CSCI 3160 算法设计与分析

> Design and Analysis of Algorithms

①②③④⑤⑥⑦⑧⑨⑩⑪⑫⑬⑭⑮⑯⑰⑱⑲⑳㉑㉒㉓㉔㉕㉖㉗㉘㉙㉚㉛㉜㉝㉞㉟㊱㊲㊳㊴㊵㊶㊷㊸㊹㊺㊻㊼㊽㊾㊿

Week 1

Lec 0 Logistics

本课研究算法，而非编程.

课程网站:

https://xiao-liang.github.io/Resources/Courses/CSCI3160-25Fall/CSCI3160-25Fall.html

不计 attendance.

3 Quizzes：24%，课堂进行.

Midterm：26%，Oct. 22 (Wed)，课堂进行.

Final：50%

Lec 1 RAM 计算模型

> the RAM Computation Model

RAM（Random Access Machine / 随机存储机）：A machine has a memory and a CPU.

1.1 内存

> Memory

An infinite sequence of cells, each of which contains the same number w of bits.

Each cell has an address: the first cell of memory has address 1, the second cell 2, and so on.

> w 是字长（word length），定义了内存单元和 CPU 寄存器的大小.

w 位组成的序列被称为一个字（word）.

w 在不同计算机中可能不同. 现代计算机通常是 32 位或 64 位.

内存地址是为了在海量内存单元中定位数据，而 CPU 的寄存器数量很少（本课 8 个），CPU 通过指令集名称/编号直接引用某个寄存器，不需要地址. 此外，访问寄存器中的整数/访问寄存器中存放的内存地址都可视作瞬间完成，是原子操作的一部分，而不是原子操作本身. 作为对比，访问内存中的数据（包括）则视为一个原子操作.

注意：RAM 模型的内存并不是指电脑上的 C/D/E 盘（这些是**硬盘**，C/D/E 是一块硬盘上虚拟划分出的逻辑区域，称为**分区**或**卷**）. RAM 模型说的 Memory 是 Random Access Memory，对应电脑的**内存条**（运行内存）. CPU 访问内存的速度要远大于访问硬盘的速度. 准确地说，CPU 不会跳过 RAM 直接访问硬盘，所有数据必须先通过 I/O 总线加载到主内存（RAM），CPU 再从 RAM 中读取和处理.

学习算法时通常不关注 w 的具体数值，在大多数算法分析中，我们只需要知道它是一个**存在且一致**的值即可. 因为 RAM 模型旨在提供一个足够通用（即 w 可根据实际调整）和灵敏的抽象，它捕捉了现代计算机处理器的核心特性：

- 统一的单元大小 —— CPU 的寄存器和内存单元都具有相同的大小 w（因为要互相 overwrite，所以要对齐）.
- 常数时间操作：CPU 的所有原子操作（如加减乘除、比较、内存访问）都假设在 w 位的数字上可以在**一个时间单位内**完成.

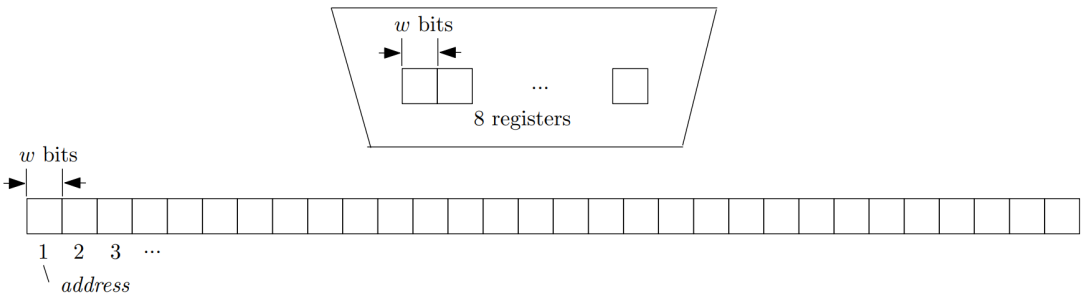
1.2 CPU

> Central Processing Unit

Contains a fixed number —— 8 in this course —— of registers, each of which has w bits (same as a memory cell).

实际的个人电脑 CPU 中，寄存器的数量远不止个位数，并且分为多种类型. 本课程为简化算法分析设定为 8 个.

> 注意区分寄存器（register）和内存单元（memory cell）.



> 这幅图中，上图是 CPU 内的 8 个寄存器，下图是 RAM 的若干个（远大于 8 个）内存单元. 但同一个模型 / 同一台电脑中，寄存器和内存单元的字长 w 相同.

1.2.1 原子操作

> Atomic Operations
>
> 注意：原子操作是 RAM 模型的理想化概念，实际电脑要考虑更多因素.
>
> 这里具体某个行为消耗 2 个、3 个还是 4 个原子操作不重要，因为算法分析

CPU 可以执行以下原子操作，每个操作的执行时间计为一个单位：

① 寄存器初始化

> Register (Re-)Initialization

将寄存器设置为固定值（如 $0, -1, 100$）或另一个寄存器的内容.

`寄存器初始化` 视为一个原子操作，强调赋值. 在其他原子操作中也有 “把数据写入寄存器” 的过程，但它们不强调赋值而是强调其他动作，因此在其他原子操作中 “把数据写入寄存器” 不视为一个完整的原子操作. 注意区分.

② 算数运算

> Arithmetic

读取两个寄存器中的整数 a, b，计算 $a + b$ 或 $a - b$ 或 $a \cdot b$ 或 a/b，并将结果存入一个寄存器.

> 注意，这里 a/b 是整数除法，即多数编程语言中使用的 “向 0 取整”（忽略余数）.
>
> 例：$6/3 = 2$，$5/3 = 1$.

`算数运算` 强调 “计算” 的动作，这里将结果存入寄存器就不视为单独的原子操作.

③ 比较/分支

> Comparison/Branching

读取两个寄存器中的整数 a, b，比较它们，and learn which of the following is true:

$a < b, a = b, a > b$.

这里强调 “比较” 的动作.

“分支”（Branching）指程序根据某个条件选择执行不同的代码路径. 最常见的就是高级编程语言中的条件语句（if/else）.

在 RAM 模型中，“比较” 的结果作为 “分支” 的决策依据. 分支利用比较的结果决定程序指针（下一条指令的地址）应该跳转到哪里.

在 RAM 模型中，假设可以在一个时间单位内完成 “读取-比较-根据结果跳转” 整个过程. 即整个过程视为一个原子操作.

④ 内存访问

> Memory Access
>
> 注意是访问内存，不是访问寄存器，访问寄存器快得多，即原子操作中的 “读取” 动作.

Take a memory address A currently stored in a register. Do one of the following:

- Read the content of the memory cell with address A into a designated register (overwriting the bits there).

  > 包括 “读取地址-把地址对应内容写入寄存器” 全过程.

- Write the content of a designated register into the memory cell with address A (overwriting the bits there).

  > 包括 “读取地址-把寄存器内容写入”

注意，如果我的内容占 $2w$ 位，即使内容是连续的（在内存中连续存放，可以只通过**一个**起始地址 A 来定位），也必须视为两份，每份 w 位，先读取第一个 w 位到第一个寄存器，再读取第二个 w 位到第二个寄存器. 需要 2 个原子操作.

`内存访问` 视为一个原子操作，强调数据在内存单元和寄存器之间的传输（而非读取寄存器中的内存地址 / 把内容写入寄存器）. 每传输 w 位就是一个原子操作，而不是看读了几个地址（因为连续存放的数据可能只需要一个起始地址）. 并且 "把 w 位数据写入一个寄存器" 在这里也不像 `寄存器初始化` 那样被单独视为一个原子操作.

⑤ 随机数生成

> Randomness

`RANDOM(x,y)` : Given integers x and y (satisfying $x \leq y$), this operation returns an integer chosen uniformly at random in $[x, y]$, and places the random integer in a register.

`随机数生成` 强调 “生成” 的动作.

1.3 RAM 模型

> The Random Access Machine (RAM) model

An execution is a sequence of atomic operations.

Its cost (also called its running time, or simply, time) is the length of the sequence, namely, the number of atomic operations.

A word is a sequence of w bits, where w is called the word length.

> In other words. each memory cell and CPU register store a word.
>
> 本课不关心 w 的具体值.

1.4 算法

> Algorithm

An input refers to the initial state of the registers and the memory before an execution starts.

An algorithm is a piece of description that ,given an input, can be utilized to unambiguously (明确地) produce a sequence of atomic operations, namely, the execution of the algorithm.

In other words, it should be always clear that what the next atomic operation should be, given the outcome of all the previous atomic operations.

> 这两段话强调了算法的两个核心要求：明确性和可预测性.

The cost of an algorithm on an input is the length of its execution on that input (i.e., the number of atomic operations required).

> 对应时间复杂度.

The space of an algorithm on an input is the largest memory address accessed by the algorithm's execution on that input.

> 对应空间复杂度.

1.5 确定性算法 vs. 随机化算法

> Deterministic Algorithms vs. Random Algorithms

1.5.1 确定性算法

> Deterministic Algorithms

确定性算法：从不调用原子操作 `RANDOM`. 其 cost 是一个固定的整数 —— it remains the same every time you execute the algorithm.

1.5.2 随机化算法

> Randomized Algorithms

随机化算法：调用 `RANDOM`. The cost of a randomized algorithm is a random variable. Even on the same input, the cost may change each time the algorithm is executed.

例：

```
1. do
2. r = RANDOM(0, 1)
3. until r = 1
```

我们不知道 Line 2 会执行多少次. Every time the above algorithm is executed, it may produce a new sequence of atomic operations.

① 预期成本

> Expected Cost of a Randomized Algorithm

Let X be a random variable that equals the cost of an algorithm on an input. The expected cost of the algorithm on the input is the expectation of X.

不同的分布（方差很大的和方差很小的）可能有同样的期望，那为什么预期成本仍然是分析随机算法的好指标?

- 单一数值，且比其他统计量更好算、更容易分析.
- 线性 —— 可以把复杂算法分解成小部分，求和它们各自的预期成本.
- 当 cost 方差很大时，可以用马尔可夫不等式（Markov inequality）、切比雪夫不等式（Chebyshev inequality）、切尔诺夫界（Chernoff bounds）等 Concentration Bounds 来避免预期成本带来的误导.

② 马尔可夫不等式

> Markov's Inequality
>
> 见 `大二 term 1 ESTR 2018 概率论 13.5 马尔可夫不等式`.

Markov's Inequality provides an upper bound on the probability that a **non-negative** random variable is much larger than its expectation.

> 注意，这里的非负很好匹配了 cost，因为 cost 本来就非负.

Statement

Let X be a non-negative random variable and $a > 0$. Then

$$Pr[X \geq a] \leq \frac{\mathbb{E}[X]}{a}$$

这意味着，if the average value (expecation) of X is small, the chance that X is very large must be small.

例：Suppose $\mathbb{E}[X] = 5$. Then,

$$Pr[X \geq 25] \leq \frac{\mathbb{E}[X]}{25} = 0.2$$

So there's at most a 20% change that X exceeds 25.

把 Markov's Inequality 应用到随机化算法的 cost analysis:

Let T be the the random variable for the running time of a randomized algorithm.

Suppose:

- $T \geq 0$;
- $\mathbb{E}[T] = \mu$.

Then for any $c > 1$:

$$Pr[T \geq c\mu] \leq \frac{1}{c}$$

> cost 大于等于 c 倍期望的概率小于等于 $\frac{1}{c}$.

If $\mu = 10$, then

$$Pr[T \geq 100] \leq \frac{1}{10}$$

③ 拉斯维加斯算法

> Example: Las Vegas Algorithm

定义：总是给出正确结果，但运行时间是随机的算法.

应用：若算法预期运行时间为 μ，设置一个 $c \cdot \mu$ 的超时限制.

> 超时则返回失败.

- 超时概率 $Pr[\text{timeout}] \leq \frac{1}{c}$.
- 例，设置 $c = 3$，失败（超时）概率 $\leq \frac{1}{3}$.
- 重复执行 3 次，至少成功一次的概率 $\geq 1 - \left(\frac{1}{3}\right)^3 \approx 0.963$.

Lec 2 最差输入

> Measuring the Efficiency of an Algorithm by the Worst Input

Computer Science 的一个重要部分就是找到 RAM 模型解决每个实际问题的最快算法.

至于如何定义 “快”，One appoach is to look at the algorithm's cost on the worst input.

2.1 基本概念

问题规模（Problem Size）：用整数 n 表示.

> 某个特定问题的输入规模，如对 n 个整数进行处理.

输入集合 $\mathcal{I}_n$：所有具有相同问题规模 n 的输入的集合.

算法成本 $X_A(I)$：Given an input $I \in \mathcal{I}_n$, the cost $X_A(I)$ of an algorithm A is the length of its execution on I.

- 最差情况成本：The worst-case cost of A under the problem size n is the maximum $X_A(I)$ of all $I \in \mathcal{I}_n$.
- 最差预期成本：The worst-expected cost of A under the problem size n is the maximum $\mathbb{E}[X_A(I)]$ of all $I \in \mathcal{I}_n$.

  > 主要用来衡量随机化算法. 因为随机化算法的最差情况成本可能是 ∞，用最差预期成本能更好反映算法实际性能.

对于确定性算法，最差预期成本等于最差情况成本. 因为确定性算法对给定输入 I 的执行过程是唯一的，所以对任意输入 I，确定性算法的预期成本都等于算法成本. 即

$$\mathbb{E}[X_A(I)] = X_A(I)$$

for a Deterministic Algorithm A and all $I \in \mathcal{I}_n$.

2.2 最差情况分析

例 1 - 字典搜索问题

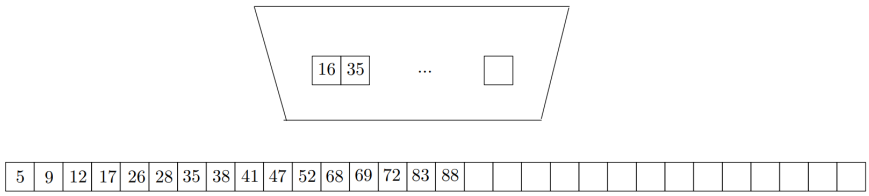
> Example: Dictionary Search

问题：在一个按升序排列的 n 个整数的集合 S 中，确定整数 v 是否存在.

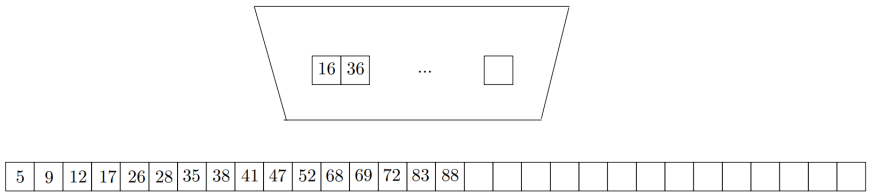
- n is the problem size.
- $\mathcal{I}_n$ is the set of all possible (S, v).

Problem Input: In the memory, a set S of n integers have been arranged in ascending order at the memory cells from address 1 to n. The value of n has been placed in Register 1 of the CPU. Another integer v has been placed in Register 2 of the CPU.

A "yes"-input with $n = 16$



A "no"-input with $n = 16$



这个问题可以用多种算法解决. 这里考虑二分搜索算法:

The worst-case cost of the binary search algorithm is $O(\log n)$.

In other words, on any input in $\mathcal{I}_n$, the maximum number $f(n)$ of atomic operations performed by the algorithm grows no faster than $\log_2 n$.

> 注意，this does **NOT** mean $f(n) = \log_2 n$.
>
> "$f(n) = O(\log n)$" only says that $f(n)$ could be functions like $10(1 + \log_2 n)$, $352 \log_3 n$, $\sqrt{\log n} + 78 \log_2(n^{83})$
>
> 见 `Tut 1 渐近分析` .

例 2 - 查 0 问题（一）

问题：在大小为 n 且至少包含一个 0 的数组 A 中找到一个 0 的位置.

考虑一个随机化算法：重复随机选择一个索引 r（$r = RANDOM(1, n)$），直到 $A[r] = 0$ 为止.

```
1. do
2. r = RANDOM(1, n)
3. until A[r] = 0
4. return r
```

不同输入，预期成本不同:

- 如果数组 A 的所有数都是 0，成本为 $O(1)$.
- 如果数组 A 中只有一个 0，预期成本为 $O(n)$. 因为每次随机选择到 0 的概率是 $\frac{1}{n}$，平均需要重复 n 次才能找到.

  > 证明：设找到时重复次数为 X. $X \sim Geometric(\frac{1}{n})$.

> $\mathbb{E}[X] = \frac{1}{p} = n.$
>
> 详细证明见 `大二 term 1 ESTR 2018 概率论 10.6 几何分布的期望`.

正因为现实问题可能有不同输入，所以我们通常考虑最差情况：

- Worst-case cost of the algorithm $= \infty$.

  > 随机的，可能永远抽不到.

- Worst expected-cost of the algorithm $= O(n)$.

  > 即数组中只有一个 0 的情况.

在本题中，我们设计的随机化算法无论是 Worst-case cost 还是 Worst expected-cost 都无法超越确定性算法中的线性扫描（也是 $O(n)$）.

> 注意，字典搜索问题中，数组是有序的，可以使用二分搜索；但查 0 问题中，数组是无序的，不能仅仅通过检查中间元素来排除左半部分或右半部分，二分搜索对这个问题无效.

例 3 - 查 0 问题（二）

Problem "Find-a-Zero": Let A be an array of n integers, among which half of them are 0. Design an algorithm to report an arbitrary position of A that contains a 0.

> 问题：在大小为 n 且一半为 0 的数组 A 中找到一个 0 的位置.
>
> 与例 2 的区别是，例 2 是 “至少一个 0”，这题是 "一半为 0".

For example, suppose $A = (9, 18, 0, 0, 15, 0, 33, 0)$. An algorithm can report $3, 4, 6$ or 8.

> 只要报一个就行，不用全报. 即报查到的第一个.

考虑一个随机化算法：同例 2.

```
1. do
2. r = RANDOM(1, n)
3. until A[r] = 0
4. return r
```

最差预期成本：$O(1)$.

事实上，该算法在所有输入 A 上，预期成本都是 $O(1)$.

> 证明：设找到时重复次数为 X. $X \sim Geometric(\frac{1}{2})$.
>
> $\mathbb{E}[X] = \frac{1}{p} = 2.$

在这道题中，即使最好的确定性算法，最差情况下也必须耗费 $\Theta(n)$ 的时间（包括 worst-case cost 和 worst-expected cost）.

证明：对抗论证（Adversary Argument）

考虑某个最优的确定性算法. 由于它是确定性的，它会根据目前为止所知道的所有信息，确定性地决定下一步要查看哪个位置 i.

对抗者（Adversary，输入构造者）在算法作出决定前，偷看 i 的位置. 为了最大化算法的运行时间， 只要将 $A[i]$ 设为非零仍然满足 "数组中有一半是 0" 的约束，那么对抗者就立即将 $A[i]$ 设为非零.

> 常见误区：确定性算法对不同的输入可能给出不同的查询顺序，因此对抗者根据当前输入构造的新输入，该算法在跑新输入时可能会改变查找顺序，这个新输入不一定够坏.
>
> 正确理解：逐步构造 $A[i]$. 例如，对于任意输入，确定性算法的第一步查询位置都是相同的（因为算法没有任何信息）. 对抗者把该位置构造为非零数（其他数未知）. 然后，对于符合该构造的任意输入，确定性算法的第二步查询位置也都相同（因为算法在查询完第一步后，得到的信息都是同一个非零数，即信息完全一致）. 那么，对抗者把第二步查询位置也构造为非零数...依次递推，直到对抗者用完 $\frac{n}{2}$ 个非零数，就构造出了最坏的 A（其余位置都是 0）.

对抗者会一直设置非零数，直到它用尽了所有允许设置非零数的位置（本题为 $\frac{n}{2}$ 个）. 这样就构造出了最坏的 A，即使最优的确定性算法也要查询 $\frac{n}{2}$ 次.

换句话说，在这道题中，我们设计的随机化算法可证明在最差预期成本上比任何确定性算法都快（不过最差情况成本还是不如确定性算法）. 这意味着**随机化在特定问题中可能超越确定性方法**.

Lec 3 递归 / 重复 / 几何级数

> Basic Techniques: Recursion, Repeating, and Geometric Series

本节主要介绍算法设计中的三种基本技术：递归 (Recursion)、几何级数 (Geometric Series) 和重复直至成功 (Repeating till Success).

介绍两个著名问题：

Hanoi Tower - recursion

k-selection - geometric series & repeating till success

3.1 递归

> Recursion

核心思想：当遇到一个子问题（与原问题相同，但输入规模更小）时，假设子问题已**解决**，并用子问题的输出来继续算法设计.

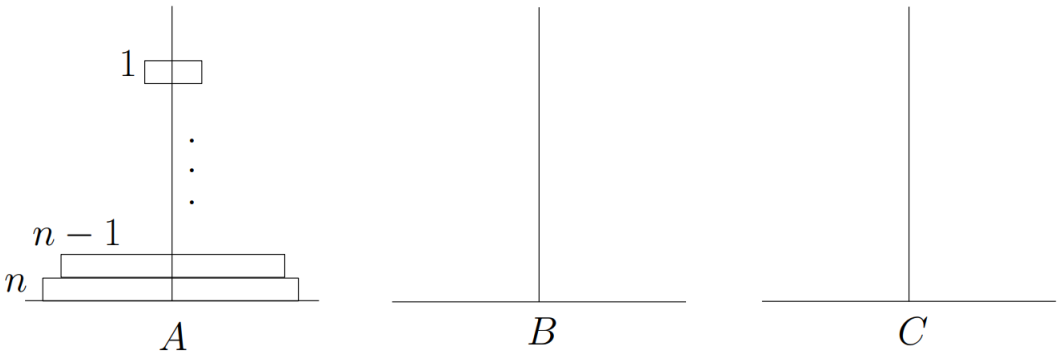
> When dealing with a subproblem (same problem but with a smaller input), consider it solved, and use the subproblem's output to continue the algorithm design.

3.1.1 汉诺塔问题

> Hanoi Tower
>
> rod：杆，棍子，柱子（pillar）
>
> disk：圆盘

例：汉诺塔问题

There are 3 rods A, B and C.



On rod A, n disks of different sizes are stacked in such a way that no disk of a larger size is above a disk of a smaller size.

The other two rods are empty.

Permitted operation: Move the top-most disk of a rod to another rod.

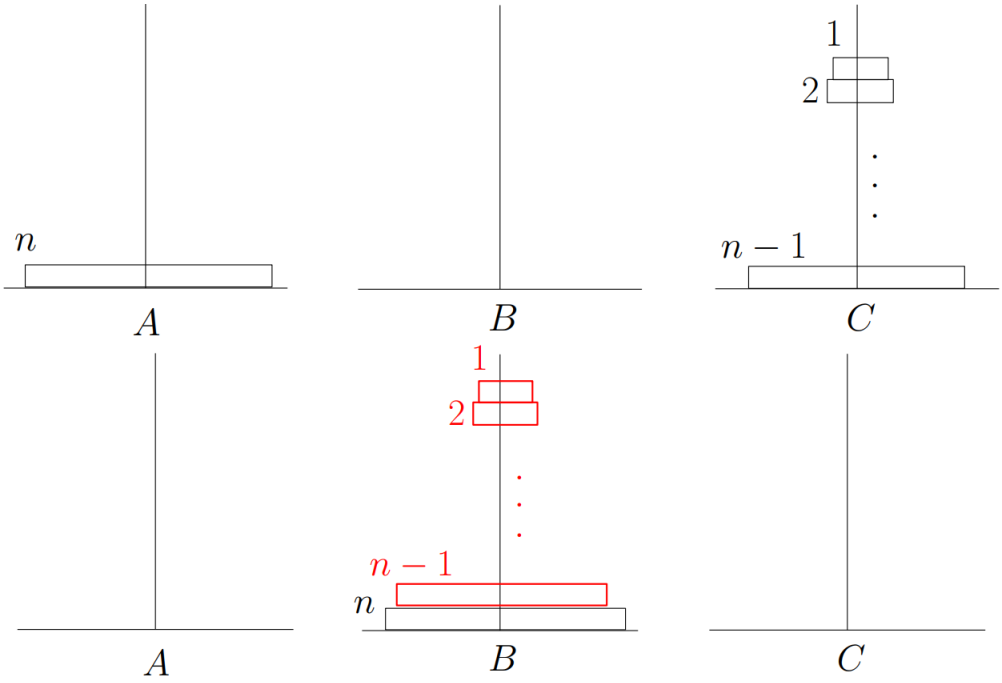
Constraint: No disk of a larger size can be above a disk of a smaller size.

Goal: Design an algorithm to move all the disks to rod B.

> 注意，rod B 和 rod C 都是空柱子，数学上等价.

直觉：**必须**先把前 $n-1$ 个移到 C，然后把最底层圆盘 n 移到 B，最后把 $n-1$ 个从 C 移回 B.

> 即解决子问题：把 $n-1$ 个圆盘从一个柱移到另一个空柱.
>
> 因为子问题完全不涉及最底层的圆盘 n，可以看作它消失了或者和地面融为一体了. 所以子问题和原问题除了规模，其他完全一致.
>
> “必须” 意思是我们考虑下界，即任意算法（移动方法）都不会比这个更优.



复杂度下界：基于递归直觉，形式化得到移动 n 个圆盘所需操作次数 $f(n)$ 满足

$$f(n) \geq 2f(n-1) + 1$$

解得

$$f(n) \geq 2^n - 1$$

结论：The best time complexity (even for randomized algorithms) for solving the Tower of Hanoi problem with n disks is $\Omega(2^n)$.

> $\Omega(2^n)$ 是下界. 关于渐近记号，见 `Tut 1 渐近分析` .

①* 最优性证明：数学归纳法

汉诺塔问题的递归算法被证明是最优的（达到最小移动步数）.

证明方法：数学归纳法（Mathmatical Induction）

证明过程如下：

递归算法给出的最小移动步数 $M(n)$ 满足 $M(n) = 2^n - 1$

基础情况（Base Case）

当 $n = 1$ 时，只有一个圆盘. 将圆盘从起始柱直接移动到目标柱最优.

$M(1) = 1$，递归算法最优性成立.

归纳假设（Inductive Step）

假设 $n = k$ 时，最少移动次数 $M(k) = 2^k - 1$. 证明当圆盘数 $n = k + 1$ 时，最少移动次数满足 $M(k + 1) = 2^{k+1} - 1$，结合 Base Case 则能证明 $M(n) = 2^n - 1$ 对所有正整数 n 成立.

证明：考虑将 $k + 1$ 个圆盘从起始柱 A 移动到目标柱 C，辅助柱为 B

- 为了将 $k + 1$ 个圆盘从起始柱 A 移动到目标柱 C，必须先把最大的圆盘 $(k + 1)_{th}$ 从起始柱 A 移动到 C.
- 为了移动最大的圆盘 $(k + 1)_{th}$，所有它上面的 k 个圆盘必须先全部堆放在辅助柱 B 上.
- 根据归纳假设，将 k 个圆盘从 A 移动到 B 至少需要 $M(k)$ 次移动.
- 最大圆盘从 A 移动到 C，需要 1 次移动.
- 为了完成最终目标，必须将堆放在辅助柱 B 上的 k 个圆盘移动到目标柱 C（在最大的圆盘上方）
- 根据归纳假设，将 k 个圆盘从 B 移动到 C 至少需要 $M(k)$ 次移动.
- 综上，最少移动总次数 $M(k + 1) = 2M(k) + 1 = 2^{k+1} - 1$

3.2 几何级数 & 重复直至成功

> Geometric Series and Repeating till Success

3.2.1 k-Selection 问题

> The k-Selection Problem
>
> 注意区分 "Top-k" 和 "k-Selection". "Top-k" 是找出**前** k 个最大（或最小）的元素. "k-Selection" 是找出**第** k 大（第 k 小）的元素.

问题：给定一个包含 n 个整数的集合 S 和一个整数 $k \in [1, n]$，要求找到 S 中第 k 小的整数.

> 注意：S 是无序的集合.

For example, suppose that $S = (53, 92, 85, 23, 35, 12, 68, 74)$ and $k = 3$. You should output 35.

定义 Rank:

The rank of an integer $v \in S$ is the number of elements in S smaller than or equal to v.

For example, suppose that $S = (53, 92, 85, 23, 35, 12, 68, 74)$. Then, the rank of 53 is 4, and that of 12 is 1.

显然 The rank of v can be obtained in $O(|S|)$ time.

> 一一对比，计数小于等于 v 的.
>
> 求某元素在集合中的 rank，即求它在这个集合中是第几小.

考虑下列任务：

Assume n to be a multiple of 3. Obtain a subproblem of size at most $\frac{2n}{3}$ with exactly the same result as the original problem.

Our goal is to produce a set S' and an integer k' such that

- $|S'| \leq \frac{2n}{3}$
- $k' \in [1, |S'|]$
- The element with rank k' in S' is the element with rank k in S

Note: it is possible that $k' \neq k$

有两个结论：

① 存在随机化算法 A_{sub} 以 $O(n)$ 最差预期成本完成该任务.

② 基于这个 A_{sub} 可设计一个随机化算法以 $O(n)$ 最差预期成本解决 k-Selection 问题.

① 存在随机化算法 A_{sub} 以 $O(n)$ 最差预期成本完成该任务.

A_{sub} 设计如下：

- Take an element $v \in S$ uniformly at random.

  > $O(1)$，因为只进行一次随机数生成（随机生成索引）.

- Divide S into S_1 and S_2 where
  - $S_1 =$ the set of elements in S less than or equal to v
  - $S_2 =$ the set of elements in S greater than v

  > $O(n)$，n 次比较.

- If $|S_1| \geq k$, then return $S' = S_1$ and $k' = k$;
  else return $S' = S_2$ and $k' = k - |S_1|$.

  > $O(1)$.
  >
  > (2025.12.13) 解释：$|S_1|$ 在划分时计算并存储，包含在 $O(n)$ 内，k 是已知整数，整数比大小是 $O(1)$；赋值 S' 是传递指针引用，$O(1)$；赋值 k' 是 $O(1)$ 操作. 综上，总复杂度仍然是 $O(1)$

- The algorithm succeeds if $|S'| \leq \frac{2n}{3}$, or fails otherwise.
- Repeat the algorithm until it suceeds.

解释：随机均匀选择一个元素 $v \in S$ 作为枢轴（pivot），将 S 分为 S_1（$\leq v$ 的元素）和 S_2（$> v$ 的元素）. 根据 k 的值判断第 k 小的元素在 S_1 还是 S_2，并返回相应的子集 S' 和新的目标排名 k'. 并采用重复直至成功 —— 如果 $|S'| \leq \frac{2n}{3}$ 则算法成功，否则失败并**重复直至成功**.

> 重复直至成功蕴含了一种类似拒绝采样的思想.

成功率：至少 $\frac{1}{3}$.

原因：the rank of v 落在 $[\frac{n}{3}, \frac{2n}{3}]$ 时必然成功（此时 S_1 和 S_2 都小于等于 $\frac{2n}{3}$）. 落在其他区域可能成功可能不成功. 落在 $[\frac{n}{3}, \frac{2n}{3}]$ 的概率就有 $\frac{1}{3}$. 所以成功率至少 $\frac{1}{3}$.

期望重复次数：最多为 3.

> $\mathbb{E}[X] \leq \frac{1}{\frac{1}{3}} = 3.$

Expected running time: $O(n)$.

② 基于这个 A_{sub} 可设计一个随机化算法以 $O(n)$ 最差预期成本解决 k-Selection 问题.

- $A_{sub}(S, k) \to (S_1, k_1)$. Note that $|S_1| \leq \frac{2}{3} \cdot n$.
- $A_{sub}(S_1, k_1) \to (S_2, k_2)$. Note that $|S_2| \leq (\frac{2}{3})^2 \cdot n$.
- $A_{sub}(S_2, k_2) \to (S_3, k_3)$. Note that $|S_3| \leq (\frac{2}{3})^3 \cdot n$.
- $\cdots$

> 这里应用了递归思维.

Stop until the t-th repetition such that $|S_t| = 1$, i.e,

$$(\frac{2}{3})^t = \frac{1}{n}$$

可以解出 t，然后计算总时间. 但我们用几何级数知识可以更优雅.

3.2.2 几何级数

> geometric series

A geometric sequence is an infinite sequence of the form

$$n, cn, c^2n, c^3n, \ldots$$

where n is a positive number and c is a constant satisfying $0 < c < 1$.

> 这里考虑收敛的几何级数（$c \geq 1$ 时发散）.
>
> 数列：一列有序的数.
>
> 前 n 项和：数列的前 n 项的和.
>
> 级数：无穷项的和.

Recall the formula for 等比数列/几何数列前 n 项和：

$$S_t = n \cdot \frac{1-c^t}{1-c}$$

It holds in general that

$$\sum_{t=0}^{\infty} c^t n = \lim_{t\to\infty} n \cdot \frac{1-c^t}{1-c} = \frac{n}{1-c} = O(n)$$

> 注意前提：$0 < c < 1$.

The summation $\sum_{t=0}^{\infty} c^t n$ is called a geometric series.

3.3.3 k-Selection：复杂度/上界

Recall:

② 基于这个 A_{sub} 可设计一个随机化算法以 $O(n)$ 最差预期成本解决 k-Selection 问题.

- $A_{sub}(S,k) \to (S_1,k_1)$. Note that $|S_1| \le \frac{2}{3} \cdot n$.
- $A_{sub}(S_1,k_1) \to (S_2,k_2)$. Note that $|S_2| \le (\frac{2}{3})^2 \cdot n$.
- $A_{sub}(S_2,k_2) \to (S_3,k_3)$. Note that $|S_3| \le (\frac{2}{3})^3 \cdot n$.
- $\cdots$

Stop until the t-th repetition such that $|S_t| = 1$

这里一定会运行有限的 t 次后停止，但我们求上界，可以对无穷项求和（实际的一定小于等于这个求和）.

The expected running time of our algorithm will be

$$a \cdot n + a \cdot \frac{2}{3} \cdot n + a \cdot (\frac{2}{3})^2 \cdot n + \ldots + a \cdot (\frac{2}{3})^t \cdot n$$

这里 a 是 A_{sub} 的最差期望成本的 big-O 中用到的常数.

> 即 $\max \mathbb{E}[X_{A_{sub}}(I)] \le an$ for $n \ge n_0$.

这里用到了期望的线性. 即各 execution 和的期望等于期望之和.

放缩求上界，有

$$\begin{aligned} & a \cdot n + a \cdot \frac{2}{3} \cdot n + a \cdot (\frac{2}{3})^2 \cdot n + \ldots + a \cdot (\frac{2}{3})^t \cdot n \\ \le\ & a \cdot n + a \sum_{i=1}^{\infty} (\frac{2}{3})^i \cdot n \\ =\ & O(n) \end{aligned}$$

总结：通过 “重复直至成功” 的思想构造 A_{sub}，然后运用 “递归思维" 大事化小，最后运用 ”几何级数“ 把有限项求和放缩为无限项求和，得到 $O(n)$ 的 worst-expected cost.

3.3.4* k-selection：下界

`3.3.3` 已经证明 worst-expected cost 是 $O(n)$. 我们还可以证明 worst-expected cost 也是 $\Omega(n)$，即证明下界.

> 但提到时间复杂度通常指上界，即 "big-O"

Tut 1 渐近分析

> Asymptotic Analysis: The Growth of Functions

In the lecture, we have defined the worst-case running time of an algorithm to be a function of n. However, the definition has nothing to do with "big-O". Many students hold the inaccurate view that "big-O" represents worst-case running time. 实际上，"big-O" 只说明了上界.

复杂度的理论分析中，我们不关心具体的常数因子，只关心增长速度.

1.1 渐近符号

渐近符号的详细讨论见 `大二 term 1 ENGG 2440 离散数学 5. Asymptotics`. 这里引用一小部分：

1.1.1 Big-O

表示函数增长的上界.

$$f(x) = O(g(x)) \quad if \quad \exists\, c > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad f(x) \leq cg(x)$$

> 虽然一个函数可以有多个 Big-O，但通常选择**最紧的上界**，以提供更精确的增长描述.
>
> 注意：c 大于 0.

深层理解：这里表示的是集合与成员的关系. $O(g(n))$ 实际上是一个集合，$f(n) = O(g(n))$ 表示 $f(n)$ 属于这个集合. 不使用属于符号是出于历史原因采用惯例.

TA 版：

We say that $f(n)$ grows asymptotically no faster than $g(n)$ if there is a constant $c_1 > 0$ such that

$$f(n) \leq c_1 \cdot g(n)$$

holds for all n at least a constant c_2.

We can denote this by $f(n) = O(g(n))$

注意，这里说 $f(n)$ 在渐近意义上增长速度不快于 $g(n)$，不是说 $f(n)$ 的实际增长速度一定不快于 $g(n)$，而是一定不快于 $g(n)$ 的某个倍数. 这样就可以避免常数因子的影响.

We say that an algorithm with running time $10000 \log_2 n$ is better than another one with running time $O(n)$. Big-O captures this because

$$\begin{aligned} 10000 \log_2 n &= O(n) \\ n &\neq O(10000 \log_2 n) \end{aligned}$$

> n 是 $10000 \log_2 n$ 的上界，但 $10000 \log_2 n$ 不是 n 的上界. 这足以说明 n 渐近意义上增长快于 $10000 \log_2 n$.

An interesting fact:

$$\log_a n = O(\log_b n)$$

for any constant $a > 1$ and $b > 1$

证明：

我们需要证明对于任意常数 $a > 1$ 和 $b > 1$，存在一个正常数 C 和一个阈值 n_0，使得对于所有的 $n \geq n_0$，都有：

$$\log_a n \leq C \cdot \log_b n$$

应用换底公式：

$$\log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \cdot \log_b n$$

定义常数 $C = \frac{1}{\log_b a}$.

> 因为 $a > 1, b > 1$，显然 $C > 0$

根据定义可以得到引理：$f(n) = C \cdot g(n)$ 且 C 是一个正常数，则 $f(n) = O(g(n))$.

> 显然，因为阈值 n_0 取在任意点都符合 big-O 定义（取等也满足小于等于号）

因此，$\log_a n = O(\log_b n)$

Because of the above, in computer science, we often omit constant logarithm bases in big-O. For example, instead of $O(\log_2 n)$, we will simply write $O(\log n)$.

> This says that "you are welcome to put any constant bases there, and it will be the same asymptotically"

Henceforth, we will describe the running time of an algorithm only in the asymptotical (i.e. big-O) form, which is also called the algorithm's **time complexity**.

> 一个函数可以有多个上界，尽量写最紧的.

1.1.2 Big-Ω

表示函数增长的下界.

$$f(x) = \Omega(g(x)) \quad if \quad \exists\, c > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad f(x) \geq cg(x)$$

1.1.3 Θ

同时满足 O 和 Ω 的条件，表示函数增长的确切阶.

$$f(x) = \Theta(g(x)) \quad if \quad f(x) = O(g(x)) \quad and \quad g(x) = O(f(x)).$$

> $f(x) = \Theta(g(x)) \nRightarrow f(x) = g(x)$. It just means that
> $\exists\, c_1, c_2 > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad c_1 g(x) \leq f(x) \leq c_2 g(x)$
>
> 只能说明 $f(x)$ 和 $g(x)$ 同阶.

1.1.4* Small-o

表示严格上界（严格更高阶），比 Big-O 更严格的界定.

$$f(x) = o(g(x)) \quad if \quad \lim_{x \to \infty} \frac{f(x)}{g(x)} = 0.$$

等价于

$$f(x) = o(g(x)) \quad if \quad \forall\, c > 0, \exists\, x_0 > 0\ such\ that\ 0 \leq f(x) < cg(x)\ for\ all\ x \geq x_0.$$

> 注意同阶不足以满足任意 c，$g(x)$ 要比 $f(x)$ 高阶才满足.

$$f(x) = o(g(x)) \Rightarrow f(x) = O(g(x))$$
$$f(x) = O(g(x)) \nRightarrow f(x) = o(g(x))$$

1.1.5* Small-ω

表示严格下界（严格更低阶），比 Big-Ω 更严格的界定.

$$f(x) = \omega(g(x)) \quad if \quad \lim_{x \to \infty} \frac{g(x)}{f(x)} = 0.$$

等价于

$$f(x) = \omega(g(x)) \quad if \quad \forall\, c > 0, \exists\, x_0 > 0\ such\ that\ 0 \leq cg(x) < f(x)\ for\ all\ x \geq x_0.$$

> 注意同阶不足以满足任意 c，$g(x)$ 要比 $f(x)$ 低阶才满足.

$$f(x) = \omega(g(x)) \Rightarrow f(x) = \Omega(g(x))$$
$$f(x) = \Omega(g(x)) \nRightarrow f(x) = \omega(g(x))$$

1.2 速查表 & 例题

1.2.1 常数 $= O(1)$

例：$100 = O(1)$

证明：取 $c = 100$

1.2.2 多项式 $= O(\text{增长最快项})$

> 标准多项式只能包含变量非负整数次幂运算的有限次加减乘. 这里考虑广义的准多项式，或多项超越函数式.

例：$100\sqrt{n} + 10n = O(10n) = O(n)$

> big-O 内如果只有一项则常数可忽略，易证.

1.2.3 低次幂 $= O(\text{高次幂})$

$1000n^{1.5} = O(n^2)$

> 取 $c = 1000$

1.2.4 对数次方 $= O(\text{任意次幂})$

$(\log_2 n)^3 = O(\sqrt{n})$

$(\log_2 n)^{9999} = O(n^{0.001})$

$n^{0.001} \neq O((\log_2 n)^{999})$

一般地，对于任何正数 $a > 0$ 和 $b > 0$，有 $\log^b n = O(n^a)$

> 这个结论表明，对数无论多少次方，增长速度都不如幂函数.

证明：

要证存在 $n_0 \geq 0$ 和 $c > 0$ 使得 $\log^b n \leq c \cdot n^a$ 对 $n \geq n_0$ 恒成立.

两边取 $\ln$，得

$$b \ln \log n \leq \ln c + a \ln n$$

整理得

$$b\ln\log n - a\ln n \le \ln c$$

左侧计为 $f(n) = b\ln\log n - a\ln n$，有

$$f'(n) = \frac{b}{n\log n} - \frac{a}{n}$$

易得零点满足 $\log n_0 = \frac{b}{a}$，$n < n_0$ 时递增，$n \ge n_0$ 时递减.

取 $\ln c = b\ln\log n_0 - a\ln n_0$

1.2.5 任意次幂 $= O(2^n)$

$n^{999} = O(2^n)$

$2^n \neq O(n^{999})$

> 幂函数增长速度小于指数函数.

证明：以 $n^{999} = O(2^n)$ 为例

$$n^{999} \le c \cdot 2^n$$

两边取对数，得

$$999\ln n \le \ln c + n\ln 2$$

整理得

$$999\ln n - n\ln 2 \le \ln c$$

左侧构造函数 $f(n) = 999\ln n - n\ln 2$，有

$$f'(n) = \frac{999}{n} - \ln 2$$

先增后减，转折点 $n_0 = \frac{999}{\ln 2}$

取 $\ln c = 999\ln n_0 - n_0\ln 2$

1.2.6 例题

$\log_2 n = \Omega(1)$

$0.001n = \Omega(\sqrt{n})$

$2n^2 = \Omega(n^{1.5})$

$n^{0.001} = \Omega(\log_2^{999} n)$

$\frac{2^n}{1000} = \Omega(n^{9999})$

$100 + 30\log_2 n + 1.5\sqrt{n} = \Theta(\sqrt{n})$

$100 + 30\log_2 n + 1.5n^{0.5001} \neq \Theta(\sqrt{n})$

$n^2 + 2n + 1 = \Theta(n^2)$

Tut 2 递归 / 重复 / 几何级数 练习

2.1 随机数生成另一范围随机数

Recall that our RAM model has an atomic operation RANDOM(x, y) which, given integers x, y, returns an integer chosen uniformly at random from [x, y].

> 注意，返回的是整数.

例 1：用 $\mathrm{RANDOM}(1, 128)$ 生成 $[1, 100]$ 的均匀随机数，预期时间 $O(1)$

解法：

- 调用（call） $\mathrm{RANDOM}(1, 128)$，设返回值为 z
- 如果 $z \in [1, 100]$，输出 z，结束
- 否则，从头开始重复（再次调用，重复直至成功）

复杂度分析：每次成功概率 $\frac{100}{128}$，每次 $O(1)$，预期总时间 $O(1)$

例 2：用 $\mathrm{RANDOM}(0, 1)$ 生成 $[1, n]$ 的均匀随机数，预期时间 $O(\log n)$

解法：

若 n 是 2 的幂

- 调用，返回 z
- 如果 $z = 0$，产生一个在范围前一半的均匀随机数的子问题.
- 如果 $z = 1$，产生一个在范围后一半的均匀随机数的子问题.

递归解法. 复杂度分析：$f(n) \leq f(\frac{n}{2}) + O(1)$，解得 $f(n) = O(\log n)$

扩展到一般的 n：

- 找到不小于 n 的最小的 2 的幂 m（$n \leq m < 2n$），需要 $O(\log n)$ 时间.

  > $1, 2, 4, \ldots$

- 使用上述方法，在期望 $O(\log n)$ 时间内生成一个 $[1, m]$ 范围的随机数 y.
- 如果 $y \leq n$，返回 y
- 否则，重复该算法直至成功.
- 成功率 $\frac{n}{m} > \frac{1}{2}$，预期重复 2 次，总预期成本 $O(\log n)$

2.2 k-Selection：确定性算法

Suppose there is a deterministic algorithm A_1 which returns the median of n integers in $O(n)$ time. Can you use A_1 as a blackbox to solve k-selection in $O(n)$ time?

> BFPRT 算法.

Consider the following algorithm:

- 从 $A_1(S)$ 获取中位数 v
- 将 S 分为 S_1（小于等于 v 的元素）和 S_2（大于 v 的元素）
- 根据 k 与 $|S_1|$ 的关系，确定子问题 S' 和 k'.

  > If $|S_1| \geq k$, then return $S' = S_1$ and $k' = k$; else return $S' = S_2$ and $k' = k - |S_1|$

- 由于 A_1 是确定性的，子问题的大小 $|S'|$ 总是不超过 $\lceil \frac{|S|}{2} \rceil$

时间复杂度满足 $f(n) \leq f(\frac{n}{2}) + O(n)$，解得 $f(n) = O(n)$

更一般的黑盒选择器：如果 A_1 在 $O(n)$ 时间内返回第 $\lceil \frac{4n}{5} \rceil$ 小的数，仍然可以使用 A_1 将问题规模缩小一个常数因子，根据几何级数求和，总成本仍为 $O(n)$

> From the geometric series we know that the total cost will be $O(n)$

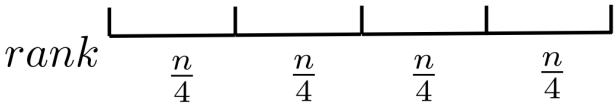
2.3 k-selection：随机性算法

课上讨论了 shrink the input size of the subproblem into at most $\frac{2}{3}n$. Now, we want to shrink the input size into at most $\frac{n}{2}$. Give an algorithm to achieve the purpose in $O(n)$ expected time.

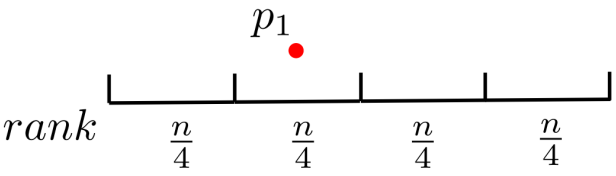
A simple solution: run our $\frac{2n}{3}$-algorithm twice. The number of remaining elements becomes at most $\frac{4n}{9}$.

另一种解法：假设 n 是 4 的倍数

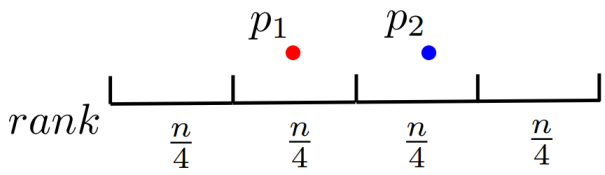
First, divide the rank space into 4 equal partitions.



Second, take an element p_1 from S uniformly at random. Repeat until $rank(p_1)$ is in range $[\frac{n}{4}, \frac{n}{2}]$.



Third, take an element p_2 from S uniformly at random. Repeat until $rank(p_2)$ is in range $[\frac{1}{2}n, \frac{3}{4}n]$.



- If $k \leq rank(p_1)$, set $S' =$ the set of elements in S less than or equal to p_1, $k' = k$.
- If $rank(p_1) < k < rank(p_2)$, set $S' =$ the set of elements in S larger than p_1 and smaller than p_2, $k' = k - rank(p_1)$.
- If $k \geq rank(p_2)$, set $S' =$ the set of elements in S larger than or equal to p_2, $k' = k - rank(p_2)$.

期望：p_1, p_2 各四次.

期望复杂度：$O(n)$

> 来自 rank 计算. 注意，我们是对 rank 空间进行四等分. 每一次选点都会进行 $O(n)$ 次计算. 期望总共 8 次选点，仍然 $O(n)$

Week 2

Lec 4 分而治之

> Divide and Conquer

Recall: Principle of recursion

核心思想：当遇到一个子问题（与原问题相同，但输入规模更小）时，假设子问题已**解决**，并用子问题的输出来继续算法设计.

> When dealing with a subproblem (same problem but with a smaller input), consider it solved, and use the subproblem's output to continue the algorithm design.

分治法即在递归（分）的基础上，添加一步（治） —— 在每步递归时解决隐藏在原问题中的核心问题，并合并子问题返回的结果.

本节通过四个问题讨论分治法：

排序问题 - 归并排序算法

逆序对计数问题 - 交叉逆序对算法

支配计数问题

矩阵乘法问题 - Strassen 算法

4.1 排序问题：归并排序

> Sorting

问题：给定一个包含 n 个不同整数的数组 A，将其按升序排列.

归并排序算法 - 分而治之思想

分：将 A 分为大致相等的两部分 A_1 和 A_2，然后递归地排序.

> 令 A_1 包含前 $\lceil \frac{n}{2} \rceil$ 个元素，A_2 包含剩下的元素.

治：将两个**已排序**的数组按升序合并. 可以在 $O(n)$ 时间内完成.

> 升序合并保证了两个子数组本身也是升序排列好的.

Running Time: Let $f(n)$ denote the worst-case cost of the algorithm on an array of size n. Then,

$$f(n) \leq 2f(\lceil \frac{n}{2} \rceil) + O(n)$$

由主定理得 $f(n) = O(n \log n)$.

4.2 逆序对计数问题

> Counting Inversions

问题：给定一个包含 n 个不同整数的数组 A. 计算逆序对数量.

> 逆序对（inversion）定义
>
> An inversion is a pair of (i, j) such that
>
> - $1 \leq i < j \leq n$, and
> - $A[i] > A[j]$

A naive algorithm：两两比对

复杂度：$(n-1) + (n-2) + \cdots + 1 = O(n^2)$

分而治之思想

分：将 A 分成 A_1（前 $\lceil \frac{n}{2} \rceil$）和 A_2. 递归计算 A_1 和 A_2 内部逆序对数量 m_1 和 m_2.

治：计算交叉逆序对（crossing inversions）的数量.

> 即 $i \in A_1$，$j \in A_2$ 的逆序对 (i, j).

$O(n \log^2 n)$ 版本

治：先对 A_1 进行归并排序（见 `4.1`），复杂度 $O(n \log n)$. 然后遍历 A_2 中的元素，对每个元素使用二分查找（$O(\log n)$）在 A_1 中找到它的排位，并计算与 A_1 中元素形成的交叉逆序对数量. 共进行 $|A_2| \approx \frac{n}{2}$ 次查找，共需 $O(n \log n)$ 时间.

递推式：$f(n) \leq 2f(\lceil \frac{n}{2} \rceil) + O(n \log n)$

由主定理解得 $f(n) = O(n \log^2 n)$

$O(n \log n)$ 版本

治：

- 假设 A_1' 和 A_2' 是递归返回的两个数组. 对 A_1' 和 A_2' 进行升序归并合并（注意：用升序归并排序的合并方法把两个子数组整合的行为保证了 A_1' 和 A_2' 本身也是提前升序排列好的，因此满足使用归并合并的前提）.

  > 复杂度 $O(n)$. 这里仅是归并排序的 "治".

- 在合并过程中同时计算交叉逆序对数量：每当 A'_2 中的元素 $A'_2[j]$ 被移动到合并后的数组 A' 中时，就意味着它比 A'_1 中所有**尚未**被移动的元素都要小. 此时所有 A'_1 中的剩余元素各自与 $A'_2[j]$ 形成一个交叉逆序对. 此时将 A'_1 中的剩余元素数量加到总的交叉逆序对计数上即可.

  > 每次计算只需要 $O(1)$ 复杂度.

注意，整个过程就是对 A 进行升序归并排序，只是在递归的过程中顺便花 $O(1)$ 代价计算一下交叉逆序对数量.

递推式：$f(n) \leq 2f(\lceil \frac{n}{2} \rceil) + O(n)$

由主定理解得 $f(n) = O(n \log n)$

易错点（Quiz 1 Q3a）

① "What are the outputs of the two subproblems, respectively?" 这里的 outputs 指两个子数组分别的逆序对总数. 注意不是子数组的交叉逆序对数. 递归的子问题返回的是这个子问题内部的总结果 / 总逆序对数. 主问题的 output 是两个子问题的 output 加上 "治" 中计算的交叉逆序对数.

> 例：$A = (70, 28, 43, 60, 80, 12, 30, 50)$. What are the outputs of the two subproblems, respectively?
>
> 答案：3 and 3.

4.3 支配计数问题

> Dominance Counting

问题：给定平面 Z^2 上 n 个 x 坐标不同的点集 P，对于 P 中的每个点 p，找出 P 中被 p 支配的点的数量.

支配：横纵坐标都大于等于被支配的点.

预先将点按 x 坐标升序排列，耗时 $O(n \log n)$.

分而治之思想

分：找到一条垂直线 l 将点集 P 分成左右两部分 P_1 和 P_2，每部分约 $\lceil \frac{n}{2} \rceil$ 个点. 递归求解 P_1 和 P_2 内部的支配计数.

治：P_1 的点不可能支配 P_2 的点. 对于 P_2 的每个点 p_2，计算它支配 P_1 中多少点. 由于 P_1 的所有点 x 坐标都小于 p_2，因此只需比较 y 坐标.

$O(n \log^2 n)$ 版本：对 P_1 按 y 坐标进行排序. 对于 P_2 中的每个点 p_2，使用二分查找在已排序的 P_1 中找出被 p_2 支配的点的数量.

> 递推式：$f(n) \leq 2f(\lceil \frac{n}{2} \rceil) + O(n \log n)$
>
> 由主定理解得 $f(n) = O(n \log^2 n)$

$O(n \log n)$ 版本：

要将支配计数问题的运行时间从 $O(\log^2 n)$ 优化到 $O(n \log n)$，我们必须在 "治" 阶段，即计算 P_2 中点支配 P_1 中点的数量时，将 $O(n \log n)$ 的工作代价降到 $O(n)$

实现这一优化的关键在于同时利用归并排序的思想来合并 y 坐标信息. 和逆序对计数问题很相似.

治：

- 假设 P_1 和 P_2 是递归返回的两个点集. 对 P_1 和 P_2 按 y 坐标进行升序归并合并（注意：用升序归并排序的合并方法把两个子点集整合的行为保证了 P_1 和 P_2 本身也是按 y 提前升序排列好的，因此满足使用归并合并的前提）.

  > 复杂度 $O(n)$. 这里仅是归并排序的 "治".

- 在合并过程中同时计算 P_2 中每个点支配的数量：每当 P_2 中的点 p_2 被移动到合并后的点集 P 中时，就意味着它比 P_1 中所有**已经**被移动的点都要大. 此时所有 P_1 中的剩余点就是不被 p_2 支配的数量（$|P_1|$ 减去剩余点数即被支配的数量）.

  > 每次计算只需要 $O(1)$ 复杂度.
  >
  > 也可以维护一个计数器，记录已从 P_1 添加到有序列表中的点的数量. 当一个点 $p \in P_2$ 被添加到有序列表时，它支配了当前计数器中记录的、所有已添加的来自 P_1 的点，因为这些点在 y 轴上都低于 p.

注意，整个过程就是对 P 进行升序归并排序，只是在递归的过程中顺便花 $O(1)$ 代价计算一下支配数量.

4.4 矩阵乘法问题

> Matrix Multiplication

问题：Given two $n \times n$ matrices A and B, compute their product AB

We store an $n \times n$ matrix with an array of length n^2 in "row-major" order.

例: $\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$ is stored as $(1,2,3,4)$

Note that any $A[i,j]$ （i 行 j 列元素）can be accessed in $O(1)$ time.

Trivial: $O(n^3)$ time

> A 的某一行分别和 B 的每一列进行共 $n\times n$ 次运算，共 n 行，总计 n^3

We will do in the class: $O(n^{2.81})$ time for n being a power of 2

Matrix Multiplication (Strassen's Algorithm)

Recall that the input A and B are order-n (i.e., $n \times n$) matrices. Assume for simplicity that n is a power of 2. Divide each of A and B into 4 submatrices of order $n/2$:

$$A = \begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}, B = \begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{bmatrix}$$

It is easy to verify:

$$AB = \begin{bmatrix}A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22}\\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}\end{bmatrix}$$

How many order-$(n/2)$ matrix multiplications do we need?

Trivial: 8.
Non-trivial: 7 — see the next slide.

Matrix Multiplication

$$\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{bmatrix} = \begin{bmatrix}p_5 + p_4 - p_2 + p_6 & p_1 + p_2\\ p_3 + p_4 & p_1 + p_5 - p_3 - p_7\end{bmatrix}$$

$$\begin{aligned}
p_1 &= A_{11}(B_{12} - B_{22})\\
p_2 &= (A_{11} + A_{12})B_{22}\\
p_3 &= (A_{21} + A_{22})B_{11}\\
p_4 &= A_{22}(B_{21} - B_{11})\\
p_5 &= (A_{11} + A_{22})(B_{11} + B_{22})\\
p_6 &= (A_{12} - A_{22})(B_{21} + B_{22})\\
p_7 &= (A_{11} - A_{21})(B_{11} + B_{12})
\end{aligned}$$

以左上角为例示范推导：

$$\begin{aligned}
&p_5 + p_4 - p_2 + p_6\\
=&(A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) - (A_{11} + A_{12})B_{22} + (A_{12} - A_{22})(B_{21} + B_{22})\\
=&A_{11}B_{11} \pm A_{11}B_{22} \pm A_{22}B_{11} \pm A_{22}B_{22} \pm A_{22}B_{21} \mp A_{12}B_{22} + A_{12}B_{21}\\
=&A_{11}B_{11} + A_{12}B_{21}
\end{aligned}$$

$\frac{n}{2} \times \frac{n}{2}$ 的矩阵进行常数次加减法只需要 $O(n^2)$ 次计算. 每个 p_i 包含一个 $\frac{n}{2} \times \frac{n}{2}$ 与 $\frac{n}{2} \times \frac{n}{2}$ 的矩阵乘法. 一共七个.

If $f(n)$ is the worst-case time of computing the product of two order-n matrices, then each of $p_i(1 \leq i \leq 7)$ can be computed in $f(\frac{n}{2}) + O(n^2)$ time.

递归式:

$$f(n) \leq 7f(\frac{n}{2}) + O(n^2)$$

由主定理解得

$$f(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

> $n^2 = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$，子问题规模占主导.
>
> 这里子问题规模占主导的含义是，乘法比加减法更耗时.

4.5 主定理

> Master Theorem

很多分治算法都能写成类似表达式:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

- n：问题规模
- a：每次递归中子问题的个数
- $\frac{n}{b}$：子问题的规模
- $f(n)$：分解问题和合并子问题所需的代价

目标：根据 $a, b, f(n)$ 来确定 $T(n)$ 的增长量级.

主定理

Case 1：子问题规模占主导

如果 $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$

那么 $T(n) = \Theta(n^{\log_b a})$

Case 2：子问题与合并代价平衡

如果 $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$，$k \geq 0$

那么 $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$

Case 3：合并代价占主导

如果 $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some $\epsilon > 0$

且满足正则性条件 $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ for some constant $0 < c < 1$ and all sufficiently large n

那么 $T(n) = \Theta(f(n))$

局限性：主定理只适用于每次递归将问题均匀划分，即子问题是 $\frac{n}{b}$ 大小. 形如
$T(n) = 2T(n-1) + O(1)$ 就不能用主定理.

4.5.1* 证明：递归树方法

> Recursion Tree Method

对于递推关系

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

根节点：原始问题，工作量为 $f(n)$.

> 考虑 "治"，即合并/分解的代价.

第一层：产生 a 个子问题，每个规模 $\frac{n}{b}$. 每个子问题工作量 $f(\frac{n}{b})$，该层总工作量 $a \cdot f(\frac{n}{b})$

第二层：产生 a^2 个子问题，每个规模 $\frac{n}{b^2}$. 该层总工作量 $a^2 \cdot f(\frac{n}{b^2})$

第 i 层：产生 a^i 子问题，每个规模为 $\frac{n}{b^i}$. 该层总工作量为 $a^i \cdot f(\frac{n}{b^i})$

递归将持续到子问题的规模达到基本情况（通常是 $T(1) = O(1)$）. 递归树深度大致为 $h = \log_b n$

总工作量求和：

$$T(n) = \sum_{i=0}^{h-1} \left(a^i \cdot f(\frac{n}{b^i}) \right) + \text{叶子节点的代价}$$

其中，叶子节点的数量为 $a^h = a^{\log_b n} = n^{\log_b a}$

叶子节点总代价为 $\Theta(n^{\log_b a})$

总工作量为

$$T(n) = \sum_{i=0}^{\log_b n-1} \left(a^i \cdot f(\frac{n}{b^i}) \right) + \Theta(n^{\log_b a})$$

它可以看作一个级数：

$$T(n) = f(n) + a \cdot f(\frac{n}{b}) + a^2 \cdot f(\frac{n}{b^2}) + \cdots + \Theta(n^{\log_b a})$$

Case 1：子问题规模占主导

实际上指叶子代价（最后一项）占主导，条件是 $f(n)$ 增长慢于 $n^{\log_b a}$. 那么每层的工作量 $W_i = a^i \cdot f(\frac{n}{b^i})$ 随着层级 i 的增加而呈几何级数递增，直到叶子层达到最大值 $\Theta(n^{\log_b a})$

由 `1.2.2`，多项式渐近看增长最大项，即最后一项 $\Theta(n^{\log_b a})$

Case 2：子问题与合并代价平衡

级数的各项增长速度相近. 条件是 $f(n)$ 与 $n^{\log_b a}$ 增长速度相近. 那么每层的工作量 W_i 近似相等. 此时 $T(n)$ 渐近等于层数 x 每层代价，因为有 $\log_b n$ 层，每层代价为 $\Theta(n^{\log_b a} \cdot \log^k n)$，总和为

$$T(n) = \Theta(\log n) \cdot \Theta(n^{\log_b a} \cdot \log^k n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$$

解释：k 是 $f(n)$ 中的对数因子. 假设 $f(n)$ 在 $n^{\log_b a}$ 的基础上额外多了 $\log^k n$ 的对数因子，虽然它们渐近不相等，但仍视为增长速度相近. 此时第 i 层工作量可写为

$$\begin{aligned} W_i &= a^i \cdot f(\frac{n}{b^i}) \\ &= \Theta(a^i \cdot (\frac{n}{b^i})^{\log_b a} \cdot \log^k(\frac{n}{b^i})) \\ &= \Theta(n^{\log_b a} \cdot \log^k(\frac{n}{b^i})) \end{aligned}$$

对所有 i 求和，

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n - 1} W_i + \Theta(n^{\log_b a}) \\ &= \Theta\left(n^{\log_b a} \sum_{i=0}^{\log_b n - 1} \log^k(\frac{n}{b^i})\right) \end{aligned}$$

可以证明 $\sum_{i=0}^{\log_b n - 1} \log^k(\frac{n}{b^i})$ 的渐近结果是 $\log^{k+1} n$

Case 3：合并代价占主导

实际上指 $f(n)$（第一项）占主导，条件是 $f(n)$ 增长快于 $n^{\log_b a}$. 那么每层的工作量 $W_i = a^i \cdot f(\frac{n}{b^i})$ 随着层级 i 的增加而呈几何级数递减，直到叶子层达到最小值 $\Theta(n^{\log_b a})$

由 `1.2.2`，多项式渐近看增长最大项，即第一项 $f(n)$

此处为严谨考虑，W_i 要递减地足够快，才能使用第一项，因此引入正则性条件 $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ for some constant $0 < c < 1$ and all sufficiently large n

Tut 3 分而治之：进阶

3.1 逆序对计数：$O(n \log n)$

> 见 `4.2`

3.2 支配计数：$O(n \log n)$

> 见 `4.3`

Week 3

Lec 5 快速傅里叶变换

> Divide and Conquer: Fast Fourier Transform

动机：我们看到分而治之可以加速矩阵乘法，那能否加速多项式乘法?

例：$(1+2x+3x^2)(2+x+4x^2) = 2+5x+12x^2+11x^3+12x^4$

More generally, for degree-d polynomials

$$A(x) = \sum_{i=0}^{d} a_i x^i, B(x) = \sum_{i=0}^{d} b_i x^i$$

their product $C(x) = A(x)B(x) = \sum_{k=0}^{2d} c_k x^k$ with

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}$$

> treat $a_i, b_i = 0$ if $i > d$

Naive time: $\Theta(d^2)$. Can we do better?

5.1 多项式的两种表示

> Key Idea: Two Representations of a Polynomial

Two equivalent representations of $A(x)$:

- Coefficients: $(a_0, a_1, \ldots, a_d)$
- 点值表示：在任意 $d+1$ 个互不相同的点上取值.

$$(x_0, A(x_0)), \ldots, (x_d, A(x_d))$$

> 一个 d 次多项式由其在任意 $d+1$ 个不同点上的取值唯一确定.

利用点值表示，可以把 $C(x)$ 的计算从计算出 $2d$ 个系数转为计算出 $2d+1$ 个点值对.

$$C(z) = A(z) \cdot B(z)$$

注意，这里只进行了 $O(d)$ 次乘法，而原先方法是 $O(d^2)$

> 暂时没考虑 $A(z), B(z)$ 内部的计算.

5.2 评估-乘法-插值范式

加速多项式乘法的核心思路是以下范式：

1. **选择**：选取 $n = 2d+1$ 个不同点 $x_0, \ldots, x_{n-1}$
2. **评估**：计算 $A(x)$ 和 $B(x)$ 在这些点上的值
3. **点值乘法**：计算 $C(x_k) = A(x_k)B(x_k)$
4. **插值**：利用点值对 $\{(x_k, C(x_k))\}_{k\in\{0,1,\ldots,n-1\}}$ 恢复 $C(x)$ 的系数

Bottlenecks: how to perform fast Evaluation and Interpolation

Evaluating a degree-n polynomial at one point costs at least $O(n)$ time. At n points, the baseline is $\Theta(n^2)$.

创新：如果能选择一组具有“高度代数结构”的特殊点来进行评估，就有可能将评估和插值的时间复杂度从 $\Theta(n^2)$ 降低

> FFT 正是通过选用 n **次单位根**作为评估点，将总成本降低到 $\Theta(n \log n)$

5.3 奇偶拆分

> Even-Odd Split

Let us decompose a given polynomial in a special manner: (assume for convenience that n is even):

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_n x^n \\ &= \left(a_0 + a_2 x^2 + a_4 x^4 + \ldots + a_n x^n\right) + \left(a_1 x + a_3 x^3 + a_5 x^5 + \ldots + a_{n-1} x^{n-1}\right) \\ &= \left(a_0 + a_2 x^2 + a_4 x^4 + \ldots + a_n x^n\right) + x \cdot \left(a_1 + a_3 x^2 + a_5 x^4 + \ldots + a_{n-1} x^{n-2}\right) \\ &= A_{even}(x^2) + x \cdot A_{odd}(x^2), \end{aligned}$$

where

$$\begin{aligned} A_{even}(x) &= a_0 + a_2 x^1 + a_4 x^2 + \ldots + a_n x^{n/2} \\ A_{odd}(x) &= a_1 + a_3 x^1 + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1} \end{aligned}$$

Observation: that $A_{even}(x)$ collects all even coefficients of $A(x)$, and A_{odd} collects all odd coefficients of $A(x)$. Both of them a of degree $\leq n/2$.

计算重叠：如果在 $\pm x_i$ 这样成对的点上求值，计算可以重叠计算重叠：如果在 $\pm x_i$ 这样成对的点上求值，计算可以重叠：

$A(x_i) = A_{\text{even}}(x_i^2) + x_i A_{\text{odd}}(x_i^2)$

$A(-x_i) = A_{\text{even}}(x_i^2) - x_i A_{\text{odd}}(x_i^2)$

这样 n 个点的求值问题可以归约到两个规模为 $n/2$ 的子问题，以及 $O(n)$ 时间的组合

因此，选择一个 special set $S = \{\pm x_0, \pm x_1, \ldots, \pm x_{\frac{n}{2}-1}\}$（共 n 个点，n 可以选择大于等于 $2d+1$ 的最小偶数）

Origin Problem: evaluate $A(x)$ on all the points in S

This problem can be reduced to first solve the same problem for $S' = \{x_0^2, x_1^2, \ldots, x_{\frac{n}{2}-1}^2\}$

Reduced problem: evaluate $A(x)$ on all the points of S'

> 治的所有部分都不超过 $O(n)$

如果可以一直分而治之，可以得到递推式

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

解得 $T(n) = O(n \log n)$

然而，**递归障碍**：这种 $(\pm x_i)$ 的结构只能归约一层，再继续向下递归时点集 S' 就不再具有 $(\pm y_i)$ 的结构了（全是正数）

Solution: Let's utilize complex numbers

例：set $S = \{1, -1, i, -i\}$, where i is the imaginary unit, i.e. $i = \sqrt{-1}$

This S supports two layer of recursion.

- $S' = \{1, -1\}$
- $S'' = \{1\}$

That is, if we want to evaluate a degree-3 polynomial $A(x)$ on every point of S, we can first do it at S', which can further be reduced to doing the evaluation on S''

This implement the above recurse $T(n) = 2T(\frac{n}{2}) + O(n)$ for the special case of $n = 4$

Important: Using a high-dimensional analog of i, we can generalize this idea to any n

5.4 n 次单位根

定义：方程 $x^n = 1$ 的 n 个解，称为 n 次单位根.

表达式：

$$x = \cos(\frac{2k\pi}{n}) + i \cdot \sin(\frac{2k\pi}{n}), \forall k \in \{0, 1, \dots, n-1\}$$

We denote $w_n = \cos(\frac{2\pi}{n}) + i \cdot \sin(\frac{2\pi}{n})$ 主 n 次单位根

所有的 n 次单位根可以表示为 $w_n^0, w_n^1, \dots, w_n^{n-1}$

> 把单位圆 n 等分.

欧拉公式：$\omega_n^k = e^{i \cdot \frac{2k\pi}{n}}$

重要性质（当 n 为偶数时）：

- $\omega_n^{k+n/2} = -\omega_n^k$
- $\{(\omega_n^k)^2 \mid k \in \{0, \dots, n-1\}\}$ 恰好是 $n/2$ 次单位根的集合 $U_{n/2}$. 更具体地，$\omega_n^{2k} = \omega_{n/2}^{k \bmod (n/2)}$

5.5 FFT 算法

目标：评估 $A(x)$ 在 n 次单位根集合 $U_n = \{\omega_n^0, \dots, \omega_n^{n-1}\}$ 上的所有值. 假设 n 是 2 的幂

递归步骤：

1. 拆分 $A(x) = A_{\text{even}}(x^2) + x \cdot A_{\text{odd}}(x^2)$ ($O(n)$ 时间)
2. 对 $A_{\text{even}}(y)$ 和 $A_{\text{odd}}(y)$ 递归调用 FFT，在 $n/2$ 次单位根 $U_{n/2}$ 上求值。($2f(n/2)$ 时间)
3. 利用性质 $\omega_n^{2k} = \omega_{n/2}^{k \bmod (n/2)}$ 以及奇偶拆分公式，计算 $A(\omega_n^k)$：

   $$A(\omega_n^k) = A_{\text{even}}(\omega_n^{2k}) + \omega_n^k A_{\text{odd}}(\omega_n^{2k})$$

   ($O(n)$ 时间)

时间复杂度：满足递推关系 $f(n) = 2f(n/2) + O(n)$，解为 $f(n) = O(n \log n)$

5.6 插值作为“逆 FFT”

- **逆 FFT**：将多项式的点值表示（在 n 次单位根上的取值）转换回系数表示的过程称为“逆 FFT”
- **时间复杂度**：逆 FFT 也可以在 $O(n \log n)$ 时间内完成

5.7 乘法范式最终版

利用 FFT 和逆 FFT，多项式乘法的 Evaluate–Multiply–Interpolate 范式最终时间复杂度为 $O(n \log n)$：

1. **选择** n：选取不小于 $2d+1$ 的最小 2 的幂作为 n
2. **评估**：使用 FFT 计算 $A(x)$ 和 $B(x)$ 在 U_n 上的值。($O(n \log n)$ 时间)
3. **点值乘法**：计算 $C(\omega_n^k) = A(\omega_n^k)B(\omega_n^k)$。($O(n)$ 时间)
4. **插值**：使用逆 FFT 从点值恢复 $C(x)$ 的系数 ($O(n \log n)$ 时间)

Lec 6 贪心：活动安排

> Greedy 1: Activity Selection
>
> 待完成.

贪心算法：每一步都做出局部最优的选择.

> 贪心不是针对某个问题的特定算法，而是一类算法的概括. 与其说贪心是一种算法，不如说是算法设计的一种思想.

贪心算法并不总能保证找到全局最优解，但有时可以. 证明 / 证伪全局最优性是其中的难点.

6.1 活动安排问题

活动安排 / 活动选择问题（Activity Selection）：规划一个只有一间会议室的小型会议，目标是在避免时间冲突的前提下，安排尽可能多的演讲（活动）.

形式化：给定一个包含 n 个区间 $[s, f]$（s 是某活动开始时间，f 是某活动结束时间）的集合 S，目标是找出一个仅包含不相交区间（各个活动没有时间冲突）的子集 $T \subseteq S$，使得 $|T|$ 最大.

6.2 贪心算法

贪心算法：选择最早结束的区间，保留它，并丢弃所有与之重叠的区间，重复此过程，直到没有剩余区间.

> 丢弃重叠区间可以保证每次的选择最紧凑，且不和之前选到的冲突.

形式化

Repeat until S becomes empty:

- Add to T the inteval $I \in S$ with the smallest finish time.
- Remove from S all the intervals intersecting I (including I itself)

时间复杂度：$O(n \log n)$

> 使用归并排序对结束时间升序排列：$O(n \log n)$.
>
> 选最早结束活动：$O(1)$，因为已经排序好.
>
> 移除所有与 I 相交的活动：活动按顺序处理，可以通过线性扫描移除不兼容的活动（从 I 当前位置向后扫描）. 每个活动在整个算法执行中只会被检查 / 移除一次，总复杂度是 $O(n)$
>
> 主导项是一开始的归并排序.

6.3 非唯一最优解

Note: there may be more than one optimal solution. Therefore, it is not accurate to say that our algorithm outputs “the” optimal solution. It is sufficient to show that our algorithm outputs an optimal solution.

贪心算法得到的解是最优，但非唯一.

6.4 最优性证明：交换论证 & 反证

Let

- $G = \{g_1, g_2, \ldots, g_k\}$ be the greedy solution
- $O = \{o_1, o_2, \ldots, o_m\}$ be an optimal solution, ordered by increasing finishing time.

证明目标是证明贪心算法的解 G 的大小 k 等于最优解 O 的大小 m，即 $k = m$

1. **基本事实:** 假设 O 是最优解，则贪心解的大小 k 必定不大于最优解的大小 m ($k \leq m$) . 因此，只需证明 k 不小于 m 即可 ($k \geq m$)

2. **证明策略:** 使用**交换论证（Exchange Argument）**，将最优解 O 逐步转换为贪心解 G，同时不减少活动数量

    - 令 $G = \{g_1, g_2, \ldots, g_k\}$ 是贪心解， $O = \{o_1, o_2, \ldots, o_m\}$ 是按结束时间排序的最优解

3. **交换步骤:**
    - 贪心算法选择 g_1（最早结束的），最优解选择 o_1
    - 由于 g_1 具有最小结束时间，所以 g_1 的结束时间 f_{g_1} 不迟于 o_1 的结束时间 f_{o_1}
    - 用 g_1 替换 O 中的 o_1。因为 g_1 结束时间不晚于 o_1，它与 O 中所有后续活动仍然兼容
    - 重复此步骤，用 g_i 替换 o_i

4. **反证法和矛盾:**
    - 假设 $k < m$ (即贪心解不是最优的)

    - 经过 k 次交换，得到 $O_k = \{g_1, g_2, \ldots, g_k, o_{k+1}, o_{k+2}, \ldots, o_m\}$

        > 已用完所有可替换的 g_i，g_k 是最后一个

    - 在 O_k 中， g_k 的结束时间早于所有剩余活动 $\{o_{k+1}, \ldots, o_m\}$ 的结束时间，且 g_k 不与它们中的任何一个重叠

- 这与贪心算法的定义矛盾：如果 g_k 是 G 中的最后一个元素，那么算法应该在 g_k 之后停止. 但 O_k 表明，仍然存在满足算法条件的活动（如 o_{k+1}）可以被添加到解集 G 中，因此算法不应停止在 g_k

- 这个矛盾证明了初始假设 $k < m$ 是错误的，因此**必有** $k = m$

Week 4 & 5

Lec 7 贪心：最小生成树

> Minimum Spanning Trees
>
> 主要内容见`大二 term 1 ESTR 2102 数据结构 8.4.4 最小生成树`.

7,1 图论基本知识

7.1.1 加权无向连通图

Let $G = (V, E)$ be an undirected graph. Let w be a function that maps each edge e of G to a positive integer value $w(e)$, which we call the weight of e.

An undirected weighted graph is defined as a pair (G, w).

一个无向加权图 (G, w) 由一个无向图 $G = (V, E)$ 和一个权重函数 w 组成，该函数将 G 的每条边 e 映射为一个正整数值 $w(e)$ 表示边的权重.

We will denote an edge between vertices u and v in G as (u, v). Note that the ordering of u, v does not matter (因为无向).

We consider that G is connected（连通的）, namely, there is a path between any two vertices in V.

> 注意：两个顶点不一定要直接相连，顶点 A, B 之间有 path 指的是顶点 A 经过若干条边之后能够到达 B.

7.1.2 无环图

Acyclic Graph: A graph contains no cycles

7.1.3 树

树：连通（connected）、无环（acyclic）、无向图（undirected）.

> 三个同时满足才叫树.

结论：any tree on $|V|$ vertices has exactly $|V|-1$ edges

(Special Exercise Set 4 Problem 1)

Let T be a tree. Prove: for any two distinct nodes u, v in the tree, there exists one and exactly one simple path from u to v (a simple path is a path where no vertex appears twice).

证明：围绕树定义的三要素展开.

① 存在性

连通：u 和 v 之间至少存在一条 path.

无环：这条 path 必然是 simple path，否则从一个点回到一个点（出现 twice）则会成环.

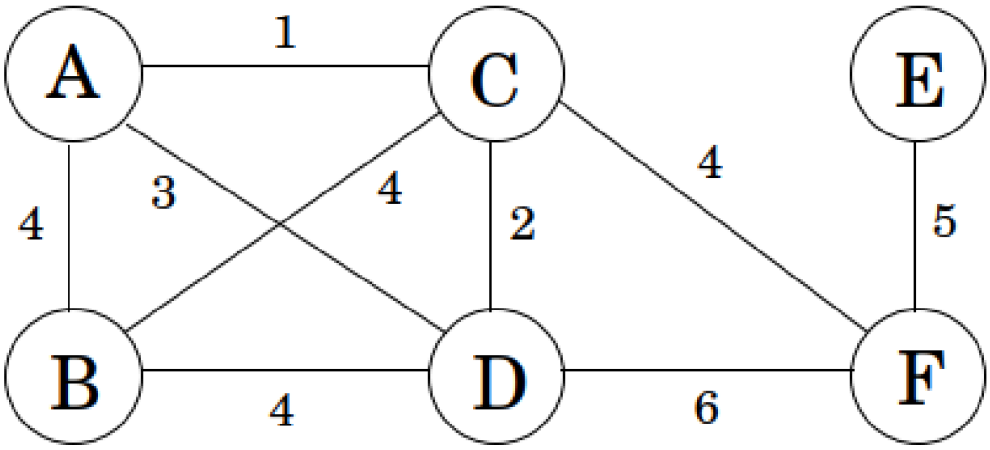
② 唯一性

反证法，假设存在不止一条 simple path. 任取两条计为 P_1, P_2.

因为 P_1 和 P_2 是不同的，它们必须在某个节点 x 分开，然后在另一个节点 y 首次汇合（x 可能是 u，y 可能是 v）.

P_1 从 x 到 y 的子路径与 P_2 从 x 到 y 的子路径是不同的. 这两条不同的子路径组合在一起会成环，与无环的定义矛盾.

7.2 最小生成树



考虑一个实际问题：Suppose you must link a collection of computers / sites / cities (as shown above).

- Nodes are computers; edges are potential links.
- Each link has a maintenance cost (edge weight).
- Goal: connect everyone while minimizing total cost.

此即最小生成树问题——计算加权无向连通图的最小生成树.

> connect everyone while minimizing total cost 的结果必然是一个 connected 且 acyclic（无环）的 undirected graph. 这样的 undirected graph 称为树. 其中 minimum total weight 的那个树叫做 Minimum Spanning Tree.

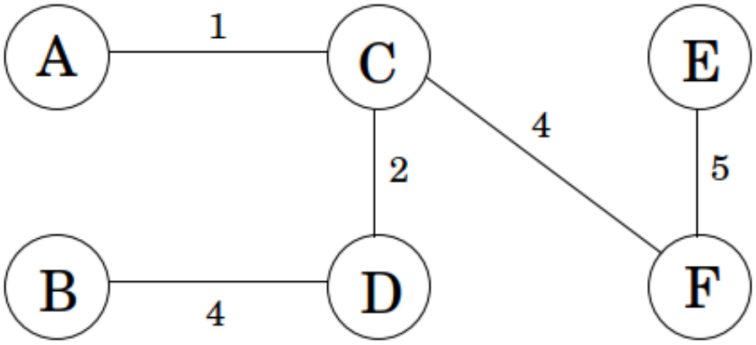


Figure: A MST for the Previous Graph

7.2.1 形式化定义

生成树（Spanning Tree）：给定一个连通的无向加权图 (G, w)，一个生成树 T 是一个子图，它是一个树（连通且无环），其顶点集与 G 相同（包含所有顶点 V），其边集是 G 边集的子集.

> Note that T must have $|V| - 1$ edges.
>
> That is, it is a tree (i.e., connected and contains no cycles) obtained by deleting some edges from G while ensuring that all the vertices of G are retained (i.e., do not become isolated).

生成树的成本（cost）定义为树中所有边的权重之和.

7.2.2 生成树 $\Leftrightarrow |V| - 1$ 条边的连通图

Let $G = (V, E)$ be an undirected graph, and let T be a subgraph of G. Then the following are equivalent:

- T is a spanning tree of G
- T is connected and has exactly $|V| - 1$ edges

证明：

生成树 $\Rightarrow |V| - 1$ 条边的连通图

Since any tree on $|V|$ vertices has exactly $|V| - 1$ edges, thus we know that T is connected and has exactly $|V| - 1$ edges.

生成树 $\Leftarrow |V| - 1$ 条边的连通图

Assume: T (being a subgraph of G) is connected and has exactly $|V| - 1$ edeges.

First, note that T must be acyclic.

> 否则，removing one edge from the cycle does not break connectivity; however, it is impossible to have a connected graph of $|V|$ vertices but with only $|V| - 2$ edegs.

Also, T must span all vertices

> 否则，T can contian at most $|V| - 1$ vertices. However, recall that T has only $|V| - 1$ edges. These two conditions imply that T must contains a cycle, contracting to the acyclic property we just established for T above

7.2.3 最小生成树问题

最小生成树（MST）：所有生成树中，成本最小的生成树.

注意：MST 可能不唯一.

Given a connected undirected weighted graph (G, w) with $G = (V, E)$, the goal of the minimum spanning tree (MST) problem is to find a spanning tree of the smallest cost.

7.3 Prim 算法：加点

> Prim's Algorithm for MST

从任意顶点开始，逐步增长一棵树 T_{mst}.

> why doesn't it matter where you start?
>
> 解释：对于任何一个割（将顶点分成 S 和 $V\backslash S$），权重最小的横切边必定属于某个 MST

将顶点集 V 分为已在 T_{mst} 中的集合 S 和其余顶点 $V\backslash S$. 连接 S 中顶点和 $V\backslash S$ 中顶点的边称为 cross edge（横切边）.

贪心：重复选择权重最小的横切边加入到 T_{mst} 中.

> 如果有多条相同，可以随机选择.

7.3.1 正确性证明：交换 & 归纳

证明采用**数学归纳法**和**交换论证**

核心不变条件 (Invariant Condition)：到目前为止，Prim 算法选择的边集 E_k 包含在**某个** MST (T^*) 中

证明依赖的图论事实 (Graph-Theoretic Facts)：

1. **事实 1 (加边成环)**：向一个树 T 中添加一条连接树中两个顶点的非树边 e，会形成**恰好一个**循环
2. **事实 2 (去环保连通)**：从连通图的循环中移除任何一条边，图仍然保持连通
3. **事实 3 (生成树的等价条件)**：对于一个子图 T，以下条件等价:
   - T 是 G 的一个生成树
   - T 是连通的且恰好有 $|V|-1$ 条边

归纳步骤 (Inductive Step)：

- **归纳假设**：假设在前 k 步选出的边集 E_k 是某个 MST T^* 的子集
- **第 $k+1$ 步**：Prim 选择了下一条最小权重跨边 e
  - **情况 1**：如果 $e \in T^*$，则 E_{k+1} 仍是 T^* 的子集
  - **情况 2**：如果 $e \notin T^*$，使用**交换论证**:
    - 在 T^* 中，边 $e=(u,v)$ 的端点 u 和 v 之间有一条唯一的路径
    - 这条路径上必然包含一条跨越割集 $(S_k, V \setminus S_k)$ 的边 f
    - 构造新图 $T' = T^* + e - f$
    - **证明 T' 是生成树**：T^*+e 包含一个循环（事实 1），移除循环中的边 f 保持连通（事实 2）由于 T^* 有 $|V|-1$ 条边，故 T' 也有 $|V|-1$ 条边。根据事实 3，T' 是一个生成树

- **证明 T' 是 MST**：因为 e 是 Prim 算法选择的最小权重跨边，所以 $w(e) \leq w(f)$。因此 T' 的总权重 $w(T') = w(T^*) + w(e) - w(f) \leq w(T^*)$。由于 T^* 是 MST，这意味着 T' 也是一个 MST。同时，E_{k+1} 是 T' 的子集。

  > Prim算法要么选到某个MST的边，要么选到不是该MST的边，但肯定和该MST的另一条跨边有相同权重，可以替换

终止：Prim 算法在 $|V| - 1$ 步后终止。此时选出的边集 P 形成了某个 MST T_{final} 的子集。由于 P 也有 $|V| - 1$ 条边，根据事实 3，P 必然等于 T_{final}。因此，Prim 算法输出一个 MST

7.3.2 基于双二叉搜索树的实现

How to implement Prim's algorithm in $O\left((|V| + |E|) \cdot \log |V|\right)$ time? 下面给出一种基于二叉搜索树的实现.

① 二叉搜索树 / 最小堆

> 待补充.

② 算法步骤

定义不变量（Invariant）：对于 $V \backslash S$ 中的每个顶点 v，$\text{best-cross}(v)$ 是连接 v 的横切边中权重最小的那一条.

> “不变” 指的是性质不变（始终为 true），而不是值不变. 这里的性质是 $\text{best-cross}(v)$ 总是连接 v 的横切边中权重最小的.
>
> 如果 $\text{best-cross}(v)$ 暂时不存在，则计为无穷大.

算法步骤：

i) 初始化 S 为权重最小的边 $\{u, v\}$ 的两端点（$S \leftarrow \{u, v\}$），并将 $\{u, v\}$ 加入 T_{mst}.

ii) 对 $V \backslash S$ 中的每个顶点 z，初始化 $\text{best-cross}(z)$ 为 $\{z, u\}$ 和 $\{z, v\}$ 中权重较小的边（如果都不存在，则视为无穷大）.

iii) 重复直到 $S = V$:

- 找到最小权重的横切边 $\{u, v\}$（其中 $u \in S, v \in V \backslash S$），将 v 加入 S，并将边 $\{u, v\}$ 加入 T_{mst}.

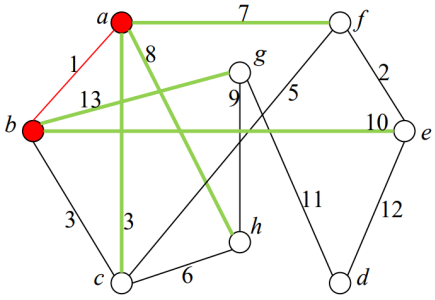
  > 只需要在 $\text{best-cross}(z)$ 中寻找，其中 $z \in V \backslash S$：因为每一步总要加入一个 $V \backslash S$ 中的新顶点，而加入的边必然是这个新顶点的 best-cross.

- 强制维护不变量性质为真（Enforce our invariant）：对于 v 的每条边 $\{v, z\}$，如果 $z \notin S$（即 $z \in V \backslash S$），且 $\{v, z\}$ 比当前的 $\text{best-cross}(z)$ 更轻，则更新 $\text{best-cross}(z) \leftarrow \{v, z\}$

> 注意，已经加入 T_{mst} 的顶点对应的 best-cross 可以不用维护，因为后面用不上了. 但实际上，我们会在该顶点及其 best-cross 加入 T_{mst} 之后，马上把它的所有信息即 $(\mathrm{id}, \mathrm{weight}, \mathrm{data})$ 在按 weight 索引的二叉搜索树中删除（即 DeleteMin 操作），方便我们在下一轮循环中快速找到最小横切边.
>
> 因为引入了新顶点 v，所以只需要看看引入这个新顶点会不会对其他 $z \in V \backslash S$ 的 best-cross 产生影响.

手写时可以维护一个 vertex, best-cross and weight 表格

Edge $\{a, b\}$ is the lightest of all. So, in the beginning $S = \{a, b\}$. The MST now has one edge $\{a, b\}$.



vertex v	best-cross and weight
a	n / a
b	n / a
c	{c, a}, 3
d	nil, ∞
e	{e, b}, 10
f	{a, f}, 7
g	{g, b}, 13
h	{a, h}, 8

③ 数据结构

原始数据结构是一个连通的无向加权图 (G, w)，通常用邻接表（Adjacency List）表示.

> 见 `大二 term 1 ESTR 2102 数据结构 8.3 数据结构`.
>
> 一种邻接表的实现：Array + Lists 的形式.
>
> 维护一个数组 `Adj[]`，长度等于顶点数量 $|V|$，每个索引 i 对应一个顶点 v_i. `Adj[i]` 内存储一个指针，指向一个链表的头部. `Adj[i]` 指向的链表包含了 v_i 的所有邻居. 准确地说，链表的节点是一个数据结构，包含以下信息：
>
> - 邻居顶点 z 的标识符（如数字、字母）.
> - 边 $\{v_i, z\}$ 的权重 $w(\{v_i, z\})$.
> - 指向下一个邻居节点的指针.
>
> 算法步骤中的某些地方需要下面提到的数据结构 P 的 Find, Insert, Delete, DeleteMin 操作和原始数据结构（邻接表）的某些操作配合完成.

为高效实现，需要一个数据结构 P 来存储形如 $(\mathrm{id}, \mathrm{weight}, \mathrm{data})$ 的元组（tuple）.

> id 是顶点的标识，例如字母 $a, b, c, d, \ldots$ 或数字 $0, 1, 2, \ldots$

> weight 是该顶点 best-cross 的权重.
>
> data 是 best-cross 本身.

并且该结构需要支持 Find, Insert, Delete, DeleteMin 等操作：

操作	说明	Prim 中的作用	复杂度
Find	给定一个整数 t，找到（返回） P 中 id 等于 t 的元组 $(\text{id}, \text{weight}, \text{data})$. 如果不存在，返回空.	i) 在 enforce invariant 时，检查新加入顶点 v 的邻居 z 是否在 $V\backslash S$ 中. 如果返回空，说明 z 对应的元组被删掉了，即 z 已经被加入到 T_{mst} ，即 z 不在 $V\backslash S$ 中，可以跳过； ii) 如果成功找到，还能够获取 z 当前的 best-cross，以便进行比较和潜在的更新.	$O(\log n)$
Insert	将一个新元组 $(\text{id}, \text{weight}, \text{data})$ 添加到 P 中.	在 enforce invariant 时，如果有 z 的 best-cross 需要更新，则要 Delete 旧元组，Insert 新元组.	$O(\log n)$
Delete	给定一个整数 t，删除 P 中 id 等于 t 的元组.	同上	$O(\log n)$
DeleteMin	从 P 中移除权重最小的元组.	在某顶点 v 及其 best-cross 加入 T_{mst} 之后，马上把它的所有信息即 $(\text{id}, \text{weight}, \text{data})$ 在按 weight 索引的二叉搜索树中删除，以便在下一轮循环中快速找到最小横切边.	$O(\log n)$

> enforce invariant 时，有两步涉及查找原始邻接表：
>
> 找 v 的邻居 z：遍历链表 `Adj[v]`，最差有 $|V|-1$ 个邻居.
>
> 经过 Find 操作，如果 $z \in V\backslash S$，要获取 $w(\{v,z\})$ 以便和 z 当前的 best-cross 进行比较和潜在的更新.
>
> 更新：Delete 旧元组 $(z, \text{old\_weight}, \text{old\_data})$，Insert 新元组 $(z, \text{new\_weight}, \text{new\_data})$. 其中，`old_weight` 和 `old_data` 是当前 z 的 best-cross 权重和边本身，`new_weight` 和 `new_data` 是 z 更新后的 best-cross 权重和边本身.
>
> 如果 $w(\{v,z\}) \geq \text{best-cross}(z)$，不需要更新. 如果更新，`new_weight` 实际上就是 $w(\{v,z\})$ ，`new_data` 实际上就是 $\{v,z\}$.

我们期望 Find, Insert, Delete, DeleteMin 都能在 $O(\log n)$ 时间内完成（n 是元组数量）. 这可以通过维护**两个**二叉搜索树 T_1（按 id 索引）和 T_2（按 weight）实现.

> 隐含条件：两个二叉树是平衡二叉树，如红黑树或 AVL 树.

四个操作分别为：

Find: given id, search the tuple in T_1.

> 因为是以 id 索引的二叉搜索树，所以 $O(\log n)$.

Insert: insert the new tuple into both T_1 and T_2

Delete: first find the tuple with id t in T_1, from which we know the weight. Now, delete the tuple from both T_1 and T_2

DeleteMin: find the tuple with the smallest weight from T_2 (which can be founded by continuously descending into left child nodes). Now we have its id t as well. Remove the tuple from both T_1 and T_2

在算法过程中，T_1 和 T_2 始终只有 $V\backslash S$ 中的顶点对应元组. 即每当一个元组加入 T_{mst}，就把它从 T_1 和 T_2 中删除. 并且，虽然 T_1 和 T_2 的索引方式不同（即元组的排列方式不同），但 T_1 和 T_2 的元组是一一对应的（数量相同、内容相同）.

④ 复杂度分析

总时间主要由三个操作组成：

- 预处理：查找初始最小权重边

  > 遍历邻接表的所有边：$O(|E|)$.

- 初始化：初始化 S 和 P.

  > 初始化 S：$O(1)$.
  >
  > 初始化 P：建两个二叉搜索树. 一共初始化 $|V|-2$ 个顶点（$|V|-2$ 个元组）. 对于每个顶点：
  >
  > - 计算 $\text{best-cross}(z)$：$O(1)$.
  > - 执行一次 Insert：每次 $O(\log|V|)$
  > - 总计：$O(|V|\cdot\log|V|)$

- 主循环：“添加一条边（和一个顶点）+ 更新不变量” 的迭代.

  > 找到并添加最小横切边：DeleteMin 操作. 每次 $O(\log|V|)$，一共 $|V|-2$ 次，故总时间 $O(|V|\log|V|)$.
  >
  > 更新不变量：对于每个新加入 S 的顶点 v，遍历其所有邻居 z（d_v 条边，故 d_v 个邻居）. 对每个 z：
  >
  > - Find：在 T_1 上查找 z 是否在 P 中，$O(\log|V|)$.
  > - 若 z 在 P 中，查邻接表的 $\{v,z\}$，$O(d_v)$. 比较 $w(\{v,z\})$ 和 $\text{best-cross}(z)$，$O(1)$. 如果 $w(\{v,z\})<\text{best-cross}(z)$，执行一次 Delete 和一次 Insert 来更新，$O(\log|V|)$.
  > - 最差情况：每次新加入顶点 v，遍历的每个邻居（几乎）都不在 T_{mst} 中，并且都需要更新. 共计 $O(\sum\limits_{v\in V}d_v\log|V|)=O(2|E|\log|V|)=O(|E|\log|V|)$.

故总时间：$O(|E|)+O(|V|\cdot\log|V|)+O(|V|\cdot\log|V|)+O(|E|\cdot\log|V|)$

课件写法：

$$
\begin{aligned}
\text{Total time} &= O(|V| \cdot \log |V| + \sum_{v \in V} \log |V| + \sum_{v \in V} d_v \log |V|) \\
&= O((2|V| + 2|E|) \cdot \log |V|) \\
&= O((|V| + |E|) \cdot \log |V|)
\end{aligned}
$$

7.4 Kruskal 算法：加边

> Kruskal's algorithm

定义跨边（cross edge）：连接 F 中不同树的边 $\{u, v\}$.

> 注意，和 Prim 算法中的 cross edge 略有不同. 因此用不同的中文翻译来区分.

维护一个森林 F，其中每个顶点恰好属于 F 中的一棵树.

初始化：$|V|$ 棵树，每棵树只包含一个顶点.

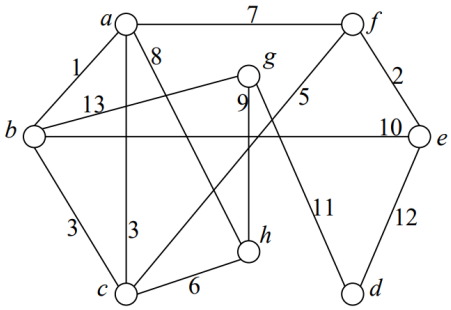
主循环：贪心. 不断选取当前最轻的跨边来合并两个树，直到所有顶点合并为一棵树（最小生成树）.

> 最轻的跨边就是最轻的不成环的边.

手写时可以维护一个 Trees, vertices 表格.

At the beginning, $|V| = 8$ trees: each tree has only one vertex.

Every edge is a cross edge at the moment.
Edge $\{a, b\}$ is the lightest cross edge.



Trees	Vertices
T_1	a
T_2	b
T_3	c
T_4	d
T_5	e
T_6	f
T_7	g
T_8	h

7.4.1 正确性证明：归纳

通过**归纳法**证明算法产生的边集合一定包含在某棵最小生成树中：

- **归纳假设**：假设前 $i-1$ 条选出的边都在某棵 MST 中
- **推导过程**：如果新选出的第 i 条边 e_i 不在原 MST 中，将其加入原 MST 会形成一个环.
- **替换策略**：在环上寻找另一条跨接边 e'，由于 Kruskal 总是选当前最小的边，因此 e_i 的权重不会高于 e'。通过将 e' 替换为 e_i，可以得到一棵包含 e_i 且成本不增加的新 MST
- Base case：$k=1$ 时，可以使用类似的替换策略.

7.4.2 实现

复杂度：$O(|E|\cdot\log|E|)$

Kruskal算法实现 $O(|E|\log|E|)$ 复杂度的核心在于两个主要步骤：**对边进行排序**以及使用**并查集（Union-Find）**数据结构来管理森林中的树.

以下是具体的实现方案：

1. **核心步骤与复杂度分析**

- **对边进行排序 ($O(|E|\log|E|)$)**:
  - Kruskal算法是一种贪心算法，其核心是反复选取当前权重最小的“跨接边”（cross edge）
  - 为了快速找到这些边，首先需要将图中所有的边按权重从大到小进行排序
  - 使用快速排序（Quicksort）或归并排序（Merge Sort）等算法，处理 $|E|$ 条边所需的时间复杂度为 $O(|E|\log|E|)$
- **维护并查集 ($O(|E|\alpha(|V|))$ 或 $O(|E|\log|V|)$)**:
  - 算法需要实时判断一条边 $\{u,v\}$ 是否为跨接边，即 u 和 v 是否属于不同的树
  - 通过**并查集（Union-Find）**可以高效地维护这种“树的集合”关系：
    - **Find(u)**：返回顶点 u 所在树的代表元素。若 $Find(u)\neq Find(v)$，则该边为跨接边。
    - **Union(u, v)**：当选定边 $\{u,v\}$ 后，将两个顶点所属的树合并为一棵
  - 使用“路径压缩”（Path Compression）和“按秩合并”（Union by Rank）优化的并查集，处理这些操作的平均复杂度极低（接近常数）

2. **总复杂度总结**

算法的总运行时间主要由排序步骤决定：

$$T(|E|,|V|)=O(|E|\log|E|)+O(|E|\alpha(|V|))=O(|E|\log|E|)$$

由于在简单图中 $|E|\leq|V|^2$，故 $\log|E|$ 与 $\log|V|$ 在渐近意义上是同阶的，因此该复杂度也常被表述为 $O(|E|\log|V|)$

3. **实现流程图**
4. **初始化**：将每个顶点初始化为一棵独立的树（此时树的数量 $t = |V|$）
5. **排序**：按边权非递减顺序排列 E
6. **遍历与合并**：
    - 取出当前最小权重的边 $\{u, v\}$
    - **检查**：若 u 和 v 属于不同树，则将此边加入 MST 并合并两棵树
    - **终止**：当所有顶点合并为一棵树（$t = 1$）时停止

Week 6

Lec 8 贪心：Huffman 编码

8.1 编码

8.1.1 编码 & 码字

Given an alphabet Σ (如英文字母表), an **encoding** is a function that maps each letter in Σ to a binary string, called a **codeword**.

编码是将字母表 Σ 中的每个字母映射到一个称为**码字**的二进制字符串.

目标：设计更短的码字来节省内存、加快传输速度、降低成本.

由于不同符号出现频率（frequency）不同，可以考虑给高频符号分配更短的码字.

> 注意：编码的定义里并没有要求所有字母分配到同样长度的码字.

8.1.2 前缀码

为了避免解码（decoding）时的歧义（ambiguity），需要强制执行一个约束（constraint）：任何字母的码字都不能是另一个字母码字的前缀（prefix）.

满足此约束的编码称为前缀码（prefix code）.

8.1.3 平均长度

For each letter $\sigma \in \Sigma$, let $\text{freq}(\sigma)$ donote the frequency of σ. Also, denote by $\text{len}(\sigma)$ the number of bits in the codeword of σ.

Given an encoding, its average length is

$$\sum_{\sigma \in \Sigma} \text{freq}(\sigma) \cdot \text{len}(\sigma)$$

> 即每个字母分配到的码字长度对频率加权平均.

平均长度是相对特定编码方式和字母表及其频率分布而言的，更换编码方式或字母表可能改变平均长度.

8.1.4 代码树

前缀码可以用二叉树来表示，称为代码树（code tree）.

- 字母对应代码树的叶子节点.
- 左边连接标记为 0，右边标记为 1.
- 码字通过连接从根节点到叶子节点的路径上的位标签（$0, 1$ 组合）生成.

字母 σ 的码字长度 $\text{len}(\sigma)$ 等于其叶子节点在代码树中的层级 $\text{level}(\sigma)$.

> 即高度 / 深度，往下几层 $\text{level}(\sigma)$ 就等于几.

平均长度可以表示为代码树的平均高度：

$$\sum_{\sigma \in \Sigma} \text{freq}(\sigma) \cdot \text{level}(\sigma)$$

（ex-05-Problem 3）根据 code tree 的定义，证明：

- The encoding produces by a code tree T is a prefix code
- Every prefix code f is produced by a code tree T

> 这里证明的是每个前缀码可以由一个码树 T 产生，只需要证明生成 f 的 T 的存在性 即可.

Solution:

① The encoding produces by a code tree T is a prefix code

如果 codeword of σ_1 is a prefix of the codeword of σ_2，根据码字的生成方式，可知 σ_1 在 T 中是 σ_2 的祖先（σ_1 会在生成 σ_2 的路径上），但这不可能发生，因为 T 的定义要求 σ_1 是 T 的叶子节点. 因此没有码字是另一个码字的前缀，该编码是 prefix code.

② Every prefix code f is produced by a code tree T

设 $S = \{f(\sigma) | \sigma \in \Sigma\}$ 是所有码字的集合. 对所有的码字 $f(\sigma)$ 按照下面的规则生成一棵二叉树 T：

- Initially, set u to the root of T
- Repeat the following until u is a leaf node
  - Let l be the level of u

    - 如果 $f(\sigma)$ 的第 l 位是 0，则尝试下降到 u 的 left child v. 如果 $f(\sigma)$ 的第 l 位是 1，则尝试下降到 u 的 right child v. 如果孩子 v 不存在，则在 T 中创建它，并用 $0/1$ 标记连接 u, v 的边.
    - 把 v 设置成新的 u（来决定下一步往哪走）
- Mark the leaf node u with the letter σ（标记为这个码字对应的字母）.

The final T is a code tree that generates f.

> 这里没有考虑最优的问题，因为这是 Σ上任意的前缀码 f.

8.2 前缀编码问题

给定字母表及其每个字母的频率，找到一个具有**最短平均长度**的**前缀码**.

该问题可以重述为：找到一棵代码树 T，使得 $\sum_{\sigma\in\Sigma}\mathrm{freq}(\sigma)\cdot\mathrm{level}(\sigma)$（加权平均深度，即前缀码的加权平均长度）最小.

8.3 Huffman 编码

> Huffman Coding

Huffman 编码是前缀编码问题的解决方案. 广泛应用于数据压缩.

贪心思想：自底向上构建代码树，始终合并两个频率最低的根节点，并把它们频率之和标记在新的根节点. 直到所有字母节点都被纳入一棵树. 这个代码树的根节点标签为 100%.

8.3.1 算法步骤

初始化：创建一个包含 $n=|\Sigma|$ 个独立叶子节点的集合 S，每个叶子节点对应一个字母及其频率.

重复步骤直到 $|S|=1$:

- 从 S 中移除两个频率最小的节点 u_1 和 u_2.

  > 这里 u_1 和 u_2 是两个树.

- 创建一个新节点 v，将 u_1 和 u_2 作为其子节点.

  > u_1 和 u_2 各自的子节点仍保持不动.但是在 S 中只保存树的根节点 v.

- 设置 v 的频率为 u_1 和 u_2 的频率之和.
- 将 v 添加到 S 中.

当 $|S|=1$ 时（一棵树），剩下的节点即为最终的代码树的根节点，由此导出的前缀码即为 Huffman 码.

8.3.2 正确性证明：归纳

> Huffman Coding: Proof of Correctness

性质 1：In an optimal code tree, every internal node of T must have two children.

证明（反证法）：

(ex-05-Problem 4): Let T be an optimal code tree on an alphabet Σ (i.e., T has the smallest average height among all the code trees on Σ). Prove: every internal node of T must have two children.

假设 T 中存在一个内部节点 u 只有一个孩子 v. 设 p 是 u 的父节点，即 $p \xrightarrow{1/0} u \xrightarrow{1/0} v$. 直接删掉 u，然后添加 $p \xrightarrow{1/0} v$（和原来的 $u \to v$ 一样标签，即不改变所有字母的码字）. 得到的新树 T' 是一个合法的码树，且生成的前缀码比原先更优（易得，因为所有以 v 为根的子树节点，包括 v 在内，深度都减少了 1）. 和 T 最优的题设相矛盾.

特殊情况：若 u 为根，直接删掉 u 然后把 v 设为新根.

性质 2：Let σ_1 and σ_2 be two letters in Σ with the lowest frequencies. There exists an optimal code tree where σ_1 and σ_2 have the same parent.

> 不是说最优码树中它们一定有共同父节点，而是它们有共同父节点的最优码树是存在的.
>
> 我们要证明的是最优性，而不是唯一性，Huffman 不一定生成唯一的最优解. 其他最优码树可能不满足 σ_1, σ_2 有共同父节点.

证明（反证法）：假设有一个最优代码树 T，其中频率最低的 σ_1 和 σ_2 没有共同父节点.

引理：任何最优代码树 T 中，频率最低的两个字母必须位于树的最深层（易证，假设有一个更深的字母，和这两个中的一个对调，都能构成更优的码树）.

考虑 σ_1 和 σ_2 没有共同父节点的最优码树 T. σ_1 和 σ_2 位于最深层，都是叶子节点. 设 σ_1 的父节点为 p_1，σ_2 的父节点为 p_2. 且 $p_1 \neq p_2$.

由于 σ_1 和 σ_2 位于最深层，它们的 sibling 也是叶子节点. 设 σ_1 的兄弟为 σ_x，设 σ_2 的兄弟为 σ_y.

交换 σ_1 和 σ_y，构建新树 T'，平均长度不变，仍然最优. 新树中 σ_1 和 σ_2 有共同父节点.

> 交换论证，我们没有证明假设是错的，而是构造了一个新树，证明存在性.

证明：霍夫曼算法产生一个**最优**的前缀码（生成加权平均深度最小的代码树）.

We will prove by induction on the size n of the alphabet Σ

Base Case: $n = 2$

In this case, Huffman's algorithm will encode one letter with 0, and the other with 1. This is clearly optimal.

Induction Step: Assuming the theorem's correctness for $n = k - 1$ where $k \geq 3$. next we show that it also holds for $n = k$

定义一些 notations：

Notations corresponding to $n = k$

- Σ: an alphabet of size k. Let σ_1 and σ_2 be two letters in Σ with the lowest frequencies.
- T: an optimal code tree on Σ, where leaves σ_1 and σ_2 have the same parent p（由性质 2 得）
- T_{huff}: output of Huffman's algorithm on Σ (in binary-tree format)

Notations corresponding to $n = k - 1$

> 我们假设的是 $n = k - 1$ 时任意字母表任意频率分布 Huffman 编码都能实现最优. 因此这里可以自己构造.

- Σ': an alphabet constructed from Σ by removing σ_1 and σ_2, and adding a letter σ^* with frequency $\text{freq}(\sigma_1) + \text{freq}(\sigma_2)$
- T': the tree obtained by removing leaves σ_1 and σ_2 from T (thus making p a leaf)

  > 不知道 T' 是不是最优. 节点 p 代表字母 σ^*

- T'_{huff}: output of Huffman's algorithm on Σ'

目标：to prove that

$$\text{avg height of } T_{\text{huff}} \leq \text{avg height of } T$$

Facts about T_{huff} and T'_{huff}

① σ_1 and σ_2 have the same parent in T_{huff}, and T'_{huff} is the tree obtained by removing leaves σ_1 and σ_2 from T_{huff}

② T'_{huff} is the optimal prefix code tree on Σ'（归纳假设）.

It follows from ① that

$$\text{avg height of } T_{\text{huff}} = \text{avg height of } T'_{\text{huff}} + \text{freq}(\sigma_1) + \text{freq}(\sigma_2)$$

> 因为 σ_1 和 σ_2 的深度相比 σ^* 增加了 1. 其他节点都不变. T_{huff} 的建树过程只多了第一步的合并 σ_1 和 σ_2

It follows from the definition of T' that

$$\text{avg height of } T = \text{avg height of } T' + \text{freq}(\sigma_1) + \text{freq}(\sigma_2)$$

> 因为 σ_1 和 σ_2 的深度相比 p 增加了 1. 其他节点都不变.

It follows from ② that

$$\text{avg height of } T'_{\text{huff}} \leq \text{avg height of } T'$$

The above three inequalities imply:

$$\text{avg height of } T_{\text{huff}} \leq \text{avg height of } T$$

> 已经证毕. 但可以进一步得到，T' 在 Σ' 上也是最优的.

8.3.3 复杂度分析

该算法可以在 $O(n \log n)$ 时间内实现.

(sp-ex-05 Problem 3) Describe how to implement Huffman's algorithm to ensure a worst-case time complexity of $O(n \log n)$, where n is the size of the alphabet Σ.

Solution: 使用一个最小优先队列 Q. 队伍中存储的是当前的节点（最初是叶子节点，之后是合并后的内部节点）. 节点的优先级是它的频率.

操作：

- 对于字母表 Σ 中的每个字母 σ，创建一个叶子节点，并将其频率设为 $\mathrm{freq}(\sigma)$
- 将这 n 个节点全部插入到最小优先队列 Q 中.
- 插入 n 个元素到最小堆中，总时间复杂度为 $O(n \log n)$

> 最小优先队列是抽象数据结构，实现它通常使用最小堆这个实际的数据结构.
>
> 最小堆：完全二叉树，父节点值总是小于等于子节点值（根节点最小）

合并过程：重复执行，直到 Q 只剩下一个节点为止

- 提取最小：从 Q 中取出频率最小的两个节点 u_1 和 u_2
- 创建新节点：创建一个新的内部节点 v，将 u_1 和 u_2 作为它的左右子节点.
- 计算频率：将新节点 v 的频率设置为 v_1 和 v_2 频率之和.
- 插入新节点：将新节点 v 重新插入到队列 Q 中.

> 最小堆维护的是所有独立树的根节点（初始 n 个，最后 1 个）
>
> Huffman 树需要另外保存，和这个最小堆无关.

时间复杂度分析：

循环总共执行 $n-1$ 次，单次循环时间复杂度为 $O(\log k)$，其中 k 是队列中当前的元素数量. 循环从 $k=n$ 到 $k=2$，总复杂度为

$$\sum_{k=2}^{n} O(\log k) = O(n \log n)$$

这里用到了斯特林近似（Stirling's Approximation），对于大 n

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

代入求和项，

$$\log(n!) \approx \log\sqrt{2\pi n} + n\log\frac{n}{e} = O(n \log n)$$

Lec 9 动态规划

9.1 动态规划

> Dynamic Programming 1: Pitfall of Recursion

9.1.1 递归的陷阱

例题

给定一个包含 n 个正整数的数组 A.

目标：计算函数 $f(n)$ 的值. 其中 $f(k)$ 定义为

$$f(k) = \begin{cases} 0 & \text{if } k = 0 \\ \max_{i=1}^{k}(A[i] + f(k-i)) & \text{if } 1 \leq k \leq n \end{cases}$$

这是递归定义，如果直接计算，运行时间为 $\Omega(2^n)$.

证明：

1. 建立运行时间的递归关系式

设 $T(k)$ 为计算 $f(k)$ 所需的时间

- 当 $k = 0$ 时，算法直接返回，耗费常数时间 $T(0) = c$
- 当 $k > 0$ 时，算法执行一个从 1 到 k 的循环. 在第 i 次迭代中，它会调用 $f(k-i)$

  因此，计算 $T(k)$ 的总时间可以表示为：

  $T(k) = \sum_{i=1}^{k} T(k-i) + C$

  （其中 C 是循环内部进行加法和比较操作的常数时间）

2. 通过代换展开关系式

为了观察增长趋势，我们对比 $T(k)$ 和 $T(k-1)$：

- $T(k) = T(k-1) + T(k-2) + \cdots + T(0) + C$
- $T(k-1) = T(k-2) + T(k-3) + \cdots + T(0) + C$

将第二个等式代入第一个等式中，可以发现：

$$T(k) = 2T(k-1)$$

3. 求解通项公式

通过上述递推关系 $T(k) = 2T(k-1)$，我们可以得到：

- $T(1) = 2T(0)$
- $T(2) = 2T(1) = 2^2T(0)$
- $T(3) = 2T(2) = 2^3T(0)$
- 以此类推，得出 $T(n) = 2^n \cdot T(0)$。

由于 $T(0)$ 是一个正常数，因此该递归算法的时间复杂度为 $\Omega(2^n)$

4. 直观理解：递归树的冗余

这种指数级增长的原因在于**冗余计算**. 例如在计算 $f(4)$ 时：

- 为了计算 $f(4)$，需要调用 $f(3), f(2), f(1), f(0)$
- 在随后的 $f(3)$ 调用中，又会再次完整地调用一遍 $f(2), f(1), f(0)$
  这种重复访问相同子问题的行为，导致了调用次数随 n 的增加呈爆炸式增长

原因：重复计算了相同的子问题.

陷阱：如果相同的子问题被反复遇到，递归算法会进行大量的冗余工作.

9.1.2 动态规划

为了获得更好的解决方案，需要满足以下条件：

- 可以存储子问题的解，以便后续再次使用；
- 子问题以特定顺序出现，允许使用较早子问题的解来解决较晚的子问题；
- 可以有效地确定（计算）这个特定顺序.

> 例如升序或降序.

这样的方案就是动态规划. 可以看作是递归的 “升级版” 或 “记忆版”.

核心原则：按照特定顺序解决子问题，并记住每个子问题的输出以避免重复计算.

例题中，按照 $f(1), f(2), \ldots, f(n)$ 的顺序计算. 每次计算后保存结果. 如果后续有需要可直接调用，仅 $O(1)$ 复杂度.

总复杂度分析：

- 计算 $f(k)$ 需要 $O(k)$ 时间，因为它依赖于 $f(1), \ldots, f(k-1)$.

  > 注意，这并不代表计算 $f(n)$ 只需要 $O(n)$ 的总时长，因为在动态规划中，要算 $f(n)$ 要先算出 $f(1), f(2), \ldots, f(n-1)$. 当前面的都算完了，最后一步算 $f(n)$ 确实只需要 $O(n)$.

- 总运行时间：$\sum_{k=1}^{n} O(k) = O(n^2)$.

  > 比递归算法快得多.

9.2 切杆问题

> Dynamic Programming 2: Rod Cutting

一根长度为 n 的杆子，以及一个长度为 n 的价格数组 P，其中 $P[i]$ 是长度为 i 的杆段的价格.

目标：将杆子切分成整数长度的段，以使总收入最大化.

穷举法：存在 $O(2^n)$ 种切割方法，太大.

动态规划思想

- 设计子问题：求解更小尺寸杆子的切割方案.
- 确定递归关系：
  - 定义 $opt(n)$ 为切割长度为 n 的杆子所能获得的最大收入.
  - 明确 $opt(0) = 0$.
  - 考虑第一次切割：第一次切割可以选择产生一个长度为 i 的杆段，价格为 $P[i]$，剩下的长度为 $n-i$ 的杆段以最优方式切割，获得 $opt(n-i)$ 的收入.
  - 第一次切割有 n 种可能的选择（i 从 1 到 n），因此递归关系为 $opt(n) = \max_{i=1}^{n}(P[i] + opt(n-i))$
- 计算顺序：$opt(0), opt(1), \ldots, opt(n)$

所有题设都和 `9.1 动态规划` 中的例题一致.

计算出 $opt(n)$ 的总复杂度：$O(n^2)$

9.2.1 Piggyback

除了计算最大收入，我们还关心实现该收入的切割方法（很切合实际）. 由此引入附带（Piggyback）技术.

- 定义 $bestSub(n)$ 为使 $opt(n)$ 达到最大时的第一次切割长度.
- 在每次计算 $opt(i)$ 时顺便用 $O(1)$ 的时间记录 $bestSub(i)$.
- 一旦 $bestSub(i)$ 全部记录完毕（i 从 1 到 n），重构最优切割方案只需要 $O(n)$ 的时间.

> 从长度 n 开始，最优切割的第一段长度是 $k_1 = bestSub(n)$.
>
> 剩下的杆子长度是 $n - k_1$.
>
> 第二段长度是 $k_2 = bestSub(n - k_1)$.
>
> 不断重复这个过程，直到剩余长度为 0.
>
> 在这个过程中，总共切割的长度之和是 n，只需要 $O(n)$ 步就可以找出所有的最优切割段.

9.3 最长递增子序列问题

> Longest Increasing Subsequences

- **任务**：给定一个包含 n 个整数的数组 $A[1...n]$，找到其最长严格递增子序列的长度.
- **特征**：子序列中的元素在原数组中不需要是连续的，但必须保持原有的相对顺序.

  > 即抽出一部分数字，按相同顺序排列得到子序列.

- **暴力解法**：尝试所有可能的子序列并检查其递增性. 这种方法的时间复杂度为 $O(2^n)$，对于大数据集不可行

9.3.1 动态规划

设 $dp[i]$ 表示以 $A[i]$ 结尾的最长递增子序列的长度.

递推关系（Recurrence Relation）：

$$dp[i] = 1 + \max(dp[j]) \text{ for all } j < i \text{ and } A[j] < A[i]$$

注意，$j < i$ 和 $A[j] < A[i]$ 要同时满足. 如果不存在这样的 j，则 $dp[i] = 1$.

理解：当 $j < i$ 且 $A[j] < A[i]$ 时，可以将 $A[i]$ 附加到以 $A[j]$ 结尾的 LIS 之后. 遍历所有符合条件的 j 取最大，得到以 $A[i]$ 结尾的最长递增子序列的长度.

最终答案：整个数组的 LIS 长度是所有 $dp[i]$ 中的最大值，即 $\max_{i \in \{1,...,n\}}\{dp[i]\}$

> 注意，$dp[i]$ 不一定随 i 递增，所以要取最大，而不是看最后一个.

复杂度：该算法的时间复杂度为 $O(n^2)$

> $O(1) + O(2) + \cdots + O(n) = O(n^2)$

9.4 进阶 - 依赖关系

当递归式很复杂的时候，我们可能需要画图来寻找依赖关系（dependency relationships）.

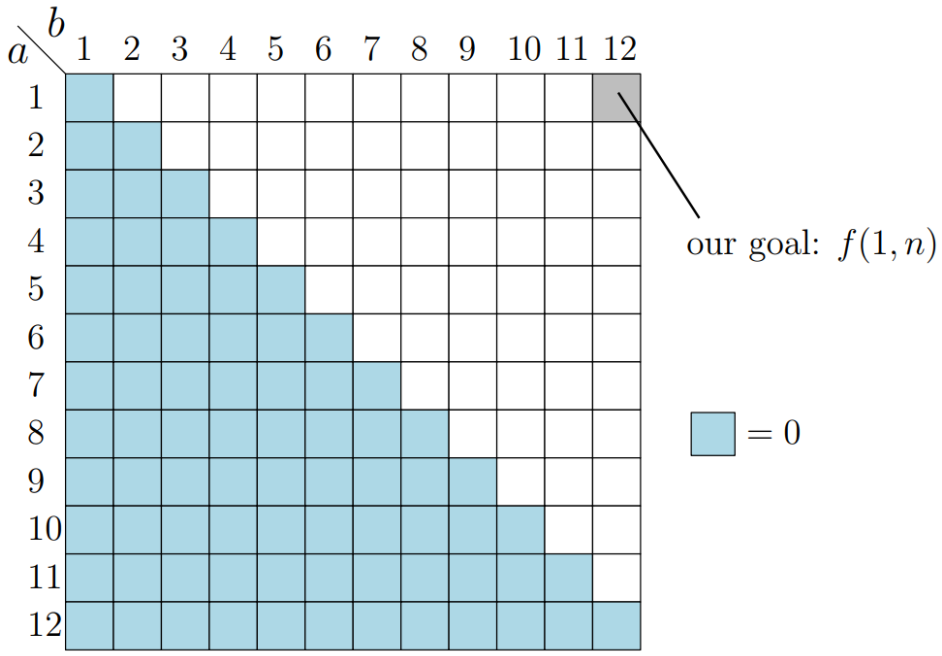
> 见 `9.7 依赖关系图`

例：Let A be an array of n integers. Define function $f(a, b)$ —— where $a \in [1, n]$ and $b \in [1, n]$ —— as follows

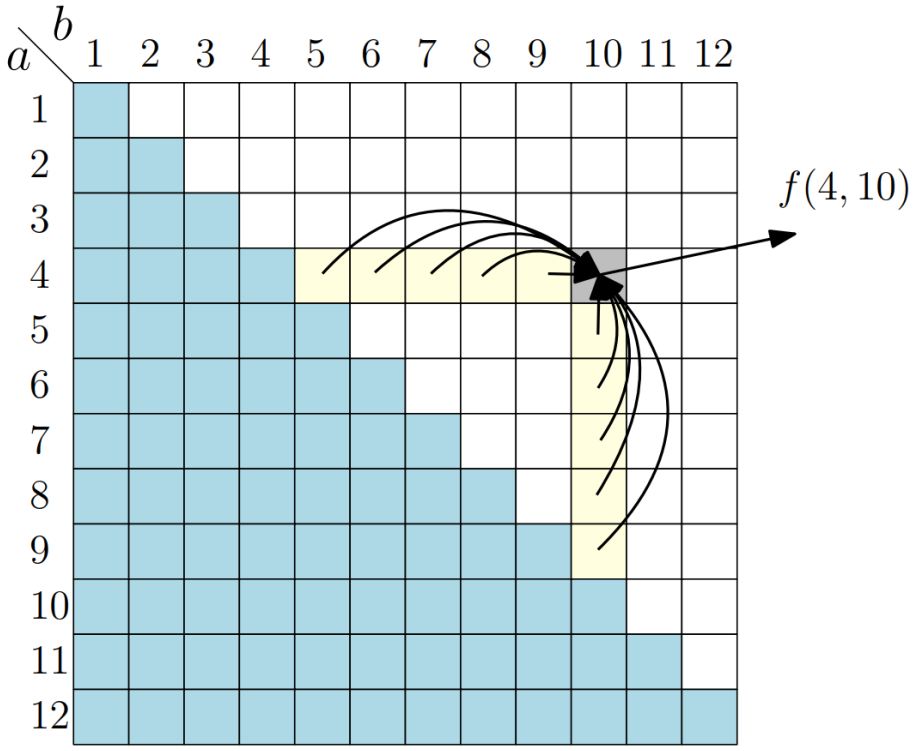
$$f(a,b) = \begin{cases} 0 & \text{if } a \geq b \\ (\sum_{c=a}^{b} A[c]) + \min_{c=a+1}^{b-1} \{f(a,c) + f(c,b)\} & \text{otherwise} \end{cases}$$

Design an algorithm to calculate $f(1, n)$ in $O(n^3)$ time.

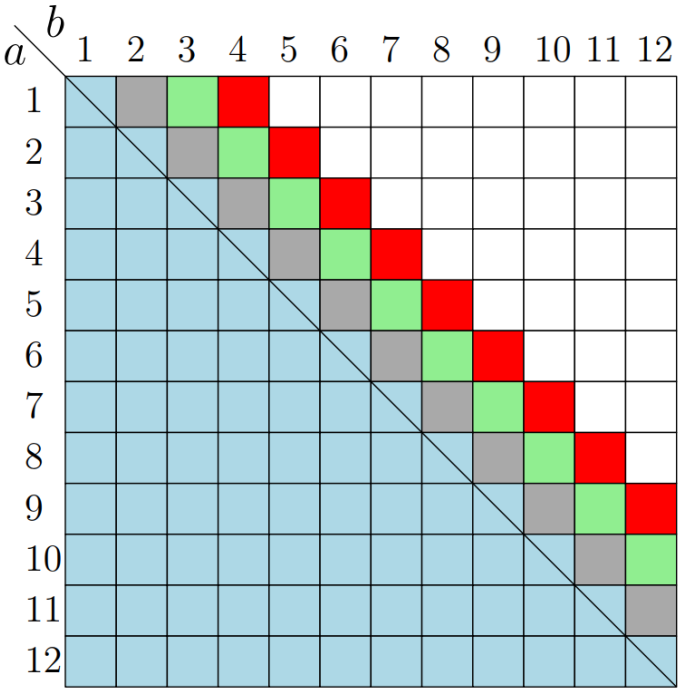
依赖关系图：



每个单元格的值只依赖正左侧和正下方：



逐层向右上角填充：



复杂度分析:

The order can be summarized as follows.

- All cells $f(a, b)$ with $b - a = 1$, each computed in $O(1)$ time.
- All cells $f(a, b)$ with $b - a = 2$, each computed in $O(2)$ time.
- ...
- All cells $f(a, b)$ with $b - a = k$, each computed in $O(k)$ time.
- ...
- All cells $f(a, b)$ with $b - a = n - 1$, each computed in $O(n - 1)$ time.

There are $O(n^2)$ values to calculate.
Total time complexity $= O(n^3)$.

9.5 编辑距离问题

> 这是 Week 7 的第一个讲义.

给定两个字符串 $A = a_1a_2 \cdots a_m$ 和 $B = b_1b_2 \cdots b_n$，计算将 A 转换成 B 所需的最少操作次数.

允许的操作:

- 插入一个字符
- 删除一个字符

- 替换一个字符

动态规划

设 $D(i,j)$ 是字符串 A 的前缀 $A[1\ldots i]$ 和字符串 B 的前缀 $B[1\ldots j]$ 之间的编辑距离. 最终目标是计算 $D(m,n)$.

边界条件:

- $D(i,0)=i$，删掉 $A[1\ldots i]$ 的所有字符
- $D(0,j)=j$，插入 $B[1\ldots j]$ 的所有字符

递归关系：对于 $i\neq 0$ 且 $j\neq 0$，编辑距离的完整递归公式为

$$D(i,j)=\min\{D(i,j-1)+1, D(i-1,j)+1, D(i-1,j-1)+![a_i==b_j]\}$$

其中:

- $D(i,j-1)+1$ 对应插入 b_j 操作.
- $D(i-1,j)+1$ 对应删除 a_i 操作.
- $D(i-1,j-1)+![a_i==b_j]$ 对应匹配或替换操作.

> 若 $a_i==b_j$ 为 true，则不用加；若 $a_i==b_j$ 为 false，则加 1（a_i 替换为 b_j）.

Pseudo-code

Input: Strings $A[1\ldots m]$, $B[1\ldots n]$
Initialize:

$$D[0][j]\leftarrow j \quad \text{for } j=0\ldots n$$
$$D[i][0]\leftarrow i \quad \text{for } i=0\ldots m$$

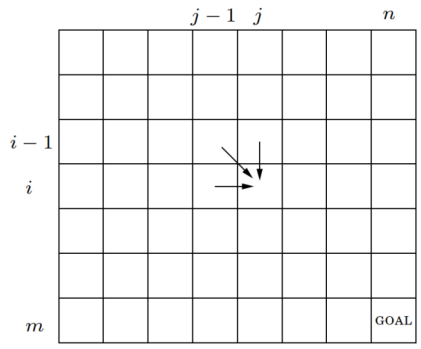
Fill table:

- For $i=1$ to m:
  - For $j=1$ to n:
    - If $A[i]=B[j]$, set $D[i][j]\leftarrow D[i-1][j-1]$
    - Else,

$$D[i][j]\leftarrow 1+\min\begin{cases} D[i-1][j] & \text{/* delete */}\\ D[i][j-1] & \text{/* insert */}\\ D[i-1][j-1] & \text{/* substitute */}\end{cases}$$

9.6 动态规划表

在 `9.5 编辑距离问题` 中，为了计算 $D(i,j)$，需要已知 $D(i-1,j)$，$D(i,j-1)$ 和 $D(i-1,j-1)$ 的解. 这启发我们使用一个二维动态规划表（Dynamic Programming Table）. 只要按合适的顺序计算并填充，就可以在 $O(mn)$ 时间内解决问题.



$A = a_1 a_2 \dots a_m$, $B = b_1 \dots b_n$

$$D(i,j) = \min\left\{D(i,j-1)+1,\ D(i-1,j)+1,\ D(i-1,j-1)+[a_i == b_j]\right\}$$

Fill the table following the suggested order. We can solve the problem **in time** $O(mn)$.

Piggybacking: 在 DP Table 的每个格子 (i, j)，除了计算并记录值 $D(i, j)$，同时记录达到 $D(i, j)$ 的最后一步所进行的操作.

然后就可以逐步回溯到 $D(0, 0)$，找到完整的编辑方式.

> 可以用箭头表示操作，例如对角箭头向左上表示匹配或替换，向上表示删除，向左表示插入.

9.7 依赖关系图

> Dependency Graph of Dynamic-Programming Subproblems
>
> Week 7 第二个课件的前半部分.

动态规划之所以有效，是因为子问题可以按一个 “良好顺序” 求解，即先前子问题的解有助于后续子问题.

为了可视化，引入一个有向图，表示子问题之间的依赖关系.

- 每个节点代表一个子问题.
- 如果求解 sub_2 需要先求解 sub_1（sub_2 依赖于 sub_1），则从 sub_1 到 sub_2 画一条有向边.
- 这个图被称为动态规划子问题的依赖关系图（Dependency Graph）.

找到这样的 “良好顺序” 是很重要的. 通过引入依赖关系图，一些简单的问题可以直接靠直觉得到 ”良好顺序“；对于复杂的依赖关系图，想要在其中找到 "良好顺序"，实际上就转化为了图论的问题.

For Rod Cutting

Recursion: $opt(n) = \max_{i=1}^{n}(P[i] + opt(n-i))$.

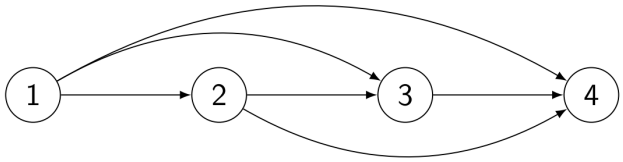


Figure: Exemplary Dependency Graph ($n = 4$)

For $i \in \{1, 2, 3, 4\}$, the i-th node represents the subproblem of computing $opt(i)$.

> Longest Increasing Subsequence 一样

For Edit Distance: the dependency graph

Recursion: $D(i,j) = \min\{D(i,j-1)+1,\ D(i-1,j)+1,\ D(i-1,j-1)+[a_i == b_j]\}$.

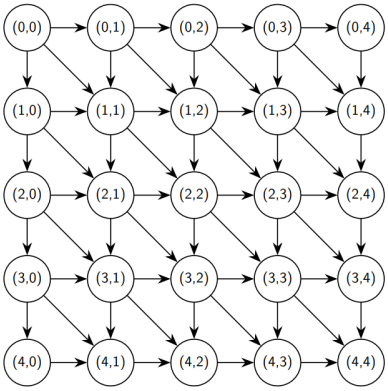


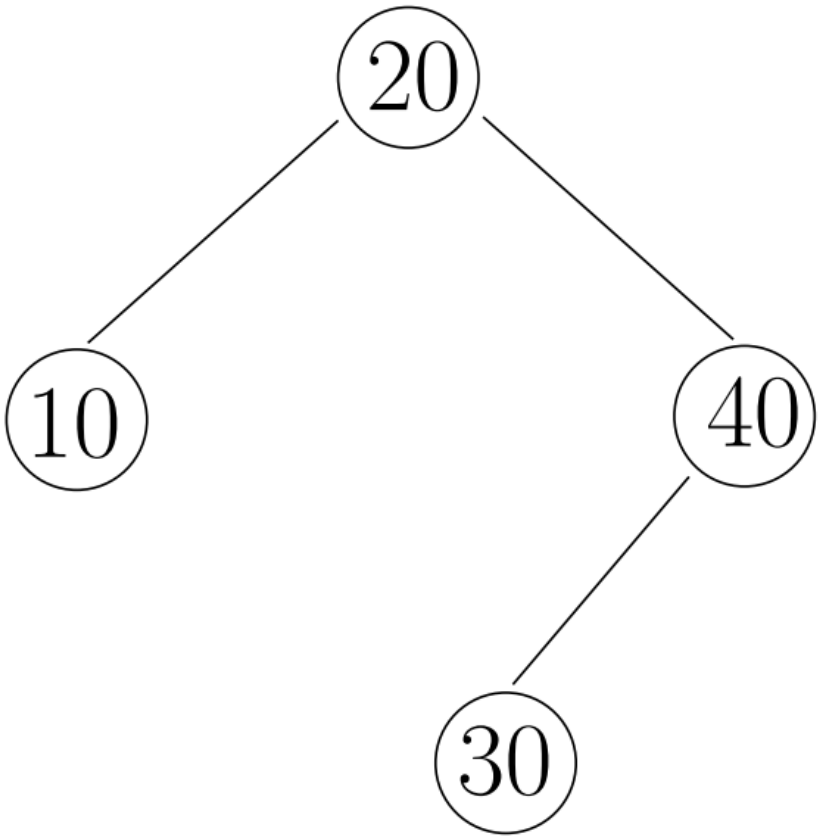
Figure: Exemplary Dependency Graph ($n = 4$)

For $i, j \in \{0, 1, 2, 3, 4\}$, the (i, j)-th node represents the subproblem of computing $D(i, j)$.

进一步讨论：见 `10.1`

9.8 最优二叉搜索树问题

> Week 8
>
> 期中考考的内容.
>
> Dynamic Programming 5: Optimal BST



9.8.1 二叉搜索树

Recall: 二叉搜索树

Each node stores a key. 内部节点 u 的键大于其左子树的任何键，小于其右子树中的任何键.

节点层级（level）： The **level** of a node u in a BST T denoted as $\text{level}_T(u)$ equals the number of edges on the path from the root to u

- The level of the root is 0

The **depth** of a tree is the maximum level of the nodes in the tree.

搜索成本（cost）：搜索节点 u 的成本与 $1 + \text{level}_T(u)$ 成正比，即访问的节点数.

9.8.2 平衡二叉搜索树

> Balanced BST

平衡 BST 是 BST 的一种改进类型，旨在通过限制树的高度来保证操作效率.

问题：一般的 BST 在最坏情况下，如果插入的键值序列是严格递增或递减的，树会退化成一个倾斜的链表，此时树的深度将达到 $O(n)$. 导致查找、插入、删除操作的时间复杂度也退化为 $O(n)$

目标：平衡二叉搜索树通过在每次插入或删除操作后进行旋转和调整，强制保持树的高度尽可能小. 确保树的深度始终保持在 $O(\log n)$ 级别，从而保证所有核心操作（查找、插入、删除）的最坏情况时间复杂度都为 $O(\log n)$

常见的平衡二叉搜索树包括：AVL 树、红黑树.

> 见 `大二 term 1 ESTR 2102 数据结构 7.5`.

9.8.3 最优 BST 问题

Let S be a set of n integers. We have learned that a balenced BST on S has depth $O(\log n)$. This is good if all the integers in S are searched with **equal probabilities**.

> 见 `9.8.2 平衡二叉搜索树`

In practice, not all keys are equally important: some are searched more often than others. This gives rise to an interesting quetion:

If we know the search frequencies of the integers in S, how to build a better BST to minimize the average search cost?

平均搜素成本

$$\text{avgcost}(T) = \sum_{i=1}^{n} W[i]\text{cost}_T(i)$$

where $\text{cost}_T(i) = 1 + \text{level}_T(i)$ is the number of nodes accessed to find the key i in T. $W[i]$ 是数字 i 的搜素频率.

最优 BST 问题

> The optimal BST problem

输入：

- A set S of n integers $\{1, 2, \ldots, n\}$
- An array W where $W[i]$ $(1 \le i \le n)$ stores a positive integer weight（数字 i 的搜素频率 / 权重）.

输出：

A BST T on S with the smallest **average cost**

We will solve a more general version of the problem.

输入：

- S and W same as before.
- Integer a, b satisfying $1 \leq a \leq b \leq n$

输出：

A BST T on $\{a, a+1, \ldots, b\}$ with the smallest average cost

$$\text{avgcost}(T) = \sum_{i=a}^{b} W[i]\text{cost}_T(i)$$

即解决最优子结构：

- 设 $r \in [a, b]$ 是 T 的根节点的键.
- 左子树 T_1 在 $\{a, \ldots, r-1\}$ 上，右子树 T_2 在 $\{r+1, \ldots, b\}$ 上.

引理

Lemma: Let T, T_1, T_2 be defined as above. Then

$$\text{avgcost}(T) = \left(\sum_{i=a}^{b} W[i]\right) + \text{avgcost}(T_1) + \text{avgcost}(T_2)$$

解释：引入新的根节点时，左右子树的所有节点的搜素成本都增加 1（因为必须经过根 r），对平均成本的贡献即增加 $W[i]$.

> 这里还没有考虑最优，只是把 T_1 和 T_2 接在 r 左右组成 T，然后给出它们 average cost 之间的关系.
>
> 当左右两个子树分别取最优，得到的树就是根取 r 时的最优解（但全局最优不一定根取 r）

证明：

$$\begin{aligned}
&\text{avgcost}(T) \\
=&\sum_{i=a}^{b} W[i] \cdot \text{cost}_T(i) \\
=&\sum_{i=a}^{b} W[i] \cdot (1 + \text{level}_T(i)) \\
=&\left(\sum_{i=a}^{b} W[i]\right) + \sum_{i=a}^{b} W[i] \cdot \text{level}_T(i) \\
=&\left(\sum_{i=a}^{b} W[i]\right) + \left(\sum_{i=a}^{r-1} W[i] \cdot \text{level}_T(i)\right) + \left(\sum_{i=r+1}^{b} W[i] \cdot \text{level}_T(i)\right) \\
=&\left(\sum_{i=a}^{b} W[i]\right) + \left(\sum_{i=a}^{r-1} W[i] \cdot (1 + \text{level}_{T_1}(i))\right) + \left(\sum_{i=r+1}^{b} W[i] \cdot (1 + \text{level}_{T_2}(i))\right) \\
=&\left(\sum_{i=a}^{b} W[i]\right) + \left(\sum_{i=a}^{r-1} W[i] \cdot \text{cost}_{T_1}(i)\right) + \left(\sum_{i=r+1}^{b} W[i] \cdot \text{cost}_{T_2}(i)\right) \\
=&\left(\sum_{i=a}^{b} W[i]\right) + \text{avgcost}(T_1) + \text{avgcost}(T_2)
\end{aligned}$$

Define $\mathrm{optavg}(a, b)$ as

- 0, if $a > b$;

  > 空子树，对最终的平均搜索成本不产生贡献.

- the smallest average cost of a BST on $\{a, a+1, \ldots, b\}$, otherwise.

Define $\mathrm{optavg}(a, b|r)$ as the optimal average cost of a BST, on condition that the BST has $r \in [a, b]$ as the key of the root.

By the previous lemma, we have

$$\mathrm{optavg}(a, b|r) = \left(\sum_{i=a}^{b} W[i]\right) + \mathrm{optavg}(a, r-1) + \mathrm{optavg}(r+1, b)$$

The recursive structure of the problem

$$\begin{aligned} &\mathrm{optavg}(a, b) \\ =&\min_{r=a}^{b} \mathrm{optavg}(a, b|r) \\ =&\left(\sum_{i=a}^{b} W[i]\right) + \min_{r=a}^{b} \{\mathrm{optavg}(a, r-1) + \mathrm{optavg}(r+1, b)\} \end{aligned}$$

复杂度分析

With dynamic programming, we can compute $\mathrm{optavg}(1, n)$ in $O(n^3)$ time (left as a special exercise).

Strictly speaking, we have not produced the optimal BST yet. However, fixing the issue should be fairly standard to you at this moment: the `9.2.1 piggyback` technique allows you to build the tree in the same time complexity as computing $\mathrm{opt}(1, n)$. This is left as a special exercise.

Tut 7 动态规划：矩阵链乘法

> Dynamic Programming: Matrix-Chain Multiplication

以下是核心内容总结：

7.1 问题背景

- **计算开销**：给定两个矩阵 A（$a \times b$）和 B（$b \times c$），计算 AB 的时间复杂度为 $O(abc)$

- **乘法结合律的影响**：对于多个矩阵相乘（如 $A_1 A_2 \ldots A_n$），不同的加括号方式（即乘法顺序）会导致计算成本出现巨大差异

- **示例**：计算 $A_1A_2A_3$（维度分别为 $m \times m, m \times m, m \times 1$）：
  - $(A_1A_2)A_3$ 的总开销为 $O(m^3)$
  - $A_1(A_2A_3)$ 的总开销仅为 $O(m^2)$

7.1.1 括号化与完全括号化

1. 什么是括号化?

- **递归定义**：在 A_k 处对矩阵链 $A_1A_2 \dots A_n$ 加括号，会将其转换为两个子表达式的乘积：$(A_1 \dots A_k)(A_{k+1} \dots A_n)$
- **继续递归**：之后，你可以对这两个子表达式分别递归地进行加括号处理

2. “完全括号化产品”的定义

一个乘积被称为是“完全括号化”的，必须满足以下两个条件之一：

- 它是一个**单一矩阵**
- 它是**两个**已经完全括号化的产品的乘积

示例对比（以 $n = 4$ 为例）：

- **属于完全括号化**：$(A_1A_2)(A_3A_4)$ 和 $((A_1A_2)A_3)A_4$。因为它们每一步都清晰地定义了哪两个矩阵或子块先相乘。
- **不属于完全括号化**：$A_1(A_2A_3A_4)$ 6。因为右侧部分 $(A_2A_3A_4)$ 内部没有明确先算哪两个，属于“未完成”状态

3. 为什么这很重要?

- **决定顺序与成本**：不同的括号化方式决定了不同的矩阵乘法顺序，而不同的顺序直接决定了最终的计算开销（Cost）。
- **核心目标**：设计一个算法，在 $O(n^3)$ 的时间复杂度内，找到一个计算成本最小的完全括号化方案

7.2 动态规划算法设计

- **目标**：在 $O(n^3)$ 时间内找到一个最优的完全括号化方案，使总计算开销最小

- **递归结构**：
  - 定义 $cost(i,j)$ 为计算矩阵序列 $A_i \dots A_j$ 的最小成本。
  - 若在 A_k 处拆分，则总成本 = $cost(i,k) + cost(k+1,j) +$ 合并这两部分的开销 $O(a_i b_k b_j)$

- **状态转移方程**：
  - 当 $i = j$ 时，$cost(i,j) = O(1)$（或 0）
  - 当 $i < j$ 时，$cost(i,j) = \min_{i \le k < j} \{cost(i,k) + cost(k+1,j) + O(a_i b_k b_j)\}$

    > 注意，这里 $O(a_i b_k b_j)$ 相对 n 是常数，因此等价于 $O(1)$

二维动态规划，最终算出 cost(1,n) 需要 $O(n^3)$

> 类似 `9.4 进阶-依赖关系` 的例题.

推导矩阵链乘法算法的时间复杂度为 $O(n^3)$，主要可以从**子问题的数量**和**解决每个子问题的开销**这两个维度来看：

1. 子问题的总数

在状态转移公式中，我们需要计算所有的 $cost(i,j)$，其中 $1 \le i \le j \le n$

- 这相当于在 $n \times n$ 的表格中填充上三角部分

- 子问题的个数大约是 $\frac{n(n-1)}{2}$，其数量级为 $O(n^2)$

2. 单个子问题的计算开销

对于每一个特定的 $cost(i,j)$，我们需要尝试所有可能的切分点 k：

- k 的取值范围是从 i 到 $j-1$
- 在最坏的情况下（比如计算 $cost(1,n)$ 时），k 需要遍历约 n 个位置
- 对于每一个 k，执行的是常数次的加法和乘法运算（获取已有的子问题值并加上合并代价 $O(a_i b_k b_j)$）

- 因此，解决一个子问题的时间复杂度为 $O(n)$

3. 总复杂度计算

将上述两部分相乘：

$(\text{子问题数量 } O(n^2)) \times (\text{每个子问题的处理时间 } O(n)) = \mathbf{O(n^3)}$

3. 最优方案的构造

- **Piggyback 技术**：在计算 $cost(i,j)$ 的同时，记录下使成本最小的拆分点 k，定义为 $bestSub(i,j)$
- **递归追溯**：通过 $bestSub$ 表可以追溯出完整的加括号顺序

  - **案例分析**：对于矩阵链 A_1, A_2, A_3, A_4，若 $bestSub(1,4) = 3$ 且 $bestSub(1,3) = 1$，则最优路径为 $(A_1(A_2A_3))A_4$

i \ j	1	2	3	4
1	$O(1)$	$O(m^3)$	$O(m^2)$	$O(m^2)$
2	0	$O(1)$	$O(m^2)$	$O(m^2)$
3	0	0	$O(1)$	$O(m^2)$
4	0	0	0	$O(1)$

$\boldsymbol{A}_1$: $m \times m$
$\boldsymbol{A}_2$: $m \times m$
$\boldsymbol{A}_3$: $m \times 1$
$\boldsymbol{A}_4$: $1 \times m$

Example:
$bestSub(1,4) = 3$, i.e., the best way to calculate $\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3\boldsymbol{A}_4$ is $(\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3)\boldsymbol{A}_4$.

Similarly, $bestSub(1,3) = 1$, i.e., the best way to calculate $\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3$ is $\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3)$.

Therefore, an optimal fully parenthesized product of $\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3\boldsymbol{A}_4$ is $(\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3))\boldsymbol{A}_4$.

4. 复杂度

- 该算法的时间复杂度为 $O(n^3)$

Week 7

Lec 10 拓扑排序

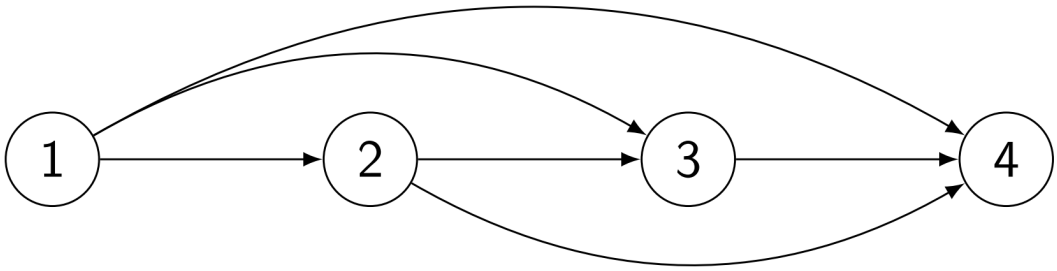
> Topological Sorting

10.1 有向无环图

`9.7 依赖关系图` 中介绍了动态规划问题的依赖关系图. 以三个经典动态规划问题为例：

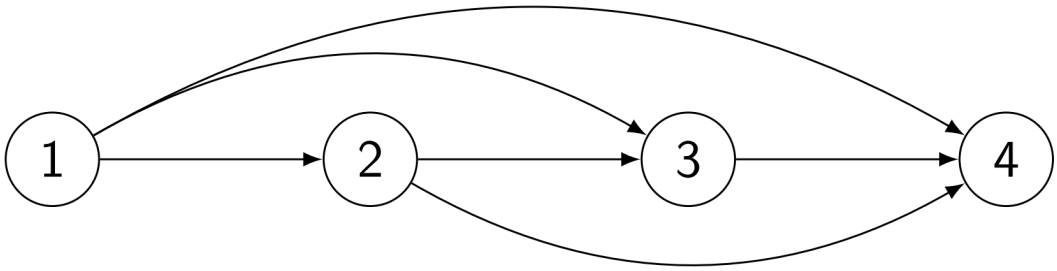
`9.2 切杆问题`：$opt(n) = \max_{i=1}^{n}(P[i] + opt(n-i))$

以 $n = 4$ 为例，依赖关系图为：



`9.3 最长递增子序列问题`：$dp[i] = 1 + \max(dp[j])$ for all $j < i$ and $A[j] < A[i]$

以 $n = 4$ 为例，依赖关系图为：

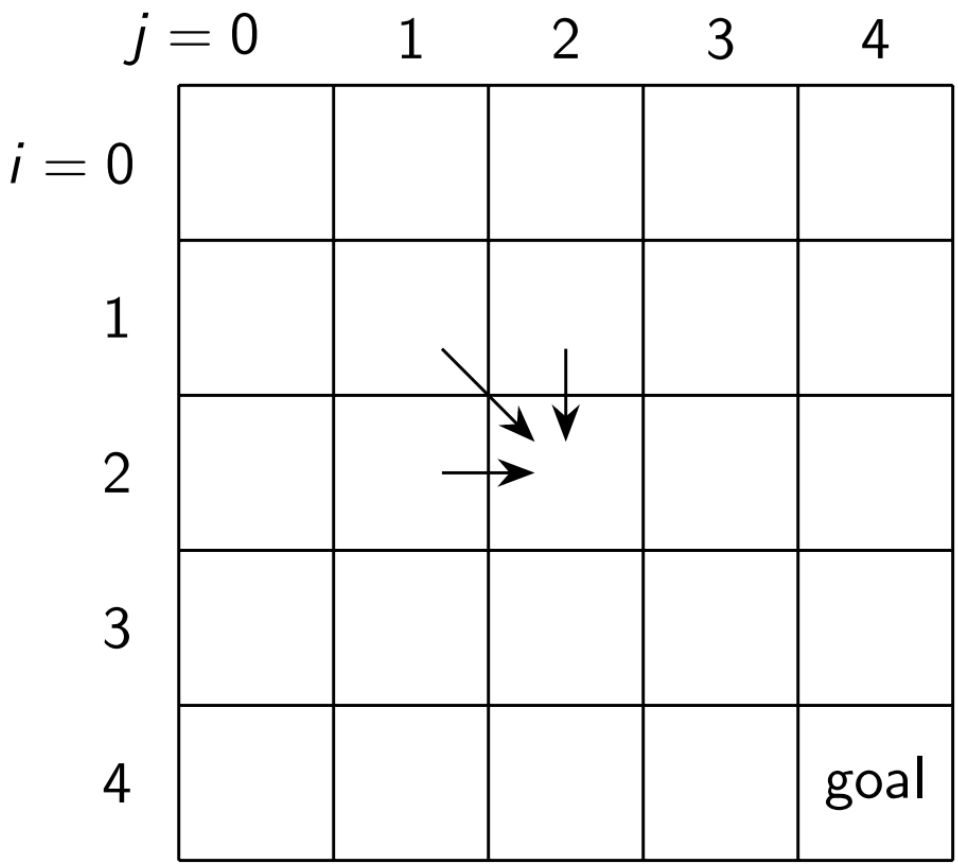


> 和 Rod Cutting 一样.

`9.5 编辑距离问题`：

$$D(i,j) = \min\{D(i,j-1)+1, D(i-1,j)+1, D(i-1,j-1)+![a_i == b_j]\}$$

以 $m = n = 4$ 为例，依赖关系图为：



一个关键特征：这些依赖关系图不包含环.

> 这种无环特性对于高效地找到子问题的 “良好顺序” 是必须的.
>
> ”不含环“ 意味着前面的问题（通常是变量较小、规模较小的子问题）不需要依赖后面的问题（变量较大、规模较大的子问题）. 这确保了 DP 能够从基本情况（Base Case）开始，依次递推或自底向上地解决所有问题.

这样的图叫做有向无环图（Directed Acyclic Graph, DAG）：不包含有向循环的有向有限图.

A **Directed Acyclic Graph (DAG)** is a finite directed graph with no directed cycles. That is, it consists of vertices connected by directed edges (arrows), and it is not possible to start at any vertex and follow a sequence of edges that eventually loops back to the starting vertex.

10.2 拓扑排序

> Topological Sorting

寻找 “良好排序” 可以转化为将依赖关系图（通常为 DAG）的节点排列成一个序列，使得对于每条有向边 $u \to v$，节点 u 在序列中都出现在 v 之前.

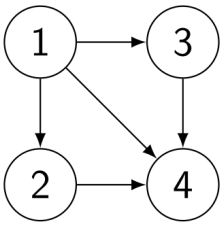
> 即如果 v 的计算要依赖 u，u 必须先于 v 已经算好，这样才是良好的顺序.
>
> 得到符合要求的序列后，直接按照序列的顺序计算子问题.

这种顺序称为拓扑顺序（topological ordering），计算这个顺序的过程称为拓扑排序（Topological Sorting）或线性化（linearization）.

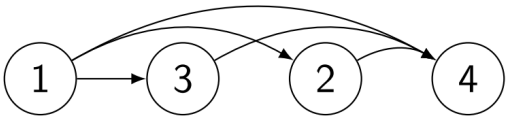
- 拓扑排序可能不唯一.
- 所有 DAG 都可以被线性化 / 被拓扑排序.

10.2.1 拓扑排序可能不唯一

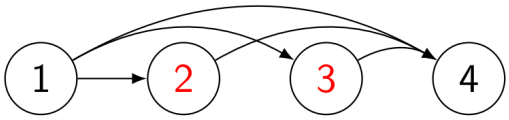
以 `9.5 编辑距离问题` 为例，$m = n = 2$ 时:



(a) Original graph



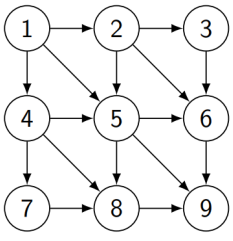
(b) Linearized (i)



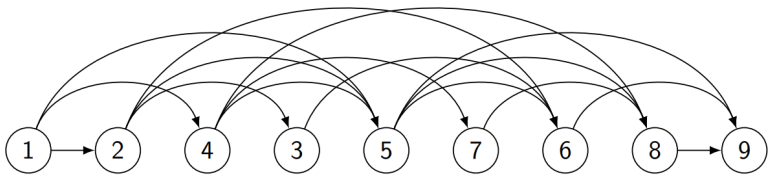
(c) Linearized (ii)

这里，线性化后，先算 2 还是 3 没有影响. 因为它们都只使用了 1.

更大的例子，以 $m = n = 3$ 为例:



(a) Original graph



(b) Linearized (i)

10.2.2 所有 DAG 都可线性化

根据 DAG 的定义和构造性排序的存在性：

① 因为 DAG 不存在环，在一个非空的有限 DAG 中，必然存在至少一个入度（Incoming Degree）为零的顶点（即没有其他顶点指向它的顶点，我们称之为源点）

证明：假设一个非空 DAG 中所有顶点的入度都大于零. 那么对于任何一个顶点 v_1，它都必定有一个前驱 v_0（即存在边 $v_0 \to v_1$）. 重复寻找前驱的过程，由于图是有限的，最终序列中必然会出现重复的顶点，从而成环，和 DAG 的定义矛盾.

② 存在一个构造性排序：DFS 算法

- 对 DAG 执行 DFS 遍历
- 记录每个节点完成（标记为黑）的时间
- 按时间递减的顺序输出所有节点. 即先排入最后完成的.

Algorithm description

- Let $G = (V, E)$ be a directed simple graph.
- 初始化：
- In the beginning, color all vertices in the graph white.
- Create an empty tree T
- Create a stack S, and then
  - Pick an arbitrary vertex v
  - Push v into S, and color it gray
  - Make v the root of T
- 主循环过程：
- Repeat the following until S is empty
  - Let v be the vertex that currently tops the stack S
  - Does v still have a white out-neighbor?

    > 注意，这里是 graph 不一定是树，所以术语是 out-neighbor，而不是 child

  - If yes: let it be u
    - Puth u into S and color u gray

      > u 变成新栈顶.

    - Make u a child of v in the DFS-tree T
  - If no: pop v from S and color v black
- If there are still white vertices, repeat the above by restarting from an arbitrary white vertex v', creating a new DFS-tree rooted at v'

实际上 DFS 有两种实现：

方法一：基于DFS树。运行DFS后，按DFS树标号的降序排列树，并在每棵树内按层级升序输出节点（同层节点从右向左输出）

方法二：基于完成时间。记录每个节点变黑（从栈中弹出）的时间，最后按**完成时间的降序**（即逆序）输出节点

复杂度：两种方法的时间复杂度均为 $O(|V|+|E|)$

> 因为对于相同的 DFS 遍历，两种方法输出的拓扑序列是一模一样的. **方法二其实是在逻辑上对方法一的另一种实现或"验证"**

Lec 11 深度优先搜索 & SCC 问题

> Depth First Search

11.1 算法步骤

Recall DFS:

- Let $G=(V,E)$ be a directed simple graph.

  > simple 指没有自环.

- In the beginning, color all vertices in the graph white.
- Create an empty tree T.
- Create a stack S, and then:
  - Pick an arbitrary vertex v
  - Push v into S, and color it gray
  - Make v the root of T
- Repeat the following until S is empty.
  - Let v be the vertex that currently tops the stack S
  - Does v still have a white out-neighbor?

    If YES, let it be u. Push u into S and color u gray

    Make u a child of v in the DFS-tree T

    If NO, pop v from S and color v black（表示该节点已完成）
- If there are still white vertices, repeat the above by restarting from an arbitrary white vertex v', creating a new DFS-tree rooted at v'

  > 保证所有顶点都被考虑. 允许有多棵树（DFS-forest）.

11.2 复杂度分析

DFS 的复杂度：$O(|V|+|E|)$

其中 $|V|$ 是图中顶点数量，$|E|$ 是图中边数量.

分析：DFS 算法保证对图中的每一个元素（顶点或边）都只进行常数次的操作.

1. 访问顶点：$O(|V|)$

   DFS 算法的核心是通过使用颜色（白、灰、黑）或访问数组来跟踪每个顶点的状态.

   每个顶点 $v \in V$ 只会被访问一次（只会从白色变为灰色一次，从灰色变为黑色一次）

   一旦一个顶点被访问并从栈中弹出（标记为黑），它就不会再被处理.

   即使对于非连通图，DFS 也会通过外部循环确保从每一个未访问的白色顶点开始新的遍历，直到所有 $|V|$ 个顶点都被处理.

   因此，花在处理所有顶点上的总时间正比于顶点数量，即 $O(|V|)$

2. 探索所有边：$O(|E|)$

   对于图中每条有向边 $u \to v$，这条边只会在处理其起点 u 时被检查一次（当 DFS 遍历 u 的邻接列表或邻接矩阵时，会检查 u 的所有出边）. 而检查一条边可能的操作（如检查邻居是否为白色、将其推入栈中、建立父子关系）都是常数时间的操作.

   因此，花在检查所有 $|E|$ 条边上的总时间是正比于边数量的，即 $O(|E|)$

总复杂度：$O(|V| + |E|)$

11.3 基于 DFS 的拓扑排序算法

DFS 可以高效地对 DAG 线性化.

> $O(|V| + |E|)$. 即 DFS 算法本身的复杂度.

11.3.1 方法一

引理 1：在 DFS 执行顺序中，不会有较早的树 T_i 中的任何节点指向较晚的树 $T_m (m > i)$ 中的任何节点的边.

> 否则这条边会被较早的树收录.

If a DFS excution on a DAG $G = (V, E)$ output k DFS trees in order: $(T_1, T_2, \ldots, T_k)$, then there is no (directed) edge (in the original edge set E) from any node of an earlier tree to any node of a later tree.

Proof: If there is a node u in an earlier tree T_i, that has an out-neighbor v, then v must have already been included in

- the current DFS tree T_i, or
- some earlier tree T_m with $m < i$

This is because the only reason that v is not added in T_i before we mark u as being done while building T_i, is that v is already done (i.e., marked black), which means that it must appear in an earlier tree T_m with $m < i$.

引理 2：在每棵 DFS 树 T_i 内部，不会有从较低层级的节点到同分支较高层级节点的边（否则成环，矛盾）.

> 不会有到右侧分支（否则提前收录），可能有到左侧分支（但是）

If a DFS execution on a DAG $G = (V, E)$ output k DFS trees in order: $(T_1, T_2, \ldots, T_k)$, then for each T_i, there is no (directed) edge (in the original edge set E) from a lower-level node to any upper-level node.

> 注意，这里的 lower-level 是较低的层级，即 level 较大的（越深/越低 level 越大）. 后面的 ascending order 指的是从上往下排，即从 upper-level 往 lower-level 排. 虽然是从 upper 到 lower，但是 level 的值的大小是逐渐增大.

Therefore, if we only want to produce a topological ordering of the nodes in Ti , we can simply sort them in ascending order of their levels. (The nodes within each level can appear in any order.)

（根据引理 2，每棵树中，节点已经是拓扑排序了.）

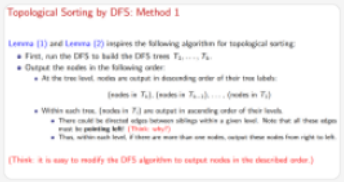
由引理 1 和引理 2，给出基于 DFS 的拓扑排序算法

Method 1：

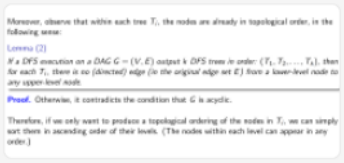
- First, run the DFS to build the DFS trees $T_1, T_2, \ldots, T_k$
- Output the nodes in the following order:
    - At the tree level, nodes are output in descending order of their tree labels: 从 T_k 到 T_1. 因为由引理 2，较晚的树不会依赖较早的树，但是较早的树中的节点可能依赖较晚的树中的节点（即从较晚树节点指向较早树节点的边可能存在，但较晚树建树时，较早树节点已经被标记为黑，所以没有加入较晚树）. 所以从较晚的树开始加入拓扑排序.
    - Within the tree, nodes are output in ascending order of their levels（自顶向下）. 此时同一 level 可能有节点指向 sibling，但是一定能保证这些边都是指向左的（建树时同层从左向右构建）. 因为如果指向右，右邻居在建树时就被加到自己的 child 了. 因为所有可能的同层箭头都向左，所以同层节点从右往左加入拓扑排序.

依赖关系图拓扑排序口诀：DFS，从晚到早，从上到下，从右往左

方法一似乎有些问题：



你觉得这个按晚树-早树、同树上层-下层、同层右-左 的输出顺序合理吗



怎么感觉这个lemma2不太对劲

如果右侧分支指向左侧分支的上层节点不是不会成环吗

同一棵树内应该只能按Method2那样，finishing time反向排序

DFS对DAG生成DFS森林，然后从晚树到早树，同一棵树从上层到下层，同一层从右到左，这样顺序输出等价于按出栈时间的逆序输出吗?

Show thinking

简单直接的回答：不等价。

虽然你在"树与树之间"的顺序（从晚到早）是正确的，但是**在同一棵树内部**，你描述的"分层输出"（从上层到下层）与"DFS出栈时间的逆序"是不一样的。

你描述的方式本质上是一种**BFS（广度优先搜索）的变体，而DFS出栈时间的逆序是一种递归/深度的结构**。

11.3.2 方法二

> Method 2
>
> 比法一简单，只要按照 DFS 弹出（涂黑）的顺序逆向输出.

There is another (only slightly different) DFS-based algorithm for topological ordering.

Input: a DAG G

Output: a list of the nodes of G in topological order.

Algorithm:

- Run a DFS on G. During the traversal, use a counter to record the finishing time of each node -- that is, the time at which the node is popped from the stack (when it is colored black)
- After the DFS completes, output the nodes inn decreasing order of their finishing times.

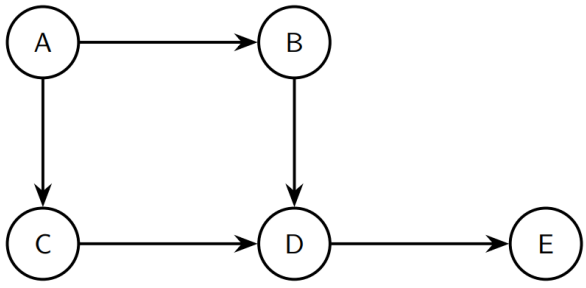
> 从最后涂黑的节点开始. 即 T_k 的根节点.

Time Complexity: $O(|V| + |E|)$ (same as DFS)

正确性证明：遍历过程中，始终满足后遍历的树的节点完成时间更晚，同一棵树中上层节点比下层节点完成时间更晚，同一层中右边节点比左边节点完成时间更晚，因此按时间倒序输出的顺序和方法一相同.

引理 3：Consider an execution of DFS that output trees $(T_1, \ldots, T_k)$. During this execution, in any T_i, a node u was colored black only after all of its descendants in T_i have been colored black; moreover, within any level containing multiple nodes, the nodes were colored black in left-to-right order.

例:



Example traversal:

- Starting node *A*.
- Visit A $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E
- Then, **pop** *E*, **pop** *D*, **pop** *C*
- Backtrack and visit B $\rightarrow$ already visited D. Thus, **pop** *B*.
- Backtrack and **pop** *A*.
- Output order (reverse of finish): A, B, C, D, E

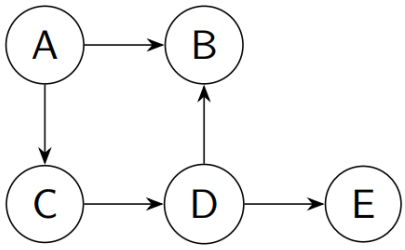
DFS vs BFS

- 拓扑排序是少数几个 DFS 比 BFS 更适合解决的问题之一.
- 普通的 BFS 不适合用于拓扑排序.

Pseudocode for BSF

```
Initialize all nodes as unvisited
Initialize an empty queue

For each node s in the graph :
    If s is unvisited :
        Mark s as visited and enqueue it
        While the queue is not empty :
            Dequeue a node u
            For each unvisited neighbor v of u :
                Mark v as visited
                Enqueue v
```



BFS starting at A:

- Dequeue A; Enqueue B and C
- Dequeue B; /* no out-neighbor */
- Dequeue C; Enqueue D
- Dequeue D; Enqueue E /* B has already been visited */
- Dequeue E

很多情况下，普通的 BFS 无论使用入栈顺序还是出栈顺序、正向还是反向，都无法实现拓扑排序.

Suppose we run BFS starting from A:

- Enqueue A;
- Dequeue A; Enqueue C
- Dequeue C;
- Enqueue B; /* the queue is empty but B is not visited */
- Dequeue B

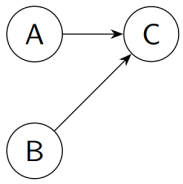


Figure: An example DAG

Why is this invalid?

- Enqueuing order? (A, C, B) /* invalid topological order */
- Dequeuing order? (B, C, A) /* invalid topological order */

Note that what we claimed is that the **plain** BFS is not suitable for topological sorting problem. But there do exist modified version of BFS (or BFS-based) algorithms that could solve this problem. One famous example is **Kahn's algorithm**.

> 没讲.

11.4 白路径定理

> White Path Theorem

Recall DFS:

- Let $G = (V, E)$ be a directed simple graph.

  > simple 指没有自环.

- In the beginning, color all vertices in the graph white.
- Create an empty tree T.
- Create a stack S, and then:
  - Pick an arbitrary vertex v
  - Push v into S, and color it gray
  - Make v the root of T
- Repeat the following until S is empty.
  - Let v be the vertex that currently tops the stack S
  - Does v still have a white out-neighbor?

    If YES, let it be u. Push u into S and color u gray

    Make u a child of v in the DFS-tree T

    If NO, pop v from S and color v black（表示该节点已完成）

- If there are still white vertices, repeat the above by restarting from an arbitrary white vertex v', creating a new DFS-tree rooted at v'

  > 保证所有顶点都被考虑. 允许有多棵树（DFS-forest）.

白路径定理：设 u 是图 G 中的一个顶点，在 u 进入栈的那一刻，另一个顶点 v 将成为 u 在 DFS 森林中的真后代，当且仅当从 u 到 v 存在一条只经过白色顶点的路径（白路径）.

11.5 强连通分量

> Strongly Connected Components, SCC

Let $G = (V, E)$ be a directed graph.

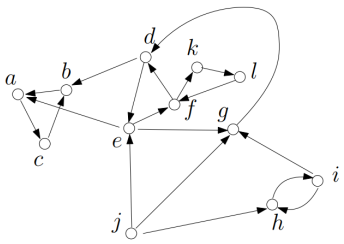
A strong connected component (SCC) of G is a subset S of V s.t.

- for any two vertices $u, v \in S$, graph G has a path from u to v and a path from v to u

  > 只要有 path 即可，不一定要有边 (u, v) 或 (v, u)

- S is **maximal** in the sense that we cannot put any more vertex into S without breaking the above property.

Example



- $\{a, b, c\}$ is an SCC.
- $\{a, b, c, d\}$ is not an SCC.
- $\{d, e, f, k, l\}$ is not an SCC (because we can still add vertex g).
- $\{e, d, f, k, l, g\}$ is an SCC.

11.5.1 性质

Lemma 1：Suppose S_1 and S_2 are both SCCs of G. Then, $S_1 \cap S_2 = \emptyset$

> 任意两个不同的 SCC 不相交（没有公共顶点）

11.5.2 SCC 问题

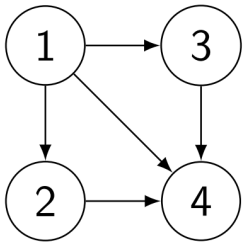
> strongly connected components problem

目标：Given a directed graph $G = (V, E)$, 将顶点集 V 划分为若干个不相交的子集，每个子集都是一个 SCC.

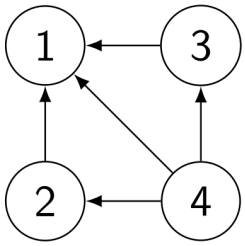
11.6 基于 DFS 的 SCC 算法

> DFS-based Algorithm for SCC

该算法通过两次调用 DFS 解决 SCC 问题. 涉及原图 G 和反向图（reverse graph） G_{rev}



(a) Original graph G



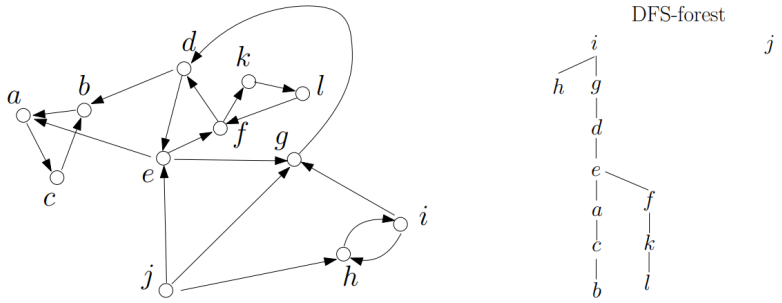
(b) Reverse Graph G^{rev}

11.6.1 算法步骤

Step 1：在 G 上运行 DFS

- 记录顶点从栈中弹出（变黑）的顺序
- 如果顶点 $u \in V$ 是第 i 个变黑的，则给 u 标记标签 i.

Example

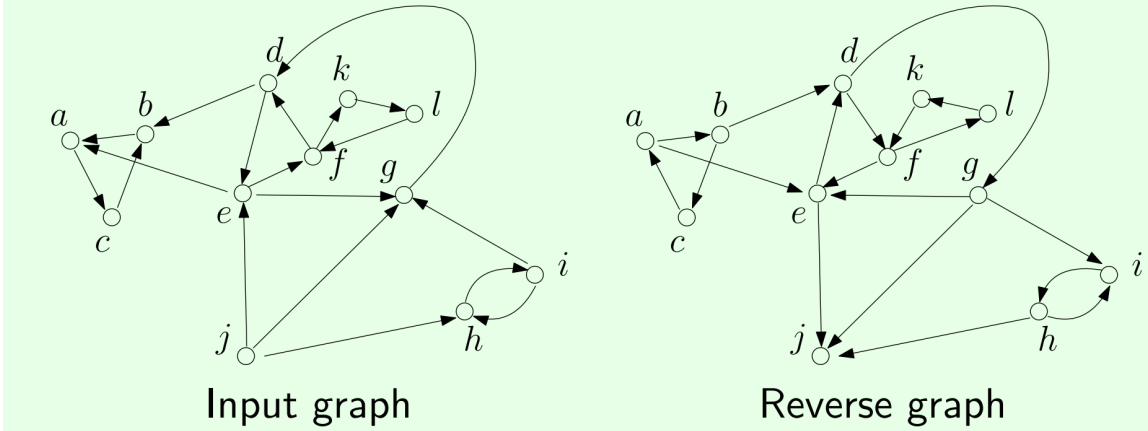


Start DFS from *i* and re-start from *j*.
The following is a possible turn-black order: *h*, *b*, *c*, *a*, *l*, *k*, *f*, *e*, *d*, *g*, *i*, *j*. E.g.:

- The label of *c* is 3.
- The label of *g* is 10.
- The label of *i* is 11.
- The label of *j* is 12.

Step 2：获取反向图 G_{rev}.

- 通过反转 G 中所有边的方向得到 G_{rev}



Step 3：在 G_{rev} 上运行 DFS.

- 规则 1：从具有最大标签的顶点开始 DFS.
- 规则 2：当需要重新启动 DFS 时，从具有最大标签的白色顶点开始.
- 输出：每一个 DFS 树中的顶点集合就是一个 SCC.

11.6.2 复杂度分析

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

原图上运行：$O(|V|+|E|)$

构造反向图（初始化新的邻接表，以及遍历每条边）：$O(|V|+|E|)$

反向图运行 DFS：$O(|V|+|E|)$

> 注意这里不需要像 Dijkstra 一样维护一个二叉堆来找最大标签，因为 DFS 的出栈本身就有时间顺序，而我们的标签就是出栈时间. 实际上只需要再维护一个栈，每次出栈的顶点压入这个栈即可. 最后一个出栈的（最后一棵树的顶点）会在这个栈的栈顶. 往下则是第一轮 DFS 中越来越早出栈的. 栈底是第一棵树的左下角叶子.

总的仍然是 $O(|V|+|E|)$

11.6.3 正确性证明

> Proof of Correctness

定义 SCC graph：

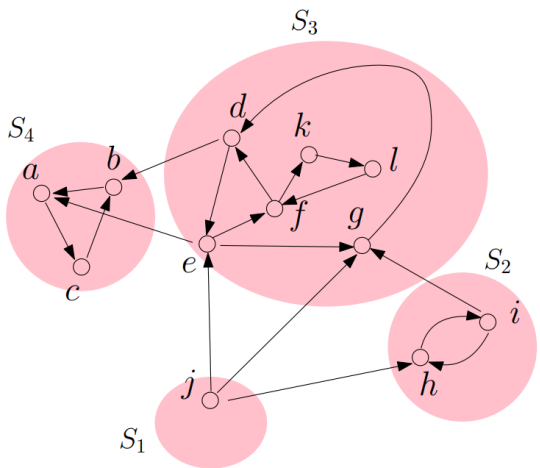
Suppose that the input graph G has SCCs $S_1, S_2, \ldots, S_t$ for some $t \geq 1$.

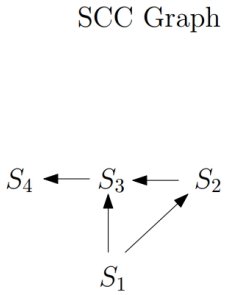
The SCC graph G^{scc} is defined as follows:

- Each vertex in G^{scc} is a distinct SCC in G（类似标签，用 vertex 表示 SCC，并不是说顶点是图）
- 如果 SCC S_i 中顶点有指向 SCC S_j 的边，则 G^{scc} 有边 $S_i \to S_j$

注意：SCC 的定义决定了 G^{scc} 是一个有向无环图（DAG）. 因为如果 SCC 图 G^{scc} 中存在一个循环，如 $S_i \to S_j \to \cdots \to S_i$，意味着 S_i 中的所有顶点都可以到达 S_j 中的所有顶点，且 S_j 中的所有顶点都可以到达 S_i 中的所有顶点. 根据 SCC 的极大性定义（强连通分量是图 G 中具有相互可达性的极大子集），S_i 和 S_j 会被合并成同一个 SCC.

例：





For each SCC $S_i (i \in [1, t])$，定义：

$$\text{label}(S_i) = \max_{v \in S_i} \text{label of } v$$

为 S_i 中顶点的最大标签.

引理：如果在 G_{scc} 中有边 $S_i \to S_j$，则 $\text{label}(S_i) > \text{label}(S_j)$

证明：Let u be the first vertex in $S_i \cup S_j$ that turns gray in DFS (i.e. u is the first vertex in $S_i \cup S_j$ discovered by DFS)

If $u \in S_i$, u has a white path to every vertex in $S_i \cup S_j$（因为有边 $S_i \to S_j$，所以 S_i 中的点可以走到 S_j；又因为 u 是并集中第一个被搜到的，所以其他点此时都是白色）. By the white path theorem:

- u turns black after all the vertices in S_j;
- u is the last vertex in S_i turning black

The above facts imply $\text{label}(S_i) > \text{label}(S_j)$.

If $u \in S_j$, first note that: u has a path to every vertex in S_j（内部强连通），but no path to any vertex in S_i.

> 因为当 $S_i \to S_j$，则一定没有边 $S_j \to S_i$（SCC 是有向无环图），并且一定没有任意能够从 S_j 到 S_i 的路径 $S_j \to \cdots \to S_i$

By the white path theorem, u turns back after all the vertices in S_j and before every vertex in S_i（DFS 从 u 开始探索，并变黑 S_j 中的所有顶点，由于 u 无法到达 S_i 中的任何顶点，因此在u 弹出前 S_i 的顶点都是白色的）

由上述引理，对 t 个 SCC 进行重排，such that：

$$\text{label}(S_1) > \text{label}(S_2) > \ldots > \text{label}(S_t)$$

重排后，有结论：对于任意 $u \in S_i$，如果在 G^{rev} 中有一条边 (v, u) 且 $v \notin S_i$，那么 v 必然属于某个 S_j with $j > i$.

> 注意方向，reverse 后的边和原图相反.

引理：在 G^{rev} 上运行 DFS，第 i 棵 DFS 树中的顶点集合恰好是 S_i

证明（归纳法）：

We will prove the claim by induction on i

Case $i = 1$:

Let u be the vertex in S_1 having the largest label; u is the root of the first DFS-tree. Consider the beginning moment of the first DFS on G^{rev}

- As S_1 is an SCC, u has a white path to every other vertex in S_1

- u has no white path to any vertex outside S_1（因为根据我们的重排规则，原图中只有可能 S_1 指向其他，不可能其他指向 S_1，reverse 后就是只有可能其他指向 S_1）

By the white path theorem, all and only the vertices in S_1 are descendants of u in the first DFS tree. The claim thus holds for $i = 1$

Case $i = k$:

Assuming that the claim holds for $i = k - 1$ (where $k \geq 2$), next we prove its correctness for $i = k$

Let u be the vertex in S_k having the largest label; u is the root of the k-th DFS-tree. Consider the beginning moment of the k-th DFS on G^{rev}.

- All the vertices in $S_1, S_2, \ldots, S_{k-1}$ are black.
- As S_k is an SCC, u has a white path to every other vertex in S_k.
- u has no white path to any vertex $S_{k+1}, S_{k+2}, \ldots, S_t$.

By the white path theorem, all and only（因为前 $k-1$ 棵树的顶点都是黑的，不可能有白路径）the vertices in S_k are descendants of u in the k-th DFS tree. The claim holds for $i = k$.

归纳可得，在 G^{rev} 上运行 DFS，第 i 棵 DFS 树中的顶点集合恰好是 S_i，即基于两次运行 DFS 的算法可以找到所有 SCC.

Tut 8 SCC：进阶

> Further Insights into SCCs

8.1 SCC 问题的定义

给定一个有向图 $G = (V, E)$，目标是将顶点集合 V 划分为互不相交的子集，使得每个子集都是一个强连通分量 3。

8.2 标准 SCC 算法步骤

文档介绍了一个时间复杂度为 $O(|V| + |E|)$ 的高效算法：

- **步骤 1：** 在原图 G 上运行深度优先搜索 (DFS)，并记录每个顶点变黑（完成访问）的顺序，将其标记为标签 i 。

- **步骤 2：** 通过翻转 G 中所有边的方向，获得反向图 G^{rev} 。

- **步骤 3：** 在 G^{rev} 上运行 DFS，遵循以下规则：

- 从标签最大的顶点开始 。

- 需要重启时，从剩余未访问顶点中标签最大的开始 。

- **输出：** 每一个生成的 DFS 树即为一个 SCC 。

8.3 算法的实现细节

- **存储：** 使用数组 A 来记录顶点变黑的顺序，其中 $A[i]$ 存储标签为 i 的顶点 11。

- **反向图构建：** 通过扫描 G 的邻接表，将原图中的边 (u, v) 转换为 G^{rev} 中的边 (v, u)，这一过程同样在 $O(|V| + |E|)$ 时间内完成 12121212。

8.4 为什么使用反向图

文档揭示了该算法的核心逻辑——**收点强连通分量 (Sink SCC)** 策略：

- **概念策略：** 理想情况下，我们应该反复找到一个“收点 SCC”（没有出边的 SCC），运行 DFS 提取它，然后从图中删除它
- **核心难题：** 在原图中直接按顺序寻找“收点 SCC”是非常困难的
- **解决方案：** 在反向图 G^{rev} 上，寻找“收点 SCC”变得非常容易 。通过在反向图上从大标签顶点开始 DFS，算法实际上是按照特定顺序提取原图中的分量，这正是该算法能够正确工作的数学基础

Week 8 期中 & Week 9 停课

无内容.

Week 10

Lec 12 最短路径问题

12.1 SSSP 问题 - 非负权重

> Notation and the SSSP Problem
>
> Single Source Shortest Paths with Non-Negative Weights
>
> 带非负权重的单源最短路径（SSSP）问题

12.1.1 加权图

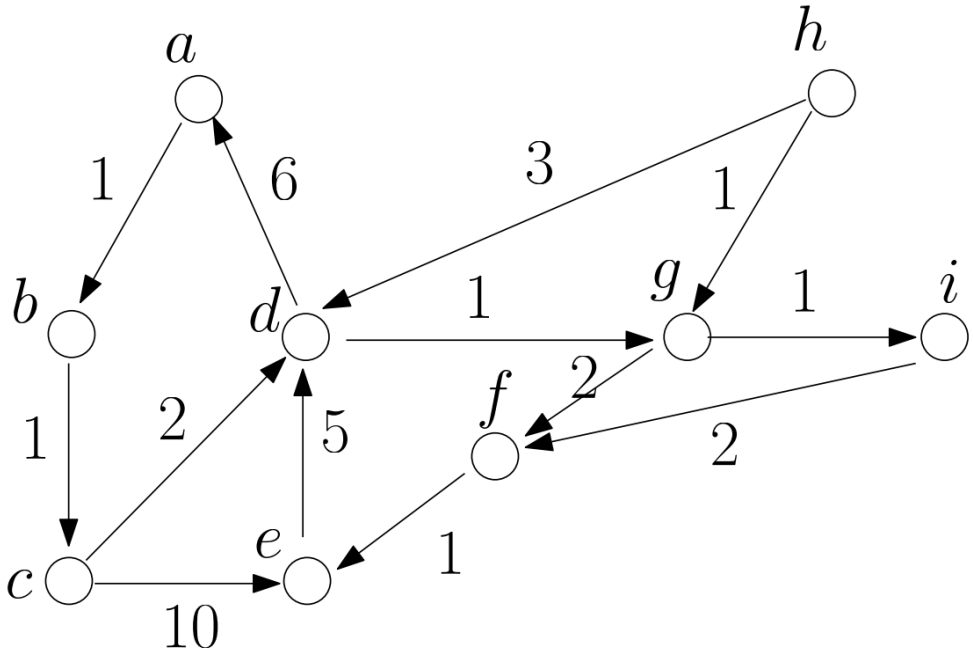
> Weighted Graphs

一个简单有向图 $G = (V, E)$ 上的函数 w 将每条边 $e \in E$ 映射到一个**非负整数** $w(e)$，称为边 e 的权重（weight）

G and w together define a weighted simple directed graph.

simple 意味着没有自环.

Example:



The integer on each edge indicates its weight. For example, $w(d, g) = 1$, $w(g, f) = 2$, and $w(c, e) = 10$.

12.1.2 最短路径

一条路径的长度是所有边权重之和：

Consider a path in G: $(v_1, v_2), (v_2, v_3), \ldots, (v_t, v_{t+1})$, for some integer $t \geq 1$. We define the path's length as

$$\sum_{i=1}^{t} w(v_i, v_{i+1})$$

A **shortest path** from u to v has the minimum length among all the paths from u to v. Denote by $\text{spdist}(u, v)$ the length of a shortest path from u to v.

If v is unreachable from u, $\text{spdist}(u, v) = \infty$

12.1.3 最短路径问题

Problem Statement

Single Source Shortest Path (SSSP) with Non-Negative Weights

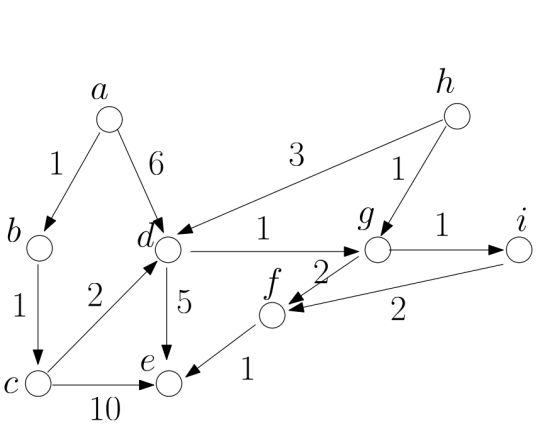
Let $G = (V, E)$ be a simple directed graph, where function $w(\cdot)$ maps every edge of E to a non-negative value. Given a source vertex s in V, we want to find a shortest path from s to t for every vertex $t \in V$ reachable from s.

The output is a shortest path tree T:

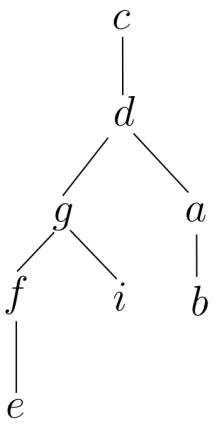
- The vertex set of T contains all vertices reachable from s
- The root of T is s

- For each node $u \in V$, the root-to-u path of T is a shortest path from s to u in G.

Example:



(a) Original graph G



(b) A shortest path tree for source vertex c

12.1.4 Dijkstra 算法

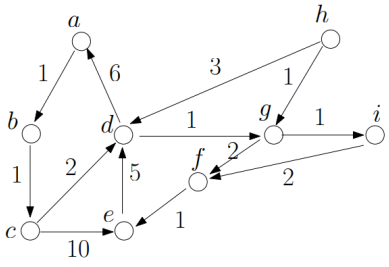
核心思想：维护一个值 $dist(v)$，表示目前找到的从 s 到 v 的最短路径长度.

松弛（Edge Relaxation）：对于边 (u, v)，如果 $dist(u) + w(u, v) < dist(v)$，则更新 $dist(v) \leftarrow dist(u) + w(u, v)$，并设置 $parent(v) \leftarrow u$

> 非负权重可以保证任意时刻，一定不会 relax 之前被 remove 掉的点（在每个点被 remove 的时候，它的 dist 就是最小的，不会有另一个路径更小了）.

Example

Suppose that the source vertex is c.



vertex v	$dist(v)$	$parent(v)$
a	∞	nil
b	∞	nil
c	0	nil
d	∞	nil
e	∞	nil
f	∞	nil
g	∞	nil
h	∞	nil
i	∞	nil

$S = \{a, b, c, d, e, f, g, h, i\}$.
/* S now contains all the vertices, and c is the vertex with the smallest $dist()$. */

算法步骤：

1. 初始化：$dist(s) \leftarrow 0$，其他 $dist(v) \leftarrow \infty$；所有 $parent(v) \leftarrow nil$.

2. 维护一个集合 $S = V$
3. 重复直到 S 为空：从 S 中移除 $dist(u)$ 最小的顶点 u. 松弛所有从 u 出发的边 (u, v). 如果 $dist(v)$ 减少，则 $parent(v) \leftarrow u$

> 这里的 v 只要考虑未被移除的点.

运行时间：Dijkstra 算法可以实现为 $O((|V| + |E|) \cdot \log |V|)$，使用高级数据结构可达 $O(|V| \log |V| + |E|)$

复杂度分析：

常见数据结构：二叉堆

- 提取最小的 $\mathrm{dist}(u)$：重复 $|V|$ 次，每次 $O(\log |V|)$
- 松弛：重复 $|E|$ 次. 若松弛成功，要更新 v 的位置，$O(\log |V|)$
- 总复杂度：$O((|V| + |E|) \cdot \log |V|)$

理论上，使用斐波那契堆可以获得更优的渐近复杂度 $O(|V| \log |V| + |E|)$

> 待补充.

局限性：Dijkstra's algorithm does not work if edges can take negative weights.

12.2 SSSP 问题 - 任意权重

> SSSP with Arbitrary Weights

12.2.1 负权重

注意：Dijkstra's algorithm does not work if edges can take negative weights.

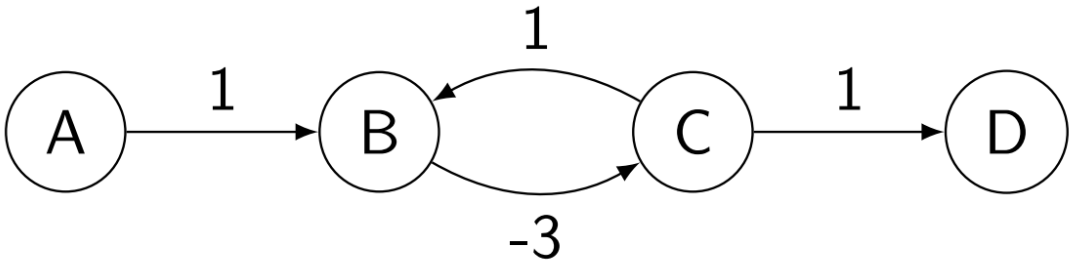


Figure: Graph with negative cycle

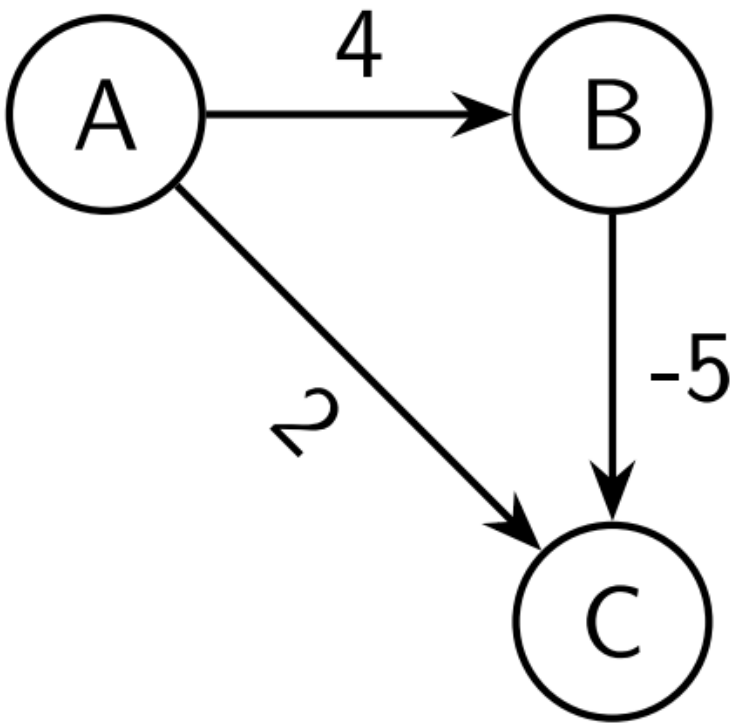
如果图中存在负环，最短路径长度可能是未定义的（无限小）

此时，最短**简单**路径问题（Shortest Simple Path, SSP）是有定义的（路径不能重复访问顶点），但它是一个 NP-hard 问题.

> NP-hard：meaning that no polynomial-time algorithm is currently known for solving it —— including Dijkstra's algorithm
>
> 待补充.

即使图中没有负环，Dijkstra 算法也无法解决最短路径问题.

一个反例：



目标：Find the shortest path from A to C

如果使用 Dijkstra's Algorithm，会给出 A 到 C 的最短路径为 $A \to C$，而不是正确答案 $A \to B \to C$，因为 C 在 B 之前就被移除，因此没有机会 relax the edge $B \to C$

Dijkstra's algorithm 失效的原因：

有负权边但是无负环也不能用 Dijkstra，因为正权重的图可以保证从小到大移除，后面移除的顶点一定没有机会 relax 之前移除的顶点（因为 dist 已经更大了，加一个非负值不可能变成更小），但是负权边不能保证当下删掉的顶点的 dist 就是最终的最小值，因为一个在算法早期看起来距离 “较远” 的节点（因为它通过正边到达）可能通过一条负权重边 “反向” 影响一个距离 “较近” 且已被确定的节点.

总结：Shortest path problem

- Non-negative weights - Dijkstra's algorithm
- Negative weights
  - negative cycles - Shortest Paths are not well-defined. Shortest Simple Paths（无环）are well defined. However, this is a NP-hard problem.
  - no negative cycles - Shortest Paths are well-defined. But Dijkstra's algorithm does not work.

What should we do with graphs that contain no negative cycles?

- Bellman-Ford algorithm.

12.2.2 Bellman-Ford 算法

> the Bellman-Ford algorithm

Problem statement:

SSSP Problem: Let $G = (V, E)$ be a directed simple graph, where function w maps every edge of E to an arbitrary integer（可正可负）. It is guaranteed that G has no negative cycles（我们只考虑没有负环的情形）. Given a source vertex s in V, we want to find a shortest path from s to t for every vertex $t \in V$ reachable from s

The output is a shortest path tree T:

- The vertex set of T contains all vertices reachable from s
- The root of T is s
- For each node $u \in V$, the root-to-u path of T is a shortest path from s to u in G

下面介绍 Bellman-Ford algorithm that solves this problem in $O(|V||E|)$ time.

Note:

- We will focus on computing $\text{spdist}(s, v)$, namely, the shortest distance from the source vertex s to every vertex $v \in V$.
- Constructing the shortest paths is easy and will be left to you.

Bellman-Ford algorithm 算法步骤

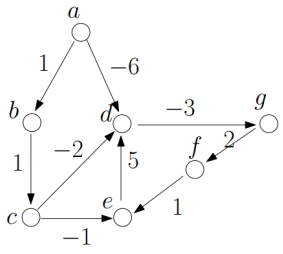
1. 初始化：设置源点 s 的距离 $\text{dist}(s) \leftarrow 0$. 对于其他顶点 $v \in V \setminus \{s\}$，设置 $\text{dist}(v) \leftarrow \infty$.
2. 核心步骤：重复 $|V| - 1$ 次以下操作：
   - 对 E 中的所有边进行松弛操作（松弛顺序不影响）.
   - 松弛：对于边 (u, v)，如果 $\text{dist}(v) > \text{dist}(u) + w(u, v)$，则将 $\text{dist}(v)$ 更新为 $\text{dist}(u) + w(u, v)$

注意，严格按照算法要进行 $|V|-1$ 轮 relax，但是实际上只需要在某轮 relax 没有变化后停止.

复杂度：The running time is clearly $O(|V||E|)$

Example

Suppose that the source vertex is *a*.



vertex *v*	*dist*(*v*)
a	0
b	∞
c	∞
d	∞
e	∞
f	∞
g	∞

For illustration purposes, we will relax the edges in alphabetic order shown below:

$$(a, b), (a, d), (b, c), (c, d), (c, e), (d, g), (e, d), (f, e), (g, f).$$

正确性证明

- 引理：在无负环图中，从 s 到任意顶点 v 的最短路径中，至少有一条是简单路径（没有顶点重复出现）
- 推论：从 s 到 v 的最短路径至多包含 $|V|-1$ 条边.
- 定理：如果从 s 到 v 的最短路径有 l 条边，那么经过 l 轮边松弛后，$\mathrm{dist}(v)$ 必然等于 $\mathrm{sptest}(v)$.
- 由于最短简单路径最多有 $|V|-1$ 条边，因此算法执行 $|V|-1$ 轮松弛后，所有可达顶点的最短路径距离都能被找到.

定理证明：Consider any vertex v, suppose that there is a shortest path from s to v that has l edges. Then, after l rounds of edge relaxations, it must hold that $\mathrm{dist}(v) = \mathrm{spdist}(v)$

Proof: We will prove the theorem by induction on l. If $l = 0$, then $v = s$, in which case the theorem is obviously correct. Next, assuming the satement's correctness for $l < i$ where i is an integer at least 1, we will prove it holds for $l = i$ as well.

Denote by π the shortest path from s to v, namely, π has i edges.

Let p be the vertex right before v on π.

By the inductive assumption, we know that $\mathrm{dist}(p)$ was already equal to $\mathrm{spdist}(p)$ after the $(i-1)$-th round of edge relaxations.

In the i-th round, by relaxing edge (p, v), we make sure:

$$\begin{aligned}\text{dist}(v) &\le \text{dist}(p) + w(p, v)\\ &= \text{spdist}(p) + w(p, v)\\ &= \text{spdist}(v)\end{aligned}$$

核心思想：每一轮松弛都将最短路径的 “覆盖范围” 扩大一步. 第 i 轮松弛确保了所有最短路径长度为 i 的顶点，都能找到其正确的距离. 由于在无负环图中，最短路径最多有 $|V| - 1$ 条边，因此算法只需要 $|V| - 1$ 轮松弛就能找到所有最短路径距离.

有一种逐步扩散的感觉.

12.3 APSP 问题

> All-Pairs Shortest Paths

12.3.1 问题描述

问题描述：全源最短路径问题

给定一个有向图 $G = (V, E)$，其中边权 w 可以是正数、零或负数，但保证图不含负权环. 目标是找到所有顶点对 (s, t) 之间的最短路径. 具体输出为以 V 中每个顶点为根的最短路径树.

> 注意，只是不含负权环，不一定不含零权环.

已知解决方案的局限性：

- 如果所有边权都非负，可以运行 $|V|$ 次 Dijkstra 算法，总时间复杂度为 $O(|V|(|V| + |E|) \log |V|)$
- 对于包含任意权值（可能为负）的一般 APSP 问题，可以运行 $|V|$ 次 Bellman-Ford 算法，总时间复杂度为 $O(|V|^2|E|)$

局限：Dijkstra 的复杂度可以接受，但是对问题有额外限制；Bellman-Ford 没有限制，但是时间复杂度太高.

下面介绍 Johnson's algorithm，能够用 Dijkstra 的复杂度解决不受限制的原始 APSP 问题.

12.3.2 Johnson 算法

> Johnson's algorithm

核心思想：**重新赋权**（Re-weighting），然后用 Dijkstra.

> 赋权函数是给每个顶点赋权，但是最终目的是重写边权
>
> Johnson 算法的核心逻辑就是通过"顶点赋权"来实现"边权转换"

思考：We cannot apply Dijkstra's algorithm because our graph may have edges with negative weights. Can we convert all the weights into non-negative values while preserving all shortest paths?

The answer is YES.

> 如果可以给负权重新赋权非负值，为什么要学 Bellman-Ford 算法?
>
> Answer：因为赋权的过程使用了 Bellman-Ford.

注意一个关键问题：不能简单的给所有边赋一个相同常数（例如，把最小的边升到正数，其他的边提升相同的值）. 因为这样会导致最短路径改变. 因为路径经过的边数越多，提升越多，原来的最短路径可能包含很多个小边，在提升过后可能就不是最短路径了.

① 赋权函数

目标：找到一个重新赋权函数，能保证将所有边权转为非负值的同时保持最短路径不变.

> 使用虚拟顶点技巧 + Bellman-Ford，将函数 $h(\cdot)$ 定义为 $h(u)=\mathrm{spdist}(v_{dummy},u)$，其中 $u\in V$

重新赋权函数：选取任意函数 $h:V\to\mathbb{Z}$，将每条边 $(u,v)\in E$ 的新权值 $W(u,v)$ 定义为

$$W(u,v)=w(u,v)+h(u)-h(v)$$

最短路径不变性：对于图 G 中的任意一条路径 $v_1\to v_2\to\cdots\to v_x$，如果它在 G 中的长度为 l，那么它在重新赋权后的图 G' 中的长度为 $l+h(v_1)-h(v_x)$. 对于 v_1 到 v_x 的所有可能路径后两项都一样，因此 l 最小的赋权完 $l+h(v_1)-h(v_x)$ 还是最小的.

② 虚拟顶点技巧

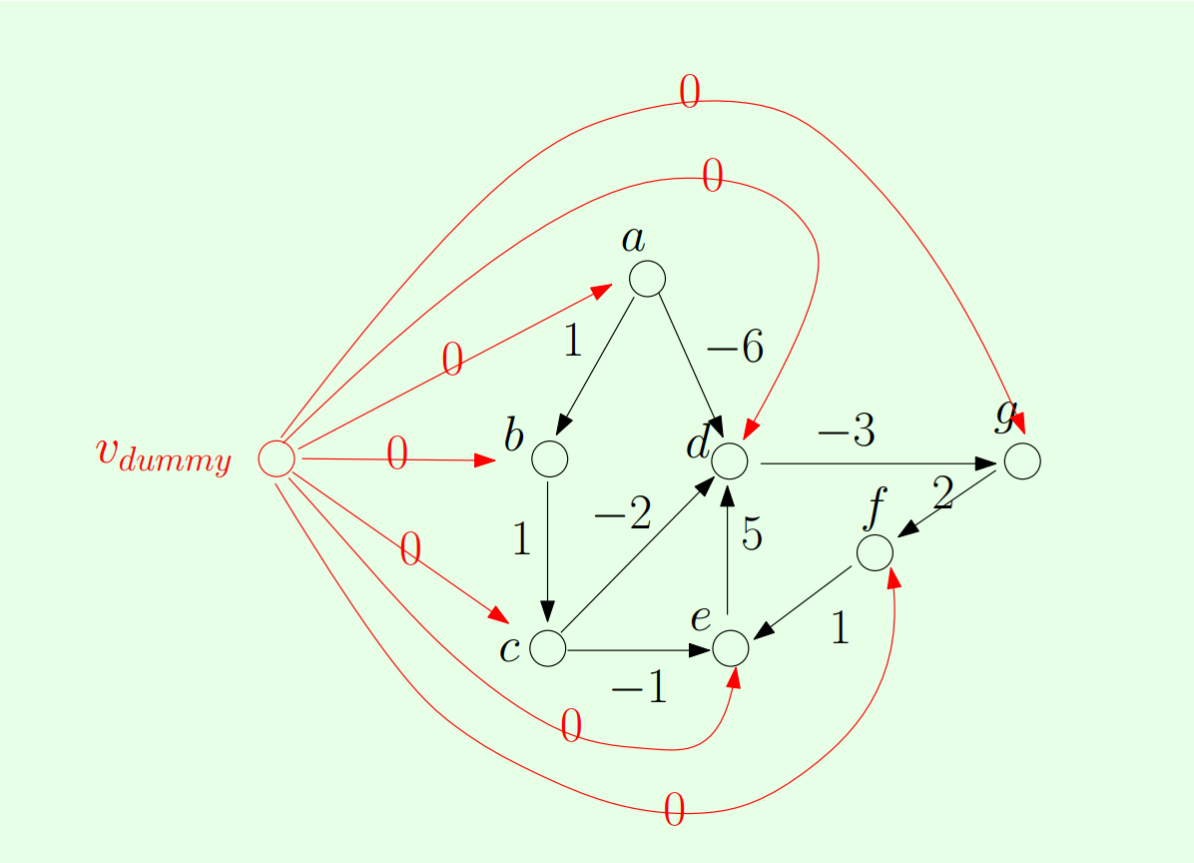
> dummy-vertex trick

我们已经设计好赋权函数的形式，新的目标是：高效地找到一个具体的赋权函数 $h(\cdot)$，使得重新赋权后的图 G' 中所有边权 $W(u,v)\geq 0$.

引入虚拟顶点技巧：

1. 从原图 $G=(V,E)$ 构造新图 $G_\Delta=(V_\Delta,E_\Delta)$
2. $V_\Delta=V\cup\{v_{\mathrm{dummy}}\}$
3. E_Δ 包含 E 中的所有边，并增加 v_{dummy} 到 V 中所有顶点的边.
4. 继承自 E 的边权不变，新增加的边权设为 0.

例：



然后确定 $h(\cdot)$：在 G_Δ 中，使用 Bellman-Ford 算法计算从 v_{dummy} 到所有其他顶点的最短路径距离 $\text{spdist}(v_{dummy}, u)$

将函数 $h(\cdot)$ 定义为 $h(u) = \text{spdist}(v_{dummy}, u)$，其中 $u \in V$

结果：使用这个 $h(\cdot)$ 重新赋权后，图 G'（重新赋权后的图，不是 G_Δ）的所有边权都是非负数. Bellman-Ford 这一步的计算时间复杂度为 $O(|V||E|)$

证明：dummy trick 重新赋权后所有边权都非负

三角不等式（松弛条件）：

$$\text{spdist}(v_{\text{dummy}}, v) \le \text{spdist}(v_{\text{dummy}}, u) + w(u, v)$$

> 这个不等式是所有最短路径算法的基础：从 v_{dummy} 到 v 的最短路径距离，不可能比 “先从 v_{dummy} 走到 u，再通过边 (u, v) 走到 v” 的距离更长.

势能函数（赋值函数）的定义：

$$h(u) = \text{spdist}(v_{\text{dummy}}, u)$$

代入三角不等式：

$$h(v) \le h(u) + w(u, v)$$

重排：

$$W(u, v) = w(u, v) + h(u) - h(v) \ge 0$$

证毕.

③ 算法步骤

> Johnson 算法步骤

1. 执行 `② 虚拟顶点技巧` 和 Bellman-Ford 算法，找到合适的 $h(\cdot)$
2. 利用 $h(\cdot)$ 重新赋权，得到非负权图 G',
3. 对 G' 中的每个顶点 $u \in V$，运行一次 Dijkstra 算法，每次得到一棵顶点为 u 的最短路径树（共 $|V|$ 次运行，得到 $|V|$ 棵树，算法结束）.

④ 复杂度分析

总时间复杂度：

$$
\begin{aligned}
&O(|V|(|V|+|E|)\log|V|)+O(|V||E|)\\
=&O(|V|(|V|+|E|)\log|V|)
\end{aligned}
$$

> Bellman-Ford 带来的复杂度可以忽略不计.

Tut 9 APSP：Floyd-Warshall 算法

9.1 问题定义与背景

- **输入**：一个不含负环的有向简单图 $G=(V,E)$，边权可以为正、零或负。
- **目标**：寻找图中所有顶点对 (s,t) 之间的最短路径。
- **性能对比**：
  - **Dijkstra 算法**：若边权非负，运行 $|V|$ 次的时间复杂度为 $O(|V|(|V|+|E|)\log|V|)$ 。
  - **Johnson 算法**：适用于含负权边（无负环）的情况，复杂度同上。
  - **Floyd-Warshall 算法**：采用**动态规划**思想，时间复杂度为 $O(|V|^3)$ 。当图非常稠密（如 $|E|=\Theta(|V|^2)$）时，该算法效率优于前两者 。

9.2 算法核心原理：动态规划

算法通过将顶点编号为 1 到 n，逐步扩大允许作为中间节点的顶点集合来求解 。

- **状态定义**：$spdist(i,j|\leq k)$ 表示从顶点 i 到 j 的最短路径长度，且该路径仅允许使用编号 $\leq k$ 的顶点作为中间节点 9。

- 转移方程：

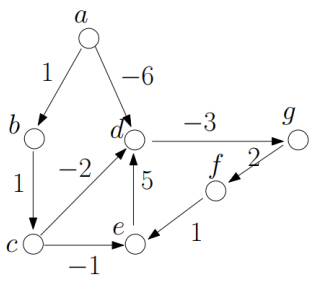
$$spdist(i,j|\leq k)=\min\begin{cases}spdist(i,j|\leq k-1) & (\text{不经过顶点 } k)\\ spdist(i,k|\leq k-1)+spdist(k,j|\leq k-1) & (\text{经过顶点 } k)\end{cases}$$

- **初始状态 ($k=0$)**：
  - 若 $i=j$，距离为 0。
  - 若 $(i,j)\in E$，距离为边权 $w(i,j)$ 。
  - 否则为 ∞

9.3 实例演示 (7个顶点)

Example

First, decide $spdist(i,j\mid\leq 0)$ for all $i,j\in[1,7]$.



vertex v	a	b	c	d	e	f	g
a	0	1	∞	-6	∞	∞	∞
b	∞	0	1	∞	∞	∞	∞
c	∞	∞	0	-2	-1	∞	∞
d	∞	∞	∞	0	∞	∞	-3
e	∞	∞	∞	5	0	∞	∞
f	∞	∞	∞	∞	1	0	∞
g	∞	∞	∞	∞	∞	2	0

文档通过一个包含顶点 a 到 g 的示例，展示了从 $k=0$ 到 $k=7$ 的矩阵演变过程：

- $k=0$：仅包含直接相连的边权 。

- $k = 1 \ldots 7$：逐步引入中间节点。例如在 $k = 3$（节点 c）时，发现了经过 c 到达 d 的更短路径。

- **最终状态 ($k = n$)**：$spdist(i, j| \leq n)$ 即为最终的全源最短路径距离。

9.4 路径恢复：Piggyback

- **路径恢复**：虽然算法主要计算距离，但可以通过"**搭便车技术 (piggyback technique)**"来扩展，从而输出每个顶点的最短路径树

Tut 10：Bellman-Ford 检测负环

1. 算法背景与复习

- **基础功能**：在没有负权环的情况下，Bellman-Ford 算法用于解决单源最短路径（SSSP）问题 2。

- **核心机制**：通过**松弛（Relaxation）**操作更新顶点的 $dist(v)$ 3。如果 $dist(v) > dist(u) + w(u, v)$，则更新 $dist(v)$ 并记录 $parent(v) = u$ 4444。

- **正常流程**：算法通常重复执行 $|V| - 1$ 轮松弛操作 5。

2. 负权环检测算法

为了检测负权环，算法在完成标准的 $|V| - 1$ 轮松弛后，增加**第 $|V|$ 轮**松弛 6666：

- **检测原理**：在没有负权环的情况下，第 $|V|$ 轮松弛不会改善任何 $dist$ 值 7。

- **判断依据**：如果在第 $|V|$ 轮松弛中，仍有任何顶点的 $dist(v)$ 能够被减小，则说明图中**存在负权环** 888888888。

- **复杂度**：时间复杂度保持在 $O(|V||E|)$ 9。

3. 正确性证明

文档分两个方向证明了算法的有效性 10：

- **方向 1**：如果第 $|V|$ 轮松弛改善了某个顶点的距离，则图中必然存在负权环（证明略） 11111111。

- **方向 2**：如果图中存在负权环，则第 $|V|$ 轮松弛一定会改善至少一个顶点的距离 12121212。该证明通过累加环上各边的松弛不等式，推导出矛盾（$0 \leq$ 负值），从而证实结论 13。

4. 如何提取负权环

一旦在第 $|V|$ 轮检测到负权环（例如通过松弛边 (u, v) 触发），可以通过以下方式找到该环：

- **追踪父节点**：从受影响的顶点开始，沿着 `parent` 指针回溯 15。

- **识别循环**：持续回溯直到**同一个顶点第二次出现。**这两个相同顶点之间路径上的所有顶点即构成了一个负权环 16。

Week 11-13 NP-难问题与近似算法

NP-难问题 & P=NP? & 近似算法

许多具有实际意义的问题（如图3着色、子集求和、旅行商问题等）目前没有已知的有效（即多项式时间）算法，它们属于**NP-难问题**.

> A graph coloring assigns colors to the vertices of a graph so that no two adjacent vertices share the same color. In the Graph-3-Coloring problem, we ask: Can the vertices of a given graph be colored with at most 3 colors such that adjacent vertices have different colors?

三个最基础的分类：

- **P (Polynomial time)**：可以在“多项式时间”内**解决**的问题。这类问题通常被认为是“易解”或“高效处理”的（例如：排序、搜索）。
- **NP (Nondeterministic Polynomial time)**：其解可以在多项式时间内被**验证**的问题。虽然我们不一定能快速算出答案，但如果有人给了你一个答案，你能很快检查它对不对（例如：数独、拼图）。
- **NP-complete (NP-完全)**：NP类中最难的问题。这是一个神奇的类别：**如果你能找到一种快速解决其中任何一个问题的方法，那么所有 NP 类问题都可以被快速解决。**

  > 要证明 $P = NP$，仅仅找到“某个” NP 问题的多项式时间解法是不够的，你必须找到的是那个**“最难的问题”**。
  >
  > 在复杂性理论中，我们把这类特殊的 NP 问题称为 **NP-完全（NP-Complete）** 问题
  >
  > 例：

对于这些难解问题，计算机科学发展出两条研究路径：不可解性理论（NP-完全性）和**近似算法**

> 本课重点：近似算法

不可解理论，计算机科学界最著名的未解之谜：**P 是否等于 NP?**

- 翻译成大白话就是：如果一个问题的解可以被“快速验证”，那它是否一定也能被“快速找到”？（目前大多数科学家认为 $P \neq NP$，但尚未证明）
- 如果 $P = NP$，那么所有的 NP 类问题（无论它们看起来多么不同）都可以通过**多项式时间归约**，转化成 P 类中的问题

近似算法目标：设计高效的算法，在多项式时间内找到**接近最优**的解，并能证明所得解的价值与最优解之间存在一个可证明的比率（即**近似比** ρ）

顶点覆盖问题 & MAX-3SAT 问题

> Vertex Cover & MAX-3SAT

Vertex Cover：顶点覆盖问题

1. 问题描述

$G = (V, E)$ is a simple undirected graph.

> simple：没有自环.

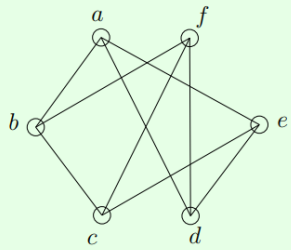
A subset $S \subseteq V$ is a **vertex cover** of G if every edge $\{u, v\} \in E$ is incident to at least one vertex in S

解释：想象在图中顶点上部署 “警卫”. 对于图中每一条边，必须保证这条边两端至少有一个 “警卫”（都有也可以）. 这样的警卫集合就是一个 vertex cover.

The V.C. Problem: Find a vertex cover of the smallest size.

> **顶点覆盖问题**通常指的是寻找一个**最小顶点覆盖 (Minimum Vertex Cover)**

Example:



An optimal solution is $\{a, f, c, e\}$.

它在许多实际问题中都有应用，例如：网络安全（在网络节点中选择最小数量的节点来监控所有连接）、电路设计、排班优化等.

找到一个图的最小顶点覆盖是一个**NP-难**的问题. 这意味着对于大规模的图，我们目前没有已知的、能够在多项式时间内找到最优解的快速算法.

2. 近似算法：两点全选

$A =$ an algorithm that, given any legal input $G = (V, E)$, returns a vertex cover of G

$OPT_G =$ the smallest size of all the vertex cover of G

A is a ρ-approximate algorithm for the vertex cover problem if, for any legal input $G = (V, E)$, A can return a vertex cover with size at most $\rho \cdot OPT_G$

The value ρ is the **approximation ratio**.

We say that A achieves an approximation ratio of ρ

考虑下列算法：

Input: $G = (V, E)$

$S = \emptyset$

```
while E is not empty do
    pick an arbitrary edge {u,v} in E
    add u,v to S
    remove from E all the edges of u and all the edges of v

return S
```

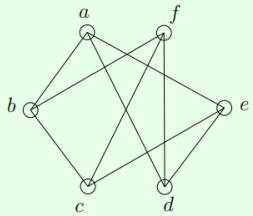
易证：

- S is a vertex cover of G
- The algorithm runs in time polynomial to $|V|$ and $|E|$

We will prove later that the algorithm is 2-approximate.

例：

Example:



Suppose we start by picking edge $\{b, c\}$.
Then, $S = \{b, c\}$ and $E = \{\{a, e\}, \{a, d\}, \{d, e\}, \{d, f\}\}$.

Any edge in E can then be chosen. Suppose we pick $\{a, e\}$.
Then, $S = \{a, b, c, e\}$ and $E = \{\{d, f\}\}$.

Finally, pick $\{d, f\}$.
$S = \{a, b, c, d, e, f\}$ and $E = \emptyset$.

定理：

The algorithm returns a set of at most $2 \cdot OPT_G$ vertices.

证明：Let M be the set of edges picked.

Lemma 1: the edges do not share any vertices.

> 反证：若某两条被选取的边有公共点 a，记先选中的边为 e_1，后选中的为 e_2，则选中 e_1 时会把所有 a 的边都 remove，包括 e_2，自相矛盾.

Lemma 2: $|M| \leq OPT_G$

> 任意 vertex cover，M 中每条边至少要有一个顶点在其中（因为整个 Graph 的所有边都至少有个顶点在其中）. 因为M 中边与边没有公共点，因此 $|M| \leq OPT_G$.

而该算法返回的 vertex cover 有 $2|M|$ 个顶点（每条边的两个点都被选取），$2|M| \leq 2OPT_G$，因此近似比为 2

MAX-3SAT 问题

The MAX-3SAT Problem

这是布尔逻辑运算相关的问题. 先定义一些概念：

variable: 一个布尔未知数 x，值为 0 或 1

literal（文字）：x 或它的取反 $\overline{x}$

3-literal clause（3字子句）：三个来自**不同变量**文字的 OR 运算

例：$(x_1 \vee \overline{x}_2 \vee x_3)$

S：子句的集合（MAX-3SAT问题中，指3字子句的集合）

$\mathcal{X}$：S 中出现过的变量的集合

A **truth assignment**（真值赋值）$f : \mathcal{X} \to \{0, 1\}$

它的作用是给 $\mathcal{X}$ 集合中的每个变量都指定一个布尔值（0或1）

满足子句：

A truth assignment f satisfies a clause in S if the clause evaluates to 1 under f

> 如果在这个赋值 f 下，这个子句的结果为 1，则说 f 满足该子句.
>
> 子句是三个文字的 OR 运算，即只要 f 让子句中任何一个文字被赋值为 1，该子句就被满足

MAX-3SAT 问题：给定一个由 n 个3字子句组成的集合 S，找到一个 f 给变量集 $\mathcal{X}$，使得被满足的子句数量最大化.

Example:

$$
\begin{aligned}
S = \{&x_1 \vee x_2 \vee x_3,\\
&x_1 \vee x_2 \vee \bar{x_3},\\
&x_1 \vee \bar{x_2} \vee x_3,\\
&x_1 \vee \bar{x_2} \vee \bar{x_3},\\
&\bar{x_1} \vee x_3 \vee x_4,\\
&\bar{x_1} \vee x_3 \vee \bar{x_4},\\
&\bar{x_1} \vee \bar{x_3} \vee x_4.\\
&\bar{x_1} \vee \bar{x_3} \vee \bar{x_4}\}.
\end{aligned}
$$

$n = 8$ and $\mathcal{X} = \{x_1, x_2, x_3, x_4\}$.

The truth assignment $x_1 = x_2 = x_3 = x_4 = 1$ satisfies 7 clauses. It is impossible to satisfy 8.

MAX-3SAT 是一个 NP难问题 & NP 完全.

近似算法：随机赋值

$A =$ an algorithm that, given any legal input S, returns a truth assignment of S

$OPT_S =$ the largest number of clauses that a truth assignment of S can satisfy.

$Z_S =$ the number of clauses satisfied by the truth assignment A returns

> 如果 A 是随机化算法，则 Z_S 是随机变量. 此时我们关注它的期望.

A is a randomized ρ-approximate algorithm for MAX-3SAT if $E[Z_S] \geq \rho \cdot OPT_S$ holds for any legal input S

> 注意方向，现在要求的是最大值，因此我们近似算法需要的是不小于最大值 ρ 倍的解（$\rho \leq 1$）

The value ρ is the **approximation ratio**.

We also say that A achieves an approximation ratio of ρ **in expectation**.

Consider the following algorithm

Input: a set S of clauses with variable set $\mathcal{X}$

```
for each variable x∈X do
    toss a fair coin
    if the coin comes up heads then x<-1
    else x<-0
```

全部随机赋值

It is clear that the algorithm runs in $O(n)$ time

Theorem 2: The algorithm produces a truth assignment that satisfies $\frac{7}{8}n$ clauses in expectation.

> n 是 S 的子句数量.

每个子句有 $\frac{1}{8}$ 的概率不被满足（三个都不满足子句才不被满足）

期望上，n 个子句有 $\frac{7}{8}n$ 个被满足.

$$E[Z_S] = \frac{7}{8}n \geq \frac{7}{8} \cdot OPT_S$$

因为 S 能被满足的子句数量最多为总数（即全满足）

近似比：$\frac{7}{8}$

旅行商问题

> Traveling Salesman Problem, TSP

完全图：任意两个不同顶点之间都有一条边.

$G = (V, E)$ is a complete undirected graph

Each edge $e \in E$ carries a non-nagative weight $w(e)$

> 表示成本：距离/时间/费用...

A **Hamiltonian cycle** of G is a cycle passing every vertex of V once

> 推销员的行程路线。它必须从一个城市出发，经过所有其他城市**恰好一次**，并最终回到起始城市，形成一个闭合的环路
>
> 哈密顿回路可以用有序的顶点排列来表示. 起点出发再回到起点.

The traveling salesman problem: Find a Hamiltonian cycle with the shortest length.

找到最短路径：NP-难

判定是否存在路径，总距离小于 K：NP完全

近似算法：最小生成树 + 欧拉环

A = an algorithm that, given any legal input (G,w), returns a Hamiltonian cycle of G.

记最短路径为 $OPT_{G,w}$

A is a ρ-approximate algorithm for the travelling salesman problem if, for any legal input (G, w), A can return a Hamiltonian cycle with length at most $\rho \cdot OPT_{G,w}$

然而，TSP 在任意 ρ 下的近似算法都是 NP 难的.

因此，我们考虑结构更好的图：

We additionally assume that the graph G satisfies **triangle inequality**:

For any $x, y, z \in V$, it holds that $w(x, z) \leq w(x, y) + w(y, z)$

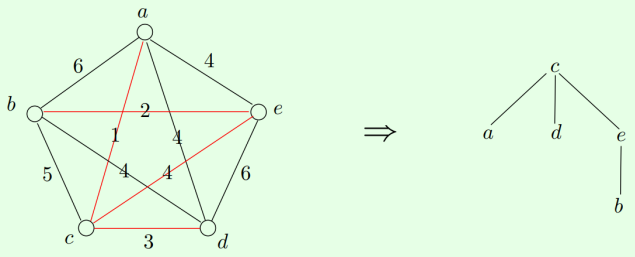
确保两点之间直接通行最短. 这至少满足了现实情境中的欧拉距离.

在满足三角不等式的图中，可以找到一个 $\rho = 2$ 的近似算法找最小哈密顿

算法包含三步：

（1）获取一个 MST（最小生成树）T of G

Example:



(2) 获取一个闭合通路 π，where every edge of T appears on π exactly twice.（欧拉环）

> 从根开始进行深度优先搜索（任何一个点都可以当作根）.
>
> 因此是 $O(|V|)$ 时间

(3) 构建哈密顿回路的顶点序列 σ（顺序敏感）：

首先，添加 π 的第一个顶点到 σ

然后，遍历 π

当到达任意顶点 v，如果未加入 σ，加入

如果已经被加入，不操作

最后，把 π 的最后一个点加入 σ

定理：该算法返回一个哈密顿回路，长度最多 $2OPT_{G,w}$

> $\rho = 2$

Lemma 1: $OPT_{G,w} \geq w(T)$

> 因为任意哈密顿环的边集移掉一条都是生成树，但不一定最小

Lemma 2: The walk π has length $2 \cdot w(T)$

> Proof: Every edge of T appears twice in π.

Lemma 3: The length of our Hamiltonian cycle σ is at most the length of π

应用三角不等式：

Let the vertex sequence in π be $u_1, u_2, \ldots, u_t$ for some $t \geq 1$

Let σ be the vertex sequence $u_{i_1}, u_{i_2}, \ldots, u_{i_{|V|+1}}$ where

$i_1 = 1 < i_2 < \ldots < i_{|V|} < i_{|V|+1} = t$

- σ 的顶点从 π 的顶点序列中选出
- $u_{i_1} = u_1$ 是起点，$u_{i_{|V|+1}} = u_t$ 是终点
- 哈密顿环的顶点序列可以看作保留欧拉环的起点终点，然后把路径上重复顶点去掉. 因为是基于欧拉环删去重复顶点，所以相对顺序仍然按欧拉环的下标递增（但是会跳过一些，而不是逐1递增）

By 三角不等式，we have for each $j \in [1, |V|]$

$$w(u_{i_j}, u_{i_{j+1}}) \leq \sum_{k=i_j}^{i_{j+1}-1} w(u_k, u_{k+1})$$

注意是递增序列，$i_{j+1} - 1 \geq i_j$

> 沿着哈密顿环走一条边，小于等于对应欧拉环中间节点都算上的路线（因为欧拉环可能绕回去又走出来）

Hence:

length of $\sigma = \sum_{j=1}^{|V|} w(u_{i_j}, u_{i_{j+1}}) \leq \sum_{k=1}^{t-1} w(u_k, u_{k+1}) =$ length of π

该算法得到的哈密顿环 $\leq$ 对应欧拉环 $=$ 2倍最小生成树 $\leq$ 2倍最小哈密顿环

集合覆盖问题 & 击中集问题

> Set Cover & Hitting Set

Set Cover：集合覆盖问题

给定一个集合的集合 $\mathcal{S}$，其并集为全集 $U = \bigcup_{S \in \mathcal{S}} S$。**集合覆盖** 是 $\mathcal{S}$ 的一个子集 $\mathcal{C} \subseteq \mathcal{S}$，使得 U 中的每个元素都至少出现在 $\mathcal{C}$ 中的一个集合里

问题目标: 找到一个规模最小的集合覆盖

贪心：

每一步，选择一个**能覆盖最多当前未覆盖元素**的集合加入到覆盖C中

The algorithm runs in time polynomial to n

该算法是 1 + $ln|U|$ 近似

$1 + (\ln |U|) \cdot OPT_S \leq (1 + \ln |U|) \cdot OPT_S$

$$\rho = 1 + \ln |U|$$

证明贪心算法具有 $1 + \ln |U|$ 的近似比，主要通过分析**每一步选择后剩余未覆盖元素的减少速率。**

以下是基于课件内容的详细证明步骤：

1. 符号定义

- 设 C^* 为最优解，其大小为 $OPT_S = |C^*|$。
- 设贪心算法选择了 t 个集合：$S_1, S_2, \ldots, S_t$
- 令 z_i 表示在选择了 S_i 之后，全集 U 中**仍未被覆盖**的元素数量
- 初始状态 $z_0 = |U|$，结束状态 $z_t = 0$

2. 核心引理：剩余元素的缩减率

引理 1： 对于每一次迭代 $i \in [1, t]$，有：

$$z_i \leq z_{i-1} \cdot \left(1 - \frac{1}{OPT_S}\right)$$

证明逻辑（均摊分析）：

1. 在选择 S_i 之前，还有 z_{i-1} 个元素未覆盖

2. 既然最优解 C^* 只用了 OPT_S 个集合就能覆盖全部元素，那么根据**平均值原理**，在 C^* 中一定至少存在一个集合，它能覆盖至少 $\frac{z_{i-1}}{OPT_S}$ 个当前未覆盖的元素

   > 解释：如果这样的集合一个都不存在，那么仅仅覆盖这些未覆盖的元素都要用掉大于 OPT_s 个集合.

3. 贪心算法每次都会选择“当前收益最大”的集合，因此贪心选择的 S_i 所覆盖的新元素数量必然满足 $|F \cap S_i| \geq \frac{z_{i-1}}{OPT_S}$

4. 因此，剩余元素 $z_i = z_{i-1} - |F \cap S_i| \leq z_{i-1} - \frac{z_{i-1}}{OPT_S} = z_{i-1}(1 - \frac{1}{OPT_S})$

3. 递归求解

通过递推公式，我们可以得到第 $t-1$ 步后的剩余元素数量：

$$z_{t-1} \leq z_0 \cdot \left(1 - \frac{1}{OPT_S}\right)^{t-1} = |U| \cdot \left(1 - \frac{1}{OPT_S}\right)^{t-1}$$

利用数学不等式 $1 + x \leq e^x$（当 $x = -1/OPT_S$ 时），得：

$$z_{t-1} \leq |U| \cdot e^{-\frac{t-1}{OPT_S}}$$

4. 得出结论

因为算法在第 t 步才结束，说明在 $t-1$ 步时至少还有一个元素没被覆盖，即 $z_{t-1} \geq 1$

结合上述不等式：

$$1 \leq |U| \cdot e^{-\frac{t-1}{OPT_S}}$$

两边取自然对数 $\ln$：

$$0 \leq \ln|U| - \frac{t-1}{OPT_S} \implies \frac{t-1}{OPT_S} \leq \ln|U|$$

整理得出：

$$t \leq 1 + OPT_S \cdot \ln|U|$$

这证明了贪心算法产生的集合数量 t 不会超过最优解的 $(1+\ln|U|)$ 倍

Hitting Set：击中集问题

定义: 给定全集 U 和一个集合的集合 $\mathcal{S}$ 。**击中集** 是 U 的一个子集 $H \subseteq U$，使得 H 与 $\mathcal{S}$ 中的所有集合 S 都有交集（即 $H \cap S \neq \emptyset$）。

问题目标: 找到一个规模最小的击中集 H 。

复杂度: 击中集问题也是 **NP-难** 的。

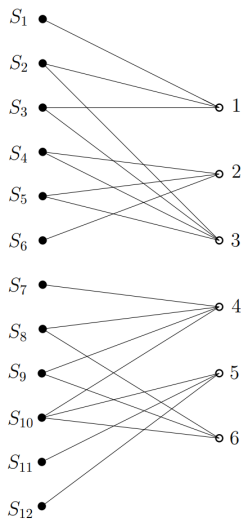
与集合覆盖的关系: 击中集问题和集合覆盖问题本质上是**相同的**问题。

- 通过将击中集问题转化为集合覆盖问题，可以得到一个具有 $\mathbf{1}+\ln|\mathcal{S}|$ 近似比的多项式时间击中集算法。

- **转换思想:** 考虑一个二分图，左侧顶点对应 $\mathcal{S}$ 中的集合 S，右侧顶点对应全集 U 中的元素 e 。

    - **击中集问题** 等价于找到一个最小的右侧顶点集 R，使得每个左侧顶点都与 R 中的至少一个顶点相邻。

- 这恰好转化为 **集合覆盖问题** 的形式



Consider the hitting set example on Slide 40. Let us create a bipartite graph G (shown left).

Each set $S \in \mathcal{S}$ corresponds to a vertex on the left of G.
Each element $e \in U$ corresponds to a vertex on the right of G.
An edge exists between vertex S and vertex e if and only if $e \in S$.

> Solving the hitting set problem is equivalent to finding a smallest set R of **right** vertices such that every left vertex is adjacent to at least one vertex in R.

This gives rise to the set cover example on Slide 3.

对称

k-中心问题

> k-Center Problem

给定二维空间中的一组点 P 和一个整数 k，找到一个大小为 k 的中心子集 $C \subseteq P$，使得惩罚值（P 中任意一点到其最近中心的距离的最大值）最小.

> C 的每个点都是中心，任意一点到中心子集 C 的距离等于它到 C 的所有中心的距离的最小值（到最近的那个中心的距离）.
>
> C 的惩罚值：所有点到 C 的距离中的最大值. 即离 C 最远的那个点到 C 的距离.
>
> 问题：找到一个 C，使得离 C 最远的点到 C 的距离达到最小

数学表达：最小化 $pen(C) = \max_{p \in P}\{\min_{c \in C} dist(p, c)\}$

NP 难.

判定问题：是否可以选 k 个中心，使得最大距离不超过 r?

> NP 完全.

$\rho = 2$ 的近似算法：贪心

Input P, k

$C \leftarrow \emptyset$

add to C an arbitrary point in P

for i=2 to k do

    p<-a point in P with the maximum dist_C(p)

    add p to C

return C

> 第一个任选，之后每个都选离当前 C 最远的，直到 k 个

讲义介绍了一个简单的**贪心算法**，其近似比为 2（即所得结果的惩罚值不超过最优解的 2 倍）：

1. 初始时，随机选择 P 中的一个点加入集合 C 。

2. 重复执行以下步骤直到 C 中有 k 个点：
    - 在 P 中找到一个距离当前集合 C **最远**的点 p 。
    - 将该点 p 加入 C 。
3. 返回集合 C 。

Theorem 1: the algorithm returns a set C with pen(C)$\leq$ 2OPT_P

> 即 $\rho = 2$

证明：待补充.

讲义通过两个部分证明了该算法的 **2-近似保证** ($pen(C_{ours}) \leq 2 \cdot OPT_P$)：

- **情况 1**：如果算法选出的 k 个点恰好分布在最优解的 k 个理想聚类区域中，利用**三角不等式**可以证明最大距离不会超过 $2 \cdot OPT$ 。
- **情况 2**：如果算法在同一个最优聚类区域内选了两个点，通过分析贪心选择的过程，证明任何点到当前中心集的距离都不会超过这两个点之间的距离，而这两个点之间的距离同样受限于 $2 \cdot OPT$ 。

Tut 11 归约

. 核心概念回顾

- **P 类问题**：可以在确定性图灵机上以多项式时间解决的问题 1。

- **NP 类问题**：可以在非确定性图灵机上以多项式时间解决的问题 2。

- **NP-hard 问题**：除非 $P = NP$，否则这类问题不存在多项式时间算法 3。

2. 归约方法 (Reduction)

要证明一个新问题 P_1 是 NP-hard，可以遵循以下两个步骤 4：

1. **确定已知问题**：找出一个已知是 NP-hard 的问题 P_2 5。

2. **建立转化关系**：证明如果存在求解 P_1 的多项式时间算法 $\mathcal{A}_1$，那么也可以利用它设计出求解 P_2 的多项式时间算法 $\mathcal{A}_2$ 6。

3. **结论**：因为 P_2 是 NP-hard，所以 P_1 也必定是 NP-hard 7。

3. 案例分析：证明团判定问题为 NP-hard

本教材通过将 **3-SAT 问题**（已知为 NP-hard）归约为**团判定问题**（The Clique Decision Problem）来完成证明。

- **相关定义**：
  -

3-SAT 问题：判断一个布尔公式（由最多包含 3 个文字的子句进行“与”运算组成）是否存在使结果为 1 的真值指派 。

- **团判定问题 (Clique)**：在无向图 G 中，判断是否存在一个至少包含 k 个顶点的集合 S，使得 S 中任意两个顶点都相连（即 k-团） 。

- **归约构造过程** ：
  - 对于 3-SAT 公式中的每一个子句，为其中的每个文字创建一个顶点。
  - 在两顶点间连边的条件是：它们**不在同一个子句**中，且它们**互不为否定关系**（例如 x_1 和 $\overline{x_1}$ 之间不连边）。
- **正确性证明**：
  - **充分性**：如果 3-SAT 公式有解，则每一子句中必有一个文字为 1，这些文字对应的顶点在图中构成一个 k-团 。
  - **必要性**：如果图中存在 k-团，这些顶点对应的文字互不冲突且来自不同子句，据此可构造出 3-SAT 的有效赋值。

4. 扩展思考

- **最大团问题 (Maximum Clique Problem)**：这是团判定问题的优化版本，目标是找到图中最大的 k 值。

- **结论**：团判定问题已被证明是 NP-hard。教材最后引导学生思考如何证明最大团问题在 $P \neq NP$ 的前提下也无法在多项式时间内解决。

期末复习

1. 2024T1 final 题解

Problem 2 (10 marks)

题目： 考虑下方的带权有向图 $G = (V, E)$。将源点设为 a，运行 Bellman-Ford 算法，执行 4 轮边松弛操作。请展示每轮之后每个顶点 $v \in V$ 的 $dist(v)$ 值。

答案： （本题在提供的答案文件中标注为“Free marks”，即送分题，未提供具体数值步骤）。

Problem 3 (10 marks)

题目： 令 S 为包含 n 个互不相同整数的集合，k_1 和 k_2 是满足 $1 \leq k_1 \leq k_2 \leq n$ 的任意整数。假设 S 存储在数组中。请设计一个期望时间复杂度为 $O(n)$ 的算法，报告 S 中所有排名在 $[k_1, k_2]$ 范围内的整数。注：整数在 S 中的排名等于 S 中小于或等于该整数的元素个数。

答案： 1. 首先应用 **k-selection** 算法，在 $O(n)$ 期望时间内找到第 k_1 小的整数 x_1。4

2. 再次应用 **k-selection** 算法，在 $O(n)$ 期望时间内找到第 k_2 小的整数 x_2。

3. **最后再扫描一遍** S，报告所有满足 $x_1 \leq y \leq x_2$ 的数字 $y \in S$。最后一步耗时 $O(n)$。

Problem 4 (10 marks)

题目： 令 $G = (V, E)$ 为简单无向图。5-环是指拥有 5 条边的环。若从 E 中移除边集 D 后能破坏 G 中所有的 5-环，则称子集 $D \subseteq E$ 为“5-环破坏者”。令 D^* 为规模最小的 5-环破坏者。设计一个算法，在 $|V|$ 的多项式时间内找到一个规模为 $O(|D^*| \cdot \log |V|)$ 的 5-环破坏者。

+1

答案： 1. 首先，计算 G 中所有 5-环的集合 S。为此，枚举所有可能的序列 $(v_1, v_2, v_3, v_4, v_5)$（其中 $v_i \in V$），并检查每条边 $(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_1)$ 是否存在于图 G 中。

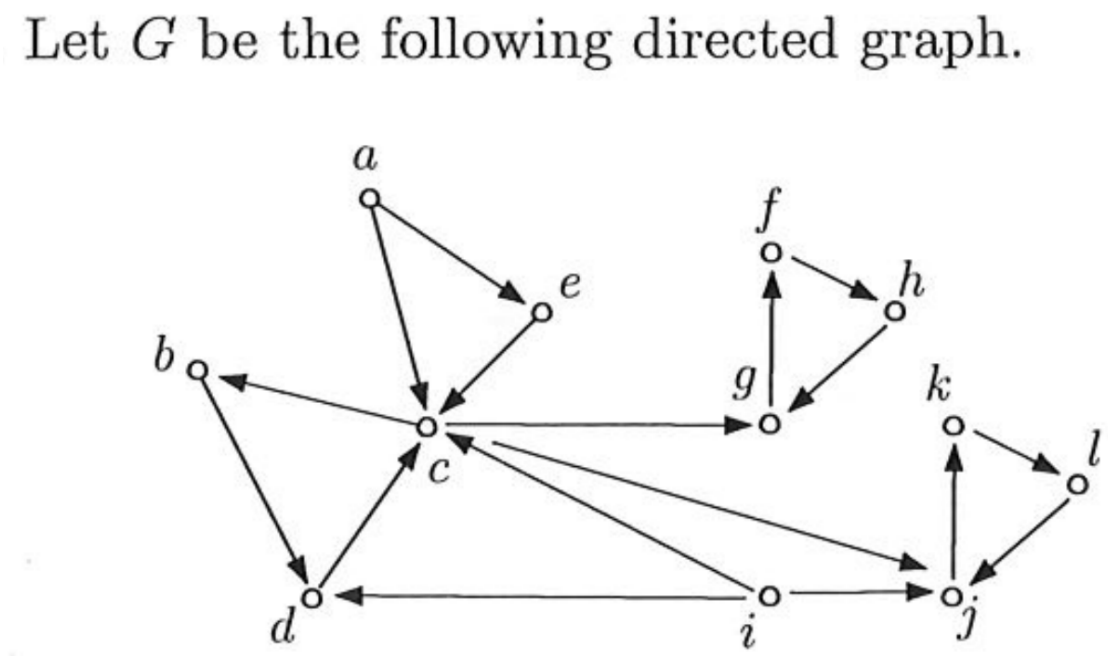
> 枚举所有 5-环集合的时间复杂度是 $O(|V|^5)$，S 的规模 $|S| = O(|V|^5)$
>
> 注意，我们后面用的近似算法不需要枚举.

2. 该问题现在转化为寻找一个最小边集来覆盖 S 中的每一个 5-环，这是一个**命中集问题 (Hitting Set Problem)** 的实例。

3. 应用标准的近似算法。运行时间显然是关于 V 的多项式级别。近似比至多为 $1 + \ln |S| = O(\log |V|)$。

Problem 5 (5 marks)

题目： 假设 Goofy 教授在有向图 G 上运行 DFS 一段时间后暂停。他告知以下信息：c 是下一个将被压入栈的顶点；当前除了 j, k, l 外，所有顶点均为白色。当 DFS 最终完成后，哪些顶点会成为 DFS 森林中 c 的后代？请说明理由。



答案： 根据**白路径定理 (White Path Theorem)**，c 在 DFS 森林中的后代为：b, d, g, f, h（注：答案原文中写的是 e 的后代，根据题意应指代起始点 c 及其可达的白色顶点）。

Problem 6 (5 marks)

题目： 在顶点覆盖问题中，除非 $P = NP$，否则没有多项式时间算法。Goofy 教授考虑了一个变体：给定 $k \leq n$，寻找一个规模至多为 k 的顶点覆盖。他声称找到了一个运行时间为 $O(n^2 \cdot k^{100})$ 的算法。证明该算法暗示 $P = NP$。

答案： Goofy 教授的算法运行时间为 $O(n^{102})$（因为 $k \leq n$）。为了找到最小规模的顶点覆盖，我们可以分别设定 $k = 1, 2, 3, \ldots$ 运行该算法，并在算法第一次返回顶点覆盖时停止。总运行时间将是 $O(n^{103})$，属于多项式时间，因此暗示 $P = NP$。

Problem 7 (10 marks)

题目： 要将一根 n 米长的钢棒切割成 n 段 1 米长的短棒。切割 l 米长棒的价格为 $A[l]$。设计一个动态规划算法，在 $O(n^2)$ 时间内计算出最小总成本。需提供递归函数、算法描述及时间分析。16161616

+1

答案： 1. **递归函数：** 令 $OPT(l)$ 为将 l 米长的棒切割成 1 米段的最优成本。则 $OPT(1) = 0$。对于 $l \geq 2$: $OPT(l) = A[l] + \min_{i=1}^{l-1}(OPT(i) + OPT(l-i))$。17

2. **算法：** 按照 l 升序计算 $OPT(l)$。18

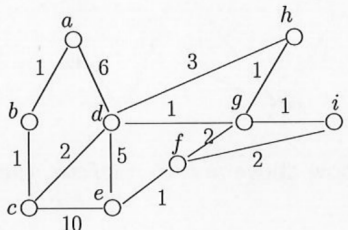
3. **分析：** 计算每个 $OPT(l)$ 需要 $O(l)$ 时间，计算 $OPT(n)$ 的总成本为 $\sum_{l=1}^{n} O(l) = O(n^2)$。19

Problem 8 (10 marks)

> 旅行商问题的非完全图变体

题目： 定义“S-往返行走”为从源点 s 出发，访问 S 中所有其他目标顶点至少一次，最后回到 s。设计一个多项式时间算法，寻找长度至多为 $2 \cdot OPT$ 的 S-往返行走，其中 OPT 是最小往返长度。

Let S be a subset of V, which includes a source vertex denoted as s, and k other destination vertices, where k is an integer satisfying $1 \leq k \leq |V| - 1$. Define an *S-round-trip walk* as a walk that starts from s, visits every destination vertex of S at least once (the order by which those vertices are visited does not matter), and then ends at s. Recall that a walk is a vertex sequence $u_0, u_1, u_2, ..., u_\ell$, where $\{u_i, u_{i+1}\}$ is an edge in E for all $i \in [0, \ell - 1]$. The *length* of the walk equals the total length of all the edges traversed. For example, consider G to be the following graph. If S has source vertex d and $k = 2$ destination vertices a and g, then an S-round-trip walk is $dabcdgfed$, whose length is $6 + 1 + 1 + 2 + 1 + 2 + 1 + 5 = 19$.



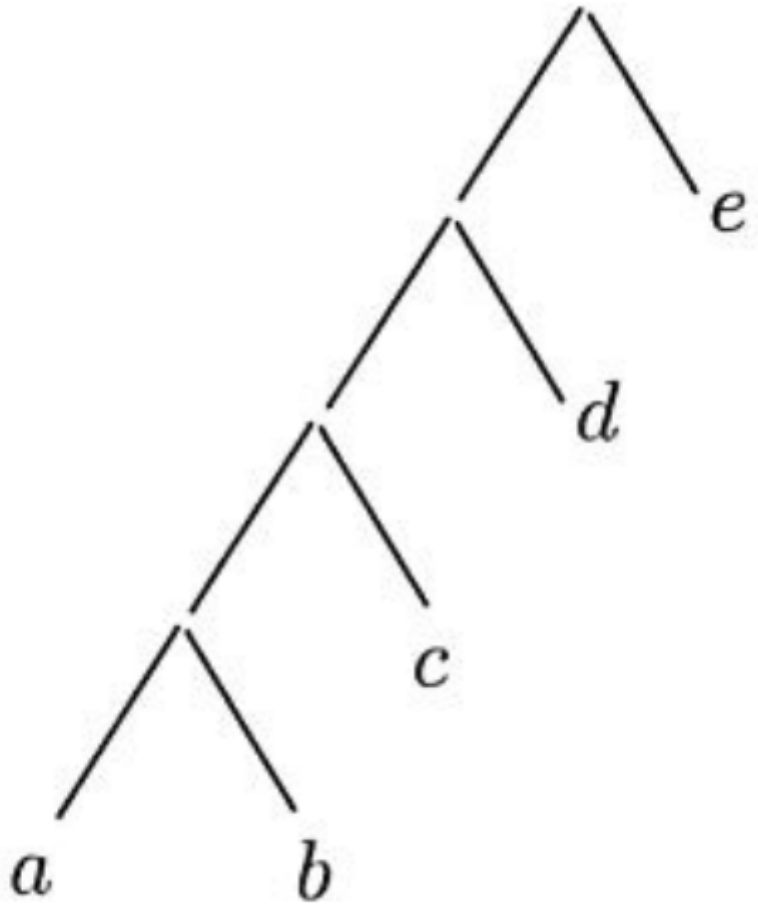
答案： 1. 首先生成一个完全图（团）$G^* = (S, E^*)$。对于 S 中任意两个顶点 u, v，边 $\{u, v\}$ 的权重等于原图 G 中 u 与 v 之间的最短路径距离。这可以通过多项式时间内的最短路径计算完成。21

2. OPT 的值对应于 G^* 中所有哈密顿回路的最小长度。寻找一个长度至多为 $2 \cdot OPT$ 的哈密顿回路 C（如课上讨论的近似算法）。

3. 将 C 转化为原图中的行走：对于 C 上每两个连续顶点 u, v，将边 $\{u, v\}$ 替换为原图中 u 到 v 的最短路径。得到的行走长度与 C 相同，至多为 $2 \cdot OPT$。

Problem 9 (10 marks)

题目： Goofy 教授对字母 $\{a, b, c, d, e\}$ 运行 Huffman 算法并得到了一棵特定的代码树（如图）。已知 a 和 b 的频率均为 $1/8$。求 c, d, e 的频率并说明理由。



答案： 根据 Huffman 算法的合并过程：

1. 算法必须首先合并 a, b 为节点 z（频率 $1/4$），然后将某个节点与 e 或其他节点逐步合并。

2. 推导得出：$f_c \geq f_a = 1/8$；$f_d \geq f_a + f_b = 2/8$；$f_e \geq f_a + f_b + f_c = 2/8 + f_c$。26

3. 由于总频率和为 1：$1 = f_a + f_b + f_c + f_d + f_e \geq 1/8 + 1/8 + f_c + 2/8 + (2/8 + f_c)$。解得 $f_c \leq 1/8$，结合前提得 $f_c = 1/8$

4. 进一步推导得 $f_d = 2/8$（即 $1/4$），$f_e = 3/8$

Problem 10 (10 marks)

题目： 令 C 为图 G 中的任意环，e^* 是 C 中权重最大的边。证明：G 存在一棵不包含 e^* 的最小生成树 (MST)。

答案： 1. 令 T^* 为 G 的任意一棵 MST。如果 e^* 不在 T^* 中，命题得证。

2. 如果 e^* 在 T^* 中：从 T^* 中移除 e^* 会将其分成两个连通分量 T_1 和 T_2。

3. 环 C 必然包含另一条边 $e \neq e^*$，且 e 连接 T_1 和 T_2 中的顶点

4. 用 e 连接 T_1 和 T_2 得到一棵新树 T，其总权重不会超过 T^*（因为 e^* 是环中权重最大的）。因此 T 也是一棵 MST 且不含 e^*

Problem 11 (10 marks)

题目： 图 G 无负环。给出一种算法，在 $O(|V||E|)$ 时间内检测 G 是否存在长度为 0 的环。需证明正确性并分析时间复杂度。

答案： 1. **重新权衡 (Re-weighting)：** 观察到重新权衡不会改变环的权重。使用 Johnson 算法中的方法，找到一个函数 $h : V \to \mathbb{R}$ 使得新权重 $w'(u, v) = w(u, v) + h(u) - h(v) \geq 0$。由于无负环，这可以在 $O(|V||E|)$ 时间内完成。

2. **检测 0 环：** 移除 G 中所有新权重 w' 为正的边，保留权重为 0 的边得到子图 G'。

3. G 包含 0 权环当且仅当 G' 包含环。使用 DFS 可在 $O(|V| + |E|)$ 时间内检测 G' 是否有环。总时间复杂度由 Johnson 算法主导，为 $O(|V||E|)$

2. sp-ex 题解

3. 知识点梳理