. 不需要固定容量，长度可动态变化。

Trees

1. Trees 的定义

a. 来源: 如果把 List 的“每个节点只能指向一个下一个节点”扩展为“每个节点可以指向多个后继节点”，我们就得到了 Tree。这里的例子展示了一个 Binary Tree，每个节点有最多两个“指向下一个节点（next1, next2）”的引用。

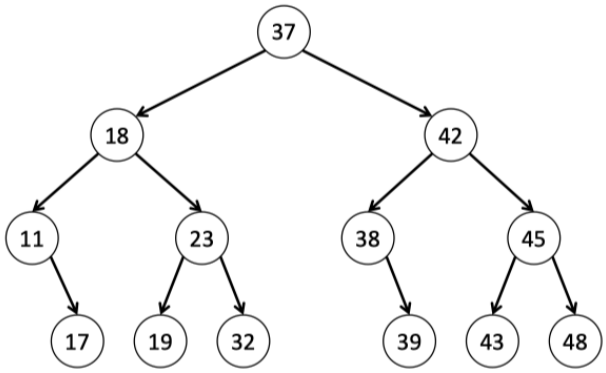
```
class TreeNode<E> {
    E value;
    TreeNode<E> next1;
    TreeNode<E> next2;
}
```



b. Children: The (immediately) next nodes are also called children.

```
class TreeNode<E> {
    E value;
    TreeNode<E> child1;
    TreeNode<E> child2;
}
```

2. Tree: Terminology



a. Terminology
   i. children（子节点）：当前节点的直接后继。
   ii. parents（父节点）：当前节点的直接前驱。
   iii. root（根节点）：唯一没有父节点的节点。
   iv. leaf / leaf node（叶节点）：没有任何子节点的节点。

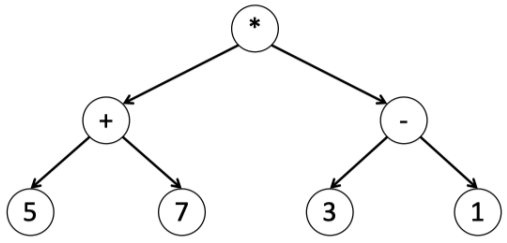
v. inner node（内部节点）：不是叶节点的节点。
vi. edge（边）：两个直接相连节点之间的连接线。
vii. Path（路径）：一连串直接相连的节点。
viii. Length of Path（路径长度）：路径上的边的数量。
ix. Degree of Node（节点的度）：一个节点的子节点个数。
x. Arity of Tree（树的度）：整棵树中节点度的最大值。如果 arity=2 → 叫 二叉树（binary tree）
xi. Height of Tree（树高）：从根节点到最深叶节点的路径长度。

b. Examples
i. Node 18: inner node, children of 37, parent of 11 and 23, sibling of 42.
ii. Path(37→18→23): 2
iii. Degree(42)：2
iv. Degree(38)：1
v. Degree(17)：0
vi. Arity(整个树)：2
vii. Height(树高)：3

3. Properties of Trees
a. Trees are cycle-free.（树结构中不会有循环路径，不能从一个节点经过若干步又回到自己。）
b. A tree with n nodes has exactly n − 1 edges.（因为每加一个新节点，只需一条边把它连到树上。）

4. Relation between Height and #Nodes: 对 binary tree 来说，如果高度是 h，那么
a. 最多有$2^{h+1} - 1$个节点
b. 至少有$h + 1$个节点
c. 最多有$2^h - 1$个 inner nodes
d. 最多有2^h个 leaf nodes

5. Tree Traversal



a. Depth-First Traversal
i. Preorder / Prefix: 先访问节点本身，再访问左右子节点。顺序：根 → 左 → 右。例子：* + 5 7 - 3 1。

ii. Inorder / Infix: 先访问左子节点，再访问节点本身，最后访问右子节点。顺序：左 → 根 → 右。例子：5 + 7 * 3 - 1。
iii. Postorder / Postfix: 先访问子节点，最后访问节点本身。顺序：左 → 右 → 根。例子：5 7 + 3 1 - *。

b. Breadth-First Traversal: 按“层级”来访问节点（从上到下、从左到右）。就像一层层“扫描”整棵树。例子：* + - 5 7 3 1。

Lecture 3: Efficiency & Complexity

Efficiency of Algorithms and Complexity

1. Efficiency of Algorithms: 算法效率 = 资源使用的多少，影响效率的主要因素包括
    a. Run-time (number of operations)
    b. Memory usage
    c. Number of accesses to a secondary storage (e.g., hard disk, cloud)
    d. Communication cost (e.g., network)

    实际性能还受到:
    a. CPU clock rate (Number of operations the CPU can execute per second)
    b. Length of the input
    c. Implementation of operations (programming language dependent)
2. Complexity: The complexity of an algorithm is a function T(n) that describes the resource cost depending on input n. 通常关注的是渐进复杂度 asymptotic behavior，即 $n \to \infty$ 时的增长趋势。
3. Complexity Example: sumTo

```
int sumTo(int n) {
    int sum = 0;            // 1 operation
    for (int i = 1; i <= n; i++) {  // n iterations
        sum += i;           // n addition operations
    }
    return sum;             // 1 operation
}
```

    a. int sum = 0;
        i. 操作类型: 赋值
        ii. 执行次数: 1
        iii. 操作数量: Constant
    b. for (int i = 1; i <= n; i++)
        i. 操作类型: 初始化一次 + 比较 (n+1) 次 + 自增 n 次
        ii. 执行次数: 2n+2
        iii. 操作数量: Linear
    c. sum += i;
        i. 操作类型: 加法 + 赋值
        ii. 执行次数: n
        iii. 操作数量: Linear
    d. return sum;
        i. 操作类型: 返回

ii. 执行次数: 1
iii. 操作数量: Constant

我们得到 T(n) = 3n + 3 → O(n)。

4. Asymptotic Efficiency of Algorithms: 在实际代码中，我们可以把算法执行的操作数（或时间）写成一个函数 T(n)，比如对于 sumTo $T(n) = c_1 n + c_2$。这些式子里面的$c_1, c_2, \dots, c_7$是常数。但是我们关心的不是常数，而是增长趋势。因为随着输入变大，常数项几乎可以忽略并且较小的幂次项也会被主导项盖过去。比如

$$T(n) = c_5 n^3 + c_6 n^2 + c_7$$

当n非常大时，$T(n) \approx c_5 n^3$。因此我们说这个算法是$T(n) = O(n^3)$。

5. Complexity of an Algorithm and Complexity of a Problem
    a. Algorithm Complexity: 衡量一个具体算法运行所需的时间或步骤随输入规模 n 增长的关系。
        i. Bubble Sort → $O(n^2)$
        ii. Merge Sort → $O(n \log n)$
    b. Problem Complexity: 解决某个问题的所有算法中，最优（最紧）复杂度是多少？也就是说，我们不再看单个算法，而是看："有没有可能存在更快的算法？"
        i. "排序问题（Sorting）" 本身的复杂度是 $\Omega(n \log n)$。这意味着：无论你用什么排序算法（只要是基于比较的），你都不可能比 O(n log n) 更快。

Asymptotic Complexity: O-Notation

1. O-Notation: 像我们刚刚提到的，我们只关心 Asymptotic Complexity。那如何只关注最高阶项n^3，在$T(n) = c_5 n^3 + c_6 n^2 + c_7$中？我们使用 O-Notation $O(n^3)$。比如

$$\begin{cases} T_1(n) = 3n^2 + 5n^2 \\ T_2(n) = 2n^3 + 7n^2 + n \\ T_3(n) = n^3 + n^2 + n + 50 \end{cases}$$

它们虽然常数和低阶项不同，但增长趋势都一样（立方增长），所以都是$O(n^3)$。

2. O-Notation 的意义在于
    a. 忽略常数因子与低阶项；
    b. 描述算法随输入规模增长的 "上界"；
    c. 表达算法性能的增长速度级别。
3. Limit Definition of O-Notation

$$O(g(n)) = \{f: \mathbb{N} \to \mathbb{R}^+ \mid \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty\}$$

a. $g(n)$ is an asymptotic upper bound for $f(n)$. $f(n)$的最高阶项，甚至比最高阶项还要大的函数。
b. $f(n)$ grows at most as fast as $g(n)$. $f(n)$是我们之前求得的$T(n)$。
c. $f(n)$ is in $O\big(g(n)\big)$.
d. $f(n) \in O\big(g(n)\big)$.
e. 这说明$f(n)$的增长速度不会比$g(n)$快太多。

4. Example: Complexity of sumTo(n)

```
int sumTo(int n) {
    int sum = 0;            // 1 operation
    for (int i = 1; i <= n; i++) {  // n iterations
        sum += i;           // n addition operations
    }
    return sum;             // 1 operation
}
```

a. Total number of operations

$$f(n) = 1 + (1 + 1 + 1)n + 1 = 3n + 2$$

b. Using the limit definition of O-notation $g(n) = n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n+2}{n} = 3 < \infty$$

c. Since $3 < \infty$: $f(n) \in O\big(g(n)\big) = O(n)$.

5. O-Notation: Properties
   a. Constants: If $f(x) = a \in \mathbb{R}$ is a constant function, then $f \in O(1)$.
      i. $3 \in O(1)$
      ii. $100000 \in O(1)$
      iii. $O\big(\sqrt{42}\big) = O(1) = O(\pi)$
   b. Scalar Multiplication: Let $f \in O(g)$ and $a \in \mathbb{R}$, then $a \cdot f \in O(g)$.
      i. $1000n \in O(n)$
      ii. $23n^5 \in O(23n^5) = O(n^5)$
      iii. $2^{n+a} \in 2^n 2^a \in O(2^n)$
   c. Addition: Let $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $f_1 + f_2 \in O(\max(g_1, g_2))$.
      i. $2n^5 + 25n^3 + 8n^2 + 42n + 8 \in O(n^5)$
      ii. $\log(n) + n \in O(n)$
   d. Multiplication: Let $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$.
      i. $(\log n + 7n + n^2) \cdot \sqrt{n} \in O\big(n^2\sqrt{n}\big) = O(n^{2.5})$
6. Orders of Common Functions

Order	Description	Example	N=10	N=100	N=1000
$O(1)$	constant	A single operation	1	1	1
$O(\log n)$	logarithmic	Binary search	4	7	11
$O(n)$	linear	sequential search	10	100	1000
$O(n \cdot \log^k n)$	log-linear quasi-linear	sorting of an array	40	700	11000
$O(n^2)$	quadratic	matrix addition	100	10000	1000000
$O(n^3)$	cubic	matrix multiplication	1000	1000000	1000000000
$O(n^k)$	polynomial				
$O(2^n)$	exponential	combinations	1024	10^{30}	10^{301}
$O(n!)$	factorial	permutations	10^7	10^{158}	10^{2568}
$O(n^n)$	superexponential	n-Queens problem	10^{10}	10^{200}	10^{3000}

7. Relationship between Classes of Complexity: 复杂度类之间的包含关系如下

$$O(1) \subset O(\log\ n) \subset O(n) \subset O(n\log\ n) \subset O(n^2) \subset O(n^3)$$

   a. 多项式函数之间的顺序: $\forall a \leq b: O(n^a) \subset O(n^b)$，例如$O(n) \subset O(n^2)$。
   b. 指数函数比任何多项式都增长得更快: $O(n^a) \subset O(b^n)$，例如$O(n^3)$远远小于$O(2^n)$。
   c. 对数函数比任何多项式都增长得更慢: $\forall a \geq 1: O(\log\ n) \subset O(n^a)$，例如$O(\log\ n)$小于$O(n)$。

Big Theta: Θ-Notation and Big-Omega: Ω-Notation

1. Limit Definition of Θ-Notation

$$\Theta\big(g(n)\big) = \{f: \mathbb{N} \to \mathbb{R}^+ \mid 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty\}$$

2. Limit Definition of Ω-Notation

$$\Omega\big(g(n)\big) = \{f: \mathbb{N} \to \mathbb{R}^+ \mid 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)}\}$$

3. Big-O、Big-Ω、Big-Θ 的本质区别

符号	含义	直觉比喻	数学关系	取最高阶项吗？
$O\big(g(n)\big)$	上界（增长不会比 g(n)快）	最多这么快	$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$	不一定，只要比它慢或一样快都行
$\Omega\big(g(n)\big)$	下界（增长不会比 g(n)慢）	至少这么快	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)}$	不一定，只要比它快或一样快都行
$\Theta\big(g(n)\big)$	紧界（增长跟 g(n)一样快）	差不多就是这个级别	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$	一定取最高阶项

4. Big-O、Big-Ω、Big-Θ 案例: 对于$f(n) = n^3 + 2n^2 + 100$
   a. Big-O: 只要上界比它大就行。$f(n) \in O(n^3), O(n^4), O(2^n)$。因为这些增长得更快或相等。

b. Big-Ω: 只要下界比它小就行。$f(n) \in \Omega(n), \Omega(n^2), \Omega(n^3)$。因为这些增长得更慢或相等。
c. Big-Θ: 必须“刚好”增长速度一样。$f(n) \in \Theta(n^3)$。只取最高阶项。

Formal Definition of O-Notation

1. Formal Definition of O-Notation

$$O\big(g(x)\big) = \{f: \exists c > 0, \exists x_0 > 0, \forall x \geq x_0, |f(x)| \leq c|g(x)|\}$$

a. 当 x大于某个阈值 x_0之后，$f(x)$的增长不会超过 $c \cdot g(x)$的增长。
b. $3n + 2 \in O(n)$: 因为当 $n > 1$时，$3n + 2 \leq 5n$。
c. $n^2 + 3n + 4 \in O(n^2)$: 因为 $n^2 + 3n + 4 \leq 2n^2$对于 $n \geq 4$。

2. Example of Formal Definition of O-Notation

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

每次循环做 1 个操作，循环 n 次，所以$f(n) = 3n + 2$。我们要找常数 c和 n_0，使 $f(n) \leq c \cdot n$。取 $c = 4, n_0 = 3$即成立，因此 $f(n) \in O(n)$。说明：O(n) 并不是唯一答案。比如 $c = 2000$也可以，$O(n^2)$也 technically 成立，但 O(n) 是最紧的上界。

3. 多变量 O 表示法 Two Variables: 有时算法依赖两个独立参数，比如$f(n, m) = n \cdot m$，定义就扩展为

$$O(f(n,m)) = \{g: \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, n_0 > 0, m_0 > 0, \forall n \geq n_0, m \geq m_0, g(n,m) \leq c \cdot f(n,m)\}$$

当 n和 m都足够大时，$g(n,m)$ 的增长速度不超过 $f(n,m)$的常数倍。

Big-O 推断技巧

1. 对于 single-level loop，若 update expression 为加法或减法，其时间复杂度为 O(n)；若 update expression 为乘法或除法，其时间复杂度为 O(logn)。
2. 对于 nested loops，如果内部循环依赖于外部循环。
   a. 外层循环采用加法和减法运算，内层循环同样采用加法和减法运算，其时间复杂度为 O(n²)。
   b. 【例外】外层循环执行乘除运算，内层循环执行加减运算，其时间复杂度为 O(n)。
   c. 外层循环执行加减运算，内层循环执行乘除运算，其时间复杂度为 O(nlogn)。

d. 外层循环执行乘法和除法运算，内层循环同样执行乘法和除法运算，其时间复杂度为 $O((\log n)^2)$。

e. 总而言之，在大多数情况下，嵌套循环的时间复杂度等于内层循环与外层循环时间复杂度的乘积。但当内层循环的边界依赖于外层循环，且外层循环呈乘法增长（例如 i *= 2），而内层循环呈加法增长（例如 j++）时，则例外。

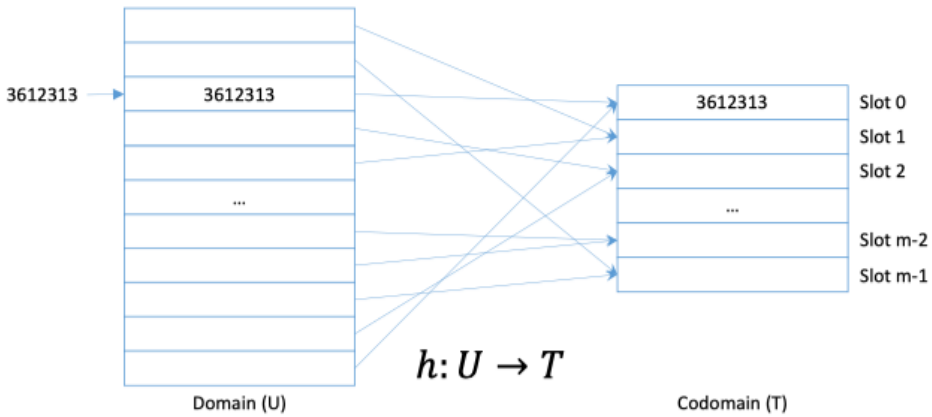
3. 对于嵌套循环，若内层循环不依赖外层循环，则直接将内层时间复杂度乘以外层时间复杂度即可得到整体时间复杂度。
4. 若两个循环是顺序执行的（即非嵌套），则总时间复杂度取两者中的较大值。

$O(n) > O((\log n)) > O(\log n)$。

Lecture 4: Hashing

Motivation

1. Can we directly access the address of a key?
    a. Direct Address Table: 假设每个键 key 都能直接作为内存地址，那查找就能在 O(1)时间内完成。例如学生 ID=3612313，就直接去内存地址 3612313 取数据，速度极快，不需要比较、不需要排序。
    b. 理想方法不现实: 在 Java 中，int 类型是 32 位（4 字节）的有符号整数。取值范围是从-2^{31}到2^{31}，也就是大约 −21 亿 到 +21 亿。如果我们要为每个可能的整数都分配一个 slot，那需要的内存是 2^32 * 4 bytes = 2^34 bytes = 16 GB。也就是说，仅仅为了能支持 O(1)的直接寻址，就得用掉 16GB 的内存。而现实中，我们通常根本不需要支持所有整数。例如学生 ID 只有 7 位数，最多一千万个学生，不到 2^{32}的 1/400。这时分配 16GB 空间是极端浪费的。
2. Hashing: Idea
    a. Idea: 我们不再直接用“Key = Memory Address”，而是用一个函数 h(k)，把大的“键空间”（domain, U）映射到一个较小的“槽空间”（codomain, T）。



    b. 新挑战: 不同的键可能映射到同一个槽，这就是“哈希碰撞 collision”。

Hash Function

1. Definition of Hash Function: Maps keys to (slots in) the hash table.
    a. 给定一个键 k，哈希函数 h(k) 计算出它应该放在哪个槽（slot / bucket）。哈希表只有 m 个槽，而所有可能的键有 n 个。通常 m ≪ n，因此 collisions 是不可避免的。
    b. Loading Factor: $\alpha = n/m$。意思是平均每个槽中存放多少个元素。α越大，碰撞越多，性能越差。

c. 好的 Hash Function 要满足什么条件
   i. 函数形式: $h: U \to \{0,1,\ldots,m-1\}$。把所有可能的键映射到有限的槽编号中。要求满射（surjective）——即尽量利用所有槽。
   ii. 计算高效: h(k)必须能在 O(1)时间算出，否则整个哈希表就失去了意义。
   iii. 均匀分布: 各个键映射的结果应尽量平均分布在 0 到 m−1 之间，否则某些槽太拥挤、某些几乎空，会导致严重碰撞。
   iv. 输入独立性: 相近的键不应映射到相近（或相同）的位置。比如学生 ID 100001 和 100002 不应该都落在相邻的槽。否则哈希表的性能会受到数据模式的影响（例如学号连续）。

d. 最简单的例子: 取模函数

$$h(k) = k \ mod \ m$$

用键除以表长 m，取余数作为槽号。例如
   i. m = 11
   ii. k = 25 → 25 mod 11 = 3
   iii. k = 18 → 18 mod 11 = 7

这样每个整数 key 都能映射到一个 0~10 之间的槽号。取模法简单、计算快（O(1)），但均匀性依赖于 m 的选择。若 m 取不当（比如 m 是 2 的幂而 key 都为偶数），就会形成偏斜。

e. Hash Function for String Values: 整数可以直接取模，但字符串不是数字，没法直接计算 mod m。所以必须先把字符串转化为 integer，再进行取模运算。换句话说：哈希函数必须定义一个从“字符序列”到“整数”的映射。

$$h(\text{STR}) = \left(\sum_{i=1}^{\text{len(STR)}} \text{ord}(\text{STR}[i]) \right) \text{mod} m$$

   i. ord(c) 表示字母 c 在字母表中的序号（例如 a→1, b→2, ⋯, z→26）。
   ii. 先把字符串中每个字符的序号求和。
   iii. 再对表长 m 取模，得到槽号。
   iv. 以字符串 "JAN" 为例: ord(J) = 10, ord(A) = 1, ord(N) = 14

$$h("JAN") = (10 + 1 + 14)\text{mod}17 = 25\text{mod}17 = 8$$

   所以 "JAN" 被映射到槽号 8。
   v. 这种方法简单但不够好，主要因为：不区分字符顺序："JAN" 和 "NAJ" 的哈希值相同（都等于 25 mod 17）;容易产生碰撞：不同字符串可能得到相同的和。

2. Birthday Paradox and Probability of Collisions

a. Birthday Paradox: 在一个有 m = 365 天的世界里，如果随机挑选 n 个人，有多大概率至少有两个人生日相同？
   i. 我们要计算 P(至少有一个相同生日) = 1 − P(没有相同生日)。
   ii. 没有相同生日的概率是

$$P(\text{noshared}) = \frac{365 \times 364 \times 363 \times \cdots \times (365 - n + 1)}{365^n}$$

   iii. 于是

$$P(\text{at least one shared birthday}) = 1 - \frac{365 \times 364 \times \cdots \times (365 - n + 1)}{365^n}$$

   iv. 计算结果出乎直觉

人数 n	至少有一对同生日的概率
10	0.11695
20	0.41144
30	0.50730
40	0.70632
50	0.89123
60	0.97037

只要 23 个人，就有超过 50% 的概率有两人同生日！当有 50 人时，几乎可以确定（97%）至少有一对生日重复。这非常反直觉：365 个可能的“槽”，23 个“键”，加载率才 6%，但碰撞概率已经超过一半。

b. Probability of Collisions: if we have n keys and m buckets

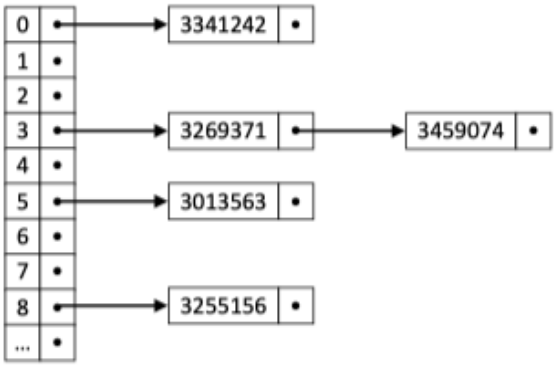
$$P = 1 - \frac{m(m-1)(m-2)(m-3)\ldots(m-n+1)}{n} = 1 - \prod_{k=0}^{n-1} \frac{m-k}{m}$$

$$= 1 - \prod_{k=0}^{n-1} \left(1 - \frac{k}{m}\right) \approx 1 - e^{-\frac{n(n-1)}{2m}}$$

这页展示的公式告诉我们一个非常重要的事实：当 n 增加时，碰撞概率上升速度非常快。即使 n 远小于 m，也会在较小的负载率时出现非忽略的碰撞。

Handling Collisions

1. Chaining: 当多个键（keys）被哈希到同一个槽（slot）时，我们不给它们抢位置，而是在该槽后面挂一个链表（list），把这些键都串在一起。



a. Chaining 的基本思想
   i. 左边是一列槽（0,1,2,…）。
   ii. 每个槽对应一条“链”。
   iii. 例如槽 3（index 3）后面挂着两个元素：3269371 → 3459074，说明这两个 key 的哈希值相同（都映射到 3）。
   iv. 简言之：哈希表主结构是一组桶；每个桶再包含一个链（或小集合），来存储发生碰撞的元素。
b. Chaining 的平均查找效率: 给定 m 是哈希表的槽数、n 是存储的元素数，则 Loading Factor: $\alpha = n/m$是平均每个槽上的元素数量。则平均查找复杂度为

$$O(1+\alpha)$$

当α较小时（如 < 0.7），查找仍接近 O(1)。这说明 Chaining 虽然允许碰撞，但只要负载控制得当，它的性能依然非常接近常数时间。

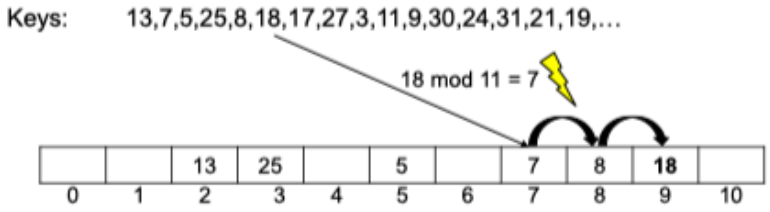
2. Open Addressing: All elements are stored directly in the hash table itself. 换句话说，每个 slot 要么是空的，要么正好存一个元素。没有“挂链表”的结构。
   a. 如何处理碰撞: When a collision occurs, the search for another ‘open’ slot happens within the same table.
      i. 计算 h(3459074)=3，结果槽 3 已经被占用，那就寻找槽 4、槽 5、槽 6……直到找到空位，把这个元素放进去。
      ii. 这种“在表里找下一个空位”的过程就叫 probing。
   b. Loading Factor 限制: One consequence is that the load factor can never exceed 1.

$$\alpha = \frac{n}{m} \leq 1$$

3. Linear Probing: A simple probing function simply checks (iteratively) the next slot in the hash table until an open slot is found. 当碰撞发生时，不使用链表、不跳跃，而是一个一个地往后看，直到找到下一个空槽为止。

a. 数学表达: $h_i(k) = (h(k) + i)\ mod\ m$，其中$i = 0,1,2,3,4$ ...表示第一次探测原位置，第二次看下一个槽，以此类推，循环到表头。

b. 案例



插入 18 的时候，h(18)=7，但槽 7 已经被 7 占用。所以依次探测：Slot 8 → 被 8 占用；Slot 9 → 空位→把 18 存到 Slot 9。

c. 线性探测的问题 —— Clustering: Large groups of consecutive occupied slots (clusters) become more likely to grow even larger.

i. 当表中出现一段连续被占用的区域时，新的键若哈希落在这段区域前后，很容易继续“撞”进去，导致这块区域越堆越长，称为 Primary Clustering

ii. 这会造成:

- 查找时间越来越长；
- 插入性能下降；
- 哈希分布变得不均匀。

4. Polynomial Probing: 设每个键 k的初始位置 home address 为 $h(k)$。当发生碰撞时，我们不只是简单地往后移动，而是按一个多项式计算新的探测位置

$$h(k,j) = (h(k) + c_1 j + c_2 j^2 + \cdots) \bmod m$$

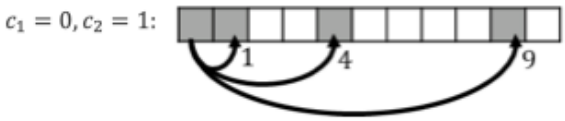
其中$j = 0,1,2,...$是第几次探测；$c_1, c_2, ...$是常数系数；m是哈希表的大小。

a. Linear Probing 是它的特例

$$h(k,j) = (h(k) + c_1 j) \bmod m$$

当 $c_1 = 1$时，就是我们前面学的 Linear Probing。如果 $c_1 = 3$，那么每次探测间隔是 3 个槽，变成“步长为 3 的 Linear Probing”。

b. Quadratic Probing: 避免连续槽的 clustering。



$$h(k,j) = (h(k) + c_1 j + c_2 j^2) \bmod m$$

最常用的形式是 $c_1 = 0, c_2 = 1$，也就是

$$h(k,j) = (h(k) + j^2) \bmod m$$

探测序列就成了平方序列： $h(k) + 1^2, h(k) + 2^2, h(k) + 3^2, ...$

c. Polynomial Probing 的问题与局限

i. All keys having the same home slot will follow the same probing sequence. 例如$h(k) = (k) \bmod 11$，那么键 5、16、27、38、49、60……都会先落在槽 5，碰撞后探测的顺序也完全一样。于是这些键之间还是会相互干扰。这叫做 Secondary Clustering。
ii. Potentially long probing sequences.
iii. Polynomial probing doesn't avoid clustering. Clusters just aren't consecutive in the hash table (but they are still there).

5. Double Hashing: 不要只用一个哈希函数来决定位置，而是用两个$h_1(k)$和$h_2(k)$。
   a. Double Hashing 探测公式: Double Hashing 用两个$h_1(k)$和$h_2(k)$，其中
      i. $h_1(k)$: 决定 home slot；
      ii. $h_2(k)$: 决定 step size。

      也就是说，每次碰撞后跳多远，由第二个哈希函数决定。

      $$h(k,i) = \big(h_1(k) + i \cdot h_2(k)\big) \bmod m$$

      其中
      i. i是第几次探测；
      ii. 每个键k都有自己独特的步长$h_2(k)$。
   b. 独立性的意义: 如果两个哈希函数是“独立的”，即

      $$P\big(h_1(k) = h_2(k)\big) = \frac{1}{m}$$

      那每个键的探测路径几乎是独一无二的。这意味着即使两个键最初落在同一个槽，它们之后的探测路径也完全不同，从而几乎完全避免了线性探测和二次探测的 clustering。
   c. Double Hashing 限制条件: 让$h_2(k) = 1 + (k \ mod \ (m'))$其中$m'$通常取$m - 1$，保证步长始终在 1 到 m−1 之间。
   d. 实际案例: 假设$h_1(k) = k \bmod m, h_2(k) = 1 + (k \bmod (m'))$。
      i. 假设我们有一个 Hashing Table 共有 m = 11 个槽，我们要插入 27, 38, 49, 59 这五个数值。我们使用 Double Hashing

         $$h_1(k) = k \bmod 11, h_2(k) = 1 + (k \bmod 10)$$

         探测公式: $h(k,i) = \big(h_1(k) + i \cdot h_2(k)\big) \bmod 11$。
      ii. 插入 27: $h_1(27) = 27 \bmod 11 = 5, h_2(27) = 1 + (27 \bmod 10) = 8$

         $$h(27,0) = (5 + 0 \cdot 8) \bmod 11 = 5$$

         槽 5 空 → 插入到槽 5。
      iii. 插入 38: $h_1(38) = 38 \bmod 11 = 5, h_2(38) = 1 + (38 \bmod 10) = 9$

         $$h(38,0) = (5 + 0 \cdot 9) \bmod 11 = 5$$

         槽 5 被占 → 第二次探测

         $$h(38,1) = (5 + 1 \cdot 9) \bmod 11 = 3$$

         槽 3 空 → 插入槽 3。

6. Open Addressing: Deletion
    a. 不能直接删除（清空）槽: 如果你直接把一个槽设为空（empty），那么搜索过程在到达这里时，会误以为后面都没有该元素了，导致搜索失败。比如
    key1 → 插在槽 3
    key2 → 冲突，从槽 3 探测到槽 4
    如果你删除了 key1 并把槽 3 清空，那查找 key2 时会以为："槽 3 空了，那 key2 肯定不存在。"
    b. 墓碑标记 Tombstone: 当你删除一个键：不是真的把槽设为空；而是标记为 tombstone。意思是：这里原本有数据，被删除了，但别停下搜索，还要继续往后探测。
7. Chaining vs. Open Addressing

维度	Chaining	Open Addressing
Memory Overhead	每个槽位都要额外存储指针（因为每个桶对应一个链表），所以会占用额外内存。	所有元素都直接放在哈希表里，没有指针，更节省内存。
Performance with Load Factor	性能在负载因子增大时比较稳定，只要哈希函数够均匀。	随着负载因子上升，性能明显下降（探测序列越来越长）。
Deletion	删除很简单：从链表中删掉节点即可，不影响其他元素。	删除比较复杂，要用 tombstone（墓碑标记），否则会破坏探测序列。
Cache Performance	因为链表节点在内存中分散，缓存命中率差。	元素存在连续的数组中，缓存命中率高，访问更快。
Use Cases	适合元素数量不确定、冲突频繁、或者数据量可能扩张的场景。	适合内存有限、且元素数量较少且可预估的场景。

Rehashing

1. Rehashing: 当哈希表太满时，创建一个更大的新表，然后 recompute 所有已有元素的哈希值并放进去。
    a. 哈希表性能依赖于它的 Load Factor: $\alpha = n/m$。当 Load Factor 太高时（比如超过 0.7 或 0.8）：冲突（collision）会显著增加；平均查找/插入时间会从 O(1) 退化到 O(n)。
    b. Rehashing 的主要目的是 减少冲突、保持插入和查找的 O(1) 效率。
2. Rehashing 的时间复杂度分析

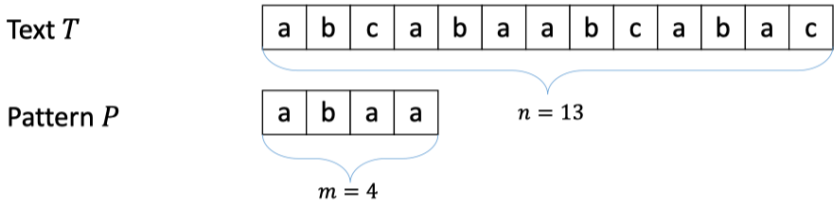
a. Rehashing 是一个 线性时间操作 $O(1) + O(n) + O(n) = O(n)$。不过它不需要每次插入都做——Rehashing 是**偶尔发生**的，一旦发生，摊到每次插入上的平均成本仍然是 O(1)。这就是所谓的 amortized O(1) 时间复杂度。
    i. Step 1 创建新表: 分配一个新的、更大的哈希表空间。这个步骤的时间复杂度是 O(1)，因为只是一次性分配。
    ii. Step 2 遍历旧表中的所有元素: 要把旧表中所有 n 个元素取出：这个步骤是 O(n)。
    iii. Step 3 把每个元素插入新表: 对每个元素重新计算 hash 值并插入新表。单次插入平均是 O(1)，但总共有 n 个元素，所以：插入所有元素共耗时 O(n)。

Lecture 5: Text Processing – I

String Matching Problem

1. String Matching Problem 的定义：“String Matching”就是找到一个模式（pattern）在文本（text）中出现的位置。
   a. 案例: Ctrl + F（或 Mac 上的 Command + F）就是 String Matching。
   b. 数学定义:
      i. The text is given as an array $T[1:n]$ of length n.
      ii. The pattern is array $P[1:m]$ of length $m \leq n$.

      两者的元素都来自同一个有限字符集 alphabet，记作Σ，比如: $\{a, b, c, \ldots, z\}$ 或者{0,1}（在二进制匹配中）。



      String Matching 的任务是：在 T 中找出所有子串，看看有没有与 P 完全相等的。

2. Shift: 如果 Pattern P和 Text T的某一部分对齐，且满足

$$T[s+1:s+m] = P[1:m]$$

   那么这个平移量 s 就称为一个 valid shift。如果不匹配，就是 invalid shift。比如说，假设文本是 T = a b c a b a a a b c（长度 n = 10）；模式是 P = a b a a（长度 m = 4）。
   a. Shift $s = 0$
      T: [$a\ b\ c\ a\ b\ a\ a\ a\ b\ c$]
      P: [$a\ b\ a\ a$]
         i. 对齐的是前四个字符，这个时候
            - a = a 正确
            - b = b 正确
            - c ≠ a 错误
         ii. 这个 shift 是 invalid。
   b. Shift $s = 1$
      T: a [$b\ c\ a\ b\ a\ a\ a\ b\ c$]
      P: [$a\ b\ a\ a$]
         i. 此时对齐部分是 b c a b，这个时候
            - a ≠ b 错误
         ii. 这个 shift 是 invalid。

c. Shift $s = 2$
   T: $a\ b\ [c\ a\ b\ a\ a\ a\ b\ c]$
   P: $[a\ b\ a\ a]$
   i. 对齐部分是 c a b a，这个时候
      ○ a ≠ c 错误
   ii. 这个 shift 是 invalid。

3. Notation and Terminology
   a. Σ^*: the set of all finite-length strings formed using characters from the alphabet Σ. 比如 $\Sigma^* = \{\varepsilon, a, b, ab, ba, aaa, abab, \dots\}$，只要是用$\Sigma$中的字符拼成的、长度有限的字符串，都属于$\Sigma^*$。
   b. Empty string ε：长度为 0 的字符串叫空串，它也属于Σ^*。空串的一个特点是，任何字符串 x 拼接 ε 都不会变
   $$x\varepsilon = \varepsilon x = x.$$
   c. The length of a string x is denoted $|x|$. 比如 $|\text{abc}| = 3$。
   d. Concatenation: 两个字符串 x 和 y 可以拼接成一个新字符串，拼接记作 xy。例如 x = "ab"、y = "cd"、xy = "abcd"。拼接后的长度是
   $$|xy| = |x| + |y|.$$
   e. Prefix: 如果一个字符串 w 是另一个字符串 x 的前面一部分，就说“w 是 x 的 prefix”，记作: $w \sqsubseteq x$。数学上定义，如果存在某个字符串y，使得$x = wy$，那么w 是 x 的 prefix。比如 x = "abcca"、w = "ab"，因为“abcca” = “ab” + “cca”，所以“ab”是“abcca”的 prefix。
   f. Suffix: 如果一个字符串 w 是另一个字符串 x 的末尾一部分，就说“w 是 x 的 suffix”，记作: $w \sqsupseteq x$。数学上定义，如果存在某个字符串y，使得$x = yw$，那么 w 是 x 的 prsuffixefix。比如 x = "abcca"、w = "cca"，因为“abcca” = “ab” + “cca”，所以“cca”是“abcca”的 suffix。
   g. Proper Prefix/Suffix: 不包括字符串本身的 prefix/suffix。比如 s = "abcca"
      i. 所有 Prefix: a, ab, abc, abcc, abcca
      ii. 所有 Suffix: a, ca, cca, bcca, abcca
      iii. Proper Prefix: a, ab, abc, abcc
      iv. Proper Suffix: a, ca, cca, bcca

Naïve String-Matching Algorithm

1. Naïve String-Matching Algorithm 流程
   a. 问题设置: $T = abcde\ (n = 5); P = abf (m = 3)$
   b. 对齐位置个数: $n - m + 1 = 3$
   c. Step 1: 【第一次对齐】
   $T = [abcde]$

$P = [abf]$

i. 比较 1: T[0]=a 和 P[0]=a 正确

ii. 比较 2: T[1]=b 和 P[1]=b 正确

iii. 比较 3: T[2]=c 和 P[2]=f → 此时立即停止本次比较，不会再比较 P 后面的字符（因为已经确定这次不匹配）

d. Step 2: 模式串右移一个位置，再从新的起点重新开始比较，【第二次对齐】

$T = a[bcd]e$

$P = \ [abf]$

i. 比较 1: T[0]=b 和 P[0]=a → 此时立即停止本次比较

e. Step 3: 模式串右移一个位置，再从新的起点重新开始比较，【第三次对齐】

$T = ab[cde]$

$P = \quad [abf]$

i. 比较 1: T[0]=c 和 P[0]=a → 此时立即停止本次比较

f. 再往右移就会让 P 的最后一个字符超出 T 的范围，因此算法停止。输出为未匹配。

2. Naïve String-Matching Algorithm 伪代码

```
Naïve-String-Matcher(T, P, n, m)
1. for s = 0 to n - m
2.     match = true
3.     for j = 1 to m
4.         if P[j] != T[j + s]
5.             match = false
6.             break
7.     if match == true
8.         print "Pattern occurs with shift s"
```

3. Naïve String-Matching Algorithm 时间复杂度

a. 总体时间复杂度: $\Theta((n-m+1)\times m)$

b. 最坏情况: $\Theta(nm)$

c. 最好情况是每次对齐的第一个字符就失败: $\Theta(n)$

Boyer-Moore-Horspool Algorithm

1. Bad-Character Rule 概念建立

a. Notation

i. Text: T

ii. Pattern: P

iii. 当前模式起始在 T 中的位置 shift: s

iv. 当前在模式中的索引: j

v. 模式的长度: m

b. Mismatch: 此时我们从右往左比较当发现 $T[s+j] \neq P[j]$ 时，发生了 mismatch，触发 Bad Character Rule，这时 Bad Character 是$T[s+j]$

i. 如果 Bad Character 在模式 P 中出现过（位置为 l）:

$$shift = j - l.$$

ii. 如果 Bad Character 不在 P 中:

$$shift = j + 1.$$

即整个模式右移超过坏字符所在的位置。

c. 推导相对位移

i. 定义 λ: $\lambda(x)$表示字符 x 在模式 P 中距离最右端有多远。如果字符 出现在离右边第 l 个位置，那么距离右端的距离是 $m-l$。

ii. 相对位移

$$\lambda\big[T[s+j]\big] - (m-j)$$

- $\lambda\big[T[s+j]\big]$: 模式中该 Bad Character 距离末尾的距离；
- $(m-j)$: 我们现在比较到的字符距离末尾的距离；
- 相减 → 就得到了模式需要向右移动多少。

iii. 代入具体字符: 假设 a 是 Bad Character，代入公式

$$\lambda[a] - (m-j)$$

由于$\lambda[a] = m-l$，得到$m-l-(m-j) = j-l$

2. Bad-Character Rule 例子

a. 问题设置: $T = ABCDEF$ $(n=6)$; $P = ACE (m=3)$

b. Step 1: 【第一次对齐 $s=0$】

$T = [ABC]DEF$

$P = [ACE]$

i. C vs E → mismatch，坏字符 x = C

ii. C 在 P 中的最右位置是 l = 2

iii. shift = j − l = 3 − 2 = 1

c. Step 2: 【第二次对齐 $s=1$】

$T = A[BCD]EF$

$P = \ \ [ACE]$

i. D vs E → mismatch，坏字符 x = D

ii. D 不在 P 中

iii. shift = j + 1 = 3 + 1 = 4

3. Boyer–Moore–Horspool Algorithm

a. BMH 算法核心: 当出现 mismatch 时，不是挪 1 个字符，而是根据坏字符在模式中的位置来决定跳多少。

b. Boyer–Moore–Horspool Algorithm 流程

i. 问题设置: $T = ABCDEF\ (n = 6); P = ACE (m = 3)$

ii. Step 1: 构建 Bad Character Table。BMH 先对模式 P 的前 m−1 个字符进行预处理。构建规则: $\lambda[x] =$ 从右端算起，字符 x 到末尾的距离（不包括最后一个字符）；如果字符不在模式中，$\lambda[x] = m$。

字符	位置	$\lambda[x]$
A 需要预处理的	1	3-1=2
C 需要预处理的	2	3-2=1
E	3	3
其他字符	不在 P 中	3

iii. Step 2: 匹配阶段

iv. Step 2.1:【第一次对齐 $s = 0$】

$T = [ABC]DEF$

$P = [ACE]$

- C vs E → mismatch，坏字符 x = C
- $\lambda[C] = 1$
- Shift = 1

v. Step 2.2:【第二次对齐 $s = 1$】

$T = A[BCD]EF$

$P =\ \ [ACE]$

- D vs E → mismatch，坏字符 x = D
- $\lambda[D] = 3$
- Shift = 3

vi. Step 2.3:【第三次对齐 $s = 4$】

$T = ABCD[EF]$

$P =\ \ \ \ \ [ACE]$

- 已越界（n = 6）
- 结束匹配

vii. Step 3: 最终结果: 没有找到完整匹配。

4. Boyer–Moore–Horspool Algorithm 伪代码

```
BMH-Matcher(T, P, n, m)
1.  λ = badCharProcessing(P)
2.  s = 0
3.  while s ≤ n − m do
4.      j = m
5.      while j > 0 and T[s + j] == P[j] do
6.          j = j − 1
7.      if j ≤ 0 then
```

```
8.          return s
9.      s = s + max(1, λ[T[s + j]] + j − m)
10. return −1

badCharProcessing(P, m)
1.  Σ = the set of all alphabets
2.  M = |Σ|
3.  λ = a new array of size M
4.  for i = 1 to M do
5.      λ[i] = m
6.  for l = 1 to m do
7.      λ[P[l]] = m − l
8.  return λ
```

5. Boyer–Moore–Horspool Algorithm 时间复杂度
    a. Average Case: $O(n)$.
    b. Worst Case: $O(nm)$.
        i. 文本长度为 n，模式串长度为 m。
        ii. 在最糟糕的情况下，算法可能需要进行大约 $n \times m$次字符比较。

Knuth-Morris-Pratt (KMP) Algorithm

1. Prefix Function: 对于一个模式串$P[1..m]$，Prefix Function 是一个长度为 m 的数组 π

$$\pi[q] = P[:q]\text{的最长前缀长度(proper prefix)} = \text{后缀(suffix)}$$

换句话说: $\pi[q]$表示：在模式串前 q 个字符中，既是前缀又是后缀的最长公共部分的长度。

    a. 案例: P = a b a b a c a，我们要计算$\pi[1..7]$。
        i. P[:1] = a → 没有真前缀、没有真后缀 → $\pi[1] = 0$
        ii. P[:2] = a b → 前缀 a/后缀 b 不相等 → $\pi[2] = 0$
        iii. P[:3] = a b a → 前缀 a, ab/后缀 a, ba → 最长匹配是 a → $\pi[3] = 1$
        iv. 以此类推
    b. Prefix Function 的意义: $\pi[q]$告诉我们："如果匹配到第 q 个字符失败了，下次从哪里重新开始比较最合理。"
2. Knuth-Morris-Pratt (KMP) Algorithm 流程
    a. 问题设置: $T = a\,b\,a\,c\,a\,a\,b\,a\,c\,c\,a\,b\,a\,c\,a\,b\,a\,a\,b\,b$; $P = a\,b\,a\,c\,a\,b$
    b. Step 1: 计算 Prefix Function

Q	1	2	3	4	5	6
P[q]	a	b	a	c	a	b
$\pi[q]$	0	0	1	0	1	2

c. Step 2: 匹配阶段
d. Step 2.1
   $T = [a\ b\ a\ c\ a\ a]\ b\ a\ c\ c\ a\ b\ a\ c\ a\ b\ a\ a\ b\ b$
   $P = [a\ b\ a\ c\ a\ b]$
   i. Comparison 1-5: match
   ii. Comparison 6: $T[6] \neq$ P[6] mismatch
      - 我们查询$\pi[5] = 1$，根据
        Shift = 已匹配字符数 − 可复用前缀长度
        因此我们需要移动 P，5 − 1 = 4 个位置。
e. Step 2.2 (Shift 1)
   $T = a\ b\ a\ c\ [a\ a\ b\ a\ c\ c]\ a\ b\ a\ c\ a\ b\ a\ a\ b\ b$
   $P =\ \ \ \ \ \ [a\ b\ a\ c\ a\ b]$
   i. a = a，这里不计入 Comparison 个数
   ii. Comparison 7: $T[6] \neq$ P[2] mismatch
      - 我们查询$\pi[1] = 0$，因此我们需要移动 P，1 − 0 = 1 个位置。
f. Step 2.3 (Shift 2)
   $T = a\ b\ a\ c\ a\ [a\ b\ a\ c\ c\ a]\ b\ a\ c\ a\ b\ a\ a\ b\ b$
   $P =\ \ \ \ \ \ \ \ [a\ b\ a\ c\ a\ b]$
   i. Comparison 8-11: match
   ii. Comparison 12: $T[10] \neq$ P[5] mismatch
      - 我们查询$\pi[4] = 0$，因此我们需要移动 P，4 − 0 = 4 个位置。
g. Step 2.4 (Shift 3)
   $T = a\ b\ a\ c\ a\ a\ b\ a\ c\ [c\ a\ b\ a\ c\ a]\ b\ a\ a\ b\ b$
   $P =\ \ \ \ \ \ \ \ \ \ \ \ [a\ b\ a\ c\ a\ b]$
   i. Comparison 13: $T[10] \neq$ P[1] mismatch
      - 我们无需查询，因为没有已匹配字符数，直接后移 1 个位置
h. Step 2.5 (Shift 4)
   $T = a\ b\ a\ c\ a\ a\ b\ a\ c\ c\ [a\ b\ a\ c\ a\ b]\ a\ a\ b\ b$
   $P =\ \ \ \ \ \ \ \ \ \ \ \ \ \ [a\ b\ a\ c\ a\ b]$
   i. Comparison 14-19: match，结束。匹配成功（q = m = 6）→ 输出匹配位置或算法停止。

3. Knuth-Morris-Pratt (KMP) Algorithm 伪代码

```
KMP-Matcher(T, P, n, m)
1.  π = ComputePrefixFunction(P, m)
2.  q = 0              // number of characters matched
3.  for i = 1 to n do  // scan the text from left to right
4.      while q > 0 and P[q + 1] ≠ T[i] do
5.          q = π[q]   // next character does not match
```

```
6.      if P[q + 1] == T[i] then
7.          q = q + 1  // next character matches
8.      if q == m then // is all of P matched?
9.          print "Pattern occurs with shift", i - m
10.         q = π[q]   // look for the next match

ComputePrefixFunction(P, m)
1.  let π[1 : m] be a new array
2.  π[1] = 0
3.  k = 0
4.  for q = 2 to m do
5.      while k > 0 and P[k + 1] ≠ P[q] do
6.          k = π[k]
7.      if P[k + 1] == P[q] then
8.          k = k + 1        // next character matches
9.      π[q] = k
10. return π
```

4. Knuth-Morris-Pratt (KMP) Algorithm 时间复杂度
    a. 预处理阶段 ComputePrefixFunction: $O(m)$.
    b. 搜索阶段 KMP-Matcher: $O(n)$.
    c. 总时间复杂度: $O(n+m)$.

Summary

1. String Matching Problem: The problem of finding all valid shifts with which a given pattern P occurs in a given text T.
2. 时间复杂度

Algorithm	Preprocessing Time	Matching Time
Naïve	0	$O\big((n-m+1)m\big)$
Boyer-Moore-Horspool	$O(m+\|\Sigma\|)$	$O\big((n-m+1)m\big)$
Knuth-Morris-Pratt	$O(m)$	$O(n)$

Lecture 6: Text Processing – II

Problem of Pattern Matching Methods

1. 普通的 Pattern Matching 算法的局限
    a. Naïve、BMH 和 KMP 都适合单次匹配: 当你有一个文本和一个模式串，要查找该模式是否存在时，它们都很好用。但是，当文本很长而查询很多时（比如 10 万次搜索），每次重新扫描整个文本就变得非常低效。
    b. 问题"固定文本 + 多次查询": 如果文本是固定的（比如一本书或一个词典），但我们要不断进行不同的搜索，比如做自动补全（autocomplete）或拼写检查（spell check），那应该怎么做？
2. 前缀树 Trie 的动机
    a. Trie 的核心思想是
        i. 用树结构保存所有单词的前缀。
        ii. 每条从根到叶子的路径代表一个单词。
        iii. 查找、插入、删除的复杂度只与字符串长度有关，O(m)。
    b. 如何应用到具体问题中
        i. 对于自动补全（autocomplete），我们只需找到输入前缀对应的节点，再输出其所有子节点。
        ii. 对于拼写检查（spell check），我们可以快速找到前缀相近的候选词。
        iii. 对于海量查询（百万级别的查询），Trie 的预处理开销很快就能"回本"。

Trie

1. Trie 的定义
    a. Trie: A tree-based data structure for storing strings in order to support fast pattern matching.
        i. Trie 是一种树形数据结构；
        ii. 它用于存储一组字符串（例如词典里的单词）；
        iii. 设计目的就是为了实现快速的模式匹配和前缀查询。
    b. 支持的主要操作:
        i. Pattern matching: 比如判断某个单词是否存在于集合中。例：search("bear") → True。
        ii. Prefix matching: 比如查找所有以某个前缀开头的单词。 例：startsWith("be") → ["bear", "bell", "best", "bench", ...]。
    c. Trie 的形式化定义与结构特征

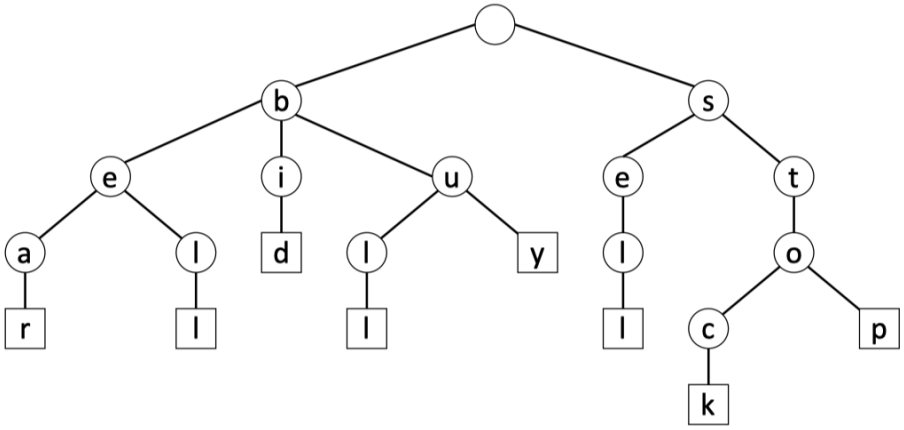
i. 设:
   - S = 一组字符串集合（比如{"bear", "bell", "bid"}）。
   - Σ = alphabet（比如{a, b, c, ..., z}）。

ii. 条件: S 中没有一个字符串是另一个字符串的前缀（否则会有歧义，比如 “be” 和 “bear”）。

iii. Trie 的构造规则: A standard trie for S is an ordered tree T.

d. Properties of T
   i. Each node of *T*, except the root, is labeled with a character of Σ.
   ii. The ordering of the children of an internal node of *T* is determined by a canonical ordering of the alphabet Σ.
   iii. The concatenation of all the characters on the path from the root to a leaf node is a string in the set.
   iv. Every internal node of *T* has at most |Σ| children.
   v. The number of the strings = the number of the external nodes.
   vi. The height of *T* is equal to the length of the longest string in *S*.

2. Trie: Example



a. 节点构成
   i. 圆形节点（internal node）: 内部节点，表示中间的字符路径。
   ii. 方形节点（external node）: 外部节点（叶子节点），表示一个完整的单词。

b. 观察结构
   i. 你能看到两个主要的分支: 左边以字母 b 开头；右边以字母 s 开头。这表明所有单词要么以 “b” 开头，要么以 “s” 开头。然后每个分支继续展开:
      - b → e → a → r
      - b → e → l → l
      - b → i → d

- b → u → l → l
- b → u → y
- s → e → l → l
- s → t → o → c → k
- s → t → o → p

ii. 这些路径就是我们最终的单词集合:

{bear, bell, bid, bull, buy, sell, stock, stop}

c. Height of T: Trie 的高度与最长字符串长度相等

$$h(T) = \max_{s_i \in S} |s_i|$$

比如对于上面这个，最长的单词是 "stock"（是 5 个字母）。

d. Number of Nodes of T: 数出图上所有圆形节点（internal nodes）和方形节点（external nodes）的数量。

3. Trie: Insertion

a. 我们要把字符串 bell 插入到空的 Trie 中:

i. 从根节点开始: Trie 一开始是空的，只有一个根节点（不存任何字符）。

ii. 处理第一个字符 'b':

- 计算该字符在字母表中的索引位置: index = 'b' - 'a' = 1
- 检查根节点的 children[1] 是否为空。
  是空的 → 新建一个节点代表 'b'。

iii. 处理第二个字符 'e':

- 进入 'b' 节点;
- 计算 'e' 的索引：index = 'e' - 'a' = 4;
- children[4] 为空 → 创建新节点 'e'。

iv. 处理第三个字符 'l':

- 进入 'e' 节点;
- 创建 'l' 节点。

v. 处理第四个字符 'l'（最后一个）:

- 再创建一个 'l' 子节点;
- 标记该节点 isEndOfWord = true，表示 "bell" 插入完成。

b. 插入多个单词 {bear, bell}: 现在我们在已有 "bell" 的 Trie 上，再插入 "bear"

i. "b"：根节点已经有 'b' → 直接沿用;

ii. "e"：'b' 节点下已经有 'e' → 继续沿用;

iii. 接下来遇到 "a":

- 'e' 节点下原本有 'l'（来自 "bell"）;

- 'a' 还没有 → 新建 'a' 节点；

iv. "r"：在 'a' 下创建 'r' 节点；

v. 标记 'r' 为单词结束。

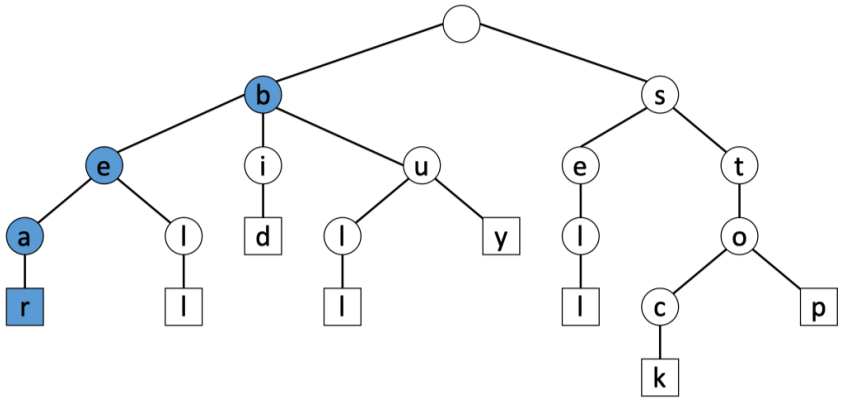
c. 重点: Trie 的高效性在于 共享公共前缀。 这让存储多个相似单词时不重复占用空间。

4. Trie: Searching

a. Trie 中已经存储了以下单词集合

{bear, bell, bid, bull, buy, sell, stock, stop}

我们要查找 "bear" 是否存在。



i. 从根节点开始。

- 当前字符：'b'
- 查找 children['b' - 'a']，找到 'b' 节点 ✅

ii. 进入 'b' 节点

- 下一个字符：'e'
- 查找 children['e' - 'a']，找到 'e' 节点 ✅

iii. 进入 'e' 节点

- 下一个字符：'a'
- 查找 children['a' - 'a']，找到 'a' 节点 ✅

iv. 进入 'a' 节点

- 下一个字符：'r'
- 查找 children['r' - 'a']，找到 'r' 节点 ✅

v. 到达 'r' 节点

- 检查该节点的 isEndOfWord 标记；
- 标记为 true → 单词 "bear" 确实存在。

b. 这次我们查找 "be"。

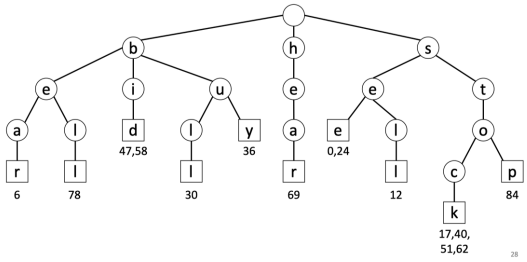
i. 从根节点 → 'b' ✅

ii. 'b' → 'e' ✅

iii. 已经走完字符串 "be"，但 'e' 的 isEndOfWord = false ✗

5. Trie: Deletion
   a. 如果 key 不存在: 直接返回，不做任何操作。例如：Trie 没有 "bids"，就不用动。
   b. 如果 key 是唯一单词
      i. 整条路径只属于这个单词；
      ii. 没有任何其他单词共享它的节点；
      iii. 那就可以安全地删掉整条路径。
      iv. 比如: T = {dog}；delete("dog") → 整个 Trie 清空。
   c. 如果 key 是其他单词的前缀: 不能删节点，只能取消“单词结束”标记。
      i. T = {bell, belly}
      ii. delete("bell") → 保留所有节点，只取消 “bell” 的结束标记。
   d. 如果 key 存在但共享前缀
      i. 单词存在；
      ii. 有其他单词共享部分前缀；
      iii. 从最后一个节点往回删，直到遇到一个仍然是其他单词路径的节点。
      iv. 比如: T = {bear, bell}；delete("bell") → 删除最后两个 "l" 节点，但保留 "be"。
6. Trie Construction from Text

s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
c	k	!		s	e	e		a		b	u	l	l	?		b	u	y	
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
s	t	o	c	k	!		b	i	d		s	t	o	c	k	!		b	i
40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
d		s	t	o	c	k	!		h	e	a	r		t	h	e		b	e
60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
l	l	?		s	t	o	p	!											
80	81	82	83	84	85	86	87	88											

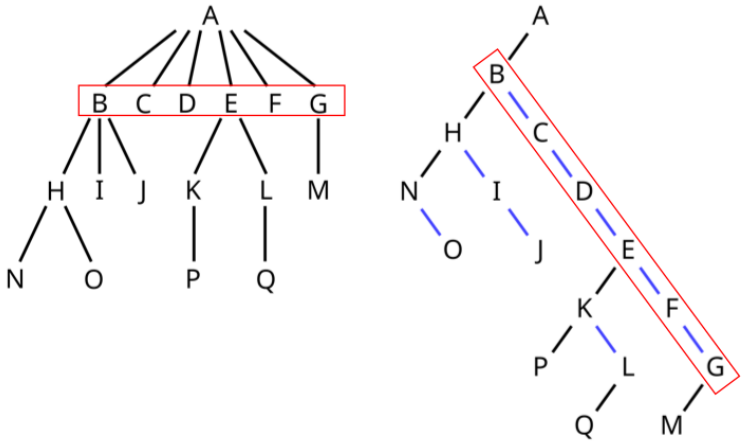


   a. Step 1: 索引。
   b. Step 2: 移除停用词（Stop Words），我们会过滤掉一些没有意义的常见词（如 a, the, of, in 等）。
   c. Step 3: 构建 Trie: 数字（如 6, 47,58, 0,24）表示该单词在原文本中的起始字符位置。这个 Trie 不只是用来“存单词”，它还记录了:
      i. 每个单词在原文中的位置；
      ii. 可以快速查找单词出现的次数和位置；
      iii. 可以做 关键词搜索、词频统计、文本索引（Text Indexing）。
7. Complexity of Standard Trie
   a. Insertion: $O(m)$, where m is the length of string s.
   b. Searching: $O(m)$, where m is the length of string s.

c. Deletion: $O(m \cdot |\Sigma|)$, where m is the length of string s and Σ is the set of alphabets.
d. Construction of the entire trie for set S: $O(n \cdot |\Sigma|)$, where n is the total length of the strings of S.
e. Space complexity: $O(n \cdot |\Sigma|)$, where n is the total length of the strings of S.

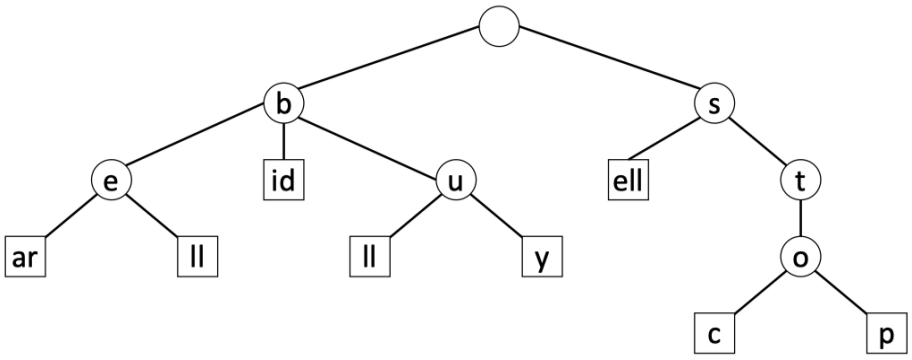
Variants of Trie

1. Left-child right-sibling binary tree



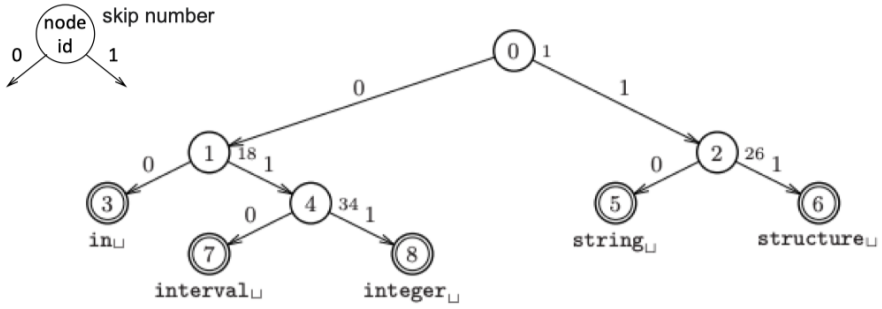
对任意多叉树节点：
- 它的 左指针（Left Child） 指向它的第一个孩子；
- 它的 右指针（Right Sibling） 指向它的下一个兄弟节点。

2. Compressed Trie



如果一条路径上每个节点都只有一个子节点（就是直下去不分叉），就把这一整条路径压缩成一个节点。

3. Patricia Tree: 它是一种 压缩的二进制 Trie
- 每个字符串被转换成二进制序列；
- Trie 的每一层表示字符串的一个 bit（0 或 1）；
- 如果路径上没有分叉（即只有一个子节点），就被压缩；
- 每个节点存储一个 “skip number”，表示跳过了多少位。



Decimal and binary ASCII codes of the symbols

		7	6	5	4	3	2	1	0
␣	32	0	0	1	0	0	0	0	0
a	97	0	1	1	0	0	0	0	1
c	99	0	1	1	0	0	0	1	1
e	101	0	1	1	0	0	1	0	1
g	103	0	1	1	0	0	1	1	1
i	105	0	1	1	0	1	0	0	1
l	108	0	1	1	0	1	1	0	0
n	110	0	1	1	0	1	1	1	0
r	114	0	1	1	1	0	0	1	0
s	115	0	1	1	1	0	0	1	1
t	116	0	1	1	1	0	1	0	0
u	117	0	1	1	1	0	1	0	1
v	118	0	1	1	1	0	1	1	0

Strings of X as sequences of bits

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	
in␣	1	**0**	0	1	0	1	1	0	0	1	1	1	0	1	1	0	0	0	**0**	0	0	1	0	0													
integer␣	1	**0**	0	1	0	1	1	0	0	1	1	1	0	1	1	0	0	0	**1**	0	1	1	1	0	1	0	1	0	0	1	1	0	0	1	**1**	0	⋯
interval␣	1	**0**	0	1	0	1	1	0	0	1	1	1	0	1	1	0	0	0	**1**	0	1	1	1	0	1	0	1	0	0	1	1	0	0	1	**0**	0	⋯
string␣	1	**1**	0	0	1	1	1	0	0	0	1	0	1	1	1	0	0	1	0	0	1	1	1	0	1	0	**0**	1	0	1	1	0	0	1	1	1	⋯
structure␣	1	**1**	0	0	1	1	1	0	0	0	1	0	1	1	1	0	0	1	0	0	1	1	1	0	1	0	**1**	0	1	1	1	0	1	1	0	0	⋯

Lecture 7: Algorithmic Paradigms

Divide & Conquer

1. Divide & Conquer 的基本概念
    a. 核心思想: 把大问题拆成结构一样但规模更小的小问题→递归地解决这些小问题→再把答案组合起来解决原问题。
        i. 大问题难解，小问题容易处理。
        ii. 把复杂度分散，通过递归让问题逐层变简单。
        iii. 子问题结构一样，可以反复套同一个方法。
    b. 三个步骤 Divide / Conquer / Combine
        i. Divide: 把问题分成若干个规模更小、类型相同的子问题。比如对 n 个元素的数组取中点，把它分成两个大小 n/2 的子数组。
        ii. Conquer: 递归地解决子问题；如果子问题已经小到不能再分（例如只剩一个元素），就直接“解”。
        iii. Combine: 把子问题的解合并成整体解。例如：两个有序数组合并成一个有序数组。
2. Divide & Conquer: Example
    a. Example 1: MergeSort
        i. Divide: 把长度 n 的数组分成两个大小为 n/2 的数组。
        ii. Conquer: 递归排序两个子数组。若子数组大小是 1，按定义它已经排好序（不需要做任何事）。
        iii. Combine: 把两个排好序的子数组用“线性扫描”方式合并成一个整体有序数组。
    b. 其他分治算法的例子
        i. Binary Search（二分查找）:每次把问题规模减半（搜索区间减半）。
        ii. Quicksort（快速排序）: 选一个 pivot，把数组分成左右两部分，然后递归排序左右部分。

Exhaustive Search

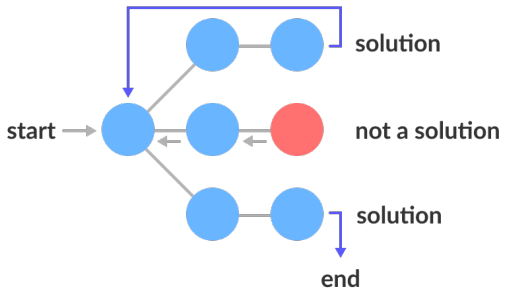
1. Exhaustive Search 的定义与特点
    a. 定义: Exhaustive Search 是一种系统地检查所有可能解或候选解的方法，以找到最优解或验证是否存在解。(把所有可能情况都试一遍。)
    b. 关键性质:
        i. Complete search: 如果解存在，它一定可以找到。
        ii. Brute-force approach: 不使用任何聪明的优化技巧，只是把所有可能性一一尝试。

iii. Unoptimized: 不剪枝、不加速，就是直接试完所有组合。
iv. Computationally expensive: 通常是指数复杂度，如: O(n!)、$O(2^n)$、$O(k^n)$ → 显示它非常慢，只适用于问题规模很小的情况。

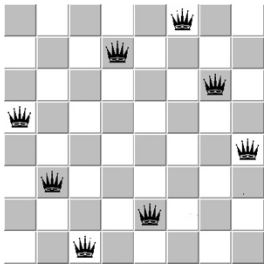
2. Password Cracking 示例: 如果不知道正确组合，又没有线索，你能做的就是把所有组合都试一遍。这就是最纯粹的 Brute Force（暴力破解）。

Backtracking

1. Backtracking 的核心: 逐步构造部分解 partial solution，一旦发现不能扩展成完整解，就立即撤销 undo 并尝试其他选择。

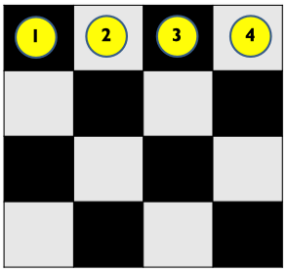


   a. Solving through "trial and error": "试错法"地解决问题。你尝试一个选择，看是否能往下走；如果不行，则回退。
   b. Iterative extension of partial solutions: 通过"逐步扩展部分解"的方式构造完整解。 例如：
      i. Sudoku: 逐格填写数字
      ii. N-Queens Problem: 逐行放皇后
      iii. Graph Coloring: 逐个节点上色
   c. If a partial solution cannot be extended, undo and try alternatives: 如果当前部分解不可能扩展成最终合法解
      i. Undo: 撤销上一步选择
      ii. Try alternatives: 换另一条路径继续尝试
2. N-Queens Problem
   a. 问题定义: 给一个 N × N 的棋盘，放置 N 个皇后（Queens），要求任意两个皇后不能互相攻击。



i. 皇后能攻击同一行、同一列、以及两个对角线方向上的任意格子。
ii. 图上展示了一个合法摆放的例子（每个皇后都不在同一行、列、或对角线上）。

b. N-Queens Problem — Exhaustive Search



i. 穷举搜索是一种“最朴素”的暴力方法：
   1. 第一位皇后：随便放在棋盘的某一个格子
   2. 第二位皇后：放在棋盘剩余的某一个格子
   3. 第三位皇后：放在剩余格子中某一个
   4. …
   5. 检查是否所有皇后互不攻击

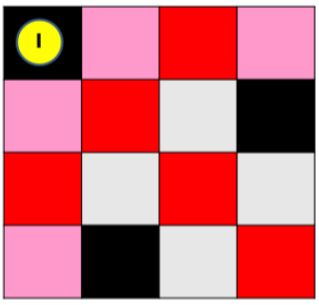
ii. 这样一共有多少可能情况？
棋盘有N^2个格子，要从中选 N 个位置摆皇后，组合数是

$$\binom{N^2}{N}$$

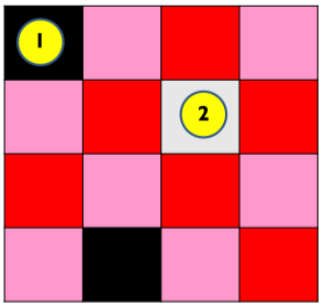
当$N = 8$的时候，$\binom{64}{8} = 4.4 \times 10^9$。四十多亿种摆法，显然就无法直接穷举了。

c. N-Queens Problem — Backtracking Example

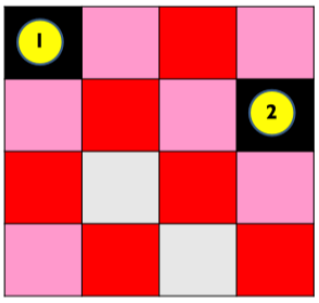
i. Step 1: The 1st queen's position. Put the 1st queen on the first cell. Mark threatened cells.



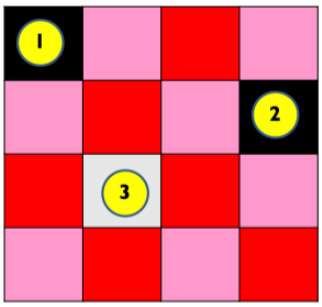
ii. Step 2: The 2nd queen's position. Put the 2nd queen on the first free cell. Mark threatened cells.
The 3rd queen's position. **Third row completely threatened.** No solution possible.
Backtracking needed.



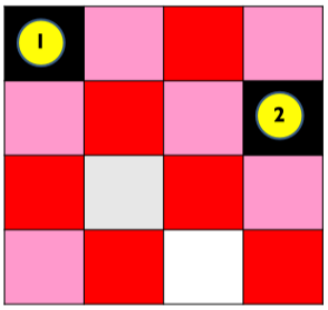
iii. Step 3: The 2nd queen's position. Put the 2nd queen on the next free cell. Mark threatened cells.

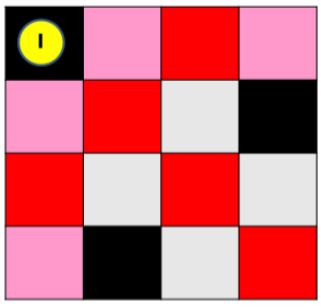


iv. Step 4: The 3rd queen's position. Put the 3rd queen on the first free cell. Mark threatened cells
The 4th queen's position. **The 4th row completely threatened.** No solution possible.
Backtracking needed.

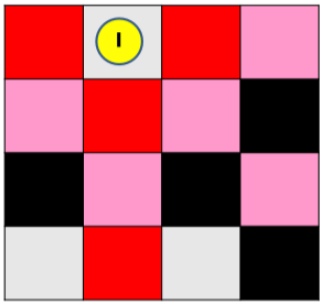


v. Step 5: **Further backtracking**

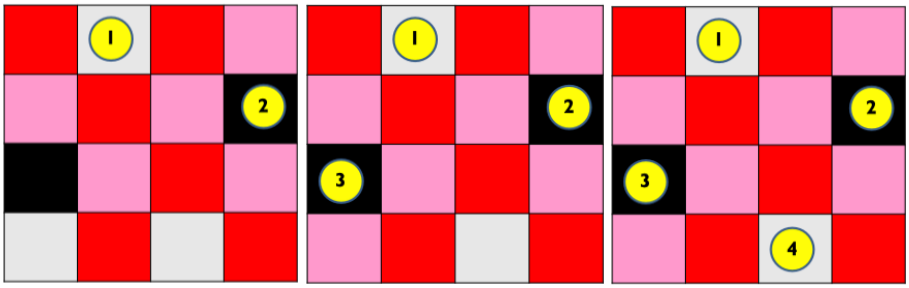




vi. Step 6: The 1st queen's position. The 1st queen on the next cell. Mark threatened cells.



vii. Step 7: Continue.



3. Advantages and Disadvantages of Backtracking
    a. Advantages
        i. 比穷举更高效
            1. 穷举会把所有可能方案都试完
            2. 回溯会在“部分解已经不合法”时立即停止，并撤销选择
            3. 这叫 剪枝 pruning
            4. 因此回溯会跳过大量不可能成功的分支，比暴力穷举快很多
        ii. 保证能找到解
            1. 回溯并不会漏掉任何可能的路径（它只剪掉“不可能成功”的路径）
            2. 只要解存在，它一定会找到
            3. 像穷举一样完整，但更聪明
        iii. 应用广泛: 回溯特别适合 Constraint Satisfaction Problems，包括
            1. 数独（Sudoku）
            2. N 皇后
            3. 图染色（Graph Coloring）
            4. 组合问题、排列问题
            5. 单词拼图、解迷宫、满足逻辑约束的问题
    b. Disadvantages
        i. 时间复杂度在最坏情况仍然可能非常大
            1. 回溯减少了很多路径，但在最坏情况下剪枝效果弱
            2. 仍然可能要探索大量搜索空间

3. 多数回溯问题最坏复杂度依然是指数级

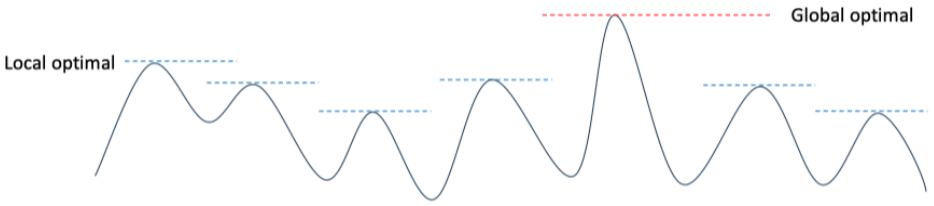
ii. 空间复杂度高（递归使用栈）

1. 回溯用递归来尝试选择
2. 深度可能达到 N 或更多
3. 每个递归层都要保存局部状态
4. 搜索树越深，占用的空间越大

Greedy Algorithms

1. Optimal Solutions
    a. Optimal/Best
        i. Optimal: The optimal solution is the one that provides the most favorable outcome according to **an objective function**.
            1. Optimal 是严格定义的，取决于问题的 objective function。
            2. Optimal 是可证明的最好的，例如最短路径、最小成本。
            3. 在算法中，最优是一个数学保证，不是主观判断。
        ii. Best: The best solution is often more **subjective or context-dependent**.
            1. Best 并不一定是数学意义上的 Optimal，而是最符合现实需求的方案。
            2. 例如: 运行时间太长 → 只能接受近似解；资源有限 → 不能求最优，只能求“够好”。
    b. Search Space: A set of all possible solutions that can be evaluated to find the optimal solution.
        i. 搜索空间就是 所有可能解的集合。
        ii. 若你想找到最优解，就必须在搜索空间中找出一个最好的。
        iii. 对某些问题，搜索空间巨大到几乎不可能穷举。比如 TSP 问题:
            1. 10 个城市: $9! = 362{,}880$
            2. 20 个城市: $19! \approx 1.216 \times 10^{17}$
            3. 无论计算机多快，也不能穷举这么大规模的搜索空间，这叫做 combinatorial explosion。
    c. Local Optimal vs. Global Optimal
        i. Optimal solutions are not always practical.
            1. Computational Complexity: 对于很多问题（如 TSP），随着规模增大，寻找 Optimal 的计算量呈指数级增长。

2. Real-World Constraints: 现实中有不确定性、不断变化的条件、不完整数据，使得 Optimal 在理论上存在，但现实中不能使用。
3. High Computational Cost for Large Problems: 即使能算出最优解，所需时间或资源也可能大到不现实。
4. Sensitivity to Parameters: 有些优化问题中，最优解对输入变化非常敏感。这意味着输入参数一点点变动，最优解可能完全不同。



ii. Global Optimal: The best solution over the entire search space.
    1. 全局最优 = 在整个解空间中价值最高（或最低）的点。
    2. 只有一个（或极少数）。
    3. 数学意义上的真正最优解。
iii. Local Optimal: Best within a limited neighborhood, but not necessarily the best overall.
    1. 局部最优是在“小范围邻域”里最好的解，但在整个空间里不一定是最优。
    2. 你当前走到的山顶可能不是最高的那一座。

2. Greedy Algorithms
    a. 核心思想: Iteratively make the locally best decision. 只看眼前，不考虑未来，不做回溯，不做全局搜索。
        i. 每一步都选择局部最优；
        ii. 不回头（irreversible choices）；
        iii. 十分快速（通常是 O(n log n) 或 O(n)）；
        iv. 但对很多问题不能保证找到全局最优解。
    b. 0/1 Knapsack Problem: 有一个背包，最多可以装 W 重量。你现在有很多物品
        i. 每个物品$o_1, o_2, o_3, \dots, o_n$
        ii. 每个物品有重量$w_1, w_2, w_3, \dots, w_n$
        iii. 每个物品有价值$v_1, v_2, v_3, \dots, v_n$

        我们的目标是，在不超过背包承重 W 的前提下，让你装进背包的东西“总价值最大”。并且 0/1 意味着，每个物品要么拿（1），要么不拿（0）。

3. 0/1 Knapsack Problem using Greedy Algorithms Example（最大承重：W = 10）

物品	1	2	3	4	5	6	7
Weight	4	4	3	1	6	2	1
Value	5	7	3	2	10	1	2
Ratio=v/w	1.25	1.75	1	2	1.67	0.5	2

Greedy Algorithms: 每一步从剩下的物品中，找 ratio 最大且能装得下的那个。

a. 第一次选择: Ratio 最大的是 4 和 7（随便选择一个，这里选 4）

Greedy	1st	2nd	3rd	4th
Object	4			
h(≤W)	1			
Cumulative value	2			

b. 第二次选择: Ratio 最大的是 7

Greedy	1st	2nd	3rd	4th
Object	4	7		
h(≤W)	1	2		
Cumulative value	2	4		

c. 第三次选择: Ratio 最大的是 2

Greedy	1st	2nd	3rd	4th
Object	4	7	2	
h(≤W)	1	2	6	
Cumulative value	2	4	11	

d. 第四次选择: Ratio 最大的是 5，但是背包只剩下 4 的重量，6 > 4。因此选择 Ratio 第二大的 1。

Greedy	1st	2nd	3rd	4th
Object	4	7	2	1
h(≤W)	1	2	6	10
Cumulative value	2	4	11	16

Text Compression

1. Text Compression 问题

a. 问题介绍: 有一个包含 100,000 个字符的文件，每个字符的出现频率已知，如何用更少的 bit 去存储它?

	a	b	c	d	e	f
Frequency	45,000	13,000	12,000	16,000	9,000	5,000

b. 传统方法: 固定长度编码 Fixed-Length Codeword

i. 定义: 如果有 n 个字符，编码每个字符至少需要

$$\lceil \log_2(n) \rceil \text{ bits.}$$

ii. 案例: 因为这边有 6 个字符 a 到 f，所以需要$\lceil \log_2(n) \rceil = \lceil \log_2(6) \rceil = \lceil 2.58496 \dots \rceil = 3$ 表示每个字符。

	a	b	c	d	e	f

Frequency	45,000	13,000	12,000	16,000	9,000	5,000
Fixed-Length Codeword	000	001	010	011	100	101

c. 问题所在: 文件长度 100,000 字符 × 每字符 3 bits = 300,000 bits。我们可以压缩的更好吗？

2. 可变长度编码 Variable-Length Codeword 与前缀码 Prefix-Free Code
   a. Variable-Length Codeword: 给每个字符分配一个不同长度的二进制串

	a	b	c	d	e	f
Frequency	45,000	13,000	12,000	16,000	9,000	5,000
Fixed-Length Codeword	000	001	010	011	100	101
Variable-Length Codeword	0	101	100	111	1101	1100

特点:

- 高频字符 a 用最短的编码 0;
- f、e 是低频，所以用了较长的 1100, 1101;
- 这是压缩的关键思想："频率高 → 编码短"。

   b. 编码案例: face 怎么编码？
      i. Fixed-Length Codeword

$$f(101) + a(000) + c(010) + e(101) \rightarrow 101000010101$$

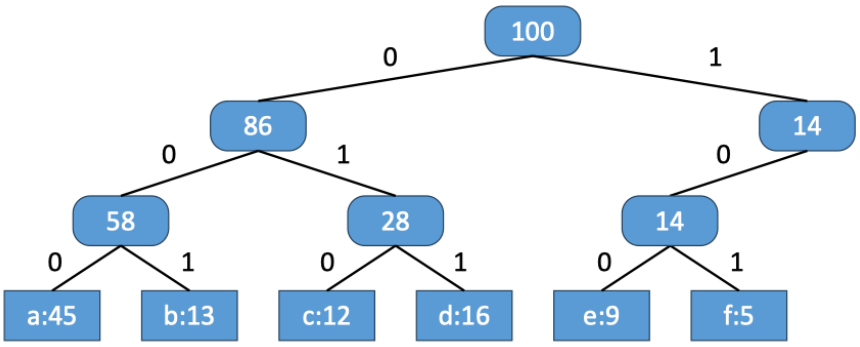
      ii. Variable-Length Codeword

$$f(1100) + a(0) + c(100) + e(1101) = 110001001101$$

   c. Prefix-Free Code（如何确保 Variable-Length Codeword 没有歧义）
      i. 问题: Variable-Length Codeword 不是固定长度，解码时你怎么知道应该从哪里切开？比如 110001001101，你怎么知道是 1100 0100 1101 还是 11 00010011 01？
      ii. Prefix-Free Code: 没有任何一个字符的编码是另一个字符编码的前缀，那么这个编码系统解码时不会产生歧义。
      iii. Variable-Length Codeword 如何解码？用 Prefix-Free Code 规则，从左到右扫描，只要匹配到一个完整编码就输出。
         1. 对 100011001101 解码:
            a. 100 → c
            b. 0 → a
            c. 1100 → f
            d. 1101 → e

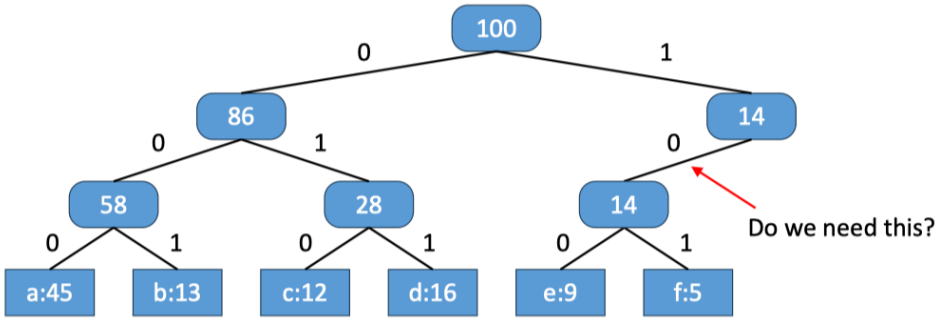
2. 得到：café

3. Prefix-free Code: Binary Tree（哈夫曼编码树 Huffman Tree）
   a. Example of Huffman Tree

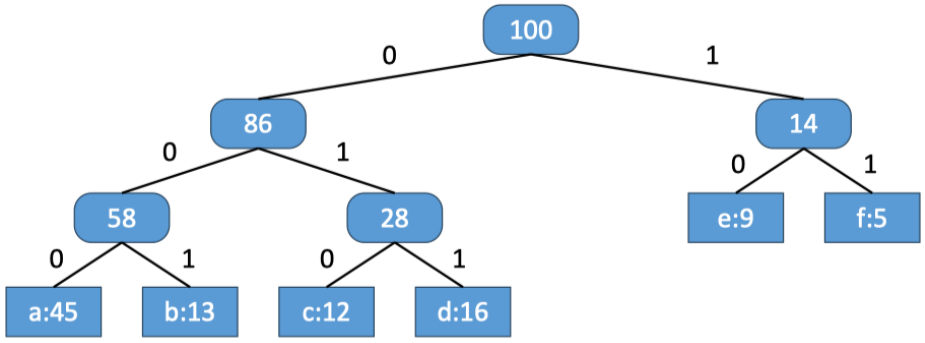


   i. 每个叶子节点代表一个字符（a, b, c, d, e, f）
   ii. 每个内部节点的数字代表其子节点频率之和（内部节点是频率）
   iii. 左边分支统一标记为 0
   iv. 右边分支统一标记为 1
   v. 从根走到某个字符叶子的“0/1 路径”就是该字符的编码。

事实上，上面的 Huffman Tree 不是 Optimal 的。



   vi. Prefix-free Code 要想达到最优压缩，树必须是 Full Binary Tree: 每个非叶子节点必须有两个孩子。
   vii. 这才是一个 Optimal Prefix-free Code



   b. 为什么它是 Prefix-free Code?

i. 因为在这棵树中，所有字符（a,b,c,d,e,f）都在叶子上，内部节点不代表任何字符，因此没有任何一个字符的编码会是另一个字符编码的前缀。

ii. 例如：

1. a = 000
2. b = 001
3. 这两个编码的关系只是“公共前缀 00”，但 b ≠ a 的扩展版本，没有编码是另一个编码完整的前缀。

c. 哈夫曼编码的核心公式

$$B(T) = \sum_{c \in C} freq(c) \cdot d_T(c)$$

i. 含义: 对每个字符 c 用它的频率乘以它的编码长度（即树的深度）总和就是压缩文件最终使用的 bit 数量。

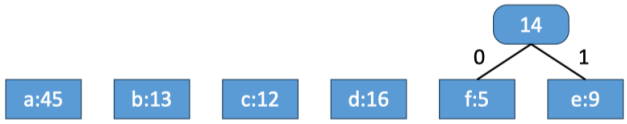
$$\text{文件所需 bit} = \sum_{\text{对每个字符}} \text{频率} \times \text{编码长度}$$

ii. 这说明，深度越浅（编码短），节省越多。这就是为什么哈夫曼算法会尽量把高频字符放在浅层。

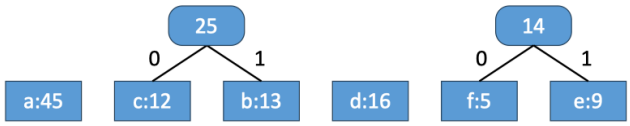
d. 如何构造 Optimal Prefix-free Code（Huffman Code: Algorithm）: 总是把当前频率最小的两个叶子合并。

i. 起点: 所有字符都是孤立的叶子。

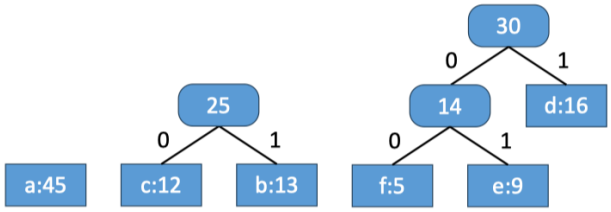
ii. Step 1: 把频率最小的两项合并，我们把 5 和 9 合并。



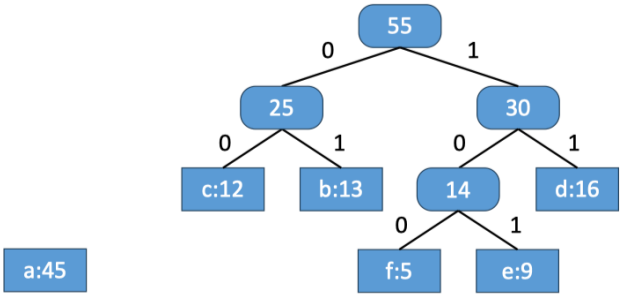
iii. Step 2: 继续合并剩下里面最小的两个，即合并 12 和 13。



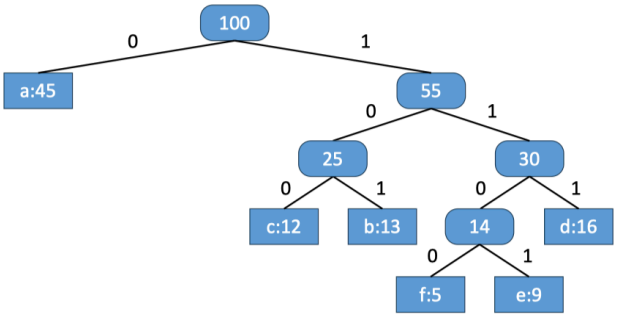
iv. Step 3: 现在最小的是 14 和 16，所以这样合并。



v. Step 4: 现在最小的是 25 和 30。



vi. Step 5: 现在最小的是 45 和 55。



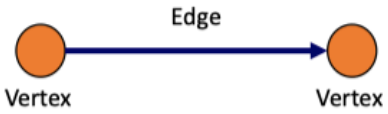
Lecture 8: Graph Algorithms

Introduction to Graphs and Networks

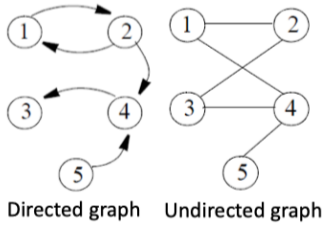
1. Definitions: Graph
   a. Graph: Set of objects (Nodes / Vertices) and relationships between objects (Edges).
      i. 顶点 Vertex / Node：图中的对象
      ii. 边 Edge / Link：对象之间的关系或连接
   b. 现实世界中的图举例

Objects	**Relationships**
Cities	Flight connection
www-pages	Hyperlinks
Social networks	Friendship

   c. Graph 的严格数学定义: A Graph G is a pair $G = (V, E)$ where
      i. V is a set of vertices. 顶点集合
      ii. E is a set of edges: $E \subseteq V \times V$. 边集合



   d. Network 是 Graph 的另一种叫法:
      i. Vertices 也叫 nodes；edges 也叫 links。
      ii. Spatial Network: 每个节点都有地理位置 a location in geo-space，边表示现实空间中的连接，例如 Road Network 是最典型的 Spatial Network:
         1. Nodes are road intersection.
         2. Edges are roads between intersections.
   e. Directed vs Undirected Graphs
      i. Directed Graph: $(v_1, v_2) \neq (v_2, v_1)$ and edges are represented by →.

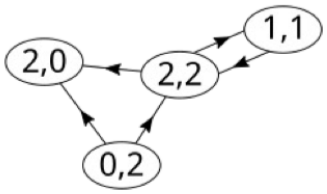


      ii. Undirected Graphs: $(v_1, v_2) = (v_2, v_1)$ and edges are represented by −.
      iii. Undirected graphs are a special case of directed graphs. So, any undirected graphs can be represented by a directed graph, not vice-versa.

1. 如何把一条无向边 {u, v} 转成有向图？把它替换成两条反向的有向边 (u, v) 和 (v, u)。

f. Subgraph and Adjacency
   i. Subgraph: A subgraph of a graph $G = (V, E)$ is a graph $G' = (V', E')$ having $V' \subseteq V$ and $E' \subseteq E$.
      1. 子图不能引入新的节点，只能从原图中选部分节点；
      2. 子图不能引入新的边，只能从原图的边中选部分边。
   ii. Adjacency
      1. Undirected graph: 只要$(v_1, v_2) \in E$，那么两个 vertices 就 adjacency。因为在 undirected graph 中，$(v_2, v_1) = (v_1, v_2)$，因此只要有边连着它们，两者就是 adjacency。
      2. Directed graph: Vertice v_1被称为 adjacency to v_2，如果 $(v_1, v_2) \in E$。这里强调 from → to，方向决定 adjacency 关系是否成立。如果存在边 A→B，那么:
         a. A adjacency to B.
         b. B NOT adjacency to A.

g. Degree
   i. Undirected graph: 某个 vertice incident（连接）多少条边，它的度就是多少。例如一个节点连了 3 条边，degree = 3。
   ii. Directed graph



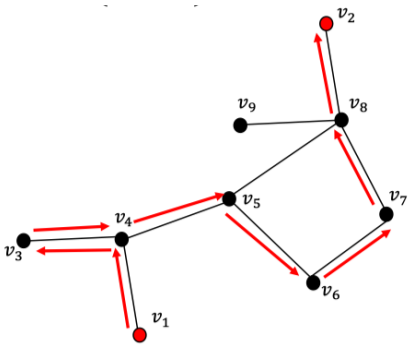
A directed graph with vertices labeled (indegree, outdegree)

      1. Indegree: 有多少条有向边终止（指向）该顶点。若有 A→X，B→X，那么 indegree(X) = 2。
      2. Outdegree: 有多少条有向边从该顶点出发。若 X→C，X→D，则 outdegree(X) = 2。

2. Definitions: Path
   a. Path of Length n: A Path of length n from vertex v_1 to vertex v_2 is a sequence of vertices $(w_0, w_1, \dots, w_n)$ having

$$w_0 = v_1, w_n = v_2, w_i \in V, 0 \le i \le n, (w_i, w_{i+1}) \in E$$

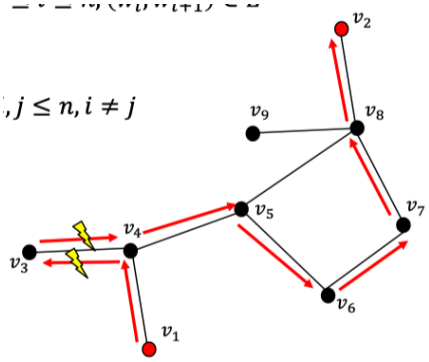
   路径 = 从起点到终点的顶点序列，相邻点必须有边连接。
   b. 路径长度 = 路径包含的边的数量。如果路径是 $(w_0, w_1, \dots, w_n)$，那么这个路径长度就是 n，因为里面有 n条边。



c. Example: A path from v_1 to v_2.

$$(v_1, v_4, v_3, v_4, v_5, v_6, v_7, v_8, v_2)$$

d. Simple Path: A Simple Path has no repeating vertices. 简单路径是不允许走回头路、不允许经过同一个节点两次的路径。

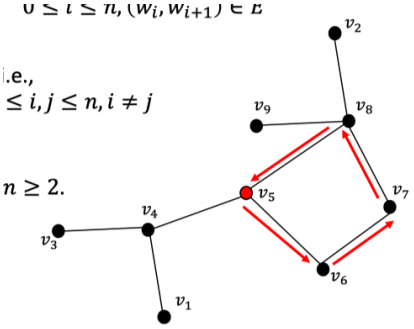


i. 在图论中，如果允许无限绕圈，路径可能无限长。

ii. 为了保证问题（如最短路径）有意义，我们通常只考虑 Simple Path，因为：

1. 重复访问节点不会得到更短的路径
2. 避免循环
3. 保证搜索空间有限

e. Simple Path 例外（Cycle）: A Cycle is a simple path **where the first and last vertices are the same**. Example cycle: (v_5, v_6, v_7, v_5).



3. Graph Connectivity

a. Undirected Graph Connectivity

i. 两个顶点 Connected: 在无向图中，如果存在从 v_i到 v_j的 path，则称
$$v_i \text{ and } v_j \text{ are connected}$$
因为边是无方向的，所以只要能走过去，一定也能走回来。

ii. Connected Graph: 如果图中任意两顶点之间都存在路径，则称无向图是 connected。也就是从图中的任何一个节点，都可以走到图中任何其他节点。

b. Directed Graph Connectivity

i. Weakly Connected: 将图中所有有向边视为无向边后，得到的无向图是 connected。（图在结构上是连通的，只是方向可能阻止你沿着图随意走动。）

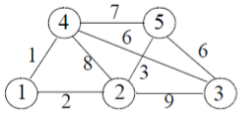
ii. Strongly Connected: 在有向图中，如果满足：

1. 从 v_i能到 v_j
2. 同时从 v_j也能到 v_i

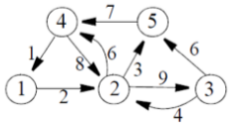
那么称 v_i和 v_j是 strongly connected。

iii. Strongly Connected Graph: 一个有向图若满足每两个顶点都 strongly connected，则整个图是 strongly connected graph。

c. Labelled Graph: Vertices and edges often have additional information → labelled graphs.



Graph G_1



Graph G_2

i. 在很多实际网络中，节点和边不只是抽象的“点”和“线”；

ii. 它们常常带有附加属性，比如距离、费用、时间、流量、容量等；

iii. 只要图中带有这些属性，我们就称其为 labelled graph。

1. Distance or travel-time: 在交通网络里，边常常表示两个地点之间的距离或开车/步行所需的时间。
2. Cost: 许多算法把边上的标号解释为 cost，比如交通运输中的费用、能耗、风险。

Graph Representation

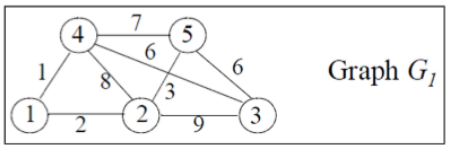
1. Adjacency Matrix

a. Adjacency Matrix 的定义: 假设有一个图 $G = (V, E)$，其中 $|V| = n$。我们可以用一个 $n \times n$的矩阵 A 来描述图中哪些顶点之间存在边。Adjacency Matrix A 的元素定义为

$$A[i,j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

也就是说如果从顶点 v_i到 v_j之间有一条边，则 A[i,j] = 1，否则为 0。

b. Adjacency Matrix Example



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	1
3	0	1	0	1	1
4	1	1	1	0	1
5	0	1	1	1	0

c. Undirect Graph 的 Adjacency Matrix 是对称的: 因为无向图里边 (i, j)与 (j, i)代表同一条边，所以矩阵天然是对称的。

$$A[i, j] = A[j, i]$$

Storage of the upper triangle matrix only.

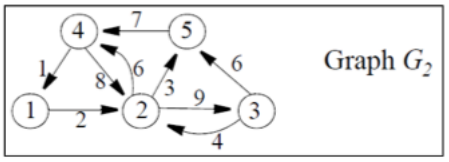
2. Cost Matrix

a. Cost Matrix 的定义: 如果图的每条边都有一个 label 表示 cost，我们用一个矩阵 A 来存这些代价，这个矩阵就叫 labelled adjacency matrix 或 cost matrix。

i. 如果两个节点之间没有边，在矩阵里用∞表示；

ii. 对角线 均为 A[i, i] = 0。

b. Cost Matrix Example



	1	2	3	4	5
1	0	2	∞	∞	∞
2	∞	0	9	6	3
3	∞	4	0	∞	6
4	1	8	∞	0	∞
5	∞	∞	∞	7	0

i. A[1,2] = 2 表示: 1 → 2 的边存在，代价为 2。

ii. A[1,3] = ∞表示: 1 → 3 没有边，无法直接到达。

iii. A[2,3] = 9 表示: 从 2 → 3 的代价是 9。

iv. A[4,1] = 1 表示: 4 → 1 边存在，代价是 1。

3. Adjacency List or Sparse Matrix Representation

a. Adjacency Matrix 的缺陷:

i. 空间复杂度太大: 邻接矩阵需要一个 n×n 的矩阵来存储边的信息，因此其空间复杂度是$O(|V|^2)$。如果 V 有百万级节点（例如真实路网、社交网络），矩阵会非常巨大，无法高效存储。

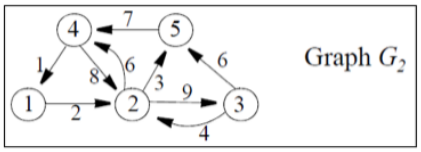
ii. 大多数应用中的矩阵非常 Sparse: 现实中的图通常不是“每个节点连接所有节点”，而是比如每个路口通常只有 3～4 条道路，这意味着

Adjacency Matrix 中绝大多数位置都是 0（或无穷，如果是 Cost Matrix）。

b. Adjacency List (AL)
   i. Adjacency List 的定义: AL 是一个数组（大小为 |V|），AL[i] 存储所有与节点 i 相连的节点。对于 Labelled Graph（带 cost），AL[i] 存储的是 (NodeID, Weight) 对，例如

$$AL[i] = [(neighbor1, cost1), (neighbor2, cost2), \dots]$$

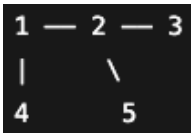
   ii. 案例 1: Labelled Graph 的 Adjacency List



Adjacency List:

[
 [(2,2)], （从 1 出发，有一条 1 → 2 边，cost = 2）
 [(3,9),(4,6),(5,3)], （从 2 出发，有三条边：2 → 3, 2 → 4, 2 → 5）
 [(2,4),(5,6)], （从 3 出发，有两条：3 → 2, 3 → 5）
 [(1,1),(2,8)], （从 4 → 1 和 4 → 2）
 [(4,7)] （从 5 → 4）
]

   iii. 案例 2: Unlabelled Graph 的 Adjacency List



AL[1] = [2, 4] （节点 1 连接到节点 2 和节点 4）
AL[2] = [1, 3, 5] （节点 2 有三条邻接边）
AL[3] = [2] （节点 3 只有一条边）
AL[4] = [1] （节点 4 只有一条边）
AL[5] = [2] （节点 5 只有一条边）

Shortest Paths – Dijkstra's Algorithm

1. Traversal Algorithms in Graph
   a. BFS in Tree
      i. 会按层（depth）逐层访问所有节点；
      ii. 因为树没有环，也没有重复路径，所以 BFS 不会遇到重复访问的问题；
      iii. 不需要记录 visited，因为一个节点永远只会被访问一次；

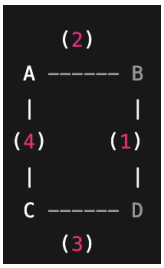
iv. 搜索总是从 root 开始。

b. BFS in Graph
   i. 同样按层访问；
   ii. 但图可能有环，或多个节点有多条路径连接；
   iii. 因此 BFS 必须记录 visited nodes，否则会在环中无限循环，或者重复遍历相同节点多次；
   iv. 起点可以是图中任意节点（因为图没有唯一根节点）。

c. DFS in Tree
   i. 沿着一个分支一直走到最深，再 backtracking；
   ii. 因为树没有环，也没有重复路径，DFS 逻辑非常清晰；
   iii. 不需要 visited（因为不可能重访节点）。

d. DFS in Graph
   i. 有环或多个路径可能通向同一节点；
   ii. DFS 必须跟踪 visited nodes 以避免重复访问和无限递归；
   iii. 搜索可以从任何节点开始，而不是固定 root。

2. Shortest-Path Search
   a. Shortest Path 的定义: The shortest path between two nodes is the path that **has the minimum total length** among all possible paths.
   b. Single-Source Shortest Path Problem 单源最短路径问题
      i. Input: Given a (**directed) graph G = (V, E)**, having **edges labelled with non-negative real-valued costs**.
      ii. Output: For a vertex find the shortest path to all other vertices in G. 不是找到一条路径，是找到所有路径。
      iii. The length of a path is the sum of costs of all its edges.
   c. Example of Single-Source Shortest Path Problem



      i. Input: 一个带权图 G=(V,E)，一个 source = A。
      ii. Output: A → B（最短路径: A → B；代价: 2）
      iii. Output: A → C（最短路径: A → C；代价: 4）
      iv. Output: A → D（最短路径: A → B → D；代价: 3）

3. Dijkstra's Algorithm Example （我们用 4 个点： A, B, C, D。A → B: 2；A → C: 5；B → C: 1；B → D: 4；C → D: 1）

a. 初始化: $A:\infty; B:\infty; C:\infty; D:\infty$。

b. Step 1: A=0 是确认的。因此为绿色（我们用绿色代表确认的距离）。此时，我们已知 A → B: 2； A → C: 5。因此$B:2; C:5$是我们目前的推测最短距离（推测最短距离用黄色表示）。

$$A:0; B:2; C:5; D:\infty$$

Shortest Path Tree: 此时我们已知 B 和 C 为 A 的孩子。

A
├── B(2)
└── C(5)

c. Step 2: 我们选出黄色里最小的数值作为新的确认距离。因此 B 确定为 2。我们已知 B → C: 1； B → D: 4。C 被更新为 3，因为 3<5；D 被更新为 6。因此，$C:3; D:6$是我们目前的推测最短距离。

$$A:0; B:2; C:3; D:6$$

Shortest Path Tree: 此时有变化，因为我们更新了 C，所以 C 变成了 B 的孩子。与此同时 D 被更新了，因此 D 也是 B 的孩子。

A
└── B (2)
    ├── C (3)
    └── D (6)

d. Step 3: 我们选出黄色里最小的数值作为新的确认距离。因此 C 确定为 3。我们已知 C → D: 1。D 被更新为 4，因为 4<6。因此，$D:4$是我们目前的推测最短距离。

$$A:0; B:2; C:3; D:4$$

Shortest Path Tree: 此时有变化，因为我们更新了 D，所以 D 变成了 C 的孩子。

A
└── B (2)
    └── C (3)
        └── D (4)

e. Step 4: 我们选出黄色里最小的数值作为新的确认距离。因此 D 确定为 4。结束。

$$A:0; B:2; C:3; D:4$$

Shortest Path Tree:

i. 到 B 的路径：A → B
ii. 到 C 的路径：A → B → C
iii. 到 D 的路径：A → B → C → D

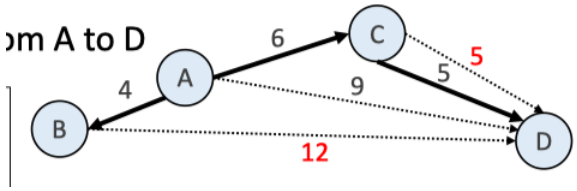
4. Dijkstra's Algorithm: Discussion
    a. Advantage
        i. Correct: Dijkstra 保证能找到从源点到所有节点的最短路径，前提是: 所有边权必须是非负的（这是 Dijkstra 的基本限制）。
        ii. Efficient: 每个节点只会被"标绿"一次（finalize 一次）。但在最坏情况下，需要维护一个包含所有未完成节点的优先队列。时间复杂度:
        $$O((|V| + |E|)\log|V|)$$
        iii. Easy to implement: 算法直观（不断选最小距离节点）；通常只需使用优先队列 + 邻接表。
        iv. Computes all shortest paths from the source: Dijkstra 默认求从源点到所有节点的最短距离。如果你只想知道从 A → 某个特定目标节点 T 的最短路径,可以在"目标节点变绿（finalized）"时提前停止，不必须跑完整个图。
    b. Disadvantage
        i. 搜索所有方向（不具有方向偏好）: Dijkstra 属于"盲搜索"，只基于当前距离最小值，不会优先朝目标方向搜索。
5. Bi-directional Dijkstra 比传统 Dijkstra 更高效
    a. 双向 Dijkstra 不是只从起点开始搜索: 它 同时从起点（绿色）与终点（红色）出发，向中间扩展，直到两边搜索的区域相遇。
        i. 从起点和终点同时扩散
        ii. 每边扩散半径只有原来的 0.5
        iii. 搜索面积只有传统的一半
        iv. 因此速度更快
    b. 传统 Dijkstra
        i. 从起点一次扩散搜索所有方向
        ii. 搜索半径大，区域大
        iii. 对稠密图计算量非常大
    c. 简洁结论: 双向 Dijkstra 更高效，因为它显著减少了搜索空间。

A*-Search

1. A* Search Purpose: Finds shortest path to a specific target node.
2. A* Search 核心: A* 会优先扩展"更接近目标的方向"的节点，如何衡量更接近目标的方向，我们使用
$$f(n) = g(n) + h(n)$$
其中
    a. g(n): 从起点到 n 的真实代价（与 Dijkstra 相同）

b. h(n): 从 n 到目标的“估计代价”（heuristic）

3. A* Search Example



比如我现在想要判断 A 到 D 的最短距离。

a. Step 1: A 访问 C，因为 11<16。
   i. f(B) = g(B) + h(B) = 4 + 12 = 16
   ii. f(C) = g(C) + h(C) = 6 + 5 = 11
b. Step 2: 访问 D。

4. Heuristic
   a. Admissibility: Heuristic 不能高估到目标的真实代价。即 h(n) ≤ 从 n 到 goal 的真实最短距离。

$$h(n) \leq \text{真实最短距离}(n, goal)$$

   b. Consistency / Monotonicity:

$$h(n) \leq cost(n, n') + h(n')$$

   从 n 直接走到 n' 再到目标，比 heuristic 估计的“直达目标的距离”不会更短。
   c. Computational Efficiency: 启发式必须快，不然每次计算 heuristic 都太慢，整体算法会变慢。

Floyd-Warshall's Algorithm

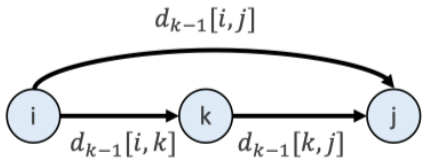
1. Floyd-Warshall 的基本概念
   a. 目标: 找出图中**任意两个节点**之间的最短路径，是一个典型的全源最短路径算法 All-Pairs Shortest Paths, APSP。
   b. 适用范围: 适用于带权图（非负或允许负权，但不能有负环）。
   c. 核心思想
      i. 你已经知道 i→j 有一条路径，是否可以通过一个新的节点 k，使得 i→k→j 更短？
      ii. 也就是我们尝试用每一个节点作为“中间节点”来改进当前已知的最短路径。
   d. 数学描述
      i. 初始化: d0[i][j] = c[i][j]
         - 初始时，最短路径矩阵 d_0 就是“直接边的权重矩阵”
         - 若 i→j 有边，则 d_0[i][j] = 权重

- 若 i→j 无边，则 $d_0[i][j] = \infty$
- 自己到自己 $d_0[i][i] = 0$

ii. 关键更新公式: 对每一个中间节点 k

$$d_k[i][j] = \min\left(d_{k-1}[i][j], d_{k-1}[i][k] + d_{k-1}[k][j]\right)$$

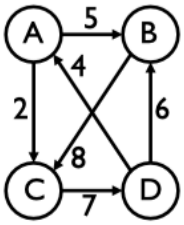
使用节点 k 作为额外可选的中间点，尝试是否能使 i 到 j 的路径变得更短。如果 i → k → j 更短，就更新。



iii. 关键技巧:

- 写出最短路径矩阵 d_0
- 固定一个 k，比如 k=A/B/C/D
- 比如 k=A，我们看“第一行 + 第一列”的交叉点（假设是 4 × 4矩阵，我们有 9 个交叉点）；每个矩阵元素是“两端之和 vs 原值”的比较；如果原值更大，就用两端之和替换。

2. Floyd-Warshall Algorithm: Example



图有 4 个点：A, B, C, D，带权有向边如下:

- A → B : 5
- A → C : 2
- B → C : 8
- C → D : 7
- D → A : 4
- D → B : 6

a. Step 1: 写出初始距离矩阵 $d^{(0)}$（也就是 k还没开始）。行、列顺序都是 A, B, C, D（行 = 起点 i，列 = 终点 j）:

$$d^{(0)} = \begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & \infty & 0 \end{pmatrix}$$

i. 对角线都是 0（自己到自己）

ii. 有边的地方是边权

iii. 没有边的是 ∞

b. Step 2: 迭代 1 k = A（问题: 经过 A 这个中间点，能否让某些 i→j 更短）

	A	B	C	D
A	0	5	2	∞
B	∞	0	8	∞
C	∞	∞	0	7
D	4	6	∞	0

因为 k = A，所以我们看第一行第一列。此时，我们有 9 个交叉点（橙色圈出的地方）。

	A	B	C	D
A	0	5	2	∞
B	∞	0	8	∞
C	∞	∞	0	7
D	4	6	∞	0

这个时候我们只需要做比对，看看“两端之和 vs 原值”的比较；如果原值更大，就用两端之和替换。

- $5 + \infty = \infty > 0$
- $5 + \infty = \infty = \infty$
- $5 + 4 = 9 > 6$
- $2 + \infty = \infty > 8$
- $2 + \infty = \infty > 0$
- $2 + 4 = 6 < \infty$（注意，此时原值更大，因此我们使用 6 替换）
- $\infty + \infty = \infty = \infty$
- $\infty + \infty = \infty > 7$
- $\infty + 4 = \infty > 0$

我们发现只有$2 + 4 = 6 < \infty$，因此使用 6 替换∞。

$$k{=}A$$
$$\begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & \mathbf{6} & 0 \end{pmatrix}$$

c. Step 3: 迭代 2 k = B（问题: 经过 B 这个中间点，能否让某些 i→j 更短）

$$k{=}A$$
$$\begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & \mathbf{6} & 0 \end{pmatrix}$$

因为 k = B，所以我们看第二行第二列。此时，我们有 9 个交叉点（橙色圈出的地方）。

经过比较，我们发现没有任何原值更大的交叉点，因此矩阵不变。

$$k=B$$
$$\begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix}$$

d. Step 4: 迭代 3 k = C（问题: 经过 C 这个中间点，能否让某些 i→j 更短）

$$k=B$$
$$\begin{pmatrix} 0 & 5 & 2 & \infty \\ \infty & 0 & 8 & \infty \\ \infty & \infty & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix}$$

因为 k = C，所以我们看第三行第三列。此时，我们有 9 个交叉点（橙色圈出的地方）。

我们发现有两处原值更大的交叉点:

- $2 + 7 = 9 < \infty$
- $8 + 7 = 15 < \infty$

因此使用 9 和 15 分别替换∞。

$$k=C$$
$$\begin{pmatrix} 0 & 5 & 2 & 9 \\ \infty & 0 & 8 & 15 \\ \infty & \infty & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix}$$

e. Step 5: 迭代 4 k = D（问题: 经过 D 这个中间点，能否让某些 i→j 更短）

$$k=C$$
$$\begin{pmatrix} 0 & 5 & 2 & 9 \\ \infty & 0 & 8 & 15 \\ \infty & \infty & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix}$$

因为 k = D，所以我们看第四行第四列。此时，我们有 9 个交叉点（橙色圈出的地方）。

我们发现有三处原值更大的交叉点:

- $4 + 15 = 19 < \infty$
- $4 + 7 = 11 < \infty$
- $6 + 7 = 13 < \infty$

因此使用 19 和 11 和 13 分别替换∞。

$$k = D$$
$$\begin{pmatrix} 0 & 5 & 2 & 9 \\ 19 & 0 & 8 & 15 \\ 11 & 13 & 0 & 7 \\ 4 & 6 & 6 & 0 \end{pmatrix}$$

f. Step 6: 更新 Predecessor Matrix

i. 先写出原始 Predecessor Matrix（Predecessor Matrix 则告诉我们“这条最短路上一条边从哪儿来”。）

	A	B	C	D
A	-	A	A	A
B	B	-	B	B
C	C	C	-	C
D	D	D	D	-

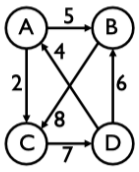
ii. 列出之前每一步的距离矩阵，找到每次更新位置的点。把 Predecessor Matrix 同样位置的元素的地方更换成对应的中间节点。

$$k=A:\begin{pmatrix}0&5&2&\infty\\ \infty&0&8&\infty\\ \infty&\infty&0&7\\ 4&6&\mathbf{6}&0\end{pmatrix}\quad k=B:\begin{pmatrix}0&5&2&\infty\\ \infty&0&8&\infty\\ \infty&\infty&0&7\\ 4&6&6&0\end{pmatrix}\quad k=C:\begin{pmatrix}0&5&2&\mathbf{9}\\ \infty&0&8&\mathbf{15}\\ \infty&\infty&0&7\\ 4&6&6&0\end{pmatrix}\quad k=D:\begin{pmatrix}0&5&2&9\\ \mathbf{19}&0&8&15\\ \mathbf{11}&\mathbf{13}&0&7\\ 4&6&6&0\end{pmatrix}$$

Predecessor Matrix

$$k=A:\begin{pmatrix}-&A&A&A\\ B&-&B&B\\ C&C&-&C\\ D&D&\mathbf{A}&-\end{pmatrix}\quad k=B:\begin{pmatrix}-&A&A&A\\ B&-&B&B\\ C&\infty&-&C\\ D&D&A&-\end{pmatrix}\quad k=C:\begin{pmatrix}-&A&A&\mathbf{C}\\ B&-&B&\mathbf{C}\\ C&C&-&C\\ D&D&A&-\end{pmatrix}\quad k=D:\begin{pmatrix}-&A&A&C\\ \mathbf{D}&-&B&C\\ \mathbf{D}&\mathbf{D}&-&C\\ D&D&A&-\end{pmatrix}$$

g. Step 7: Shortest Path Construction



Initialization:

	A	B	C	D
A	0	5	2	∞
B	∞	0	8	∞
C	∞	∞	0	7
D	4	6	∞	0

Initialization:

	A	B	C	D
A	-	A	A	A
B	B	-	B	B
C	C	C	-	C
D	D	D	D	-

Predecessor Matrix

$$\begin{pmatrix}-&A&A&C\\ D&-&B&C\\ D&D&-&C\\ D&D&A&-\end{pmatrix}$$

Origin → Destination	Backtracking using predecessor matrix	Shortest path
A → B	$p(A,B)=A$	(A, B)
B → A	$p(B,A)=D \quad p(B,D)=C \quad p(B,C)=B$	(B, C, D, A)
A → D	$p(A,D)=C \quad p(A,C)=A$	(A, C, D)
D → C	$p(D,C)=A \quad p(D,A)=D$	(D, A, C)

3. Floyd-Warshall Algorithm: Discussion
   a. Time Complexity: $O(|V|^3)$，V 是图中节点的个数。
   b. Space Complexity: $O(|V|^2)$，V 是图中节点的个数。
   c. Advantages
      i. 实现非常简单: 只需要三层嵌套循环，不需要复杂的数据结构。
      ii. 一次运行可求出所有点对的最短路径: 区别于 Dijkstra（只能求单源最短路径），Floyd-Warshall 可求任意点 i 到任意点 j 的最短路径。
      iii. 能处理负权边（但不能有负环）: Dijkstra 不能处理负权边。
   d. Disadvantages
      i. 不适合非常大的图。

ii. 对单源最短路径不够高效，如果你只想求从一个起点，到所有点的最短路径，那么 Floyd-Warshall 太慢了。

Lecture 9: Graph Algorithms (Part 2)

Topological Sorting

1. 全序 Total Order 和偏序 Partial Order
    a. Example: 一堆事情要做，哪些可以先做、哪些必须后做？比如穿衣服：
        i. 先穿 Underpants 再穿 Pants
        ii. 先穿 Pants 再系 Belt
        iii. 先穿 Shirt 再打 Tie
        iv. 先打 Tie 再穿 Jacket
        v. 先穿 Socks 再穿 Shoes

    这里每一句“先…再…”其实都是在说一个二元关系：“A 必须在 B 之前”。把所有衣物记成一个集合

    $M = \{\text{Underpants, Pants, Belt, Shirt, Tie, Socks, Shoes, Jacket, Watch}\}$

    在这个集合上，我们定义一个关系 “≤”: 写成 Underpants ≤ Pants，意思就是“穿内裤的时间不晚于穿裤子”。类似地，Socks ≤ Shoes 等等。这样，**一堆“先后约束”就是集合 M 上的一个关系 ≤**。
    b. Total Order: 所有元素都能比较。
        i. Reflexivity: 每个元素和自己比是成立的 $\forall x, x \leq x$
        ii. Transitivity: $x \leq y, y \leq z \Rightarrow x \leq z$
        iii. Antisymmetry: 如果 $x \leq y$且 $y \leq x$，那只能是 $x = y$（两个不同东西不可能互相在前）
        iv. Totality（关键）: 对任意 x, y，要么 $x \leq y$，要么 $y \leq x$。也就是说，**任意两件东西都能比较谁先谁后**。
        v. Example: 整数上的 ≤：3 ≤ 5 或 5 ≤ 3 总有一个成立；排队时人的先后：队伍里任意两个人，总有一个站在前面。
    c. Partial Order: 只要求 Reflexivity、Transitivity 和 Antisymmetry，不要求“所有人都能比”。
        i. Example: 穿衣的例子就是典型 Partial Order
            1. Socks ≤ Shoes
            2. Pants ≤ Belt
            3. Shirt ≤ Tie ≤ Jacket
        ii. 但比如: Socks 和 Shirt 根本没有约束（可以先穿袜子也可以先穿衬衫）；Watch 和大多数衣物都没有约束。
        iii. 所以: 有些对是可比较的（Socks vs Shoes）；有些对是不可比较的（Shirt vs Socks）。
2. Topological Sorting 的目标

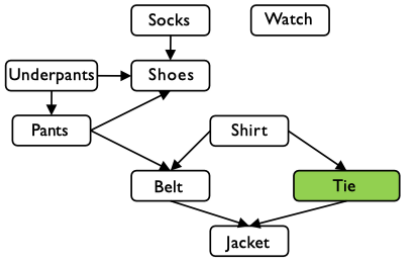
a. Topological Sorting 的目标: 已知一个 Partial Order（图给出的“先后依赖”），找到一个 Total Order（线性顺序），使得新顺序不会违反原来的所有“先后约束”。
    i. Example: 在穿衣例子中，一种合法的 Topological Sorting 可以是 Underpants, Socks, Pants, Shirt, Belt, Tie, Shoes, Jacket, Watch。我们发现所有“必须先”的关系都满足：
        1. Underpants 在 Pants 前
        2. Pants 在 Belt 前
        3. Socks 在 Shoes 前
        4. Shirt 在 Tie、Jacket 前
        5. Tie 在 Jacket 前

    但一些没有约束的对，比如 (Watch, Shoes)，在这个顺序里被“强行”排成了某个先后，这是从偏序扩展到全序所必须做的事。
b. Graph 的视角
    i. Directed Graph for Partial Order: 为了更方便处理这些关系，我们用 Directed Graph 来表达 Partial Order。
        1. 顶点: 每个元素（一件衣物、一个课程、一个任务……）。
        2. 有向边 u → v: 表示 “u 必须在 v 前面” 或 “u ⩽ v”。
    ii. Directed Acyclic Graph (DAG): 如果这个 Directed Graph 没有 cycle，就叫 DAG（Directed Acyclic Graph）。Topological Sorting 只在 DAG 上有意义，原因很简单:
        1. 比如对于 A → B → C → A。如果 A 要在 B 前，B 要在 C 前，C 又要在 A 前，那你再怎么排一条直线都满足不了，必然有人不满足要求。
        2. 因此，如果图里有环，就不存在 Topological Sorting；如果图是 DAG，就一定有至少一个 Topological Sorting。
c. Topological Sorting 的正式定义: 在有向图 G = (V, E) 上，一个序列

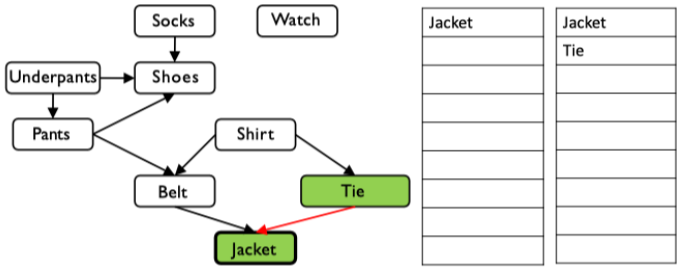
$$(v_{i_1}, v_{i_2}, \dots, v_{i_n})$$

是一个 Topological Sorting，当且仅当: 对于图中每条有向边 $(u, v) \in E$，u 在这个序列中总是出现在 v 的前面。
    i. Topological Sorting 本身是一个 total / linear order。
    ii. 把所有点排成一排；
    iii. 任何一条箭头都必须从左指向右；
    iv. 没有任何箭头是从右指向左的。
d. Topological Sorting 问题: 给定一个有向图，输出这样一个 Topological Sorting。

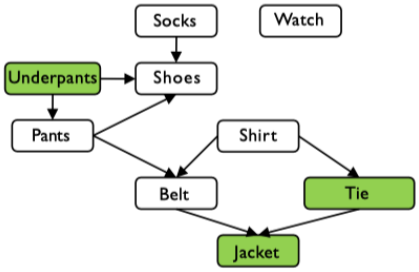
3. 方法一: DFS-based Approach
    a. 方法原则
        i. 任选一个节点开始。
        ii. 没有后继 → 立即开始回溯 → 回溯时将此节点加入列表。
        iii. 一个节点的所有后继完成后，该节点也要在回溯时加入列表。
    b. Example
        i. Step 1: 任选一个 node，比如 Tie。



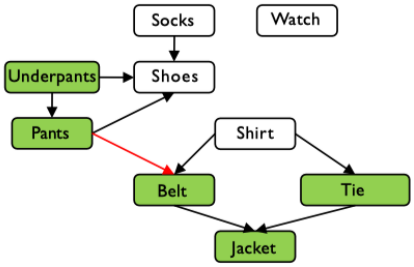
        ii. Step 2: 访问 Tie 的后续节点 Jacket。发现 Jacket 没有后继，立即开始回溯，回溯时将此节点加入列表。由于 Tie 的后继都已经完成，因此该节点也要加入列表。



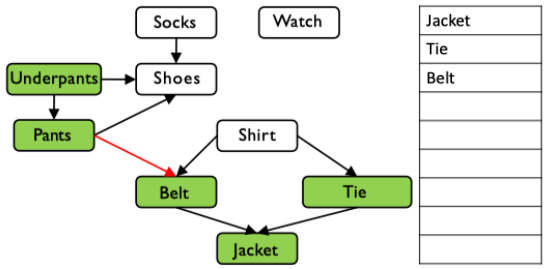
        iii. Step 3: 任选一个没有被访问过的新 node。



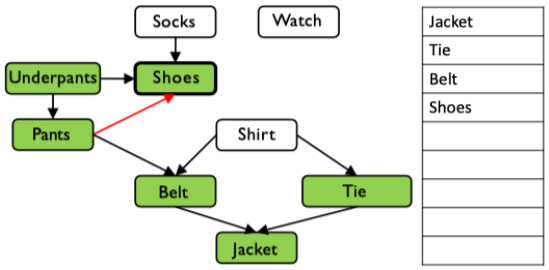
        iv. Step 4: 逐一访问其所有后继节点。



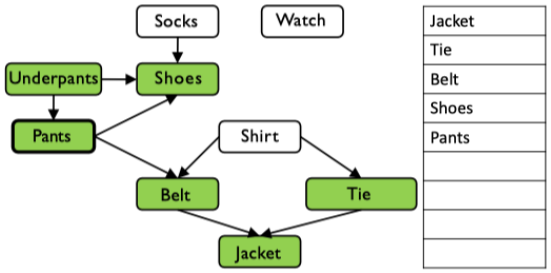
v. Step 5: 发现 Belt 的后继节点都已经完成，因此开始回溯，并把 Belt 加入列表。



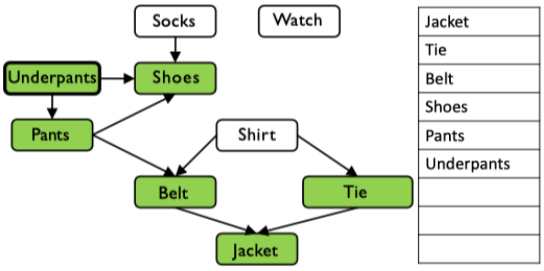
vi. Step 6: 回溯到 Pants 发现 Pants 还有一个后继节点，继续访问其后继节点，即 Shoes。并且发现 Shoes 没有后继节点，因此开始回溯，并把 Shoes 加入列表。



vii. Step 7: 此时 Pants 的后继节点都已经完成，因此开始回溯，并把 Pants 加入列表。



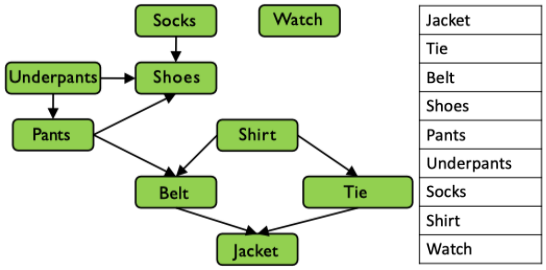
viii. Step 8: 此时 Underpants 的后继节点都已经完成，因此开始回溯，并把 Underpants 加入列表。



ix. Step 9: 任选剩余未访问节点，发现

1. Socks 的后继节点都已经完成，因此开始回溯，并把 Socks 加入列表。

2. Shirt 的后继节点都已经完成，因此开始回溯，并把 Shirt 加入列表。
3. Watch 没有后继节点，把 Watch 加入列表。



c. Time Complexity
   i. Using adjacency list

$$O(|V| + |E|)$$

   ii. Using adjacency matrix

$$O(|V|^2)$$

   iii. 含义
      1. |V|: 图中的顶点数量 number of vertices
      2. |E|: 图中的边数量 number of directed edges

4. 方法二: Kahn's Algorithm
   a. 核心逻辑: Kahn 算法就是优先取入度为 0 的节点；加入序列后就删除该节点，这会导致有其他节点的入度减少。不断重复，直到所有节点都被选出。
   b. Example:

$$A \rightarrow B \rightarrow C$$
$$A \rightarrow D$$

      i. Step 1: 计算初始入度
         1. A:0
         2. B:1
         3. C:1
         4. D:1
      ii. Step 2: A 入度为 0，A 加入序列，B 和 D 的入度-1。

$$[A]$$

         1. B:0
         2. C:1
         3. D:0
      iii. Step 3: B/D 入度为 0，任取一个，比如 B。B 加入序列，C 的入度-1。

$$[A, B]$$

         1. C:0

2. D:0

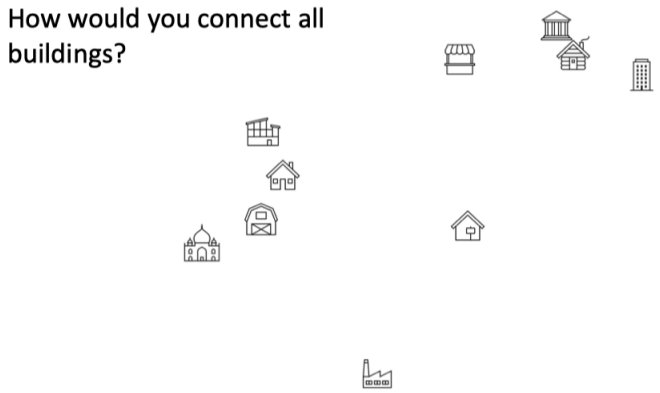
iv. Step 4: C/D 入度为 0，任取一个，比如 D。D 没有出边，那么我们继续取 C，结束。

$$[A, B, D, C]$$

5. Topological Sorting: Applications
   a. Application 1: 进程/任务的并行化 Parallelization of Processes
      i. Serialization Graph: 序列化图描述哪些操作必须在其他操作之前完成。拓扑排序作用是据依赖关系得到一个合法顺序，或者发现哪些任务可以并行执行。
      ii. 拓扑排序给你一个“全局合法顺序”，但你也能从中看出哪些可以并行。
   b. Application 2: 检测图中的环 Detecting cycles in graphs。这是因为拓扑排序只有在 DAG（有向无环图）中才可能成功。

Minimal Spanning Trees

1. Network Design and Minimum Spanning Trees (MSTs)
   a. Network Design



      i. 图中有许多建筑（房子、商店、工厂、银行…），每个都是一个 node。
      ii. 隐含问题:
         1. 我们想把所有建筑连通
         2. 例如铺设电缆、电网、道路、通信线路
         3. 我们不希望浪费资源
         4. 我们不希望造太多冗余线路
      iii. 于是我们
         1. 抽象 nodes（把现实问题 → 抽象为图论问题）。
         2. 加入所有可能连线（complete graph）。
         3. 加入 weights

iv. 我们的目标是

1. 既能连通所有点
2. 又没有环（最省边数）
3. 又总成本最少

b. Minimum Spanning Trees

i. 数学定义: 给定一个带权无向图

$$G = (V, E)$$

权重函数$c: E \to \mathbb{R}^+$，寻找满足下面条件的 subgraph $G' = (V, E')$

1. 边集 E′是原边集 E 的子集；
2. $G' = (V, E')$是连通且无环的（G'是一颗 tree）；
3. 所有生成树中，E'的总成本最小

$$\sum_{e \in E'} c(e) \leq \sum_{e \in E''} c(e)$$

ii. 总而言之，现实网络设计会遇到“怎样以最小成本连通所有节点”的问题。将其抽象成图论后，这个问题的解就是 Minimum Spanning Tree（MST）。

2. 方法一: Prim’s Algorithm

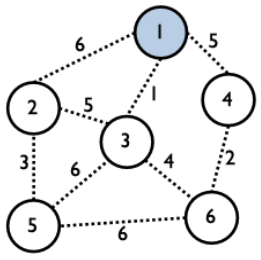
a. 核心逻辑

i. 首先任选一个 node 作为起点；

ii. 每一步都选择: 连接“已选节点集合 S″和“未选节点集合 V−S（也就是我们选择的 node 之外的剩下的 node）″之间的、代价最小的边；

iii. 每一步都扩展当前的生成树，使它更大。

b. Example

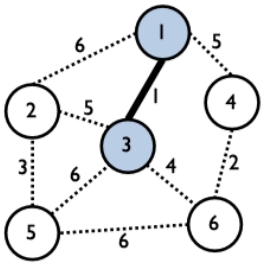


i. Step 1: 我们选择 1 作为起点。

$$S = \{1\}$$

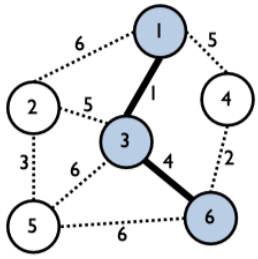
ii. Step 2: 1 的三条出边中权重最低的是 1，其连接 3，因此我们选择 3 加入集合 S。

$$S = \{1,3\}$$



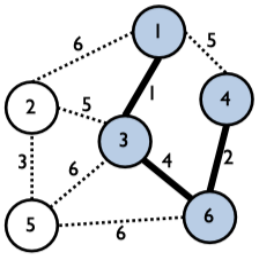
iii. Step 3: 1 和 3 剩下的 5 条出边中权重最低的是 4，其从 3 连接 6，因此我们选择 6 加入集合 S。

$$S = \{1,3,6\}$$



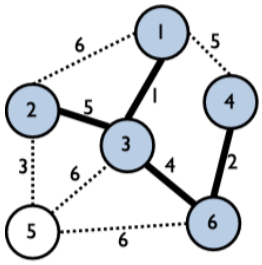
iv. Step 4: 1、3 和 6 剩下的 6 条出边中权重最低的是 2，其从 6 连接 4，因此我们选择 4 加入集合 S。

$$S = \{1,3,6,4\}$$



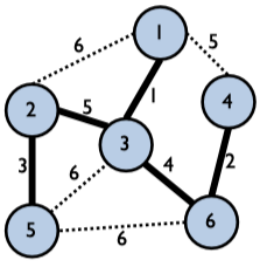
v. Step 5: 1、3、6 和 4 剩下的 6 条出边中有 2 个权重最低的，分别是 3 连接 2（权重为 5）和 4 连接 1（权重为 5）。但是，不能选择 4 连接 1，因为这样就形成环了。因此我们选择 2 加入集合 S。

$$S = \{1,3,6,4,2\}$$



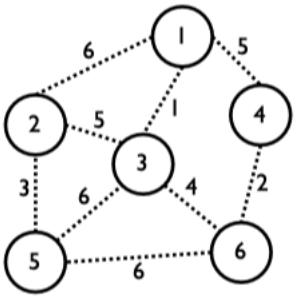
vi. Step 6: 1、3、6、4 和 2 剩下的出边中权重最低的是 2 到 5（权重为 3）。因此我们选择 5 加入集合 S。

$$S = \{1,3,6,4,2,5\}$$



c. Prim's Algorithm: Complexity
   i. General: $O(|V|^2)$。V 是顶点数量。
   ii. Using min-heap: $O(|V|\log|V| + |E|\log|V|)$。V 是顶点数量；E 是边的数量。

3. 方法二: Kruskal's Algorithm
   a. 核心逻辑
      i. 首先写出所有边的权重，按照从小到大排列；
      ii. 从最便宜的边开始考虑；
         1. 如果加入该边不会与已有的边形成 cycle → 加入；
         2. 如果会造成环 → 跳过它。
      iii. 继续检查下一条边。
   b. Example

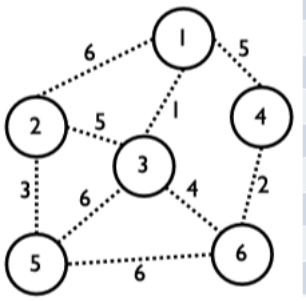


E
(1,3) → 1
(4,6) → 2
(2,5) → 3
(3,6) → 4
(1,4) → 5
(2,3) → 5
(1,2) → 6
(3,5) → 6
(5,6) → 6

      i. Step 1: 选择(1,3)。
      ii. Step 2: 选择(4,6)。
      iii. Step 3: 选择(2,5)。
      iv. Step 4: 选择(3,6)。
      v. Step 5: 跳过(1,4)，不然出现 cycle。
      vi. Step 6: 选择(2,3)。
      vii. Step 7: 跳过(1,2)，不然出现 cycle。
      viii. Step 8: 跳过(3,5)，不然出现 cycle。
      ix. Step 9: 跳过(5,6)，不然出现 cycle。
   c. 问题: 我们如何判断会不会出现 cycle?

i. Disjoint Set: A Disjoint Set (also known as Union-Find) is a data structure that is used to manage and track a collection of disjoint sets. It supports the following operations efficiently:
    1. Make-Set(x): Creates a new set whose only member (and thus representative) is x.
    2. Union(x, y): Merge two sets that contain x and y into one.
    3. Find-Set(x): Find the representative of the set that contains x.

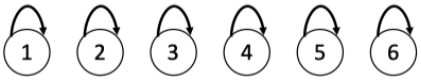
ii. Disjoint Set for Kruskal's Algorithm



E
(1,3) → 1
(4,6) → 2
(2,5) → 3
(3,6) → 4
(1,4) → 5
(2,3) → 5
(1,2) → 6
(3,5) → 6
(5,6) → 6

1. Step 1: 使用 Make-Set(x)，为每一个 node 创建一个独立集合。

{1}{2}{3}{4}{5}{6}

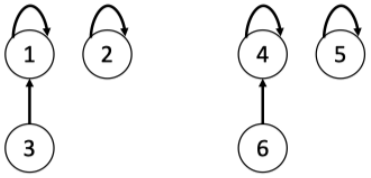


2. Step 2: 对于我们选择的每一条边，执行该操作
    a. If Find-Set(u) ≠ Find-Set(v), $A = A \cup \{(u, v)\}$，也就是合并该边进入目前的集合。
    b. If Find-Set(u) = Find-Set(v), no need to union，因为出现了 cycle。
3. Step 2.1: 选择(1,3)，Find-Set(1) = 1, Find-Set(3) = 3，因此合并集合

$$MST = \emptyset \cup \{(1,3)\} = \{(1,3)\}$$
{1,3}{2}{4}{5}{6}

4. Step 2.2: 选择(4,6)，Find-Set(4) = 4, Find-Set(6) = 6，因此合并集合

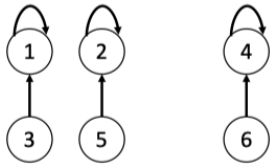
$$MST = \{(1,3)\} \cup \{(4,6)\} = \{(1,3), (4,6)\}$$
{1,3}{4,6}{2}{5}



5. Step 2.3: 选择(2,5)，Find-Set(2) = 2, Find-Set(5) = 5，因此合并集合

$$MST = \{(1,3),(4,6)\} \cup \{(2,5)\} = \{(1,3),(4,6),(2,5)\}$$

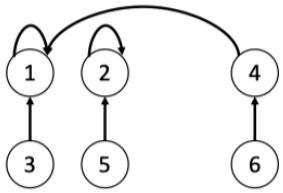
{1,3}{4,6}{2,5}



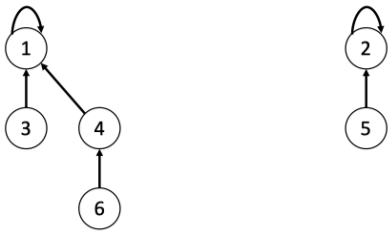
6. Step 2.4: 选择(3,6)，Find-Set(3) = 1, Find-Set(6) = 4，因此合并集合

$$MST = \{(1,3),(4,6),(2,5)\} \cup \{(3,6)\}$$
$$= \{(1,3),(4,6),(2,5),(3,6)\}$$

{1,3,4,6}{2,5}



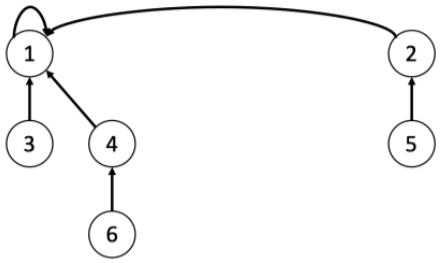
把 1 和 4 连接在一起是因为 Find-Set(3) = 1, Find-Set(6) = 4，我们要连接的是 Find-Set 的数值对应的 node



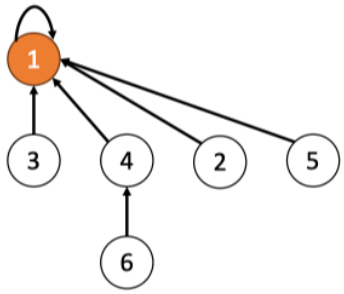
7. Step 2.5: 选择(1,4)，Find-Set(1) = 1, Find-Set(4) = 1，因此不能合并，会成 cycle，跳过。
8. Step 2.6: 选择(2,3)，Find-Set(2) = 2, Find-Set(3) = 1，因此合并集合

$$MST = \{(1,3),(4,6),(2,5),(3,6)\} \cup \{(2,3)\}$$
$$= \{(1,3),(4,6),(2,5),(3,6),(2,3)\}$$

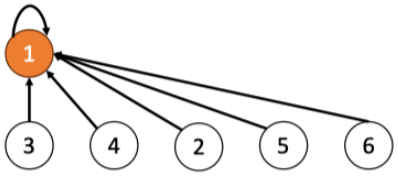
{1,3,4,6,2,5}



9. Step 2.7: 只有一个集合了，因此结束。



10. Step 2.8: 对于并查集 parent 指针树，我们进行 path compression，也就是把所有 node 连接到根节点下面。



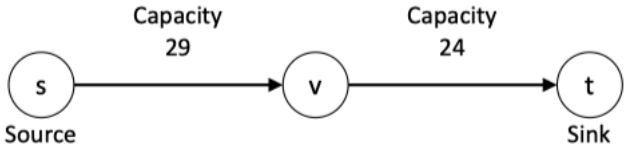
Path compression

d. Kruskal's Algorithm: Complexity
   i. Time Complexity: $O(|E|\log|E|)$，E 是边的数量。
   ii. Kruskal 是 Optimal: Kruskal 每次选的最小边不会破坏最优性，因此最终得到的树一定是 MST。

Lecture 10: Graph Algorithms (Part 3)

Flow Network

1. Flow Network



   a. 直观问题
      i. 给你两根串联的水管，它们的容量分别是 29 和 24，最大能输出多少水流量？
      ii. 直观答案：最大水流 = 两条管道中较小的容量 = 24
   b. Graph Theory 模型: 我们可以把现实世界中的“管道系统”模型化为一个 directed graph。
   c. Flow Network 的正式定义: 一个 Flow Network 是$G = (V, E)$，其中每条边(u, v)都有 nonnegative capacity $c(u, v) \geq 0$。并且我们定义一个 Flow Function $f(u, v)$，其含义是从 u 流到 v 目前的流量。Flow Function 需要满足下面 2 个 properties:
      i. Capacity Constraint: Flow Function 的值，也就是这条边的流量不能超过这条边的总 capacity，也不能是负数。
      $$0 \leq f(u, v) \leq c(u, v)$$
      ii. Flow Conservation: 对所有中间节点 u（除了源点 s 和汇点 t），其流入的总流量和流出的总流量需要相等。也就是对任意一个中间节点 u，其 incoming edges 的总流量等于 outgoing edges 的总流量。
      $$\sum_{v} f(v, u) = \sum_{v} f(u, v)$$
2. Maximum-Flow Problem
   a. Maximum-Flow Problem 的正式定义: 在一个流网络 G 中，找从 source node s 到 sink node t 能送出的最大流量，其需要满足 3 个数学条件
      i. Capacity Constraint: 和之前一样。
      $$0 \leq f(u, v) \leq c(u, v)$$
      ii. Flow Conservation: 和之前一样。
      $$\sum_{v} f(v, u) = \sum_{v} f(u, v)$$
      iii. Non-Negativity: 之前也提到过，每条边的流量不得是负数。
      $$0 \leq f(u, v)$$
   b. Maximum-Flow Problem 的 Objective Function: 最大化流量

$$\text{Maximize: } \sum_{v \in V} f(s,v) \text{ or } \sum_{v \in V} f(v,t)$$

这两个表达式的含义是从 s 流出的总流量或流入 t 的总流量。这两个表达式都可以作为目标函数，因为 s 流出的总流量 = 流入 t 的总流量。

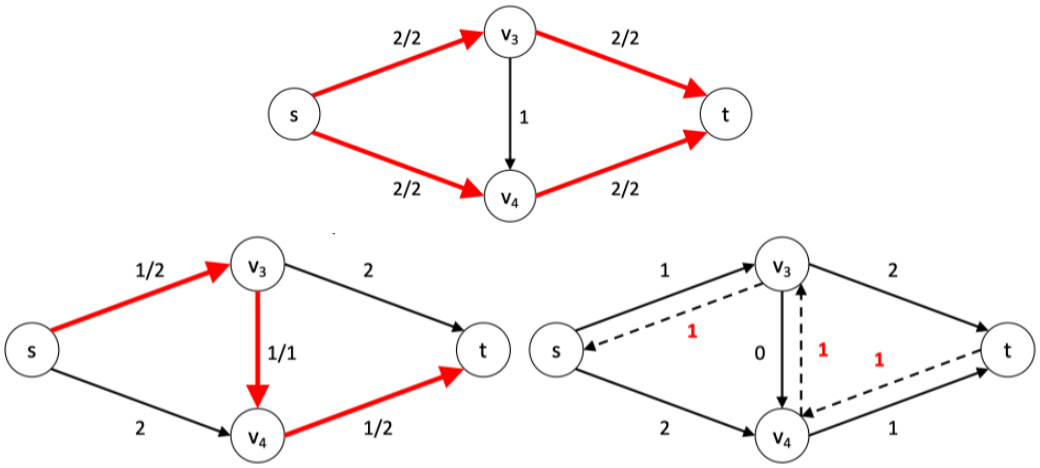
c. Maximum-Flow Problem Formulation as Linear Programming.

Ford-Fulkerson Method and Residual Network

1. Residual Network
   a. Residual Network: The residual capacity of an edge is the remaining capacity after considering the flow already sent through that edge.

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v), & \text{若原图有 } u \to v \\ f(v,u), & \text{若原图有 } v \to u \\ 0, & \text{其他情况} \end{cases}$$

      i. 正向边还能推多少（capacity − flow）；
      ii. 反向边能退多少（flow）。
   b. Residual Network 的目的: Residual Network 是 Ford-Fulkerson Method 的核心，其帮助我们找最大流量。
   c. Residual Network Example



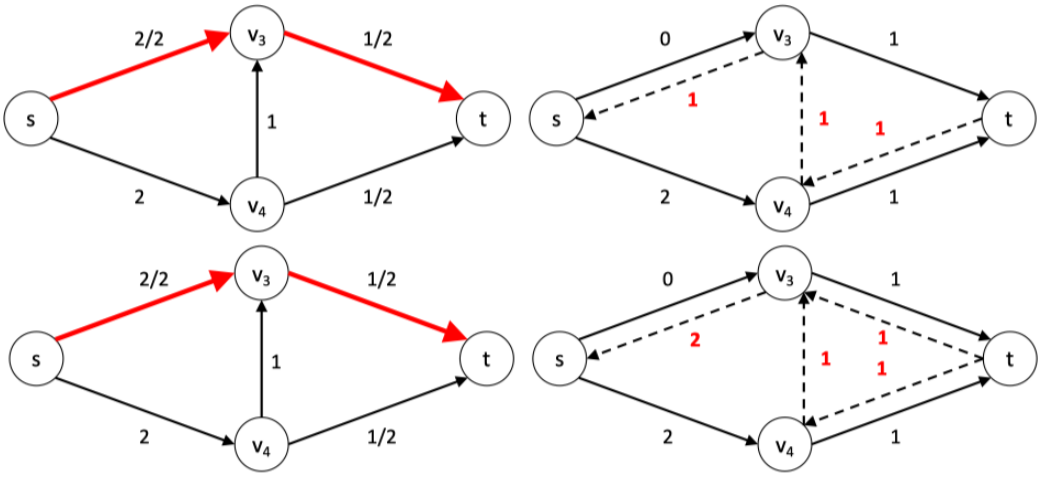
我们现在选择一条 path: $s \to v_3 \to v_4 \to t$

      i. 左边图像: 原图
         1. 对于选择的 path，我们用

$$Flow/Capacity \ (or \ f(u,v)/c(u,v))$$

         来表示其流量情况。
         2. 我们发现由于$s \to v_3 \to v_4 \to t$的最大流量应该是 1，因此我们写出 1/2，1/2 和 1/2，用来表示$Flow/Capacity$。
      ii. 右边图像: Residual Network

1. 实线: 正向边还能推多少，也就是 capacity – flow，因此我们标出 1，0 和 1。
2. 虚线: 反向边能退多少，也就是刚才实际走过的流量，这里都是 1。



我们现在选择一条新的 path: $s \to v_3 \to t$

iii. 左边图像: 原图
   1. 我们使用$Flow/Capacity$ notation，可以发现$s \to v_3 \to t$的最大流量是 1，因此我们用完了$s \to v_3$的所有流量，标注 2/2；$v_3 \to t$的流量情况是 1/2。

iv. 右边图像: Residual Network
   1. 实线: 正向边还能推多少，也就是 capacity – flow。这里$s \to v_3$的 capacity 是 1，flow 是 1，因此标注 0；$v_3 \to t$的 capacity 是 2，flow 是 1，因此标注 1。
   2. 虚线: 反向边能退多少，也就是刚才实际走过的流量。$v_3 \to s$原本标注了 1，这里再加上 1，一共是 2；而$t \to v_3$是 1。

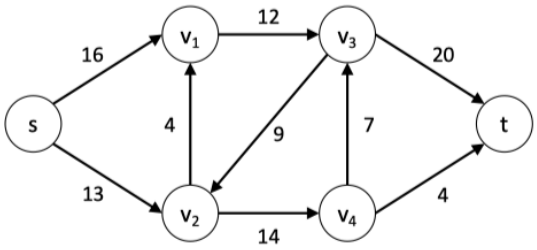
2. Ford-Fulkerson Method
   a. Augmenting Path: 从 s（源点）到 t（汇点）的一条路径，而且这条路径上的所有边的残量容量都 > 0，也就是我们之前一直在选择的 path。
   b. Ford-Fulkerson 的五个步骤
      i. Initialization: 把所有边的初始流量设为 0（我们只有原图，还没有画出 Residual Network）。
      ii. Find Augmenting Path: 用 BFS、DFS 或任何方法，从 s 到 t，找到一条可以走的路径。
      iii. Augment Flow
         1. 看路径上每条边的残量，如果是第一个路径就看其 capacity
         2. 找一个最小的残量（这叫 bottleneck）

3. 沿着整条路径推这个瓶颈值的流量

iv. Update Residual Graph: Residual Network 必须实时更新，以保持“还能走哪些路”。

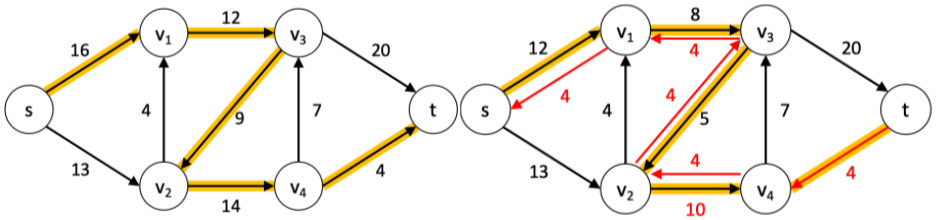
v. Repeat: 重复直到找不到 Augmenting Path。

c. Ford-Fulkerson Method Example



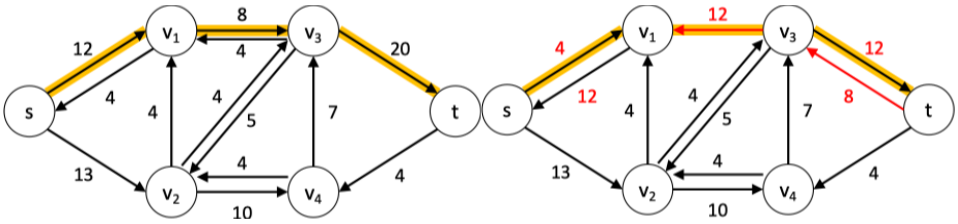
i. 第一轮: Augmenting Path $(s, v_1, v_3, v_2, v_4, t)$

这是第一个路径，因此我们看其 capacity，最小 capacity 是 4，因此 flow 为 4。我们依据此更新 Residual Graph。

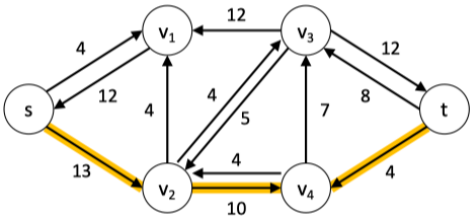


ii. 第二轮: Augmenting Path (s, v_1, v_3, t)

这是第二个路径，最小残量是 8，因此 flow 为 8。我们依据此更新 Residual Graph。

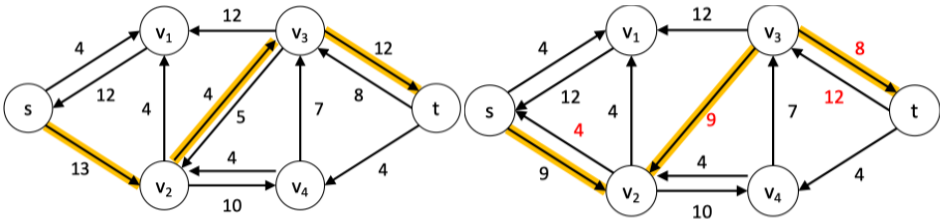


iii. 第三轮: 我们选择了错误的 Augmenting Path，请注意(s, v_2, v_4, t)并不是一个 Augmenting Path。



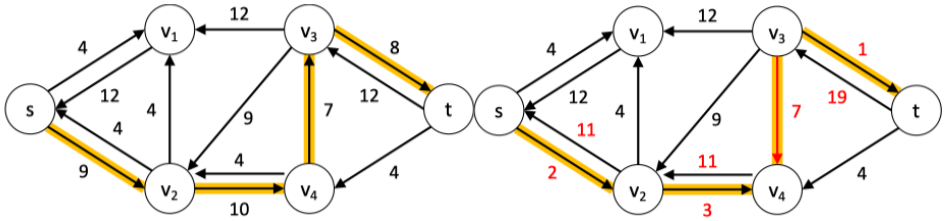
iv. 第四轮: Augmenting Path (s, v_2, v_3, t)

这是第四个路径，最小残量是 4，因此 flow 为 4。我们依据此更新 Residual Graph。

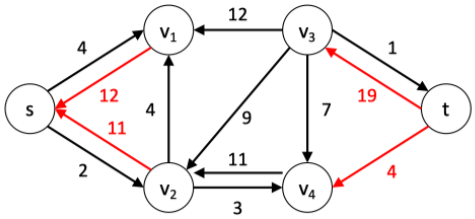


v. 第五轮: Augmenting Path (s, v_2, v_4, v_3, t)

这是第五个路径，最小残量是 7，因此 flow 为 7。我们依据此更新 Residual Graph。



vi. No more augmenting paths. Done!



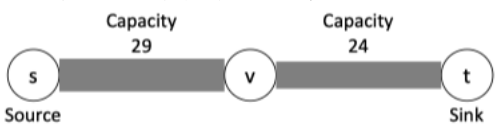
我们可以看到

$$\sum_{v \in V} f(s,v) = 12 + 11 = 23 \text{ or } \sum_{v \in V} f(v,t) = 19 + 4 = 23$$

3. Edmonds-Karp Algorithm: By using breadth-first search to find an augmenting path in the residual network, the algorithm runs in polynomial time, independent of the maximum flow value.
    a. 普通 Ford-Fulkerson
        i. 允许你随便选任何 augmenting path
        ii. 正确，但最坏情况可能需要 非常非常多 次扩增（甚至指数级）
    b. Edmonds-Karp 的核心改变
        i. 强制用 BFS 找最短路径（按边数）作为 augmenting path
        ii. 因此每次选的路径都是 离源点 s 最近的通往 t 的路径

Max-Flow Min-Cut Theorem

1. Flow Network: Bottleneck
    a. Bottleneck: 最大流由“最细的那段管道”决定。



水要从 s 流到 t，必须经过这两段管道。就像水管串联时，整条水管能通过的水量由最细的一段决定—不可能比最细段的容量大。因此，最大流量为

$$\text{maxflow} = \min(29,24) = 24$$

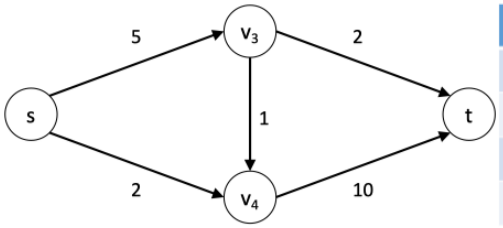
b. 如何提高最大流？
   i. 最大流取决于路径上最小容量的边，这条边叫“bottleneck edge”。
   ii. 因此:
      1. 加宽非瓶颈地方无效果
      2. 要提高最大流，必须加宽瓶颈

2. Flow Network: Cut
   a. Cut 直观解释: 用一把“刀”把所有点分成两堆
      i. 左边这堆叫 S
      ii. 右边这堆叫 T

      要求：s 必须在 S 里，t 必须在 T 里。刀从中间切过去，把图分成 “s 这一侧” 和 “t 这一侧”。
   b. Cut 容量$c(S,T)$的数学定义

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

      也就是看所有“从 S 指向 T 的边”，把这些边的容量加起来，就是这个 cut 的容量。
   c. Example of Cut



S	T	$c(S,T)$
$\{s\}$	$\{v_3, v_4, t\}$	5+2=7
$\{s, v_3\}$	$\{v_4, t\}$	2+1+2=5
$\{s, v_4\}$	$\{v_3, t\}$	5+10=15
$\{s, v_3, v_4\}$	$\{t\}$	2+10=12

   d. Minimum Cut: 最小割上的这些边的容量之和 = 最大流，它们就是整个网络的“真正瓶颈”。如上述案例所示，minimum cut 为 5。

3. Max-Flow Min-Cut Theorem: 如果 f 是一个流，G 是网络，源点 s，汇点 t，那么下面三件事是等价的（说的是同一个状态）
   a. f 是一个 maximum flow
   b. 在 f 对应的 residual network 里没有 augmenting paths
   c. 存在某个 cut (S,T)，使得

$$|f| = c(S,T)$$

      也就是“流量值 = 某个割的容量”。（Maximum Flow = Minimum Cut）

这说明: 只要跑完 Ford-Fulkerson，没有 augmenting paths 了，而且你找到一个 cut 的容量正好等于当前 flow，那就可以 100%确定已经达到 maximum flow。