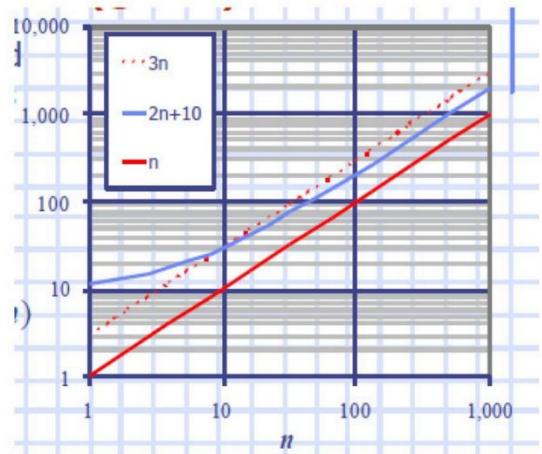


中文名称	英文名称	平均时间复杂度	最坏时间复杂度	最好时间复杂度	空间复杂度	稳定性
选择排序	Selection	n^2	n^2	n^2	1	不稳
冒泡排序	Bubble	n^2	n^2	n	1	稳
插入排序	Insertion	n^2	n^2	n	1	稳
堆排序	heap	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$	1	不稳
希尔排序	Shell	$n^{1.3}$	n^2	n	1	不稳
归并排序	Merge	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$	n	稳
快速排序	Quick	$n \log_2 n$	n^2	$n \log_2 n$	$n \log_2 n$	不稳
桶排序	Bucket	$n + k$	n^2	n	$n + k$	稳
计数排序	Counting	$n + k$	$n + k$	$n + k$	$n + k$	稳
基数排序	Radix	$n * k$	$n * k$	$n * k$	$n + k$	稳

“Big-Oh” Notation

Example: $2n + 10$ is $O(n)$

- $2n + 10 \leq cn$
- $(c - 2)n \geq 10$
- $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$



这张图展示了**大O符号 (Big-Oh notation) **的应用，具体是一个例子：表达式 $2n + 10$ 是 $O(n)$ 的证明过程。以下是图中内容的意义：

1. **示例解释：**我们要证明 $2n + 10$ 是 $O(n)$ ，即找一个常数 c 和 n_0 ，使得对于所有 $n \geq n_0$ ，都有 $2n + 10 \leq c \cdot n$ 成立。

2. **逐步推导：**

- 目标是不等式 $2n + 10 \leq c \cdot n$ 成立。
- 通过变形，得到 $(c - 2)n \geq 10$ 。
- 从中推导出 $n \geq \frac{10}{c-2}$ 。

3. **选择常数：**

- 选择 $c = 3$ 和 $n_0 = 10$ ，这样当 $n \geq 10$ 时，不等式成立。

4. **图表说明：**

- 图中展示了 $2n + 10$ 和 $3n$ 的图形，红线表示 n ，蓝线表示 $2n + 10$ ，从图中可以看到，蓝线（表示 $2n + 10$ ）在某个点之后会被红线（表示 $3n$ ）所界定，说明 $2n + 10$ 的增长速度不会超过 $3n$ 的常数倍。

这个过程展示了如何通过选择合适的常数，证明一个函数属于某个大O类，从而理解该算法的时间复杂度。

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function.
- The statement " $f(n)$ is $O(g(n))$ " means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

这张图片介绍了计算机科学中**渐进符号 (Asymptotic Notation) 的概念，主要讲解了 $\Omega(n)$ (大欧米伽) 和 $\Theta(n)$ (大西塔) 的定义，同时还提到了 $O(n)$ (大O符号) **的概念。

主要概念

1. $\Omega(n)$ (Big-Omega)

- 用于表示**下界**，即某个函数 $f(n)$ 至少增长得和 $g(n)$ 一样快 (或更快)。
- 数学定义：存在常数 c 和 n_0 使得：

$$f(n) \geq cg(n), \quad \forall n \geq n_0$$

- 直观理解：当输入规模 n 变得足够大时， $f(n)$ 的增长速度不会低于 $g(n)$ 。

2. $\Theta(n)$ (Theta)

- 用于表示**精准界限**，即 $f(n)$ 的增长速度与 $g(n)$ 相同。
- 这意味着 $f(n)$ 既是 $O(g(n))$ (上界) 也是 $\Omega(g(n))$ (下界)。
- 数学定义：存在常数 c_1, c_2 和 n_0 使得：

$$c_1g(n) \leq f(n) \leq c_2g(n), \quad \forall n \geq n_0$$

- 直观理解： $f(n)$ 的增长速度与 $g(n)$ 在相同数量级上。

3. $O(n)$ (Big-O)

- 用于表示**上界**，即 $f(n)$ 最多增长得和 $g(n)$ 一样快 (或更慢)。
- 数学定义：存在常数 c 和 n_0 使得：

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

- 直观理解：当输入规模 n 变得足够大时， $f(n)$ 的增长速度不会超过 $g(n)$ 。

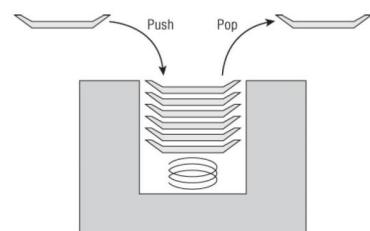
Stack Abstract Data Type

A **stack** is an Abstract Data Type (ADT) and supports the following methods:

- ▶ **push(Obj)** : Insert object *Obj* onto the top of the stack.
- ▶ **pop()** : Remove (and return) the object from the top of the stack. An error occurs if the stack is empty.

In addition to *push* and *pop* there are also typically methods like:

- ▶ **initialize()** : initialize a stack
- ▶ **isEmpty()** : returns a **true** if stack is empty, **false** otherwise.
- ▶ **isFull()** : returns a true if stack is full, false otherwise.



让我们再来看一下前缀表示法 (Prefix Notation) 和中缀表示法 (Infix Notation)，以及它们之间的区别。

1. 中缀表示法 (Infix Notation)

这是我们平常使用的常规算式表达方式，运算符位于操作数之间。

例子：

- $x + y$ ：表示 x 加 y 。
- $x * (y + z)$ ：表示先计算 $y + z$ ，然后再将结果与 x 相乘。

中缀表示法中，运算符的优先级和括号的使用决定了计算顺序，理解起来最直观，但对于计算机处理来说需要进行优先级解析。

2. 前缀表示法 (Prefix Notation)

在前缀表示法中，**运算符位于操作数之前**。也叫做**波兰表示法** (Polish Notation)。前缀表达式的特点是，运算符位于操作数的前面，计算顺序是从右到左，首先看到的是运算符，然后依次处理它后面的操作数。

例子：

- $+ x y$ ：表示 $x + y$ ，运算符 $+$ 在操作数 x 和 y 之前。
- $* + x y z$ ：表示 $x + y$ 的结果与 z 相乘，等同于 $x + y$ 后与 z 相乘： $(x + y) * z$ 。

在前缀表示法中，操作符作用于它后面紧跟的操作数，因此你不需要担心运算符的优先级问题，因为从运算符的顺序来看，它已经完全确定了计算的顺序。

2.1.2.2 后缀表达式/反向波兰表达式 (Postfix notation, Reverse Polish notation)

在后缀表达式中，操作数（如变量或数字）位于算术运算符之前，当遇到一个运算符时，它应用于之前遇到的两个操作数。例如，中缀表达式 $x + y$ 在后缀表达式中写作 $xy+$ ，中缀表达式 $(x + y) * z$ 在后缀表达式中写作 $xy+z*$ ，中缀表达式 $x * (y + z)$ 在后缀表达式中写作 $xyz+*$ ，中缀表达式的 $-x$ 的后缀表达式写作 $x - 1*$ 。

同理我们也可以用后缀表达式也可以转化为中缀表达式，后缀表达式的 $xywz/-*$ 写成中缀表达式为 $x * (y / w - z)$ 。

下图展示了如何用栈处理后缀表达式。

步骤如下：

1. 遇到操作数就将其压入栈中。
2. 遇到操作符就从栈中弹出所需的操作数，执行运算，并将结果压回栈中。
3. 最终，栈顶元素即为表达式的结果。

后缀表示法 (Postfix Notation) 的定义：

在后缀表示法中，运算符位于操作数之后。例如，表达式 $x + y$ 在后缀表示法中写作 $x\ y\ +$ ，这意味着你首先看到操作数 x 和 y ，然后运算符 $+$ 应用于这两个操作数。**运算符总是作用于它前面的两个操作数。**

例子：

- 表达式 $x\ y\ +$ ：表示 $x + y$ ，即先将 x 和 y 相加。
- 表达式 $x\ y\ z\ +\ *$ ：表示 $x * (y + z)$ 。首先计算 $y + z$ ，然后将结果与 x 相乘。

表达式 $-x$ 的后缀表示法：

在普通的中缀表示法中，负号 $-$ 是一个一元运算符，它作用于单个操作数（即 x ）。要用后缀表示法表示一元负号，可以写作 $x\ -1\ *$ 。这意味着：

- x 是操作数，
- -1 是表示负号的一个操作数，
- 然后通过乘法 $*$ 来执行操作。

实际上， $x\ -1\ *$ 是将负数 -1 与 x 相乘，从而表示 $-x$ ，在后缀表示法中，这种方式是可以接受的，因为它符合后缀表达式中运算符作用于之前操作数的规则。

总结：

后缀表示法通过将运算符置于操作数之后来表示表达式，运算符会依次作用于前面的操作数。在表示一元运算符时，也可以通过类似 $x\ -1\ *$ 的方式来实现。

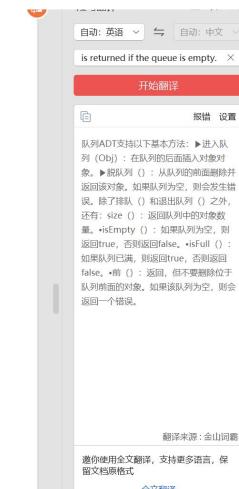
Queue ADT

A queue ADT supports the following fundamental methods:

- ▶ **enqueue(Obj)**: inserts object *Object* the *rear* of the queue.
- ▶ **dequeue()**: removes and returns the object from the *front* of the queue. An error occurs if the queue is *empty*.

In addition to **enqueue()** and **dequeue()** there are also:

- **size()**: Return the number of objects in the queue.
- **isEmpty()**: Returns true if queue is empty, and false otherwise.
- **isFull()**: Returns true if queue is full, and false otherwise.
- **front()**: Return, but do not remove, the object at the front of the queue. An error is returned if the queue is empty.



队列 (Queue)

队列 (Queue) 是一种线性数据结构，遵循先进先出 (FIFO, First In, First Out) 的原则。也就是说，最先进入队列的元素会最先被移除。你可以将它想象成一个排队的场景，比如排队买票，第一个到达的人是第一个买到票的人。

队列的基本操作：

- 入队 (enqueue)：将元素添加到队列的末尾。
- 出队 (dequeue)：移除并返回队列的第一个元素。
- 查看队列头元素 (front/peek)：返回队列的第一个元素，但不移除它。
- 查看队列是否为空 (isEmpty)：检查队列是否为空。
- 队列的大小 (size)：返回队列中元素的数量。

队列的应用：

- 任务调度：操作系统的任务调度通常使用队列，先到先执行。
- 广度优先搜索 (BFS)：广度优先搜索算法中会用到队列来存储待访问的节点。

栈 (Stack)

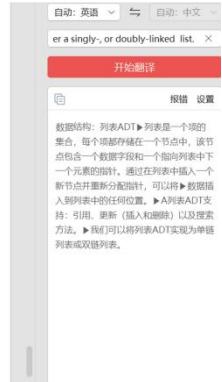
栈 (Stack) 也是一种线性数据结构，但它遵循先进后出 (LIFO, Last In, First Out) 的原则。栈的操作和队列不同，它总是先移除最近加入的元素。你可以将栈想象成一叠盘子，你只能拿到最上面那个盘子。

栈的基本操作：

- 压栈 (push)：将元素添加到栈的顶端。
- 弹栈 (pop)：移除并返回栈顶的元素。
- 查看栈顶元素 (peek/top)：返回栈顶的元素，但不移除它。
- 查看栈是否为空 (isEmpty)：检查栈是否为空。
- 栈的大小 (size)：返回栈中元素的数量。

Data Structures: List ADT

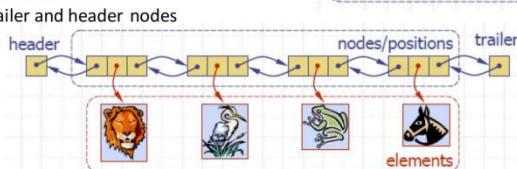
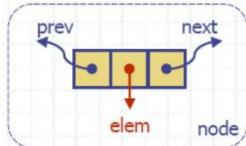
- ▶ A list is a collection of items, with each item being stored in a node which contains a data field and a pointer to the next element in a list.
- ▶ Data can be inserted anywhere in the list by inserting a new node into the list and reassigning pointers.
- ▶ A list ADT supports: referring, update (both insert and delete) as well as searching methods.
- ▶ We can implement the list ADT as either a singly-, or doubly-linked list.



Linked List

► A *node* in a *doubly-linked* list stores two links: a *next* link, pointing to the next element in list, and a *prev* link, pointing to the previous element in the list.

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
 - element
 - link to the previous node
 - link to the next node
- Special trailer and header nodes



From now on, we will concentrate on doubly-linked lists.

e.g2*

2. Multiplying two $n \times n$ square matrices $A = (a_{ij})_{1 \leq i,j \leq n}$ and $B = (b_{ij})_{1 \leq i,j \leq n}$ gives a matrix,

$$C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}$$

wherein $c_{ij} = \sum_{k=1}^n a_{ik} b_{kj} \quad \forall 1 \leq i, j \leq n$

The following pseudo-code multiplies two square matrices A and B to get matrix C:

```

indexing   for i from 1 to n    n    % loop i
indexing   for j from 1 to n    n    % loop j
           C[i][j] = 0;
           for k from 1 to n      % loop k
               C[i][j] = C[i][j] + A[i][k] * B[k][j]; n
           end for
i, j, k end for
           end for

```

$O(n^3)$

Please give its asymptotic notation $T(n) \in O(g(n))$ (i.e $T(n)$ is Big-Oh of $g(n)$).

是这样的，用 n^2 进行遍历，然后 n 进行计算，乘起来就是 n^3

电荷
Spri

Binary Search (cont.)

In order to decrease the size of the range we compare k to the key of the median, mid , of the range, i.e.,

$$mid = \lfloor (low + high)/2 \rfloor$$

Three cases are possible:

- ▷ $k = key(mid)$, the search is completed successfully.
- ▷ $k < key(mid)$, search continued with $high = mid - 1$.
- ▷ $k > key(mid)$, search continued with $low = mid + 1$.

自动: 英语 自动: 中文

o the previous element in the list. ×

开始翻译

双链接列表中的节点存储两个链接。一个是下一个链接，指向列表中的下一个元素，另一个是prev链接，指向列表中的前一个元素。

翻译来源：金山词霸

帮你使用全文翻译，支持更多语言，保留文档原格式

ch continued with low = mid + 1. ×

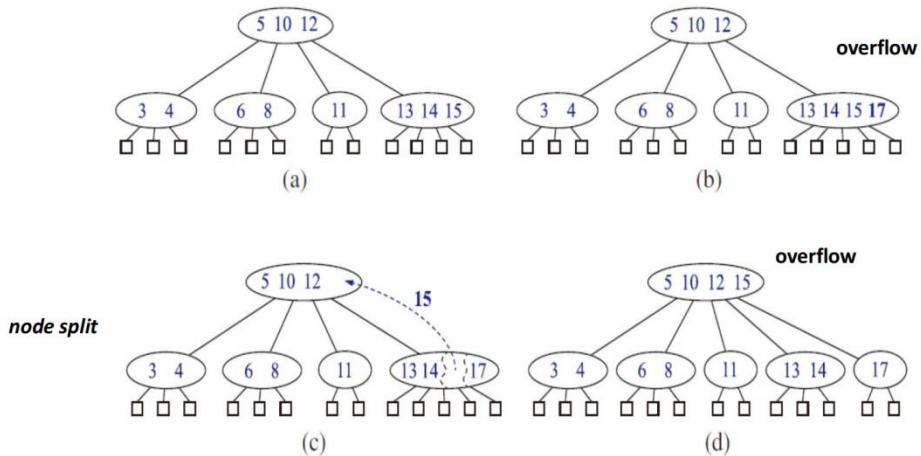
开始翻译

为了减小范围的大小，我们将 k 与范围的中位数，中位数的键进行比较，即中位数 = (低+高) / 2三种情况是可能的。 $\lvert k < \text{键} \rvert$ (中)，搜索成功完成。 $\lvert k < \text{键} \rvert$ (中)，搜索继续与高=中-1。 $\lvert k > \text{键} \rvert$ (中)，搜索继续与低=中+1。

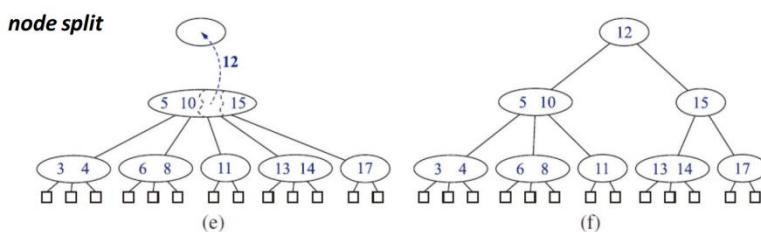
翻译来源：金山词霸

帮你使用全文翻译，支持更多语言，保留文档原格式

(2, 4) tree - Insertion



(2, 4) tree - Insertion



A split operation either eliminates the overflow or propagates it into the parent of the current node.

Indeed, this propagation can continue all the way up to the root of the search tree.
But if it does propagate all the way to the root, it will finally be resolved at that point.



❖ 什么是溢出？

在 2-3-4 树（每个节点最多 4 个 key 和 5 个子节点）中，如果 插入一个新的 key 导致一个节点变成 5-节点（即有 4 个 key 和 5 个子节点），就会发生溢出（Overflow），需要执行 分裂操作（Split）。

❖ 分裂操作（Split）的具体步骤

假设我们有一个 5-节点（有 4 个 key 和 5 个子节点）：



处理方式如下：

1. 分裂成两个节点：

- 左侧变成 3-节点，包含 k1, k2，子节点是 v1, v2, v3。
- 右侧变成 2-节点，包含 k4，子节点是 v4, v5。
- k3 提升到父节点（如果 v 是根节点，则 k3 成为新的根）。

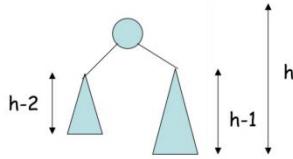


❖ 为什么「溢出可能会传播到父节点」？

- 如果 父节点已经是 4-key（即 5-节点），则 k3 插入后父节点也会溢出，然后 继续分裂，可能导致 整个树的高度增加。
- 最终，可能会创建新的根节点，使树的高度 +1。

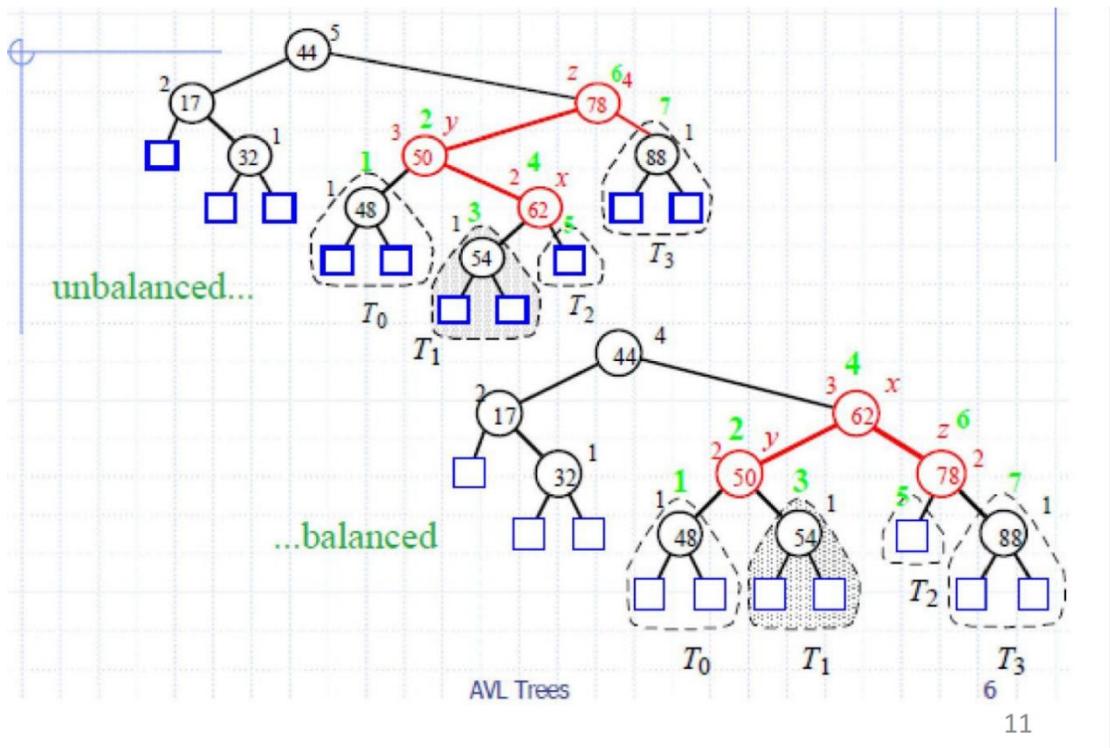
AVL trees (by Adel'son-Vel'skii and Landis)

▷ *Height-Balance Property*: for any node n , the heights of n 's left and right subtrees can differ by at most 1.

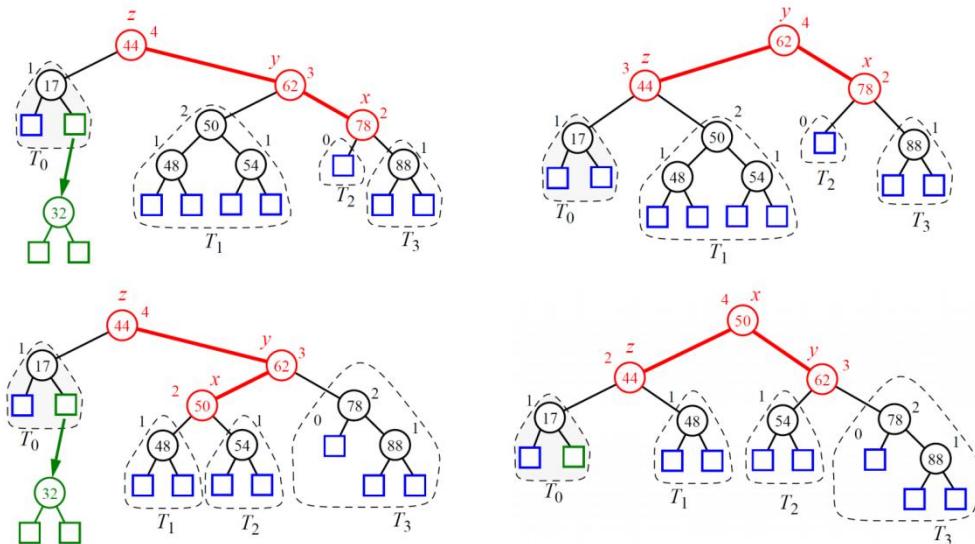


Height of an AVL Tree

- **Theorem:** The height of an AVL tree storing n keys is $O(\log n)$.



Removal in an AVL Trees



Sorting

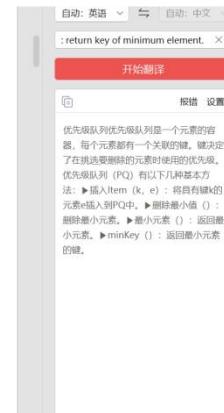
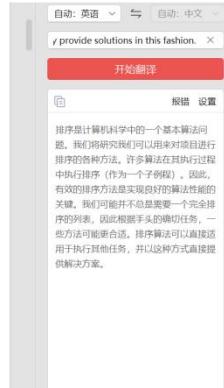
Sorting is a fundamental algorithmic problem in computer science.

We will investigate various methods that we can use to sort items.

Many algorithms perform sorting (as a subroutine) during their execution. Hence, efficient sorting methods are crucial to achieving good algorithmic performance.

We may not always require a fully sorted list, so some methods might be more appropriate depending upon the exact task at hand.

Sorting algorithms might be directly adaptable to perform additional tasks and directly provide solutions in this fashion.



Priority Queues

A **Priority Queue** is a container of elements, each having an associated **key**.

Keys determine the *priority* used in picking elements to be removed.

A **priority Queue** (PQ) has these fundamental methods:

- ▶ **insertItem(k, e)**: insert element e having key k into PQ.
- ▶ **removeMin()**: remove minimum element.
- ▶ **minElement()**: return minimum element.
- ▶ **minKey()**: return key of minimum element.

这张图片讲解了优先队列（Priority Queue）的基本概念和方法。优先队列是一个包含元素的容器，每个元素都有一个与之关联的优先级（key）。这个优先级用于决定哪个元素会被优先移除。

图片中的四个基本操作方法如下：

1. **insertItem(k, e)**: 将一个具有键值 k 的元素 e 插入到优先队列中。
2. **removeMin()**: 移除优先队列中的最小元素。
3. **minElement()**: 返回优先队列中的最小元素。
4. **minKey()**: 返回最小元素的键值。

这些操作都与优先队列中元素的优先级相关，通常会采用堆等数据结构来实现这些操作。

这张图片讲解了二叉树的两种重要类型，分别是**满二叉树** (Full Binary Tree) 和**完全二叉树** (Complete Binary Tree)。

1. 满二叉树 (Full Binary Tree) :

- 定义：如果二叉树中的每个节点要么是叶子节点，要么有且只有两个子节点，那么这个二叉树就是满二叉树。

2. 完全二叉树 (Complete Binary Tree) :

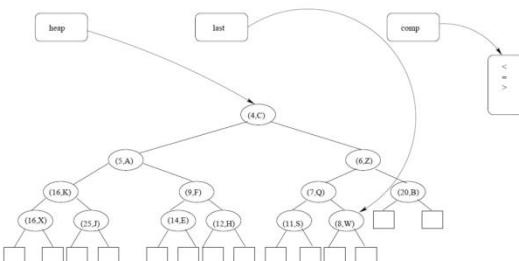
- 定义：完全二叉树的高度为 h ，满足以下两个条件：a) 在深度为 d ($d = 1, 2, \dots, h-1$) 的每一层，节点数必须为 2^d 。b) 在深度为 h 的节点必须尽可能地靠左排列。

3. 近似完全二叉树 (Nearly Complete Binary Tree) :

- 定义：近似完全二叉树的高度为 h ，满足以下条件：a) 在深度 d ($d = 1, 2, \dots, h-1$) 的每一层，节点数为 2^d 。b) 在深度为 h 的节点排列时，尽量靠左。

最后，图片提到有些书籍会将“完全”二叉树用来表示“近似完全”二叉树，并且将“完美二叉树”用来表示“完全二叉树”，二者在某些语境下可能有所混用。

PQ/Heap implementation



heap: A (nearly complete) binary tree T containing elements with keys satisfying the heap-order property, stored in an array.

last: A reference to the last used node of T in this array representation.

comp: A comparator function that defines the total order relation on keys and which is used to maintain the minimum (or maximum) element at the root of T .

短句翻译

自动: 英语 自动: 中文

maximum element at the root of T. .

开始翻译

堆: 一个(几乎完整的)二进制树, 它包含的元素和满足堆属性的键, 存储在一个数组中. 最后: 对这个数组表示中最后使用的节点的引用. comp: 一个比较器函数, 它定义键上的总顺序关系, 并用于保持T根上的最小(或最大)元 素.

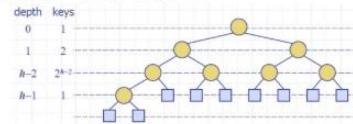
翻译来源: 金山词霸

翻译: 金山词霸

翻译: 金山词霸

From: we apply the complete binary tree property

- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \dots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \dots + 2^{h-2} + 1$
- Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



这个呢

这张图片讲解了堆 (Heap) 在实现优先队列 (PQ) 时的一些理论和性质。

定理:

一个存储 n 个键的堆的高度是 $O(\log n)$ 。

证明:

(应用完全二叉树的性质)

- 设 h 为存储 n 个键的堆的高度。
- 在二叉树的深度 i ($i = 0, 1, \dots, h-2$) 处, 每一层有 2^i 个键。并且在深度 $h-1$ 处至少有 1 个键。因此, 总键数 n 必须大于等于从深度 0 到深度 $h-1$ 的所有节点的键数量总和:

$$n \geq 1 + 2 + 4 + \dots + 2^{h-2} + 1$$
- 因此, $n \geq 2^{h-1}$, 即 $h \leq \log(n) + 1$ 。

图示:

下方的二叉树图形显示了堆的结构。每一层的节点数量遵循二叉树的规律, 且在最底层 (深度 $h-1$) 也至少包含一个节点。

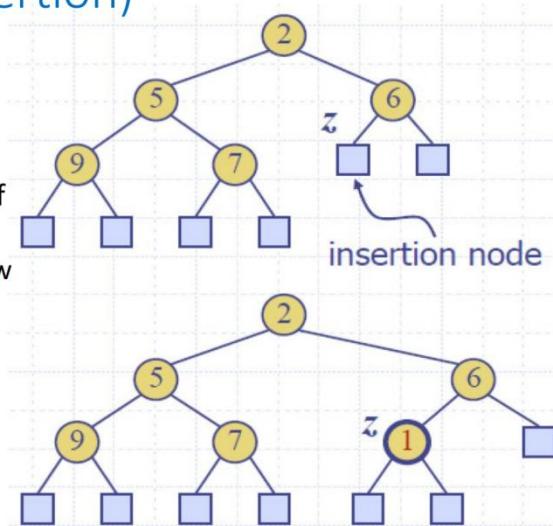
通过这个定理, 我们可以理解堆结构在存储 n 个元素时, 其高度是对数级别的。

Up-heap bubbling (insertion)

❖ Method *insertItem* of the priority queue ADT corresponds to the insertion of a key k to the heap

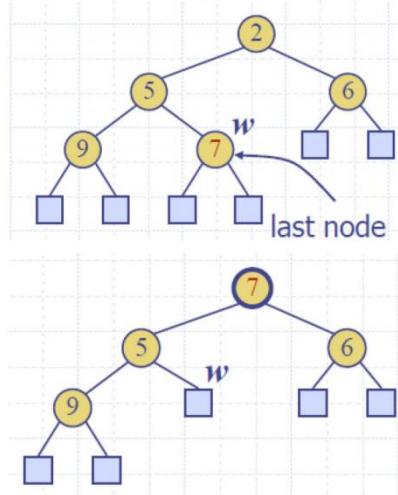
❖ The insertion algorithm consists of three steps

- Find the insertion node z (the new last node)
- Store k at z and expand z into an internal node
- Restore the heap-order property (discussed next)



Down-heap bubbling (removal of top element)

- ❖ Method `removeMin` of the priority queue ADT corresponds to the removal of the root key from the heap
- ❖ The removal algorithm consists of three steps
 - Replace the root key with the key of the last node w
 - Compress w and its children into a leaf
 - Restore the heap-order property (discussed next)



这张图片讲解了堆 (Heap) 中删除操作的过程，特别是“下沉操作” (Down-heap bubbling)，即删除堆顶元素的过程。

删除方法：

`removeMin` 方法对应于从堆中删除根节点（即最小节点）的操作。

删除算法的三步：

1. 将根节点的键值替换为最后一个节点的键值 w 。
 - 这意味着根节点被最后插入的节点替代。
2. 将节点 w 和它的子节点压缩成一个叶子节点。
 - 由于堆的结构会发生变化，所以会进行必要的调整，使得新的节点符合堆的结构。
3. 恢复堆顺序：
 - 为了恢复堆的顺序，新的根节点会下沉（即与其子节点交换位置），直到恢复堆的顺序属性（例如，在最小堆中，根节点应该是最小的）。

图示：

- 上图显示了一个堆，其中 9 是根节点，最后插入的节点 7 被标记为 w ，并准备替代根节点。
- 下图展示了删除操作后的堆结构，新的根节点是 7，并且堆顺序已经通过下沉操作恢复。

这个过程使得堆能够在删除根节点后继续保持堆的特性。

这张图片讲解了堆排序 (Heap-Sorting) 算法的基本概念。

主要内容：

1. 使用堆实现的优先队列：

- 假设有一个包含 n 个元素的优先队列，这个队列是通过堆来实现的。
- **空间复杂度**：使用堆实现的优先队列的空间复杂度是 $O(n)$ 。
- **方法复杂度**：
 - 插入和删除最小元素 (`insertItem` 和 `removeMin` 方法) 需要 $O(\log n)$ 的时间。
 - 获得队列的大小、判断是否为空、获取最小键值 (`size`、`isEmpty`、`minKey` 和 `minElement` 方法) 都可以在 $O(1)$ 时间内完成。

2. 堆排序 (Heap-Sort)：

- 使用基于堆的优先队列，可以在 $O(n \log n)$ 的时间内对 n 个元素进行排序。
- 这个排序算法称为 **堆排序 (Heap-Sort)**。

总结：

堆排序通过将元素插入到堆中并删除堆顶元素来实现 ↓，它的时间复杂度是 $O(n \log n)$ ，且在大多数情况下效率较高。

这张图片讲解了**归并排序 (Merge Sort) **算法的实现过程和步骤。

归并排序的三个步骤：

1. **分解 (Divide)**：将输入序列 S 拆分成两个子序列 S_1 和 S_2 ，每个子序列大约包含 $n/2$ 个元素。
2. **递归 (Recur)**：递归地对两个子序列 S_1 和 S_2 分别进行归并排序。
3. **合并 (Conquer)**：将排序后的两个子序列 S_1 和 S_2 合并成一个有序的序列。

伪代码：

图片中还提供了归并排序的伪代码，具体实现如下：

```
Algorithm mergeSort(S, C)
    Input: sequence S with n elements, comparator C
    Output: sequence S sorted according to C
    If S.size() > 1:
        (S1, S2) ← partition(S, n/2)
        mergeSort(S1, C)
        mergeSort(S2, C)
        S ← merge(S1, S2)
```

Copy Edit

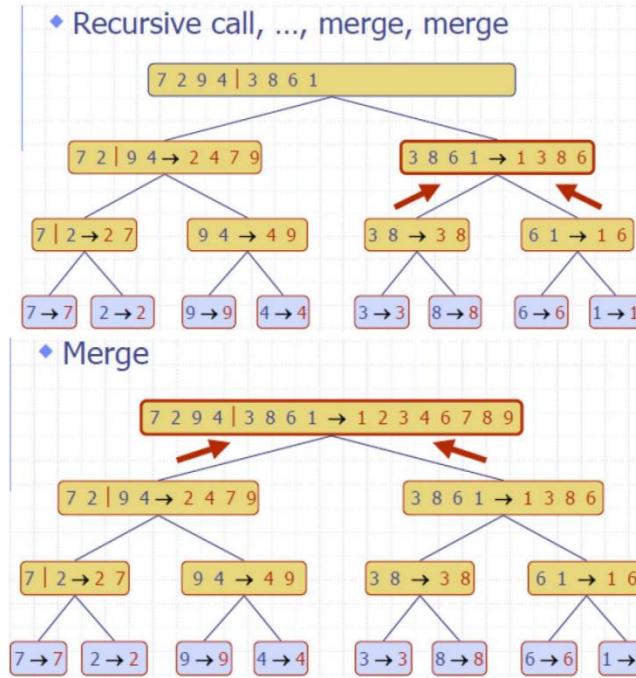
解释：

- `S.size() > 1`：判断序列 S 是否有超过一个元素，如果只有一个元素则不需要排序。
- `partition(S, n/2)`：将序列 S 分成两部分，分别是 S_1 和 S_2 。
- `merge(S1, S2)`：将排序后的两个子序列 S_1 和 S_2 合并成一个排序好的序列 S 。

总结：

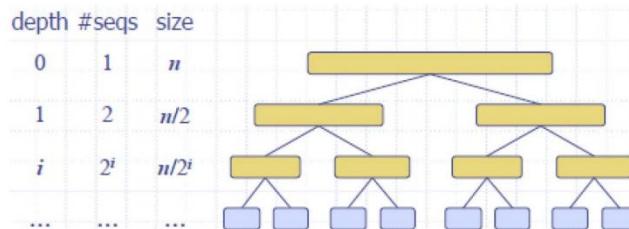
归并排序是一种基于分治法的排序算法，通过递归地[↓]解序列并合并结果，具有较好的时间复杂度 $O(n \log n)$ 。它适用于大规模数据的排序，尤其是在数据存储在外部存储设备时。

MergeSort - Example



MergeSort - Analysis

- ❖ The height h of the merge-sort tree is $O(\log n)$
 - at each recursive call we divide in half the sequence
- ❖ The overall amount or work done at the nodes of depth i is $O(n)$
- ❖ Thus, the total running time of merge-sort is $O(n \log n)$



这张图片讲解了**快速排序 (QuickSort) **算法的基本过程。

快速排序的步骤：

1. 分解 (Divide) :

- 从输入序列 S 中随机选取一个元素 x 作为“枢轴” (pivot) 。
- 然后将序列 S 分为三个部分：
 - L : 所有小于 x 的元素。
 - E : 所有等于 x 的元素。
 - G : 所有大于 x 的元素。

这个过程将序列分为三个子序列。

2. 递归 (Recur) :

- 对子序列 L 和 G 进行递归排序。

3. 合并 (Conquer) :

- 最终将排序后的 L 、 E 和 G 合并在一起，得到最终的有序序列。

图示说明：

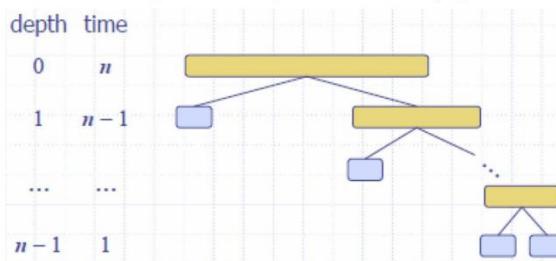
- 在图片中，黄条表示排序中的元素，蓝色条表示选中的枢轴元素 x 。
- 图示展示了如何将序列分成三个部分，并进行递归排序。

总结：

快速排序是一种基于分治法的高效排序算法，通过选择一个枢轴并将元素分区来递归地进行排序。它的时间复杂度为 $O(n \log n)$ ，在平均情况下表现很好，但在最坏情况下可能为 $O(n^2)$ 。

QuickSort -worst-case running time

- ❖ The worst case for quick-sort occurs when the pivot is the unique **minimum** or **maximum** element
- ❖ One of L and G has size $n-1$ and the other has size 0
- ❖ The running time is proportional to the sum
$$n + (n - 1) + \dots + 2 + 1$$
- ❖ Thus, the worst-case running time of quick-sort is $O(n^2)$



递归关系分析:

给定递归关系:

$$T(n) = 2T(n/2) + bn$$

我们使用替代法展开:

1. 第一次展开:

$$T(n) = 2T(n/2) + bn$$

2. 第二次展开 (对 $T(n/2)$ 继续展开) :

$$\begin{aligned} &= 2(2T(n/2^2) + b(n/2)) + bn \\ &= 2^2T(n/2^2) + 2bn \end{aligned}$$

3. 第三次展开:

$$= 2^3T(n/2^3) + 3bn$$

4. 继续展开 k 次:

$$= 2^kT(n/2^k) + kbn$$

终止条件 (Base Case)

当 $n/2^i = 1$ 时, 递归终止, 即:

$$2^i = n \Rightarrow i = \log n$$

最终结果

$$T(n) = bn + bn \log n$$

$$T(n) = O(n \log n)$$

结论

最终, 我们证明了该递归关系的时间复杂度为 $\text{O}(n \log n)$, 这通常出现在分治算法 (如归并排序、快速

The Master Method

It is a "cook-book" method to solve

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{otherwise.} \end{cases}$$

wherein $d \geq 1, a > 0, c > 0, b > 1$

这张图片介绍了**主方法 (Master Method)**，这是一种用于求解递归关系的“食谱式”方法。

主方法的递归形式

$$T(n) = \begin{cases} c, & \text{if } n < d \\ aT(n/b) + f(n), & \text{otherwise} \end{cases}$$

其中：

- $a > 0$: 表示递归调用的子问题个数
- $b > 1$: 表示问题规模缩小的因子
- $f(n)$: 表示额外的非递归工作
- $d \geq 1, c > 0$: 表示递归终止条件

主方法的作用

主方法提供了一种标准化的方法来分析递归关系，特别适用于**分治算法**的时间复杂度计算，例如**归并排序**、**快速排序**、**Strassen矩阵乘法**等。

想深入分析一个具体的递归关系，可以应用主方法的三个案例之一来确定时间复杂度。你想了解具体的主定理及其应用吗？

$T(n) = aT(\frac{n}{b}) + n^x$ 这种形式，将上面的结论写成公式形式：

$$T(n) = \begin{cases} \Theta(n^x) & \text{if } a < b^x \\ \Theta(n^x \log n) & \text{if } a = b^x \\ \Theta(n^{\log_b a}) & \text{if } a > b^x \end{cases}$$

对于任意的 $f(n)$, $T(n) = aT(\frac{n}{b}) + f(n)$ 递归形式的主方法如下：

$$T(n) = \begin{cases} \Theta(f(n)) & \text{if } n^{\log_b a} \text{ 漐进小于 } f(n) \\ \Theta(f(n) \log n) & \text{if } n^{\log_b a} \text{ 漐进等于 } f(n) \\ \Theta(n^{\log_b a}) & \text{if } n^{\log_b a} \text{ 漐进大于 } f(n) \end{cases}$$

算法解析

输入 & 输出

- **输入:** 一个包含互不相同整数的列表 L 。
- **输出:** 列表 L 中的倒置对数量。

递归步骤

1. 基准情况 (Base Case)

- 如果列表 L 只有一个元素，则它是有序的，没有倒置对，直接返回 $(0, \perp)$ 。

2. 递归划分 (Divide)

- 把列表 L 分成两半：
 - **A:** 前 $\lfloor n/2 \rfloor$ 个元素
 - **B:** 后 $\lfloor n/2 \rfloor$ 个元素
- 递归计算两个子数组中的倒置对：
 - k_A : A 中的倒置对数量
 - k_B : B 中的倒置对数量

3. 合并 & 计算跨界倒置对 (Merge & Count)

- 在归并 (Merge) 时统计**跨越 A 和 B 的倒置对**：
 - 如果 $A[i] > B[j]$, 那么 $A[i]$ 和 B 中所有剩余的元素都会形成倒置对。
- 这个部分通过 `MERGEANDCOUNT(A, B)` 完成，返回新的排序后列表 L 和跨界倒置对数 k 。

4. 最终结果

- 倒置对总数: $k_A + k_B + k$

时间复杂度分析

- **递归划分部分:**
 - 每次把列表分成两半，进行递归调用。
 - 类似归并排序，总共递归 $\log n$ 层。
 - **合并 & 计算倒置对部分:**
 - 归并操作的复杂度是 $O(n)$ 。
 - **总复杂度:**
 - 递归层数: $O(\log n)$
 - 每层的合并复杂度: $O(n)$
 - **总时间复杂度 = $O(n \log n)$** ，比暴力方法 ($O(n^2)$) 高效得多。
-

示例

假设 `L = [2, 3, 8, 6, 1]`，计算倒置对：

1. 拆分:

- `A = [2, 3]`
- `B = [8, 6, 1]`

2. 递归计算倒置对:

- `A` 本身没有倒置对（因为 $2 < 3$ ）。
- `B` 递归计算，发现 `(8, 6), (8, 1), (6, 1)` 是倒置对，共 3 个。

3. 合并时计算跨界倒置对:

- `(2,1), (3,1)` 也是倒置对，所以总倒置数 $\downarrow 5$ 。

这其实是同一个方法的两个部分，而不是两种不同的方法。

整体思路是基于“归并排序”来计算倒置对 (inversions)，主要分为两部分：

1. **COUNTINVERSIONS (L)** —— 递归拆分列表，并计算左右两部分的倒置对。
 2. **MERGEANDCOUNT (A, B)** —— 归并两个子列表，并计算跨界倒置对（跨越左右部分的倒置对）。
-

流程概览

1. COUNTINVERSIONS 负责拆分列表

- 递归地把列表拆成两半 A 和 B 。
- 分别计算 **左半部分** 和 **右半部分** 的倒置对数。
- 调用 `MERGEANDCOUNT(A, B)` 计算跨界倒置对。

2. MERGEANDCOUNT 负责合并和计算跨界倒置对

- 在合并两个有序列表时，如果发现 $b_j < a_i$ ，说明 b_j 会与 A 中所有剩下的元素构成倒置对。
 - 通过计数，快速计算出所有跨界倒置对，而不是暴力两两比较。
-

整体复杂度

- **COUNTINVERSIONS** 递归拆分，深度 $O(\log n)$ 。
- **MERGEANDCOUNT** 归并时计算跨界倒置对，复杂度 $O(n)$ 。
- **最终时间复杂度：** $O(n \log n)$ ，比 $O(n^2)$ 的暴力解法快很多。
(↓)

示例

假设有如下物品：

物品	重量 w	收益 b	单位收益 b/w
A	10	60	6.0
B	20	100	5.0
C	30	120	4.0

背包最大容量 $W = 50$ 。

步骤

1. 按照单位收益降序排列：

- A (6.0), B (5.0), C (4.0)

2. 选取物品：

- 先取 A (重量 10, 全选)
- 再取 B (重量 20, 全选)
- 还剩下 20 空间, 从 C 取 $20/30 = 2/3$ (即 20 重量)

3. 计算总收益：

- A 贡献 60
- B 贡献 100
- C 贡献 $(120 \times 20)/30 = 80$
- 总收益 = $60 + 100 + 80 = 240$

Fractional Knapsack Problem (cont.)

In the solution to an FKP, we use a *heap-based* priority queue to store items of S , where the *key* of each item is its *value index* b_i/w_i . In this case, the heap maintains the *maximum* value at the root (max-heap).

Object:	1	2	3	4
Benefit:	7	9	9	2
Weight:	3	4	5	2
Value index:	2.33	2.25	1.8	1



◆ 贪心策略与堆的应用

贪心选择性质 (Greedy-Choice Property)

- **贪心策略**: 在每一步选择中，都选择单位收益（收益/重量）最大的物品。每次选择时，从剩余物品中选择**单位收益最大的物品**，然后继续选择，直到背包无法再容纳更多物品为止。
- **堆 (Heap) 的使用**: 通过堆可以高效地**取出单位收益最大的物品**。每次选择物品时，使用堆的**删除最大值操作**来找到当前单位收益最大的物品，时间复杂度是 $O(\log n)$ 。

堆优化的优势

- 使用堆（通常是最[大堆](#)）的优势在于，每次从堆中删除单位收益最大的物品时，时间复杂度为 $O(\log n)$ 。
- 对于每次选择，堆的操作需要对物品进行排序（按单位收益排序）。选择一个物品后，堆会自动调整，保持堆的性质，使得下一个最大值仍然位于堆顶。

时间复杂度

- 排序的时间复杂度是 $O(n \log n)$ ，因此总时间复杂度由以下几部分组成：
 - 排序物品: $O(n \log n)$
 - 每次选择物品时的堆操作: $O(\log n)$ 每次选择操作最多进行 n 次。
 - 最终算法时间复杂度为 $O(n \log n)$ 。
-

◆ 解释

- 在**每次选择**时，我们优先选择单位收益最高的物品，直到背包满或物品无法继续选择。
- 使用**堆 (Heap)** 来高效管理物品的选择，减少每次选择物品的时间消耗。
- 通过堆排序，能够实现每次选择最大收益物品的[↑]程，同时使得时间复杂度保持在 $O(n \log n)$ 。

这张图片介绍的是区间调度问题 (Interval Scheduling Problem)，它是一个经典的贪心算法问题，涉及到如何在有限的资源 (如单个机器或会议室) 上安排一系列任务或活动。

❖ 区间调度问题简介

我们有一组任务 (或者请求使用某个房间的时间区间等)，并且有一个单独的机器或房间可以处理这些任务。目标是从这组任务中选择一个子集，使得我们能够最大化安排的任务数。

任务安排的限制

- 每个任务都有一个开始时间和结束时间。
 - 如果任务的时间区间重叠，不能同时安排两个任务。
 - 我们只能选择不重叠的任务进行安排。
-

❖ 贪心算法的解决思路

贪心策略

贪心算法的核心思想是：

- 每次选择结束时间最早的任务。这样选择可以给后续任务留下更多的时间，最大化可安排的任务数量。

算法步骤

1. 按照任务的结束时间排序，从早到晚。
2. 选择第一个任务 (结束时间最早)。
3. 对于剩余的任务，每次选择结束时间最早且不与已经选择的任务重叠的任务。
4. 重复直到没有更多可以选择的任务。

时间复杂度

- 排序的时间复杂度是 $O(n \log n)$ ，其中 n 是任务数量。
- 遍历任务的时间复杂度是 $O(n)$ 。

因此，整个算法的时间复杂度是 $O(n \log n)$ ，是一个非常高效的解决方案。

❖ 示例

假设有如下任务的时间区间：

任务	开始时间	结束时间
T1	1	3
T2	2	5
T3	4	6
T4	6	8
T5	5	7

◆ 动态规划的两个核心特性

1. 最优子结构 (Optimal Substructure)

- 一个问题的最优解由其子问题的最优解组成。
- 这意味着可以通过解决小规模的子问题，递推得出整个问题的最优解。

2. 重叠子问题 (Overlapping Subproblems)

- 许多子问题会在递归过程中被多次求解。
 - 传统递归算法会重复计算相同的子问题，而动态规划会存储这些子问题的解，避免重复计算，提高效率。
-

◆ 动态规划的基本思想

• 自底向上 (Bottom-up)

- 先解决最小的子问题，存储它们的解，并利用这些解来解决更大的问题。
 - 这样可以避免递归带来的大量函数调用，提升计算效率。
 - 通过构造一个表 (table)，逐步填充子问题的解，并最终求出整个问题的解。
-

◆ 变体 (Variations)

• 传统动态规划的解通常存储在一个**二维表 (table) **中，但在某些问题中：

- 表 (Table) 可以是三维的 (例如某些涉及多个状态的 DP 问题)。
- 表可以是树结构 (适用于树形 DP，如求解树的最优路径问题)。

这两张图片详细展示了**0-1 背包问题**的解决过程，通过动态规划算法求解给定的背包问题实例。

第一张图片的内容：

1. 问题描述：

- 给定一个整数 W 和一组包含 n 个物品的集合 S ，每个物品有一个正整数的价值和正整数的重量，问题是选择一个总重量不超过 W 的物品子集，要求该子集的总价值最大。该问题可以在 $O(nW)$ 时间内解决。

2. 实例介绍：

- 背包的最大承重 $W = 10$ 。
- 物品的价值和重量如下表所示：

i	1	2	3	4
b_i	25	15	20	36
w_i	7	2	3	6

- 需要解答的是，在最大承重为 $W = 10$ 的情况下，如何选择物品以最大化价值。

第二张图片的内容：

1. 动态规划的表格：

- 使用给定的动态规划算法，构建了一个二维表 $B[k, w]$ ，表示从前 k 个物品中选择，使得背包总重量不超过 w 时的最大价值。
- 表格的计算过程基于递推公式：

$$B[k, w] = \begin{cases} B[k - 1, w], & \text{如果不选择物品 } k \\ \max\{B[k - 1, w], b_k + B[k - 1, w - w_k]\}, & \text{如果选择物品 } k \end{cases}$$

- 通过填充表格，我们得到最大价值。

2. 表格示例：

- 表格显示了所有可能的物品和背包容量的组合，最终的最优解是选择物品 3 和 4，获得的最大价值为 **56**。

3. 解法总结：

- 最终解法是选择物品 3 和 4，其总重量为 $3 + 6 = 9$ （小于 10），最大价值为 **56**。

总结

- 动态规划**通过构建一个表格，逐步计算每个子问题的解，最终得到整体最优解。
- 时间复杂度是 $O(nW)$ ，其中 n 是物品数量， W 是背包容量，适用于背包问题等优化问题。

如果你需要更深入的代码实现或其他问题的解答，欢迎继续提问！😊

(I) 一层循环
解题思路：

- 1° 列出循环变量*t*及每轮循环*i*的变化值
- 2° 找到*t*与*i*的关系
- 3° 确定循环停止条件
- 4° 联立两式，解方程
- 5° 写结果

[例] $i = n * n;$
 $\text{while } (i \neq 1)$
 $i = i / 2;$

[解] $t: 0 \ 1 \ 2 \ 3 \dots$
 $i: n^2 \ \frac{n^2}{2} \ \frac{n^2}{4} \ \frac{n^2}{8} \dots$
 $i = \frac{n^2}{2^t} \quad ①$
 $i = 1 \quad ②$
联立①② $\frac{n^2}{2^t} = 1$
 $\therefore T = O(\log_2 n).$

[例] $x = 0;$
 $\text{while } (n >= (x+1)*(x+1))$
 $x = x + 1;$

[练习1] $\text{int } i = 1;$
 $\text{while } (i \leq n)$
 $i = i + 2;$

[解] $t: 0 \ 1 \ 2 \ 3 \ \dots$
 $x: 0 \ 1 \ 2 \ 3 \ \dots$
 $t = x \quad ①$
 $n = (x+1)^2 \quad ②$
联立①② $\sqrt{n}-1 = t$
 $\therefore T = O(\sqrt{n})$

[练习2] $\text{int } i = 0;$
 $\text{while } (i * i * i \leq n)$
 $i++;$

[练习3] $y = 0;$
 $\text{while } ((y+1) * (y+1) \leq n)$
 $y = y + 1;$

[答案] 1. $O(\log_2 n)$
2. $O(3\sqrt{n})$
3. $O(\sqrt{n})$

很快就可以秒杀这种题

(Ⅲ) 双层循环

解题思路：

- 1° 列出外层循环
中 i 的变化值
- 2° 列出内层语句
的执行次数
- 3° 求和，写结果

[例] `int m=0, i, j;`
`for (i=1; i<=n; i++)`
`for (j=1; j<=2*i; j++)`
`m++;`

[解] $i: 1 \ 2 \ 3 \ 4 \ 5 \ \dots \ n$

内层语句	□	□	□	□
执行行	□	□	□	□
次数	(2次)	(4次)	(6次)	(8次)	...	(2n次)

将其表示求和 $\frac{n(2+2n)}{2} = n(n+1)$

$T = O(n^2)$

(Ⅳ) 多层循环

解题思路：

方法一：

抽象为计算三维体积，

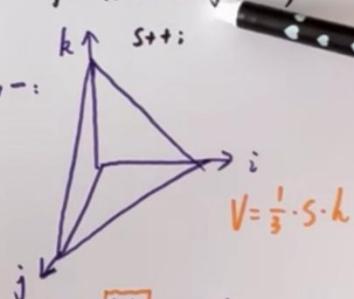
方法二：

列式求和

[例] `for (i=0; i<=n; i++)`

`for (j=0; j<=i; j++)`

`for (k=0; k<j; k++)`



$$\text{方法二: } \sum_{i=0}^n \sum_{j=0}^i \sum_{k=0}^{j-1} 1 = \sum_{i=0}^n \sum_{j=0}^i (j-1-i+1)$$

$$= \sum_{i=0}^n \sum_{j=0}^i j = \sum_{i=0}^n \frac{j(j+1)}{2}$$

$$= \frac{1}{2} \sum_{i=0}^n (i^2 + i) = \frac{1}{2} \sum_{i=0}^n i^2 + \frac{1}{2} \sum_{i=0}^n i = \frac{n(n+1)}{2} = O(n^3)$$

Q3. 代码分析

```
c  
  
for i = 1 to n do  
    for j = i to 2*i do  
        output hello
```

复制 编辑

1) 表达为求和形式:

外层 i 从 1 到 n, 内层 j 从 i 到 2i, 共输出 $2i - i + 1 = i + 1$ 次:

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

2) 化简求和:

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

- 时间复杂度: $O(n^2)$

如果你需要我整理成表格、中文报告或者要可复制的代码, 欢迎告诉我!

详细讲讲第三题

当然可以! 我们来详细拆解 Q3 的代码逻辑, 一步步分析 $T(n)$ 表达式和最终的时间复杂度。

当然可以！我们来详细拆解 Q3 的代码逻辑，一步步分析 $T(n)$ 表达式和最终的时间复杂度。

✓ 原代码：

```
c

for i = 1 to n do
    for j = i to 2*i do
        output hello
```

复制 编辑

🔍 逐步分析

我们想要计算的是：“hello” 被输出了多少次？

这意味着我们要找出整个代码中内层循环的执行次数总和，这也就是 $T(n)$ 。

✿ 外层循环（变量 i ）

- i 从 1 到 n ，所以外层循环总共执行 n 次。

✿ 内层循环（变量 j ）

- 每次 i 固定时， j 从 i 到 $2*i$ 。
- 所以 j 的次数为：

$$2i - i + 1 = i + 1 \text{ 次}$$

✓ 表达为求和公式

我们用总输出次数 $T(n)$ 表示：

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

把它拆成两部分：

$$\begin{aligned} T(n) &= \sum_{i=1}^n i + \sum_{i=1}^n 1 \\ T(n) &= \frac{n(n+1)}{2} + n \end{aligned}$$

* 进一步简化

$$T(n) = \frac{n(n+1)}{2} + n = \frac{n(n+1) + 2n}{2} = \frac{n(n+3)}{2}$$

这是输出语句的准确次数。

💡 时间复杂度分析

我们只关心渐进上界，即 **Big-O 表示法**。

$$T(n) = \frac{n(n+3)}{2} = O(n^2)$$

✓ 结论：

1. 输出 "hello" 的总次数为：

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

2. 时间复杂度为： $O(n^2)$

把内外层分开用 Σ 表示给我看看

好的，我们将第三题中这段代码的内外层循环分别用两个 Σ （求和符号）表示，让你更清晰地看到每一层循环的结构和总次数的推导过程。

✓ 原代码：

c

```
for i = 1 to n do
    for j = i to 2*i do
        output hello
```

复制 编辑

□ 用双重求和表示：

我们定义 $T(n)$ 为输出"hello"的总次数，即：

$$T(n) = \sum_{i=1}^n \sum_{j=i}^{2i} 1$$

这里的含义是：

- 外层： i 从1到 n
- 内层：每个 i 固定时， j 从 i 到 $2i$ ，输出1次（输出一次hello）

✳ 先看内层的求和：

内层 $\sum_{j=i}^{2i} 1$ 的意思就是从 $j = i$ 到 $j = 2i$ 每次加1，一共加几次？

答案是：

$$2i - i + 1 = i + 1$$

所以：



所以：

$$\sum_{j=i}^{2i} 1 = i + 1$$

回到外层：

把内层求和结果代入：

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

展开求和公式：

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

★ 最终表达总结：

- 完整双重求和表达式：

$$T(n) = \sum_{i=1}^n \sum_{j=i}^{2i} 1$$

- 化简后：

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

- 时间复杂度：

$$O(n^2)$$

讲解一下 Σ 是怎么用的，他右边的1不应该是乘以1就是可以去掉的吗

非常棒的问题！你对 Σ （求和符号）的理解已经很接近准确了。我们来详细讲解一下 Σ 的含义和使用方式，包括你提到的那个“右边的1”。

1.2 一、 Σ 的基本定义

设有：

$$\sum_{k=a}^b f(k)$$

意思是：

对从 $k = a$ 到 $k = b$ ，依次把 $f(k)$ 的值加起来。

比如：

$$\sum_{i=1}^4 i = 1 + 2 + 3 + 4 = 10$$

2. 你的例子：

$$\sum_{j=i}^{2i} 1$$

这个表示什么意思呢？

它的意思是：

从 $j = i$ 到 $j = 2i$ ，每次都加上 1。

举个例子：当 $i = 3$ 时，这个求和是：

$$\sum_{j=i}^{2i} 1 = 1 + \downarrow + 1 + 1 = 4$$

💡 那右边的 1 能不能去掉？

不能完全“去掉”，但我们可以简化表达。

Σ 后面写的是要加的内容。即使是常数 1，它也代表每一项都加 1。

我们常写：

$$\sum_{j=a}^b 1 = b - a + 1$$

这是因为你从 $j = a$ 到 $j = b$ ，总共有 $b - a + 1$ 项，每项都是 1。

✓ 所以你是对的：“乘以 1”不会改变总和，但它的存在表示我们是在“计数”，不能完全不写它，特别是在表达你在加什么的时候。

✓ 更直观的理解：

表达式	含义	值
$\sum_{j=3}^6 1$	加 1 四次 ($j = 3, 4, 5, 6$)	4
$\sum_{j=3}^6 j$	加 j 的值： $3+4+5+6$	18
$\sum_{j=3}^6 j^2$	加 j 的平方： $9+16+25+36$	86

所以：

- $\sum_{j=i}^{2i} 1$ 是在计数，从 $j = i$ 到 $2i$ 有多少个数。
- 最终结果是： $i + 1$

● 总结一句话：

Σ 是“加”的符号，后面的表达式不能省略，它说明我们每次加的是什么值。

如果是 1，就是在计数项数；如果是变量、函数或表达式，那就每次加这个表达式的结果。

✖ 为什么你会觉得是 3 次?

你把它理解成了类似“从 3 开始，每次加 1，直到等于 6”，像是在执行一串赋值计算：

text

复制 编辑

```
3 + 1 = 4  
4 + 1 = 5  
5 + 1 = 6
```

这其实更像是你在模拟一个变量变化的过程，而不是“对每个 j 的值执行某个操作”。

但 Σ 求和 是统计从起始值到结束值之间，每一项加什么内容，而不是每一项怎么变换。

💡 正确认知总结：

- Σ 是在对每个 j 的值执行求和操作，不是做连加计算。
- $\sum_{j=a}^b 1$ 就是问：“从 $j=a$ 到 $j=b$ 有多少项？”

$$\sum_{j=i}^{2i} 1 = \text{项数} = (2i - i + 1) = i + 1$$

✓ 用你提的例子 ($i = 3$) 再看一遍：

我们要执行的是：

$$\sum_{j=3}^6 1$$

展开就是：

$$1 + 1 + 1 + 1 = 4$$

因为 j 的取值是：3, 4, 5, 6，一共 4 项

所以结果是 4，不是 3。

Q1. 判断是否是最大堆 (Max-Heap)

给定数组：

csharp

复制 编辑

```
[20, 15, 18, 7, 9, 5, 12, 3, 6, 2]
```

规则：

对于索引 i ，它的左子节点在 $2i + 1$ ，右子节点在 $2i + 2$ 。最大堆要求每个父节点 \geq 子节点。

我们来验证一下（用 0-based 索引）：

i	value	left(i)	right(i)	验证
0	20	1(15)	2(18)	<input checked="" type="checkbox"/> $20 \geq 15, 18$
1	15	3(7)	4(9)	<input checked="" type="checkbox"/> $15 \geq 7, 9$
2	18	5(5)	6(12)	<input checked="" type="checkbox"/> $18 \geq 5, 12$
3	7	7(3)	8(6)	<input checked="" type="checkbox"/> $7 \geq 3, 6$
4	9	9(2)	—	<input checked="" type="checkbox"/> $9 \geq 2$

其余节点没有子节点。

结论：这是一个合法的最大堆 (max-heap)

Q2. AVL树中插入52后的最终结构

题目给出一棵 AVL 树并让你插入 52，但未提供初始结构。

由于原始树未展示，我无法直接画图。你可以上传那棵初始树的图，我再继续。

第二题：0-1 背包问题（动态规划求解）

题设：

- 物品数：4
- 背包容量：5
- 价值 (Benefits) $B = [4, 5, 2, 8]$
- 重量 (Weights) $W = [2, 2, 1, 3]$

我们使用动态规划构建一个表格 $dp[i][w]$ ，表示前 i 个物品在总重量不超过 w 的情况下能获得的最大价值。

步骤如下：

1. 建立大小为 $(n+1) \times (\text{capacity}+1)$ 的二维数组 dp ， $n=4$, $\text{capacity}=5$ 。
2. 初始化第一行 ($i=0$) 和第一列 ($w=0$) 为0。
3. 使用如下递推关系：

```
if W[i-1] > w:  
    dp[i][w] = dp[i-1][w]  
else:  
    dp[i][w] = max(dp[i-1][w], B[i-1] + dp[i-1][w - W[i-1]])
```

构建表格如下（行是物品，列是容量）：

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	4	4	4	4
2	0	0	5	5	9	9
3	0	2	5	7	9	11
4	0	2	5	8	10	13

最终答案：最大价值是 13。

选择的物品为第 2、3 和第 4 个（索引从 1 开始），即价值为 $5 + 2 + 8 = 15$ 但背包容量限制，只能装第2和第4（共5单位重量），总价值 13。

```

T3(int n) {
    if (n <= 2)
        return 1;
    else
        return T3(floor(sqrt(n))) + 1;
}

```

中文解释与分析:

这个递归函数的特点是每次调用中，把 `n` 变为 `floor(sqrt(n))`，然后继续递归，直到 `n <= 2` 为止。

我们设递归次数为 `k`，即 `T3` 需要调用多少次才会停止。

我们来解这个方程：

- 第一次调用： n
- 第二次调用： \sqrt{n}
- 第三次调用： $\sqrt{\sqrt{n}} = n^{(1/4)}$
- 第四次调用： $n^{(1/8)}$
- ...
- 第 k 次调用： $n^{(1/2^k)}$

终止条件是： $n^{(1/2^k)} \leq 2$

取对数两边：

```

pgsql                                         ⌂ 复制   ⌂ 编辑

log(n^(1/2^k)) ≤ log(2)
(1/2^k) * log n ≤ log 2
1/2^k ≤ log 2 / log n
2^k ≥ log n / log 2 = log2(n)
k ≥ log2(log2(n))

```

终止条件是: $n^{(1/2^k)} \leq 2$

取对数两边:

```
pgsql
log(n^(1/2^k)) ≤ log(2)
(1/2^k) * log n ≤ log 2
1/2^k ≤ log 2 / log n
2^k ≥ log n / log 2 = log_2(n)
k ≥ log_2(log_2(n))
```

复制 编辑

因此, 递归深度是:

$T3(n) \in \Theta(\log \log n)$

最终答案:

SCSS

复制 编辑

$T3(n)$ 的紧渐进上界是 $\Theta(\log \log n)$

这是因为每次递归都将 n 替换为 \sqrt{n} , 使得函数收敛得非常快, 直到 $n \leq 2$ 为止。

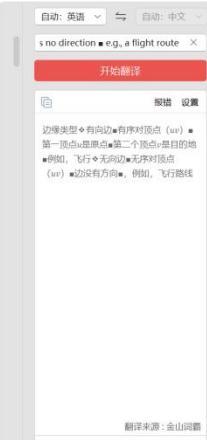
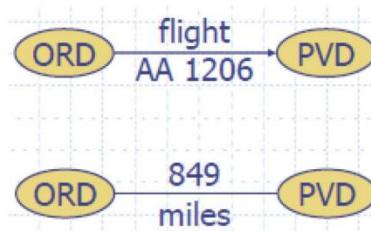
如果你还需要图示或用中文完整解释给其他人听, 我也可以帮你整理一份说明文档。

Graphs

Edge Types

❖ Directed edge

- ordered pair of vertices (u,v)
- first vertex u is the origin
- second vertex v is the destination
- e.g., a flight



Graphs

Terminology

❖ End vertices (or endpoints) of an edge

- Two vertices that an edge connects
- U and V are the endpoints of a

❖ Incident edges on a vertex

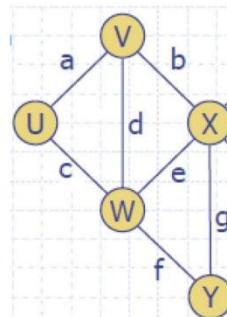
- all the edges that connect to that vertex
- a, d , and b are incident on V

❖ Adjacent vertices

- Two vertices that are directly connected by an edge
- U and V are adjacent

❖ Degree of a vertex

- Number of edges that are incident to the vertex
- V has degree 3



这张图是关于图论中一些基本概念的介绍，下面是每一部分的中文解释：

◆ 图 (Graphs)

1. 子图 (Subgraph)

子图是指一个图 G 的子集组成的图 H ，这个子图的顶点和边都是原图 G 中顶点和边的子集。

- 图中上半部分的图示说明了这一点：原图 G 边用虚线表示，而子图 H 是其中的一个部分，用实线表示。

2. 生成子图 (Spanning Subgraph)

生成子图是图 G 的一个子图，但它包含了 G 的所有顶点，只是边的数量可能比原图少。

- 图中下半部分的图示说明了生成子图：所有的顶点都在，但有些边（用虚线表示）被去掉了。

3. 连通图 (Connected Graph)

如果图中任意两个不同的顶点之间都有路径可以相连，那么这个图就是连通的。

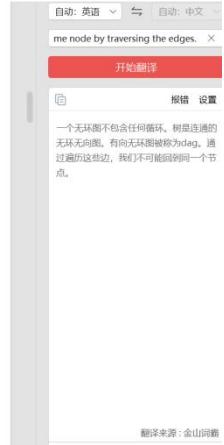
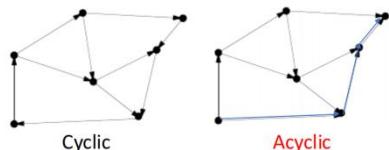
4. 连通分量 (Connected Components)

如果一个图不是连通图，那么可以将它分成若干个最大连通的子图，这些子图被称为该图的连通分量。

Graphs

An acyclic graph does not contain any cycles. Trees are connected acyclic undirected graphs.

Directed acyclic graphs are called DAGs. It's impossible to come back to the same node by traversing the edges.

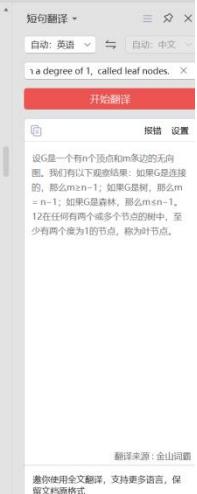
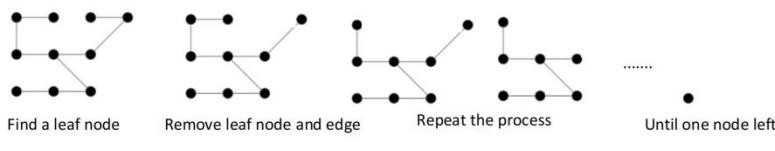


Graphs

Let G be an undirected graph with n vertices and m edges. We have the following observations:

- If G is connected, then $m \geq n - 1$;
- If G is a tree, then $m = n - 1$;
- If G is a forest, then $m \leq n - 1$.

In any tree with two or more nodes, there are at least two nodes with a degree of 1, called leaf nodes.



♣ 图 (Graphs) 性质总结:

设 G 是一个有 n 个顶点和 m 条边的 **无向图**, 有以下观察结果:

1. **如果 G 是连通的 (connected)** , 那么边数 $m \geq n - 1$
► 换句话说, 为了使图连通, 至少需要 $n - 1$ 条边。
 2. **如果 G 是树 (tree)** , 那么 $m = n - 1$
► 树是一种**无环、连通**的图, 边数刚好是 $n - 1$ 。
 3. **如果 G 是森林 (forest)** , 那么 $m \leq n - 1$
► 森林是由若干棵树组成的图 (也就是说, 不一定连通) , 边数小于等于 $n - 1$ 。
-

⌚ 树的叶子节点 (Leaf Nodes)

在一棵有两个或更多节点的树中, **至少会有两个度数为 1 的节点**, 这些节点称为**叶节点 (Leaf nodes)**。

图示过程说明:

这个过程展示了如何不断地**删除叶节点**来分析或简化一棵树:

1. **找到一个叶节点** (度为 1 的节点)。
2. **删除该叶节点和与之相连的边。**
3. **重复上述过程。**
4. 最终, **只剩下一个节点**。

这个过程中总共移除了 $n - 1$ 个节点和 $n - 1$ 条边——这也再次说明了树的一个重要性质:

- ✓ 树中边数始终为 $n - 1$ 。

邻接表的定义：

在图论中，邻接表是一种用来表示图的方式。它告诉我们每个顶点都连接到哪些其他顶点。

可以把它理解成：

“每个节点有一个朋友名单，列出它直接连着的其他节点”。

举个例子：

假设我们有一个图如下：

mathematica

复制 编辑

图中的边：

A - B
A - E
B - C
C - D
E - F
F - C

那么邻接表就可以写成：

mathematica

复制 编辑

A: B, E
B: A, C
C: B, D, F
D: C
E: A, F
F: E, C

✓ 简单回答：

因为 0 已经被访问过了，而且处理完了，所以从队列中被移除了。

■ BFS 中队列的工作方式（很像排队打饭）：

1. 起点 0 被加入队列
2. 队列 = $[0]$
3. 取出队头 0 ，访问它 → 处理它的邻居： $1, 2, 4 \rightarrow$ 加入队列
4. 这时队列 = $[1, 2, 4]$
5. 既然 0 已处理完，就从队列中移除了

💡 所以队列是这样变化的：

步骤	处理节点	队列变化	说明
1	0	$[0] \rightarrow [1, 2, 4]$	访问 0 ，加入邻居
2	1	$[1, 2, 4] \rightarrow [2, 4, 3]$	访问 1 ，加入新邻居 3
...		...	

0 是 BFS 中最早被访问的，所以它最早被“出队”。

DFS vs. BFS

DFS traversal:

- Produces a spanning tree, such that, all non-tree edges are *back edges*.
- It is used for detecting cycles.

BFS traversal:

- Finds *shortest paths* in a graph.
- Explores vertices level by level, ensuring the shortest path to each vertex.



这张图是 **DFS** (深度优先搜索) 和 **BFS** (广度优先搜索) 的对比说明。下面我来给你详细解释每一句话的意思，还有它在图论里的用途 🤝

● **DFS Traversal (深度优先遍历) :**

► "Produces a spanning tree, such that all non-tree edges are back edges."

- 意思是：DFS 生成一棵生成树（spanning tree），也就是说访问所有顶点、形成一个“树状结构”。
- 所有 **不是生成树中的边**（non-tree edges），如果连接到已经访问过的祖先节点，就叫 **back edge**（回边）。
- ❷ **Back edge** 的存在就说明图中有环（cycle）。

► "It is used for detecting cycles."

- 这句话就是在强调上面那点：
 - DFS 可以用来判断图中有没有环（特别是有向图中用来判断是否是**有向无环图 DAG**）。
-

● **BFS Traversal (广度优先遍历) :**

► "Finds shortest paths in a graph."

- BFS 在**无权图**中可以找出从起点到所有点的**最短路径**（最少经过的边数）。
- ⚠ 注意：这个只对**所有边权相等或无权图**有效。如果图中有权重，就要用 Dijkstra 算法了。

► "Explores vertices level by level, ensuring the shortest path to each vertex."

- BFS 是一层一层访问节点的：先访问距离起点为 1 的节点，再是距离为 2 的，以此类推。
- 所以它天然就保证了访问到每个节点的路径就是最短路径。

▣ **DFS vs BFS 总结对比表**

特性	DFS	BFS
结构	生成树，包含回边	生成最短路径树（层级结构）
遍历顺序	尽量往深处走	一层一层遍历（层次广度）
应用	环检测、拓扑排序、连通性分析	最短路径（无权图）、层级结构遍历
实现方式	通常用栈（或递归）	通常用队列

如果你想，我也可以用图或动画来演示 **DFS 如何检测环** 或 **BFS 如何找最短路径**，这样理解会更直观，要不要来一波？😊

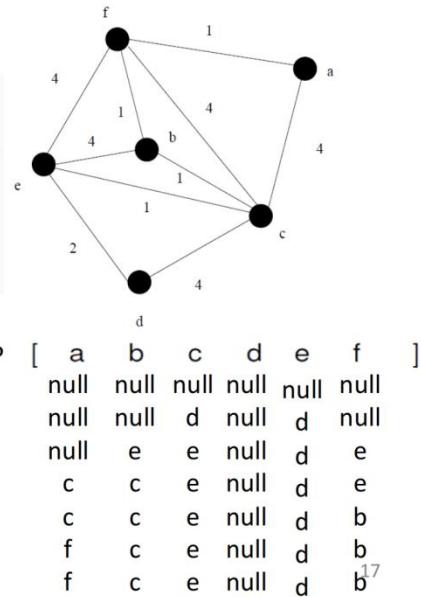
Dijkstra's algorithm

```

3     D[u] ← +∞
4     P[u] ← null // Initialize predecessor of each vertex to null
5     Let Q be a priority queue (heap)
6     Insert all vertices of G into Q using the D labels as keys.
7     Let C be an empty set // Initialize C as an empty set to store finalized vertices
8     while NOT EMPTY(Q) do
9         u ← REMOVE_MIN(Q) // Extract the vertex with minimum distance from Q
10        Add u to C // Add this vertex to the set of finalized vertices
11        for each z s.t. (u, z) ∈ E do
12            if D[u] + w(u, z) < D[z] then
13                D[z] ← D[u] + w(u, z)
14                key(z) ← D[z]
15                P[z] ← u // Update the predecessor of z to u

```

D	a	b	c	d	e	f
∞	∞	∞	0	∞	∞	
∞	∞	4	0	2	∞	
∞	6	3	0	2	6	
7	4	3	0	2	6	
7	4	3	0	2	5	
6	4	3	0	2	5	
6	4	3	0	2	5	



当然可以！这张图讲的是**最小生成树 (Minimum Spanning Tree, MST) **的概念。我们来逐条解释：

♣ 最小生成树 (MST) 是什么？

设图 G 是一个无向图，图中的每条边都有一个权重或代价。

最小生成树 (MST) 是图 G 的一个子图，它满足以下三个条件：

1. 包含图 G 的所有顶点

👉 换句话说，最小生成树覆盖了所有节点。

2. 不包含任何回路 (cycle)

👉 所以它形成的是棵树（无环图）。

3. 在所有可能的生成树中，边的总权重最小

👉 我们要找的就是那种连接所有点、代价最小的方案。

❖ 最小生成树的应用

- **网络设计**：比如计算机网络、通信网络的最小成本连接方案。
- **近似算法 / 聚类分析** 的基础：在一些机器学习或数据挖掘中，MST 也会用来做数据的聚类。

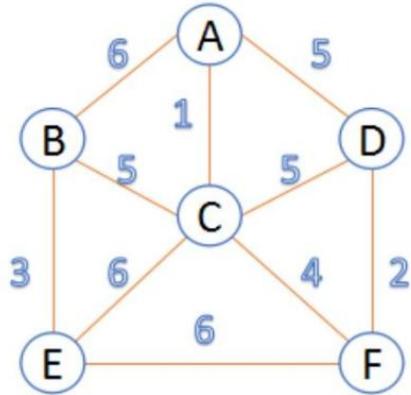
大意

在一张连通图 G 里，有 n 个节点和 m 条边，第 i 条边的权值为 w_i 。我们设图 G 的最小生成树为 T 。

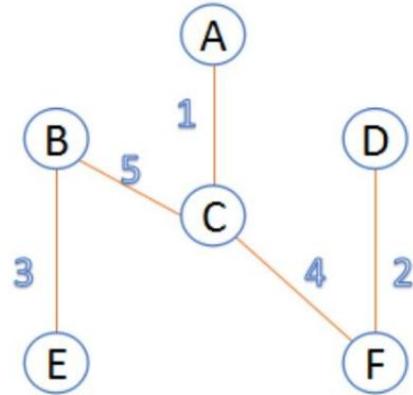
那么，图 G 的最小生成树 T 必须具备以下条件：

- T 必须包括 G 的所有节点；
- T 的边数必须等于 $n - 1$ ；
- T 的边权和必须在所有生成树中最小；
- T 必须为 G 的子图。

假设我们有如下的连通图 G （左），那么它的最小生成树 T 如图所示（右）。



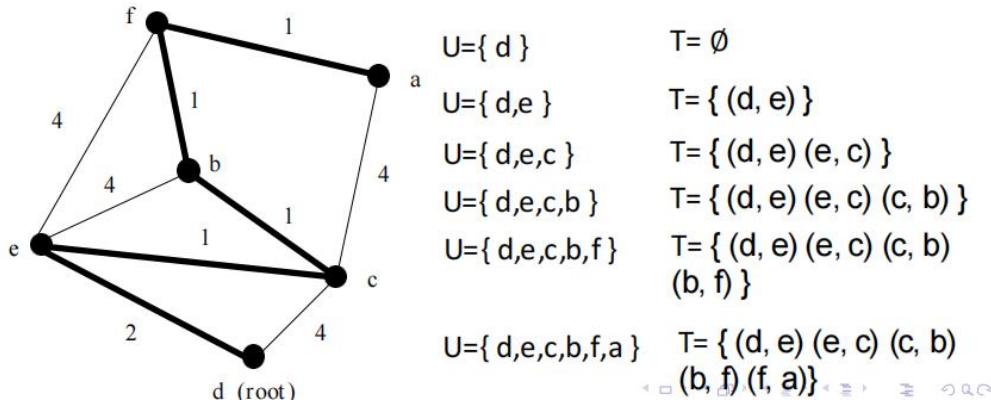
连通网G



最小生成树

Prim's Algorithm

```
T = ∅; #(Set T to be empty.)  
U = { r }; #(Start with vertex r in set U.)  
while (U ≠ V) #(Continue until U contains all vertices.)  
    let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;  
    #(Find the edge with the lowest weight where one end is in U and the other is in V-U.)  
    T = T ∪ {(u, v)}; #(Add this edge to T.)  
    U = U ∪ {v} #(Move vertex v from V-U to U.)
```



六、Prim 与 Kruskal 的对比

属性	Prim 算法	Kruskal 算法
起点依赖	是, 从一个点出发	否, 直接按边排序
适合稠密图	<input checked="" type="checkbox"/>	<input type="checkbox"/> (效率低)
实现方式	堆 + 邻接表	并查集 + 边集排序

七、应用场景

- 网络电缆布线
- 路网建设成本优化
- 聚类分析 (如Kruskal也常用于此)
- 游戏地图生成 (迷宫连通)

目 Kruskal 算法步骤 (如图中所示)

1. 将所有边按权重升序排序。
 2. 从最小的边开始检查：
 - 如果这条边连接的两个点 **不在同一棵树中** (即不会成环) , 那么就加入到生成树中。
 - 如果会成环, 就跳过这条边。
 3. 重复以上步骤, 直到构成 $n-1$ 条边为止 (n 是顶点数) 。
-

✿ 例子 (更直观地理解)

假设我们有这样几条边 (权重) :

```
css
A - B (1)
B - C (3)
A - C (2)
C - D (4)
```

复制 编辑

步骤如下:

1. 排序: A-B(1), A-C(2), B-C(3), C-D(4)
2. A-B: 加入
3. A-C: 不会成环, 加入
4. B-C: 成环 (A-B-C-A) , 跳过
5. C-D: 加入

最终的生成树为: A-B, A-C, C-D



当然可以! 你现在看到的这页讲的是**流网络 (Flow Network)** , 它是网络流问题的基础概念, 下面我详细讲讲 

⌚ 什么是 Flow Network (流网络) ?

一个流网络是一个 **带权有向图 (weighted directed graph)** , 具有如下特性:

基本组成:

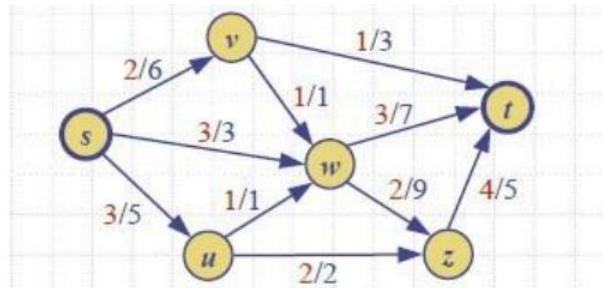
1. **顶点集合 V** : 节点代表网络中的点。
2. **边集合 E** : 每条边都有一个 **非负整数容量 (capacity)** , 用 $c(e)$ 表示, 代表这条边最多能“承载”多少流量。
3. **源点 s (source)** : 唯一的起点, 没有任何入边。
4. **汇点 t (sink)** : 唯一的终点, 没有任何出边。

- ▷ A flow f for a network N is an assignment of an integer value $f(e)$ to each edge e that satisfies the following properties:

1. Capacity Rule: For each edge e , $0 \leq f(e) \leq c(e)$
2. Conservation Rule: For each vertex $v \neq s, t$

$$\sum_{e \in E^-(v)} f(e) = \sum_{e \in E^+(v)} f(e)$$

where $E^-(v)$ and $E^+(v)$ are the incoming and outgoing edges of v , respectively.



什么是“流” (Flow) ?

在一个流网络中，流 f 是对每条边 e 赋予一个非负整数值 $f(e)$ ，表示通过这条边的实际流量。

但是这个“流”必须满足两个关键规则：

✓ 两大规则 (Flow 的性质)

1. 容量约束 (Capacity Rule)

每条边上的流量不能超过其容量。

用数学表示就是：

$$0 \leq f(e) \leq c(e)$$

意思是：实际流 $f(e)$ 一定在 0 和这条边最大容量 $c(e)$ 之间。

2. 守恒规则 (Conservation Rule)

除了源点 s 和汇点 t 之外，每个点的流入 = 流出。

数学表达式如下：

$$\sum_{e \in E^-(v)} f(e) = \sum_{e \in E^+(v)} f(e) \quad \text{对于所有 } v \neq s, t$$

这里：

- $E^-(v)$: 所有进入节点 v 的边 (入边)
- $E^+(v)$: 所有从节点 v 发出的边 (出边)

简单理解就是：“流进来多少，流出去多少”，不能无中生有或者凭空消失。

📘 英文内容讲的是什么？

❖ Cut (割) - 定义：

流网络 (flow network) 中有一个很重要的概念叫做 **cut (割)**，它是把网络中的顶点集合分成两部分：

- 一部分叫 V_s ，里面包含源点 s
- 另一部分叫 V_t ，里面包含汇点 t

然后：

- **前向边 (Forward edge)**：是从 V_s 指向 V_t 的边
- **后向边 (Backward edge)**：是从 V_t 指向 V_s 的边

❖ 中文总结：

在流网络中，**割** 就是把点集分成两部分，源点 s 在 V_s ，汇点 t 在 V_t ，然后关注从一边流向另一边的边。

- “**前向边**：割的方向是从 V_s 到 V_t ”
- “**后向边**：割的方向是从 V_t 到 V_s ”

割的作用是：用于分析最大流最小割定理，计算网络最大流时，找到限制流量的瓶颈位置。

前向边和后向边是不是只有穿过割的边才算，其他没有接触到割的都不算

很好！你这个问题问得特别对！我来帮你彻底理清楚：

✓ 是的 —— 只有穿过割的边才算！

👉 定义复习一下：

当我们对一个割 X 来计算：

- **前向边 (forward edge) :**
从 $V_s \rightarrow V_t$ 的边，而且是穿过割的边
- **后向边 (backward edge) :**
从 $V_t \rightarrow V_s$ 的边，而且是穿过割的边

 没有穿过割、不跨越两边的边，是不参与计算的。

比如：

- $s \rightarrow u$
- $u \rightarrow w$

这些在割 X 中，起点和终点都在 V_s ，完全没穿过割，所以无论流量还是容量，都不计入这个割的计算。

2 路径上的残量容量（路径 π 的增广能力）：

- 假设 π 是从源点 s 到汇点 t 的一条路径。
- 那么路径的**总残量容量**是：

$$\Delta_f(\pi) = \min_{(u,v) \in \pi} \Delta_f(u, v)$$

- 也就是说，整条路径能增广的最大流量由路径上“最窄的一条边”决定。

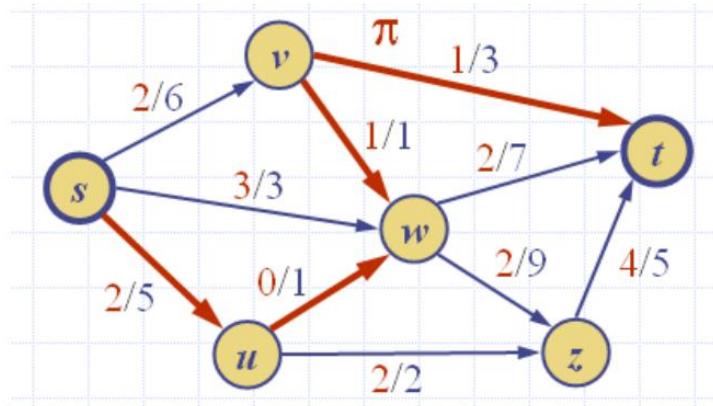
3 增广路径的判断条件：

- 如果 $\Delta_f(\pi) > 0$ ，也就是说整条路径上还有流量可以继续推送，
- 那么这条路径 π 就是一个**增广路径**。

✓ 总结成一句话：

增广路径是从源点 s 到汇点 t 的一条路径，在这条路径上，每条边都还有剩余容量可用。整条路径的增广能力是路径上最小残量容量。

Augmenting Path - Example



We get the following residual capacities and flow

- ▷ $\Delta_f(s, u) = 3, \Delta_f(u, w) = 1, \Delta_f(w, v) = 1, \Delta_f(v, t) = 2.$
- ▷ $\Delta_f(\pi) = 1$
- ▷ $|f| = 7$

当然可以，这张图是在讲 **最大流最小割定理 (Max-Flow Min-Cut Theorem)**，它是网络流理论中的一个核心结果。

🧠 图片原文总结：

❓ 如果一个网络中已经没有可增广路径怎么办？

◆ 引理4 (Lemma 4) :

如果一个网络 N 相对于某个流 f 没有增广路径，

那么 f 就是一个 **最大流**。

同时，**存在一个割 X** ，使得这个最大流的流量就是这个割的容量：

$$|f| = c(X)$$

◆ 定理2 (Theorem 2) :

最大流的值 = 最小割的容量。

▣ 合起来就是——最大流最小割定理：

在一个网络中，最大可能的流量等于最小割的容量。

也叫做 **Ford-Fulkerson 定理**，是基于我们使用“增广路径”不断推流的算法之上的。

Ford-Fulkerson 算法通过不断寻找“增广路径” (augmenting path) 来增加网络中的总流量，直到找不到增广路径为止，最终得到最大流。

步骤详解 (图片内容解读) :

✓ 初始化:

- 对于网络中的每条边 e , 初始流量 $f(e) = 0$ 。

✓ 重复操作 (直到没有增广路径为止) :

1. 搜索一条增广路径 π :

- 增广路径是一条从源点 s 到汇点 t 的路径，在**残量网络 (residual graph)** 中存在。

2. 计算瓶颈容量 (bottleneck capacity) $\Delta_f(\pi)$:

$$\Delta_f(\pi) = \min_{e \in \pi} \text{residual capacity}(e)$$

- 即路径 π 上所有边中剩余容量的最小值，决定了这次能“加多少流”。

3. 沿着路径 π 增加流量:

- 对于路径上的每条边:

- 正向边: 增加流量 $\Delta_f(\pi)$
- 反向边: 减少流量 $\Delta_f(\pi)$



▣ 循环直到找不到增广路径

当残量网络中不再存在从源点 s 到汇点 t 的路径时，算法终止。此时的流量就是最大流。

✓ 中文总结翻译优化版（比图右边更清晰一点）：

Ford-Fulkerson 算法流程：

1. 初始时，每条边的流量 $f(e) = 0$ 。
 2. 重复以下步骤，直到不存在增广路径：
 - 在残量网络中寻找一条从源点 s 到汇点 t 的增广路径 π ；
 - 计算路径上的瓶颈容量 $\Delta_f(\pi)$ ，即路径上所有边的剩余容量的最小值；
 - 沿着该路径增加流量：
 - 若是正向边，增加 $\Delta_f(\pi)$ ；
 - 若是反向边，减少 $\Delta_f(\pi)$ 。
-

！注意点：

- 如果边的容量是整数，Ford-Fulkerson 会在有限次内终止；
↓
- 如果允许浮点数残量（尤其是循环小数），可能出现**无限循环**，需要用 BFS 寻路的 Edmonds-Karp

⚠ 注意事项

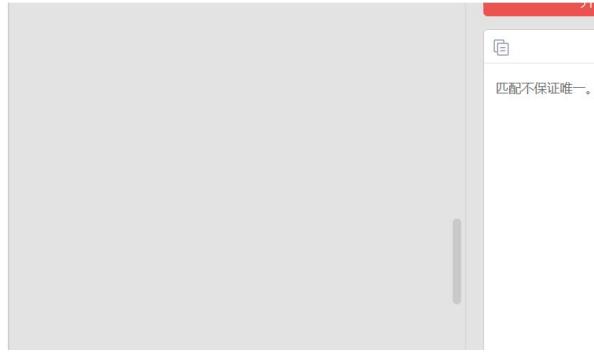
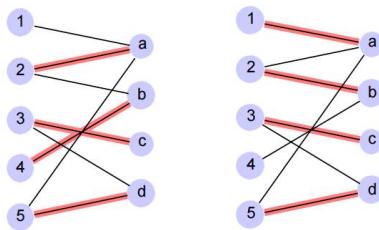
- Ford-Fulkerson 的效率取决于**增广路径的寻找方式**。
 - 如果使用 DFS 且有小数容量，可能**无限循环**。
 - 所以通常我们用它的变体：**Edmonds-Karp 算法**（使用 BFS），时间复杂度为 $O(VE^2)$
-

✓ 总结

特点	说明
问题类型	最大流
算法核心	寻找增广路径，不断增流
使用场景	网络流、路径选择、分配问题等
优化版本	Edmonds-Karp 算法（用 BFS）

Maximum Bipartite Matching

Matching not guaranteed to be unique.



你现在看到的是用**网络流 (network flow) **方法来解决最大二分图匹配问题的“建模转换”过程，也叫图的“归约” (reduction)。

下面我来一步步讲清楚图和文字里讲了什么：

什么是“网络流解法”？

最大二分图匹配问题，其实可以转换成一个**最大流问题 (Max Flow Problem) **来解。

这个思路就是：

我们把匹配问题建成一个“流网络”，然后用最大流算法（比如 Ford-Fulkerson）来求出最大匹配。

图里的结构解释（从左往右看）：

1. 源点 X (source node) :

- 图最左边的 **X** 是源点，表示“匹配流量”的起点。
- 它连到所有左侧顶点（也就是人、任务候选者）——集合 $U = \{1, 2, 3, 4, 5\}$ 。

2. 左侧顶点 U:

- 每个点代表一个人、工作者、实体等等。
- 它们之间原本的匹配边，现在变成了**有向边**，指向右侧集合 V 。

3. 右侧顶点 V:

- 每个点代表可以匹配的对象，比如任务、资源等等。
- 它们所有点都有一条出边连向最右的 **Y**。

4. 汇点 Y (sink node) :

- 所有 V 中的点都连向它，表示“匹配流量”的终点。
-

图的含义总结:

我们把问题转换成了一个标准的最大流模型：

- 流从 x 流出，必须经过：
 $x \rightarrow \text{左侧顶点 } (U) \rightarrow \text{右侧顶点 } (V) \rightarrow Y$
 - 所有边容量为 1 (代表一对匹配)。
-

最终目的:

找到 $x \rightarrow Y$ 的最大流，也就是：

- 能流出多少单位的流，
- 就相当于我们可以构造多少对“互不冲突”的匹配对。

换句话说：

最大流 = 最大匹配的大小

● Maximum Bipartite Matching 解法步骤

① 构建图

Construct the directed graph with a source and sink node.

构造一个有向图，添加：

- 一个 源点 (Source) , 连向所有左侧集合 U 的点；
 - 一个 汇点 (Sink) , 由所有右侧集合 V 的点连向它；
 - 所有匹配候选边 (即 U 到 V) 作为中间有向边，容量为 1。
-

② 寻找增广路径

Using a search method to find a path from the source to the sink.

使用搜索算法 (比如 BFS 或 DFS) 找一条从源点到汇点的路径，这条路径上边的剩余容量都要大于 0。

这种路径称为：增广路径 (Augmenting Path)

③ 更新路径

Once a path is found, reverse each edge on this path.

一旦找到一条增广路径，就将路径上的每条边方向反转 (或说更新残余网络)。

这个步骤对应最大流算法中的“增广流”和“更新残量图”↓

④ 重复直到无增广路径

Repeat step 2 and 3 until no more paths from the source to sink exist.

不断重复步骤 2 和 3，直到无法再从源点到达汇点 (即找不到增广路径)，此时算法终止。

⑤ 得到最大匹配

The final matching solution is the set of edges between U and V that are reversed (i.e., run from V to U).

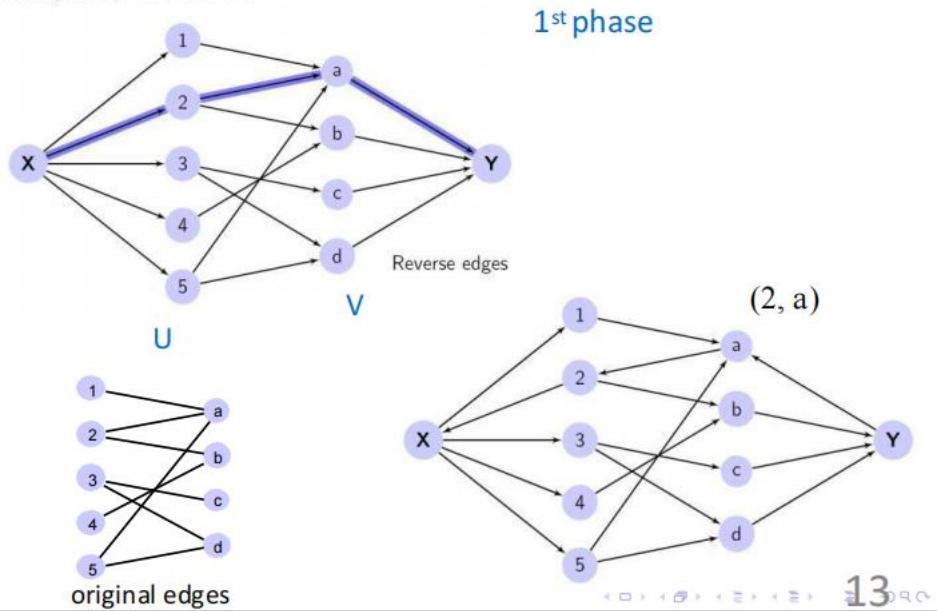
最终解就是所有从 $V \rightarrow U$ 的反向边 (也就是说，原来是 $U \rightarrow V$ 的边被“使用”了，即匹配成功了)。所以我们把这些边反转回来，就得到了一个最大匹配集合。

🧠 总结一句话：

通过找增广路径和更新残余图来“流量累加”，直到不能再流为止，最终被使用过的边就对应了最大匹配。

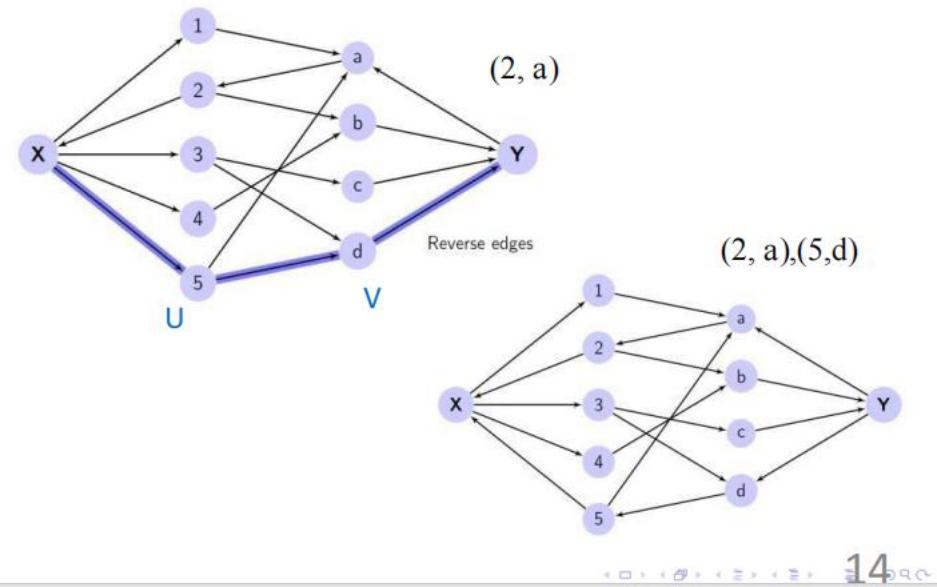
Maximum Bipartite Matching

Found path: $\mathbf{X} \rightarrow 2 \rightarrow a \rightarrow \mathbf{Y}$



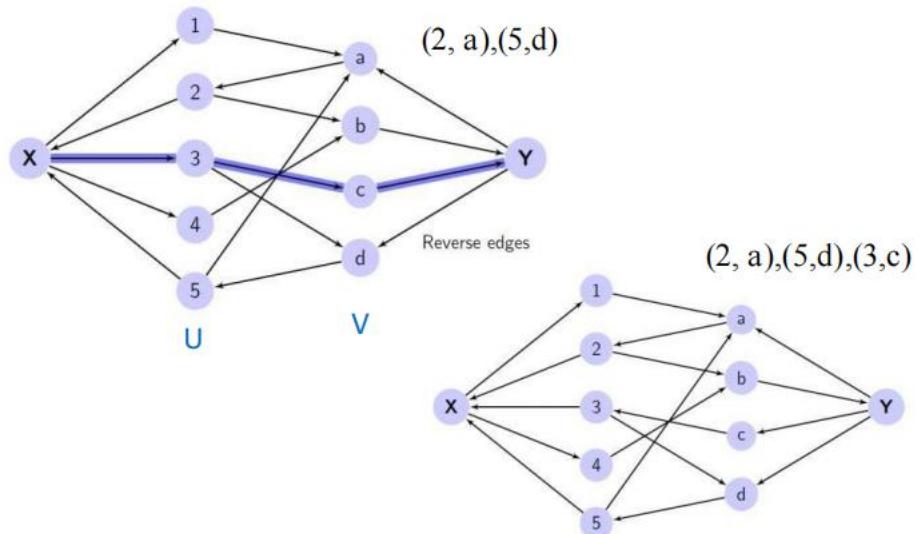
Maximum Bipartite Matching

Found path: $\mathbf{X} \rightarrow 5 \rightarrow d \rightarrow \mathbf{Y}$



Maximum Bipartite Matching

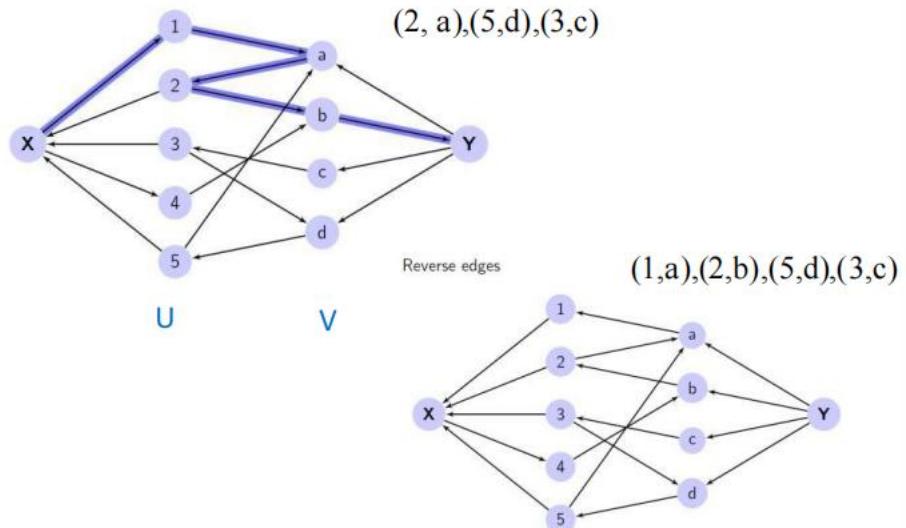
Found path: $\mathbf{X} \rightarrow 3 \rightarrow c \rightarrow \mathbf{Y}$



15

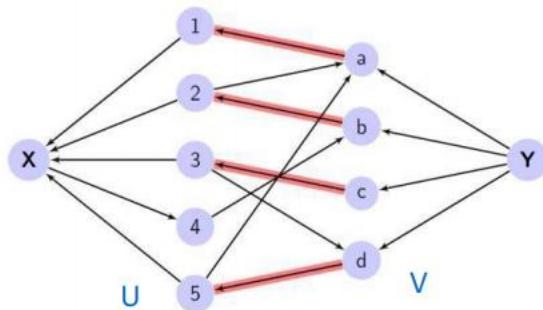
Maximum Bipartite Matching

Found path: $\mathbf{X} \rightarrow 1 \rightarrow a \rightarrow 2 \rightarrow b \rightarrow \mathbf{Y}$



Maximum Bipartite Matching

No more paths found. Matching is reversed edges.



17

第19页详解：最大公约数与欧几里得算法的引入

概念：Greatest Common Divisor (GCD)

定义：两个正整数的最大公约数是同时整除这两个数的最大整数。

举个例子：

- $\gcd(12, 18) = 6$, 因为：
 - 公约数有：1, 2, 3, 6
 - 最大的是 6。

欧几里得算法基础

Fact: $\gcd(a, 0) = a$

很直观的事实：任何数和 0 的最大公约数就是这个数本身，因为任何数都能整除 0。

引理 (Lemma) :

若 $a = bq + r$, 且 $b \neq 0$, 则：

$$\gcd(a, b) = \gcd(b, r)$$

这就是欧几里得算法的核心原则：我们可以通过“余数迭代”的方式不断减少问题规模，最终求得最大公约数。

✓ 引理再次重述 (与上一页相同) :

| 若 $a = bq + r$, 则:

$$\gcd(a, b) = \gcd(b, r)$$

★ 证明过程逐步拆解:

1. 设: $d = \gcd(a, b)$, $e = \gcd(b, r)$, 目标是证明 $d = e$ 。
2. 因为 $d | a$ 且 $d | b$, 那么 $d | (a - bq) = r \Rightarrow d | r$.

| 说明: 如果一个数能整除 a 和 b , 它也能整除 $r \Rightarrow$ 所以 d 是 b 和 r 的公因数 \Rightarrow 所以 $d \leq e$.

3. 反之, 因为 $e | b$ 且 $e | r$, 那么 $e | (bq + r) = a \Rightarrow e | a$

| 说明: 所以 e 也是 a 和 b 的公因数 \Rightarrow 所以 $e \leq d$

4. 因此: $d \leq e$ 且 $e \leq d \Rightarrow$ 得证 $d = e$

● 新定义: Relatively Prime (互素)

| 如果 $\gcd(a, b) = 1$, 则称 a 和 b 是互素的 (relatively prime)

💡 即它们没有公共因子, 除了 1.

👉 例如:

- $\gcd(8, 15) = 1 \Rightarrow 8$ 和 15 是互素的
- $\gcd(12, 18) = 6 \Rightarrow$ 不是互素



1.4 Divisibility

Extended Euclidean Algorithm

Given two integers a and b , we often need to find other two integers, s and t , such that

$$s \times a + t \times b = \gcd(a, b)$$

The extended Euclidean algorithm can calculate the $\gcd(a, b)$ and at the same time calculate the value of s and t .

Key observation remainder r in each GCD step can be represented by $a - bq$ and $\gcd(a, b)$ is the last effective remainder.

23

1.4 Divisibility

Extended Euclidean Algorithm

Given two integers a and b , we often need to find other two integers, s and t , such that

$$s \times a + t \times b = \gcd(a, b)$$

The extended Euclidean algorithm can calculate the $\gcd(a, b)$ and at the same time calculate the value of s and t .

Find the greatest common divisor of 25 and 60.

Solution

$$a = 60, b = 25, \quad q = 2, \quad r = 10$$

$$a = 25, b = 10, \quad q = 2, \quad r = 5 \quad 5 = 25 - 10 \cdot 2 = 25 - (60 - 25 \cdot 2) \cdot 2 = 5 \cdot 25 - 2 \cdot 60$$

$$a = 10, b = 5, \quad q = 2, \quad r = 0 \quad s = -2$$

$$a = 5, b = 0 \quad t = 5$$

We have $\gcd(25, 60) = 5$.

24

小提示:

r in each GCD step can be represented by $a - bq$, 而最后一个非零余数就是 $\gcd(a, b)$ 。

这为后面的“回代”过程打下基础：你可以通过前面的除法步骤反向推导出 s 和 t 的值。

第24页详解

这是上面理论部分的**具体计算示例**，用于说明如何通过扩展欧几里得算法求出 s, t 。

目标：求 $\gcd(25, 60)$ ，并求出 s, t ，使得：

$$s \times 25 + t \times 60 = \gcd(25, 60)$$

步骤 1：普通欧几里得算法（除法）

我们从 $a = 60, b = 25$ 开始：

1. $60 \div 25 = 2$, 余数 $r = 10$
2. $25 \div 10 = 2$, 余数 $r = 5$
3. $10 \div 5 = 2$, 余数 $r = 0 \Rightarrow$ 停止!

所以：

$$\gcd(60, 25) = 5$$

▣ 步骤 2：反向回代 (求 s, t)

从刚才的除法中：

- 第2步： $5 = 25 - 2 \times 10$
- 用第1步中的 $10 = 60 - 2 \times 25$ 代入：

$$5 = 25 - 2 \times (60 - 2 \times 25) = 25 - 2 \times 60 + 4 \times 25 = 5 \times 25 - 2 \times 60$$

即：

$$5 = (-2) \times 60 + 5 \times 25$$

✓ 最终结果：

$$s = -2, \quad t = 5 \Rightarrow -2 \times 60 + 5 \times 25 = 5$$

● 总结：

- 普通欧几里得算法只解决了 GCD；
- 扩展算法进一步得到了 s 和 t , 满足 Bézout 等式；
- 在密码学中（如模逆运算、RSA 公钥私钥生成）极为关键。

$$s \times a + t \times b = \gcd(a, b)$$

再详细讲讲这个式子

好的，我们再深入详细讲解这个式子：

$$s \times a + t \times b = \gcd(a, b)$$

这其实是一个**数学上的重要恒等式**，叫做**贝祖等式 (Bézout's identity)**。我们来逐个拆解它的组成部分和意义：

一、各个符号含义

- a, b : 两个整数 (可以是正数或负数)
- $\gcd(a, b)$: 表示 a 和 b 的**最大公约数** (Greatest Common Divisor)
- s, t : 是两个整数，**满足上面这个等式**。你可以把它们看成是“系数”，是通过**扩展欧几里得算法**求出来的。

二、这个等式的意义

这个等式表明：

对于任意两个整数 a 和 b , 都存在一对整数 s 和 t , 使得 $sa + tb = \gcd(a, b)$ 。

也就是说，最大公约数可以用 a 和 b 的整数线性组合的形式来表示！

三、为什么这个公式重要？

1. 计算模逆元素 (modular inverse) :

- 若 a 和 m 互质 ($\gcd(a, m) = 1$)，那么存在 s 使得 $sa \equiv 1 \pmod{m}$ ，
- 这个 s 就是 a 的模逆，正是这个等式的一个特殊情况。

2. 求解线性不定方程 $ax + by = c$

- 只有当 c 是 $\gcd(a, b)$ 的倍数时，这个方程有解，解的形式就来自这个公式。

3. RSA密码系统、中国剩余定理等很多算法中都用到这个公式。

四、举个完整的例子

我们来看一个具体例子，理解 s, t 是怎么来的：

设: $a = 30, b = 12$

第一步：先求 $\gcd(30, 12)$

用欧几里得算法：

```
matlab
30 ÷ 12 = 2 余 6
12 ÷ 6 = 2 余 0
所以 gcd(30, 12) = 6
```

第二步：扩展欧几里得法，回代求出 s, t

我们从除法步骤回代：

```
ini
6 = 30 - 2×12
```

所以：

$$6 = 1 \cdot 30 - 2 \cdot 12$$

也就是：

$$s = 1, \quad t = -2$$

那么这个等式就变成了：

$$1 \cdot 30 + (-2) \cdot 12 = 6$$

总结一句话：

这个等式 $sa + tb = \gcd(a, b)$ 是用来表示 **最大公约数可以由 a 和 b 的线性组合表示**，其中的 s 和 t 可以用扩展欧几里得算法算出来，有很多实用的数学和算法应用。

第33页: 2.1 Modulo Operator (模运算符)

✓ 定义说明:

- 模运算 (**mod**) : 也可以写作 `%` , 在很多编程语言中这样表示。
 - 它的作用是:
给定两个整数 a 和 n ($n > 0$) , 找到 a 除以 n 后的“余数”。
-

◆ 数学背景: 除法算法

公式:

$$a = q \cdot n + r$$

- a : 被除数 (任意整数)
- n : 除数 (正整数)
- q : 商 (整数)
- r : 余数 (满足 $0 \leq r < n$)

□ 这个式子是“整除”的基本公式, 用它就可以理解模运算。

◆ 模运算公式：

$$a \mod n = r$$

也就是说，模运算的结果就是除法中的余数 r 。

★ 举个例子：

$$17 \div 5 = 3 \text{ 余 } 2 \Rightarrow 17 = 3 \cdot 5 + 2 \Rightarrow 17 \mod 5 = 2$$

■ 第34页：2.2 Set of Residues (剩余类集合)

✓ 定义：

在模 n 的世界里，所有可能出现的余数（ \mod 之后的结果）组成一个集合，记作：

$$\mathbb{Z}_n = \{0, 1, 2, \dots, n - 1\}$$

◆ 举几个例子：

- $\mathbb{Z}_2 = \{0, 1\}$
- $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$
- $\mathbb{Z}_{11} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

这些集合是模运算结果的全集。

❖ 重要结论:

- 无论整数 a 是正数、负数，还是 0， $a \bmod n$ 的结果永远属于 \mathbb{Z}_n 。
 - 所以， \mathbb{Z}_n 就是“模 n ”下所有可能的输出值。
-

💡 举例说明:

- $-1 \bmod 6 = 5$
- $7 \bmod 6 = 1$

你会发现它们的结果都在 $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$ 中。

✓ 总结一句话:

模运算是将任意整数 a 映射到一个“循环的余数集合”里（就是 \mathbb{Z}_n ），而模集合就是所有可能的余数值。这种数学结构是模算术的基础，在密码学、公钥系统、哈希函数中都非常常见。

这两页继续讲述模运算中的一个核心概念：同余（Congruence）和剩余类（Residue Classes），是模算术的理论基础。下面我来详细解释图片中内容：

📘 第35页: 2.3 Congruence 同余

✓ 定义:

两个整数 a 和 b 如果在模 m 意义下余数相同，就说：

$$a \equiv b \pmod{m}$$

也就是：

- $a \bmod m = b \bmod m$
 - $m \mid (a - b)$
-

◆ 示例说明:

- $2 \equiv 12 \pmod{10}$: 因为 $2 \bmod 10 = 2$, 而 $12 \bmod 10 = 2$
- $13 \equiv 23 \pmod{10}$: 因为它们都余 3
- $3 \equiv 8 \pmod{5}$
- $8 \equiv 13 \pmod{5}$

□ 图示解析:

- 显示了整数集合 $\mathbb{Z} = \{\dots, -8, \dots, 2, \dots, 12, 22, \dots\}$
- 每个数模 10 后会“映射”到集合 $\mathbb{Z}_{10} = \{0, 1, \dots, 9\}$
- 所以像 $-8, 2, 12, 22$ 全部模 10 都等于 2，它们是同一个**同余类成员**

结论：

| 只要模数一样，结果一样，就是同余的！

■ 第36页：Residue Classes (剩余类)

✓ 定义：

一个**剩余类** $[a]$ 是所有与 a 在模 n 意义下**同余**的整数的集合：

$$[a]_n = \{x \in \mathbb{Z} \mid x \equiv a \pmod{n}\}$$

即：

$$[a]_n = \{\dots, a - 3n, a - 2n, a - n, a, a + n, a + 2n, a + 3n, \dots\}$$

◆ 示例：模 $n = 5$

构造出 5 个剩余类：

- $[0] = \{\dots, -15, -10, -5, 0, 5, 10, 15, \dots\}$
- $[1] = \{\dots, -14, -9, -4, 1, 6, 11, 16, \dots\}$
- $[2] = \{\dots, -13, -8, -3, 2, 7, 12, 17, \dots\}$
- $[3] = \{\dots, -12, -7, -2, 3, 8, 13, 18, \dots\}$
- $[4] = \{\dots, -11, -6, -1, 4, 9, 14, 19, \dots\}$

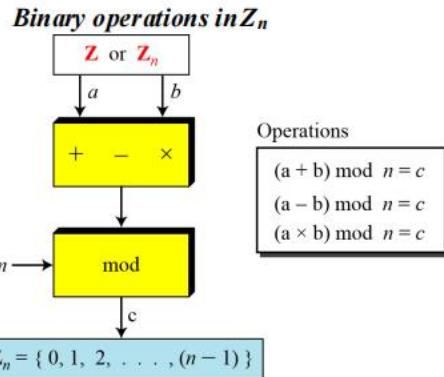
注意：每个类中的数字模 5 都是一样的。

● 总结一句话：

- 同余类 [a] 把所有和 a 模 n 同余的整数归为一类。
- 总共会有 n 个不同的类，分别代表余数是 0 到 $n - 1$ 的情况。
- 在模 n 的世界里，所有整数都会落入其中一个剩余类里。

2.4 Operation in Z_n

The three binary operations that we discussed for the set Z can also be defined for the set Z_n . The result may need to be mapped to Z_n using the mod operator



37

2.4 Operation in Z_n

Properties of mod operator

First Property: $(a + b) \text{ mod } n = [(a \text{ mod } n) + (b \text{ mod } n)] \text{ mod } n$

Second Property: $(a - b) \text{ mod } n = [(a \text{ mod } n) - (b \text{ mod } n)] \text{ mod } n$

Third Property: $(a \times b) \text{ mod } n = [(a \text{ mod } n) \times (b \text{ mod } n)] \text{ mod } n$

Example

The following shows the application of the above properties:

$$(37 + 99) \text{ mod } 6 = 136 \text{ mod } 6 = 4$$

$$[(37 \text{ mod } 6) + (99 \text{ mod } 6)] \text{ mod } 6 = (1+3) \text{ mod } 6 = 4$$

◆ 三种基本运算在模 n 中表示为：

- $(a + b) \bmod n = c$
- $(a - b) \bmod n = c$
- $(a \times b) \bmod n = c$

这就定义了 模加法、模减法 和 模乘法。

■ 第38页：模运算的性质 (Properties of Mod Operator)

这些是模运算的三个**分配性质**，也就是说你可以先取模再运算，结果不会变：

● 三个重要性质：

1. 加法性质：

$$(a + b) \bmod n = [(a \bmod n) + (b \bmod n)] \bmod n$$

2. 减法性质：

$$(a - b) \bmod n = [(a \bmod n) - (b \bmod n)] \bmod n$$

3. 乘法性质：

$$(a \times b) \bmod n = [(a \bmod n) \times (b \bmod n)] \bmod n$$

这些性质对于编程、算法设计、密码学尤其重要，比如 ↓ A 加密算法就大量使用这些规律。

这个式子 $(10 \bmod 7)^7 \bmod 7 = 3^7 \bmod 7$ 主要基于 **模运算的性质**。其核心原理就是：模运算可以分配到指数运算中。具体来说，我们可以利用以下几个重要的模运算性质：

1. 模运算的基本性质：

- 对于任意的整数 a 和 b ，以及正整数 x ，有：

$$(a \bmod x) = a - \lfloor \frac{a}{x} \rfloor \times x$$

- 最常见的规则是，若我们要计算 $(a * b) \bmod x$ ，可以先分别计算 $a \bmod x$ 和 $b \bmod x$ ，再乘起来取模，即：

$$(a * b) \bmod x = [(a \bmod x) * (b \bmod x)] \bmod x$$

2. 关于指数运算的模性质：

对于指数运算的模运算，我们可以利用以下公式：

$$(a^b) \bmod x = ((a \bmod x)^b) \bmod x$$

这意味着，我们可以先对底数 a 进行模运算，得到一个较小的数，然后再进行指数运算，最后对结果取模。

3. 应用到具体问题：

在这个问题中，我们想计算 $10^7 \bmod 7$ ，但是直接计算 10^7 会得到一个非常大的数。为了简化计算，我们使用上述的指数运算模性质。

步骤：

1. 计算 $10 \bmod 7$ ：

$$10 \bmod 7 = 3$$

所以，我们将原式 $10^7 \bmod 7$ 转换为 $3^7 \bmod 7$ ，这就是利用了模运算的性质。

2. 接下来计算 $3^7 \bmod 7$ ：根据模运算的指数性质， $(10 \bmod 7)^7 \bmod 7$ 就是 $3^7 \bmod 7$ ，这就是为什么它们是相等的。

4. 总结：

这种转化的原理在于**模运算可以提前应用到底数**。通过先计算 $10 \bmod 7$ ，我们将原式变成了更简单的 $3^7 \bmod 7$ ，使得后续的运算更加简单。这是模运算的一个重要性质，可以帮助我们简化计算，尤其是在处理大数时。

希望这个解释能帮助你理解其中的原理！

好的，关于 $3^7 \bmod 7$ ，我们可以通过逐步计算的方式来得到最终的结果。

步骤：

我们需要计算的是 $3^7 \bmod 7$ ，也就是说我们要找出 3 的 7 次方除以 7 的余数。

1. 先计算 $3^2 \bmod 7$ ：

$$3^2 = 9$$

然后，计算 9 对 7 取模：

$$9 \bmod 7 = 2$$

所以， $3^2 \bmod 7 = 2$ 。

2. 计算 $3^4 \bmod 7$ (即 $(3^2)^2 \bmod 7$)：

$$3^4 = (3^2)^2 = 9^2 = 81$$

然后，计算 81 对 7 取模：

$$81 \bmod 7 = 4$$

所以， $3^4 \bmod 7 = 4$ 。

3. 计算 $3^7 \bmod 7$ (即 $3^4 * 3^2 * 3 \bmod 7$)：

我们可以分解 3^7 为：

$$3^7 = 3^4 \times 3^2 \times 3$$

然后分别计算各部分对 7 取模：

$$3^4 \bmod 7 = 4, \quad 3^2 \bmod 7 = 2, \quad 3 \bmod 7 = 3$$

所以：

$$3^7 \bmod 7 = (3^4 \times 3^2 \times 3) \bmod 7 = (4 \times 2 \times 3) \bmod 7$$

首先计算：

$$4 \times 2 = 8$$

然后：

$$8 \times 3 = 24$$

最后：

$$24 \bmod 7 = 3$$

所以， $3^7 \bmod 7 = 3$ 。

结果：

因此， $3^7 \bmod 7 = 3$ 。通过这个计算步骤，我们逐步得到了最终的结果。

希望这个解释清楚了如何一步步计算！

✓ 第三个式子：

$$10 \mod 7 = 3 \Rightarrow 10^n \mod 7 = 3^n \mod 7$$

这个就稍微复杂一点，但逻辑一样：

1. $10 \div 7 = 1$ 余数是 3

2. 那么：

$$10^n \mod 7 = (10 \mod 7)^n \mod 7 = 3^n \mod 7$$

| 也就是说，10 的 n 次方模 7，其实就是 3 的 n 次方模 7

？举个例子：

假设 $n = 4$

- $10^4 = 10000$, 但我们不直接计算它
- $10 \mod 7 = 3$, 所以:

$$10^4 \mod 7 = 3^4 \mod 7$$

计算一下 $3^4 = 81$,

$$81 \div 7 = 11 \text{ 余 } 4$$

所以最终结果是：

$$10^4 \mod 7 = 4$$

这两张图讲的是模运算 (modular arithmetic) 中的逆元 (Inverse)，分别是加法逆元和乘法逆元。下面是每一部分的中文解释：

第一张图：加法逆元 (Additive Inverse)

内容：

- 在模 n 的整数集合 \mathbb{Z}_n 中，若两个数 a 和 b 满足：

$$a + b \equiv 0 \pmod{n}$$

则称 a 和 b 是彼此的**加法逆元**。

注释：

在模运算中，每个整数都有一个加法逆元。一个数和它的加法逆元相加，其结果模 n 等于 0。

第二张图：乘法逆元 (Multiplicative Inverse)

内容：

- 在模 n 的整数集合 \mathbb{Z}_n 中，若两个数 a 和 b 满足：

$$a \times b \equiv 1 \pmod{n}$$

则称 a 和 b 是彼此的**乘法逆元**。

注释：

在模运算中，一个整数不一定有乘法逆元。但如果它有乘法逆元，那么这个数和它的乘法逆元相乘，其结果模 n 等于 1。

✓ 解的判定方法:

设 $d = \gcd(a, n)$, 那么:

1. 如果 $d \nmid b$ (即 d 不能整除 b) :

- ✗ 无解。

2. 如果 $d \mid b$ (即 d 能整除 b) :

- ✓ 有 d 个不完全相同的模 n 解。
-

✓ 举个例子理解 (非幻灯片内容, 我补充的) :

考虑:

$$6x \equiv 8 \pmod{14}$$

我们有:

- $a = 6, b = 8, n = 14$
- $\gcd(6, 14) = 2 = d$
- 因为 $2 \mid 8$, 所以有 2 个解

解法步骤是:

1. 把方程除以 $d = 2$: 得到 $3x \equiv 4 \pmod{7}$
2. 解这个方程 (可以用逆元法或扩展欧几里得)
3. 得到一个特解, 再通过公式得到其他解

为什么要检查 $\gcd(7, 9)$?

模方程的一般形式是：

$$a \cdot x \equiv b \pmod{n}$$

其中 a 是系数， x 是未知数， b 是常数， n 是模数。

对于这个方程， $a = 7$, $b = 6$, $n = 9$, 我们需要判断这个方程是否有解。

1. 解的存在性：

如果 $\gcd(a, n) = 1$, 那么方程 **一定有唯一解**。这就是为什么我们需要检查 $\gcd(7, 9)$ 是否为 1。如果 $\gcd(a, n) = 1$, 那么我们可以找到 a 在模 n 下的逆元, 进而解出 x 。

2. 当 $\gcd(a, n) \neq 1$ 时：

如果 $\gcd(a, n)$ 不等于 1, 方程可能没有解, 或者可能有多个解。

为什么会这样呢？

假设 $\gcd(a, n) = d$, 其中 $d > 1$, 这意味着 a 和 n 之间有一个公约数 d 。此时, 方程 $a * x \equiv b \pmod{n}$ 只有解 **当且仅当** d 也能整除 b 。具体来说：

- 如果 d 能整除 b (即 $b \% d == 0$) , 那么方程有解, 但可能不是唯一的。
- 如果 d 不能整除 b , 那么方程就没有解。

举个例子：

假设方程是 $6 * x \equiv 3 \pmod{9}$, 这里 $a = 6$, $b = 3$, $n = 9$ 。

- 首先, 计算 $\gcd(6, 9)$ 。 $\gcd(6, 9) = 3$, 而 3 能整除 3 (即 b) , 所以方程有解。
- 然而, 解不会是唯一的。具体来说, 如果 $\gcd(6, 9) = 3$, 那么方程有 3 个不同的解, 因为模数 9 除以 3 得到的商是 3 , 因此解的数量是 3 。

总结：

- 当 $\gcd(a, n) = 1$ 时, 方程有且仅有一个解。
- 当 $\gcd(a, n) \neq 1$ 时, 方程可能有解, 但解的个数依赖于 $\gcd(a, n)$ 和 b 的关系。解的个数是 $\gcd(a, n)$ 的个数。



Example

Solve the equation $10x \equiv 2 \pmod{15}$.

Solution

First we find the $\gcd(10, 15) = 5$. Since 5 does not divide 2, we have no solution.

12

Example

Solve the equation $7x \equiv 6 \pmod{9}$.

Division in \mathbb{Z}_n^* is defined by the equation $a/b \equiv ab^{-1} \pmod{n}$.

Solution

Because $\gcd(7, 9) = 1$, and $1|6$, so it has only one solution.

$$x \equiv (6 \times 7^{-1}) \pmod{9}$$

$$7^{-1} \pmod{9} \equiv 4$$

$$x \equiv (6 \times 4) \pmod{9} \equiv 24 \pmod{9}$$

$$x = 6$$

1. 负数模运算的转换:

当我们在模运算中遇到负数时，我们可以通过将负数加上模数 m 来将其转换为等价的正数。具体来说，给定一个负数 $-a$ ，我们可以计算：

$$-a \mod m = (-a + m) \mod m$$

这是因为在模 m 下，负数与其对应的正数是等价的。例如：

$$-7 \mod 18 = -7 + 18 = 11$$

所以， $-7 \mod 18$ 等价于 $11 \mod 18$ 。

2. 验证:

假设我们要验证 $5 * (-7) \mod 18$ 和 $5 * 11 \mod 18$ 是否等价。

- 第一步：计算 $5 * (-7) \mod 18$ ：

$$5 \times (-7) = -35$$

然后对 -35 进行模 18 运算：

$$-35 \mod 18 = -35 + 18 \times 2 = -35 + 36 = 1$$

所以， $5 * (-7) \mod 18 = 1$ 。

- 第二步：计算 $5 * 11 \mod 18$ ：

$$5 \times 11 = 55$$

然后对 55 进行模 18 运算：

$$55 \mod 18 = 55 - 18 \times 3 = 55 - 54 = 1$$

所以， $5 * 11 \mod 18 = 1$ 。

3. 结论:

因为 $5 * (-7) \mod 18 = 1$ 且 $5 * 11 \mod 18 = 1$ ，我们可以得出结论， $5 * (-7) \mod 18$ 与 $5 * 11 \mod 18$ 是等价的。

当 $n \bmod m$, $n < m$ 时, 结果就为 n 吗

是的, 当 $n < m$ 时, $n \bmod m$ 的结果就是 n 。

原因:

模运算 $n \bmod m$ 是将 n 除以 m , 然后取余数。对于任何整数 n 和模数 m , 模运算的定义是:

$$n \bmod m = n - \left(\left\lfloor \frac{n}{m} \right\rfloor \times m \right)$$

其中 $\left\lfloor \frac{n}{m} \right\rfloor$ 是 n 除以 m 的整数部分 (商)。

当 $n < m$ 时:

- 由于 n 小于 m , 商的整数部分为 0 (即 $\left\lfloor \frac{n}{m} \right\rfloor = 0$)。
- 因此, 余数就是 n 本身。

举个例子:

- 如果 $n = 3$, $m = 7$, 那么 $3 \bmod 7 = 3$, 因为 $3 < 7$, 它本身就是余数。

总结:

当 $n < m$ 时, $n \bmod m = n$ 。

1.1.2.5 加法逆元 (Additive Inverse)

在 Z_n 中, 如果两个数 a 和 b 满足 $a + b \equiv 0 \pmod{n}$, 那么 a 和 b 互为加法逆元。

在模运算中, 每个整数都有一个加法逆元。

例子如下: 假设 $n = 5$, 所以 $Z_5 = \{0, 1, 2, 3, 4\}$

对于 $a = 1$, 它的加法逆元是 4 , 因为 $1 + 4 = 5$, 而 $5 \bmod 5 = 0$ 。

对于 $a = 2$, 它的加法逆元是 3 , 因为 $2 + 3 = 5$, 而 $5 \bmod 5 = 0$ 。

1.1.2.6 乘法逆元 (Multiplicative Inverse)

在 Z_n 中, 如果两个数 a 和 b 满足 $a \times b \equiv 1 \pmod{n}$, 那么 a 和 b 互为乘法逆元。

在模运算中, 一个整数可能有也可能没有乘法逆元。

同样的例子下:

对于 $a = 1$, 它的乘法逆元是 1 , 因为 $1 \times 1 = 1$, 而 $1 \bmod 5 = 1$ 。

对于 $a = 2$, 它的乘法逆元是 3 , 因为 $2 \times 3 = 6$, 而 $6 \bmod 5 = 1$ 。

对于 $a = 4$, 它的乘法逆元是 4 , 因为 $4 \times 4 = 16$, 而 $16 \bmod 5 = 1$ 。

扩展欧几里得算法可以用来找到 b 在 Z_n 中的乘法逆元, 前提是 n 和 b 互质, 即 $\gcd(n, b) = 1$ 。此外还会找到整数 s 和 t 使得 $s \times n + t \times b = \gcd(n, b)$ 。

由于 $\gcd(n, b) = 1$, 我们有 $s \times n + t \times b = 1$ 。

在模 n 的情况下, 这意味着 $t \times b \equiv 1 \pmod{n}$, 因此 t 就是 b 在 Z_n 中的乘法逆元。

乘法逆元通常表示为 $b^{-1} \pmod{n}$ 。

保存的记忆已满 ①

mod6是什么意思，是前面一个数对6取余的结果吗

是的，**mod 6** 的意思就是“对 6 取模”或“对 6 取余数”。

举例说明：

- $8 \bmod 6 = 2$, 因为 8 除以 6 商为 1, 余数为 2。
- $14 \bmod 6 = 2$, 因为 14 除以 6 商为 2, 余数还是 2。
- $6 \bmod 6 = 0$, 因为 6 除以 6 刚好除尽, 余数是 0。
- $5 \bmod 6 = 5$, 因为 5 小于 6, 除不尽, 余数就是它本身。

用途举例：

在加密中, 比如凯撒密码 (Caesar Cipher) 中, 如果使用的是字母表的循环 (26个字母), 就常常会用 **mod 26**, 这样可以确保偏移后仍然落在字母范围内。

同样, 如果你看到 **mod 6**, 很可能是指在一个长度为 6 的循环结构 (比如 0~5 的范围) 中处理数据。

- 1 2个质数 $p = 3; q = 11$
- 2 质数相乘 $N = p \times q = 3 \times 11 = 33$
- 3 欧拉函数 $T = (p-1) \times (q-1) = 2 \times 10 = 20$
- 4 选公钥E 质数; $1 < \text{公钥} < T$; 不是T的因子 $E = (3, 33)$
- 5 算私钥D $(D \times E) \% T = 1$ $D = (7, 33)$ 秘

你上传的两页幻灯片详细讲解了 **欧拉定理 (Euler's Theorem)**，我们来分部分进行解读：

■ 第31页: Euler's Theorem 概述与示例

★ 定理陈述:

Euler's Theorem (欧拉定理) :

设 n 是一个正整数, x 是一个与 n 互质的整数 (即 $\gcd(x, n) = 1$) , 则有:

$$x^{\varphi(n)} \equiv 1 \pmod{n}$$

其中 $\varphi(n)$ 是欧拉函数 (Euler's totient function) , 表示小于 n 且与 n 互质的整数个数。

✓ 示例 (n = 10) :

因为 $\varphi(10) = 4$, 所以对于所有与 10 互质的数 x , 都有 $x^4 \equiv 1 \pmod{10}$

- $3^4 = 81 \pmod{10} = 1$
- $7^4 = 2401 \pmod{10} = 1$
- $9^4 = 6561 \pmod{10} = 1$

✓ 这些都验证了定理。

RSA 加密方案的基本步骤 (35页)

第一步：选择两个大质数

- 选择两个足够大的质数 p 和 q 。

第二步：计算模数和欧拉函数

- 计算 $n = p \times q$
- 计算欧拉函数 $\phi(n) = (p - 1)(q - 1)$

第三步：选择加密指数 e

- 选择一个正整数 e , 满足 $\gcd(e, \phi(n)) = 1$
- 即 e 与 $\phi(n)$ 互素, 确保 e 存在模 $\phi(n)$ 的乘法逆元。

第四步：计算解密指数 d

- 使用扩展欧几里得算法 (Extended Euclidean Algorithm) 来求出 d , 满足:

$$d \cdot e \equiv 1 \pmod{\phi(n)}$$

即存在整数 k , 使得:

$$de = k \cdot \phi(n) + 1$$

密钥生成:

- 公钥 $K_E = (n, e)$ —— 公开的
- 私钥 $K_D = d$ —— 保密的

RSA 加密解密流程 (36页)

系统初始化 (Setup) :

1. 选择两个质数 p, q
 2. 计算:
 - $n = p \cdot q$
 - $\phi(n) = (p - 1)(q - 1)$
 3. 选择一个与 $\phi(n)$ 互素的整数 e
 4. 计算 d , 使得 $d \cdot e \equiv 1 \pmod{\phi(n)}$
-

密钥设置 (Keys) :

- 公钥: $K_E = (n, e)$
 - 私钥: $K_D = d$
-

加密过程 (Encryption) :

- 明文 M 是小于 n 的一个整数 (即 $M \in \mathbb{Z}_n$)
- 密文计算公式:

$$C = M^e \pmod{n}$$

解密过程 (Decryption) :

- 接收到密文 C
- 使用私钥 d 解密:

$$M = C^d \pmod{n}$$

The RSA method

RSA Cryptosystem

Setup:

- $n = p \times q$, with p and q primes.
- $\phi(n) = (p-1)(q-1)$
- $\gcd(e, \phi(n)) = 1$
- d of $e^{-1} \bmod \phi(n)$

Keys:

- Public key $K_E = (n, e)$.
- Private key $K_D = d$.

Encryption:

- Plaintext M in Z_n (i.e., $0 \leq M < n$)
- Ciphertext $C = M^e \bmod n$.

Decryption:

- Recover $M = C^d \bmod n$.

Example

Setup:

- $p = 7, q = 17$
- $n = 7 \times 17 = 119$
- $\phi(119) = 6 \times 16 = 96$
- $e = 5$
- $d = 77$ ($77 \times 5 = 96 \times 4 + 1$)

Keys:

- Public key $K_E = (119, 5)$.
- Private key $K_D = 77$.

Encryption:

- $M = 19$
- $C = 19^5 \bmod 119 = 66$

Decryption:

- $M = 66^{77} \bmod 119 = 19$.

The RSA method

Complete RSA Example

Setup:

- $p = 5, q = 11$
- $n = 5 \times 11 = 55$
- $\phi(55) = 4 \times 10 = 40$
- $e = 3$
- $d = 27$ ($27 \times 3 = 40 \times 2 + 1$)

Encryption:

- $C = M^3 \bmod 55$

Decryption:

- $M = C^{27} \bmod 55$

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	1	8	27	9	15	51	13	17	14	10	11	23	52	49	20	26	18	2
M	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
C	39	25	21	33	12	19	5	31	48	7	24	50	36	43	22	34	30	16
M	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
C	53	37	29	35	6	3	32	44	45	41	38	42	4	40	46	28	47	54

✓ 示例 1 (第37页)

设置:

- $p = 7, q = 17$
- $n = 7 \times 17 = 119$
- $\phi(n) = (7 - 1)(17 - 1) = 6 \times 16 = 96$
- 选择 $e = 5$, 因为 $\gcd(5, 96) = 1$
- 解密指数 $d = 77$, 因为:

$$77 \times 5 = 385 = 96 \times 4 + 1 \Rightarrow 77 \times 5 \equiv 1 \pmod{96}$$

密钥:

- 公钥: $(119, 5)$
- 私钥: 77

加密:

明文 $M = 19$, 加密得:

$$C = 19^5 \pmod{119} = 66$$

解密:

密文 $C = 66$, 解密得:

$$M = 66^{77} \pmod{119} = 19$$

| ✓ 成功还原明文。

✓ 示例 2 (第38页)

设置:

- $p = 5, q = 11$
- $n = 5 \times 11 = 55$
- $\phi(n) = (5 - 1)(11 - 1) = 4 \times 10 = 40$
- 选择 $e = 3$, 因为 $\gcd(3, 40) = 1$
- 解密指数 $d = 27$, 因为:

$$27 \times 3 = 81 = 40 \times 2 + 1 \Rightarrow 27 \times 3 \equiv 1 \pmod{40}$$

密钥:

- 公钥: $(55, 3)$
- 私钥: 27

加密:

加密公式为:

$$C = M^3 \pmod{55}$$

解密:

解密公式为:

$$M = C^{27} \pmod{55}$$

(可以自行代入具体明文值 M 做一次完整实验)

口 小结:

- 加密操作: $M^e \pmod{n}$
- 解密操作: $C^d \pmod{n}$
- 只要 $d \cdot e \equiv 1 \pmod{\phi(n)}$, 解密就可以成功恢复原始明文。
- 这些例子虽然数值小, 便于学习, 但在实际应用中 p, q 会是几百位的大质数, 确保安全性。

N P-completeness

NP-hard: a language M , defining some decision problem, is NP-hard if every other language L in NP is polynomial-time reducible to M . So this means that

► M is NP-hard if for every $L \in \text{NP}$, $L \xrightarrow{\text{poly}} M$.

If a language M is NP-hard and it belongs to NP itself, then M is NP-complete.

NP-complete problems are some of the *hardest problems* in NP.

N P-完备性

NP-hard: 定义一些决策问题的语言 M 是 NP-hard, 如果 NP 中的所有其他语言 L 都可以多项式时间简化为 M 。这意味着

► 如果对于每个 $L \in \text{NP}$, $L \rightarrow M$, 则 M 是 NP 难的。

如果语言 M 是 NP-hard 并且它本身属于 NP, 则 M 是 NP-complete。

NP 完全问题是 NP 中最难的问题之一。

N P-completeness

Cook-Levin Theorem

Cook-Levin Theorem states that *Circuit-SAT* is NP-complete.

[Cook 1971, Levin 1973]

Proof:

1. Circuit-SAT ∈ NP

Proof: verify an input assignment satisfies a circuit by computing the output of a finite number of gates, one of which will be the output of the circuit. This can be done in polynomial time. Thus, by definition of NP, CIRCUIT-SAT ∈ NP.

N P-完备性

库克-莱文定理

Cook-Levin 定理指出 Circuit-SAT 是 NP 完全的。[Cook 1971, Levin 1973]

证明：1. Circuit-SAT ∈ NP 证明：通过计算有限个门的输出来验证输入赋值是否满足电路，其中一个门的输出将是电路的输出。这可以在多项式时间内完成。因此，根据 NP 的定义，CIRCUIT-SAT ∈ NP。

N P-completeness proof

CNF-SAT and 3-SAT

CNF-SAT

Input: A Boolean formula in CNF.

Question: Is there an assignment of Boolean values to its variables so that the formula evaluates to 1 (i.e. is the formula *satisfiable*)?

3-SAT

Input: A CNF $C = c_1 \wedge c_2 \wedge \dots \wedge c_l$ of clauses dependent on variables x_1, x_2, \dots, x_m such that each c_i is of the form $x_{i_1} \vee x_{i_2} \vee x_{i_3}$ for $1 \leq i \leq m$.

Question: Is there a truth assignment for variables x_i that satisfies C ?

N P-完备性证明

CNF-SAT 和 3-SAT

CNF-SAT

输入：CNF 中的布尔公式。

问题：是否为其变量分配了布尔值，以便公式计算结果为 1（即公式是否可满足）？

3-SAT

输入：一个 CNF $C = c_1 \wedge c_2 \wedge \dots \wedge c_n$, 其子句依赖于变量 x_1, x_2, \dots, x_m , 并且每个 c_i 的形式为 $x_i \vee x_i \vee x_i$, 其中 $1 \leq i \leq n$ 。

问题：变量 x_i 是否存在满足 C 的真值赋值？

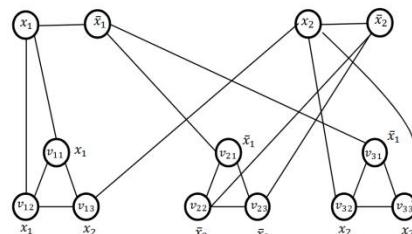
Review N P-completeness

- **P problems:** decision problems that can be solved in polynomial time ($O(n^k)$ for some constant k).
- **NP problems:** decision problems where a proposed solution can be verified in polynomial time.
- **NP-Hard problems:** Problems that are at least as hard as the hardest NP problems. Every problem in NP can be reduced to them in polynomial time. May not be in NP: Could be harder (e.g., halting problem) or unverifiable.
- **NP-Complete Problems:**
 - 1) NP-Hard (all NP problems reduce to them)
 - 2) In NP (solutions can be verified in polynomial time).
- **Polynomial-time reducibility**
 - Transforms any input s for L into an input $f(s)$ for M ,
 - s is a "yes" instance of $L \Leftrightarrow f(s)$ is a "yes" instance of M .

N P-completeness proof

VC 是 NP-complete

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$$



Assume n variables and m clauses. We choose $k=n+2m$.
In this case, $k=2+2*2=8$

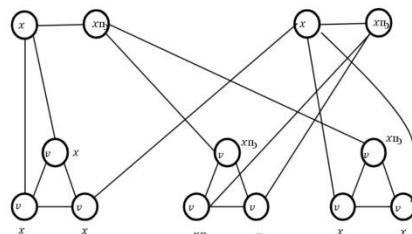
Review N P-completeness

- **P 问题：**可以在多项式时间 ($O(n^k)$ 对于某个常数 k) 内解决的决策问题。
- **NP 问题：**可以在多项式时间内验证所建议的解决方案的决策问题。
- **NP 困难问题：**至少与最困难的 NP 问题一样难的问题。NP 中的每个问题都可以在多项式时间内简化为它们。可能不在 NP 中：可能更难（例如，停止问题）或无法验证。
- **NP 完全问题：**
 - 1) NP-Hard (所有 NP 问题都归结为它们)
 - 2) 在 NP 中 (解可以在多项式时间内验证)。
- 多项式时间可约性
 - 将 L 的任何输入 s 转换为 M 的输入 $f(s)$,
 - s 是 L 的 "yes" 实例 $\Leftrightarrow f(s)$ 是 M 的 "yes" 实例。

N P 完整性证明

VC 是 NP 完备的

$$\phi = (x \vee x \vee x) \wedge (\neg x \vee x \vee x) \wedge (\neg x \vee x \vee x)$$



假设 n 个变量和 m 子句。我们选择 $k=n+2m$ 。在这种情况下， $k=2+3*2=8$

归约 (Reduction) 讲解:

规约是计算复杂性理论中的一种重要技术，它的基本思想是通过将一个已知的困难问题转换成另一个问题。如果我们能把一个已知的**NP-Complete**问题转换为另一个问题，并证明这个转换是多项式时间可行的，那么我们就能够证明这个新问题也是NP-Complete的。

归约的步骤：

1. 选择已知的**NP-Complete**问题：

- 通常从已经证明是NP-Complete的标准问题开始（如 **3-SAT**、**顶点覆盖**、**哈密顿回路** 等），这些问题的难度是已知的。

2. 构造问题之间的映射：

- 找到一个多项式时间的转换算法，将已知的NP-Complete问题的实例转换为你想证明为NP-Complete问题的实例。也就是说，给定已知问题的一个输入，我们需要在多项式时间内构建一个新问题的输入，使得这两个问题的答案是一致的。

3. 证明归约的正确性：

- 确保转换后的问题能准确地反映原问题的答案。即，如果原问题有解，那么转换后的问题也应该有解，反之亦然。这个过程通常需要通过数学证明或构造示例来验证。

归约实例：

以 **3-SAT** 归约到 **顶点覆盖 (VC)** 问题为例：

1. **3-SAT**问题：

- 给定一个3-SAT公式，它由一组子句组成，每个子句由3个文字（变量或其否定）组成。问题是：是否存在一个布尔变量的赋值，使得所有子句至少有一个文字为真？

2. **顶点覆盖问题 (VC)**：

- 给定一个图G和一个整数k，问题是：是否存在一个大小为k的顶点集，能够覆盖图中所有的边（即每条边至少有一个端点在这个顶点集中）？

归约实例：

以 **3-SAT** 归约到 **顶点覆盖 (VC)** 问题为例：

1. 3-SAT问题：

- 给定一个3-SAT公式，它由一组子句组成，每个子句由3个文字（变量或其否定）组成。问题是：是否存在一个布尔变量的赋值，使得所有子句至少有一个文字为真？

2. 顶点覆盖问题 (VC) :

- 给定一个图G和一个整数k，问题是：是否存在一个大小为k的顶点集，能够覆盖图中所有的边（即每条边至少有一个端点在这个顶点集里）？

归约过程：

- 为每个3-SAT公式中的变量构造一个“变量小工具” (variable gadget)，每个子句构造一个“子句小工具” (clause gadget)。
- 在每个变量的小工具中，选择一个节点表示变量的布尔值，而另一个表示其否定。每个子句的小工具由3个节点组成，分别对应3个文字。
- 对于每个子句，连接变量节点与子句节点，确保顶点覆盖问题的解能够覆盖所有的边。

归约的正确性：

- 如果3-SAT公式是可满足的，那么可以选择相应的变量节点，使得每个子句中至少一个文字为真，从而确保每个子句的小工具中有足够的节点被覆盖。
- 反之，若顶点覆盖问题有解，说明选择的节点覆盖了所有子句的小工具，可以对应一个可满足的变量赋值。

归约的重要性：

- 证明NP-Complete性：**通过归约，我们能够证明一个新的问题是NP-Complete，只要我们能够将一个已知的NP-Complete问题多项式时间内归约到这个新问题上。
- 复杂度分析：**归约帮助我们分析问题的难度，了解哪些问题是计算上不可行的（即无法在多项式时间内解决）。



✓ 步骤：使用 欧拉定理 (Euler's Theorem)

前提：若 $\gcd(a, n) = 1$, 那么

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

第一步：检查互质

$$\gcd(2, 505) = 1 \quad \checkmark \text{ 满足欧拉定理}$$

第二步：计算欧拉函数 $\phi(505)$

分解质因数：

$$505 = 5 \times 101 \Rightarrow \phi(505) = \phi(5) \times \phi(101) = (5 - 1)(101 - 1) = 4 \times 100 = 400$$

第三步：使用欧拉定理进行指数约简

$$2^{400} \equiv 1 \pmod{505} \Rightarrow 2^{802} = 2^{400 \cdot 2 + 2} \equiv (1)^2 \cdot 2^2 = 4 \pmod{505}$$

✓ 最终答案：

$$2^{802} \pmod{505} = 4$$

P	NP	NP-hard	NP-complete
能在多项式时间内解决	能在多项式时间内判定一个解是否正确。用非确定算法在非多项式时间内选定解，确定算法在多项式时间内判断解的正确与否	NP问题规约到NP-hard, 但不一定在多项式内可判定	既是NP-hard又是NP问题，在多项式时间内可判定
排序算法，部分背包问题	旅行商问题，01背包问题，SAT可满足性问题，3-SAT，VC顶点覆盖问题，三色图问题	SAT可满足性问题，3-SAT，VC顶点覆盖问题，三色图问题	SAT可满足性问题，3-SAT，VC顶点覆盖问题，三色图问题
证明多项式时间可解决	证明多项式时间内可判定一个解	证明由某NP问题规约来	证明多项式时间内可判定，然后找到一个NPC问题，并且证明那个NPC问题可以规约到这个问题上