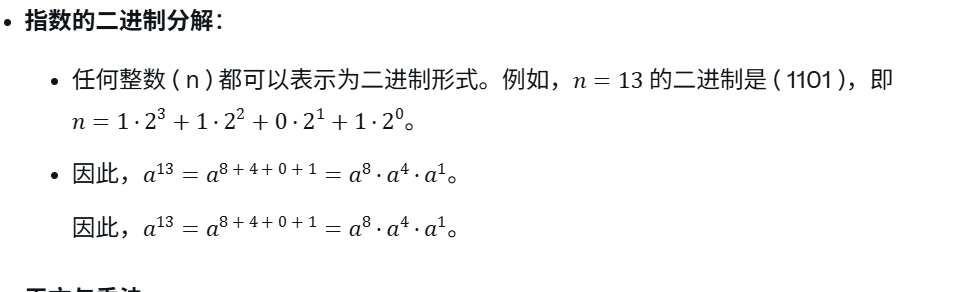
1. 费马小定理&&模幂算法

用通俗的话来说：  
如果 ( p ) 是一个质数，选一个数 ( a )，只要 ( a ) 不是 ( p ) 的倍数，那么 ( a ) 的

p−1次方除以 ( p )，余数一定是 1。

public static boolean isPrime(int n) {  
 // Corner cases  
 if (n <= 1 || n == 4) return false;  
 if (n <= 3) return true;  
  
 int k = 3; // Try k = 3 times  
 while (k > 0)  
 {  
 // Pick a random number in [2..n-2]  
 // Above corner cases make sure that n > 4  
 int a = 2 + (int)(Math.random() % (n - 4));  
  
 // Fermat's little theorem  
 if (power(a, n - 1, n) != 1)  
 return false;  
  
 k--;  
 }  
  
 return true;  
}  
  
/\*\*  
 \* This is a helper method to isPrime. You can ignore this method.  
 \* It is an iterative Function to calculate a^n mod p in log time  
 \*  
 \* @source: https://www.geeksforgeeks.org/primality-test-set-2-fermet-method/  
 \*/  
static int power(int a, int n, int p)  
{  
 // Initialize result  
 int res = 1;  
  
 // Update 'a' if 'a' >= p  
 a = a % p;  
  
 while (n > 0)  
 {  
 // If n is odd, multiply 'a' with result  
 if ((n & 1) == 1)  
 res = (res \* a) % p;  
  
 // n must be even now  
 n = n >> 1; // n = n/2  
 a = (a \* a) % p;  
 }  
 return res;  
2.以下是一个链表的练习题，写了好久:)

public class IntList {

public int first;

public IntList rest;

public IntList(int f, IntList r) {

this.first = f;

this.rest = r;

}

public static void evenOdd(IntList lst) {

if (lst==null||lst.rest==null) {

return;

}

IntList even = lst;

IntList oddhead = lst.rest;

IntList odd=oddhead;

while (even.rest!=null&&odd.rest!=null) {

even.rest=odd.rest;

even=odd.rest;

odd.rest=even.rest;

odd=even.rest;

}

even.rest=oddhead;

}

}

/\*题目要求实现一个方法 `evenOdd`，通过改变给定的 `IntList` 的顺序，使得偶数索引的元素出现在奇数索引的元素之前。例如，如果

`lst` 定义为 `IntList.list(0, 3, 1, 4, 2, 5)`，调用 `evenOdd(lst)` 后，`lst` 会被修改为 `IntList.list(0, 1, 2, 3, 4, 5)`。

你需要通过修改链表的指针来实现这个方法，而不是创建一个新的链表。你的解决方案需要能够处理链表长度为奇数和偶数的情况。

提示：确保你的解决方案对长度为奇数和偶数的链表都能正确工作。\*/（最后发现是力扣上面的原题，屑~）

3.又是一道难题

将一个链表lst均匀地分割成k个子链表，并将这些子链表存储在一个数组中返回。链表长度只能递减1. lst = 5 -> 4 -> 3 -> 2 -> 1 和 k = 2（目标：array[0] = 5 -> 3 -> 2, array[1] = 4 -> 1）

（AI的解法）

public static IntList[] partition(IntList lst, int k) {

IntList[] array = new IntList[k];

IntList[] tails = new IntList[k]; // 辅助数组，跟踪每个子链表的尾部

int index = 0;

IntList L = lst;

while (L != null) {

IntList next = L.rest; // 保存下一个节点

L.rest = null; // 断开当前节点的链接

if (array[index] == null) {

array[index] = L; // 如果子链表为空，设为头部

} else {

tails[index].rest = L; // 否则追加到尾部

}

tails[index] = L; // 更新尾部指针

L = next; // 移动到下一个节点

index = (index + 1) % k; // 循环到下一个子链表 ，使用取模保证了数组在k范围内循环添加，并且多余的部分添加到数组的前端，又学到了:)

}

return array;

}

//还是太吃理解了，唉唉~

多态的编译时类型看左边，运行时类型看右边，但是注意的是，等号右边的new不能出现父类，否则会编译错误。

编译和运行是分开来的，两种错误分开来分析

Object o = "mulchor"; //o编译时是object，运行时是字符串，向下兼容

Planet x = (Planet) o;//planet是object子类，编译没问题，但是运行时转化了string，运行错误

Exception in thread "main" java.lang.ClassCastException:

java.lang.String cannot be cast to Planet

4.一个接口的用例

public int compareTo(Dog uddaDog) {

return this.size - uddaDog.size; // Natural ordering by size

}

private static class NameComparator implements Comparator<Dog> {

public int compare(Dog a, Dog b) {

return a.name.compareTo(b.name); // Compare by name using String's compareTo

}

}

public static Comparator<Dog> getNameComparator() {

return new NameComparator();

}

}嵌套

5.循环队列

rear指向队尾的下一个位置

1.front==rear，队列为空

2.前部出队，front++，后部入队，rear++，

3.超过容量，rear通过取模到最前面（rear=（rear+i）%size）（i是移动步数）

4.front==（rear+1）%size时队列是满的

手写for-each循环

Set<String> s = new HashSet<>();

...for (String city : s) {

...

}

其原理是：

Set<String> s = new HashSet<>();

...

Iterator<String> seer = s.iterator();

while (seer.hasNext()) {

String city = seer.next();

...

}

一个arrayset迭代器，

public Iterator<T> iterator() {

return new ArraySetIterator();

}

private class ArraySetIterator implements Iterator<T> {

private int wizPos;

public ArraySetIterator() {

wizPos = 0;

}

public boolean hasNext() {

return wizPos < size;

}

public T next() {

T returnItem = items[wizPos];

wizPos += 1;

return returnItem;

}

}

使用for-each循环的条件

public class CustomIntArray implements Iterable<Integer> {

private int[] array; // 内部存储数组

private int size; // 数组大小

// 构造函数

public CustomIntArray(int[] input) {

this.array = input;

this.size = input.length;

}

// 实现Iterable接口的iterator()方法

@Override

public Iterator<Integer> iterator() {

return new CustomIntArrayIterator();

}

// 自定义迭代器类

private class CustomIntArrayIterator implements Iterator<Integer> {

private int currentIndex = 0; // 当前迭代位置

// 检查是否还有下一个元素

@Override

public boolean hasNext() {

return currentIndex < size;

}

// 获取下一个元素

@Override

public Integer next() {

if (!hasNext()) {

throw new NoSuchElementException();

}

return array[currentIndex++];

}

// 可选：实现remove方法（这里抛出不支持异常）

@Override

public void remove() {

throw new UnsupportedOperationException("Remove operation not supported");

}

}

重写object类里面的equals()

public boolean equals(Object other) {

if (this == other) {

return true;

}

if (other == null) {

return false;

}

if (other.getClass() != this.getClass()) {

return false;

}//检查类型是否相等getClass()

ArraySet<T> o = (ArraySet<T>) other;

if (o.size() != this.size()) {

return false;

}

for (T item : this) {

if (!o.contains(item)) {

return false;

}

}

return true;

}

Try-catch

try {  
 d.receivePat();  
} catch (Exception e) {  
 System.out.println("Tried to pat: " + e);  
}

try没有问题，跳过这几个语句

Try发现问题，跳转到catch，catch接收异常并操作

时间复杂度

public static void printParty(int N) {

for (int i = 1; i <= N; i = i \* 2) {

for (int j = 0; j < i; j += 1) {

System.out.println("hello");

int ZUG = 1 + 1;

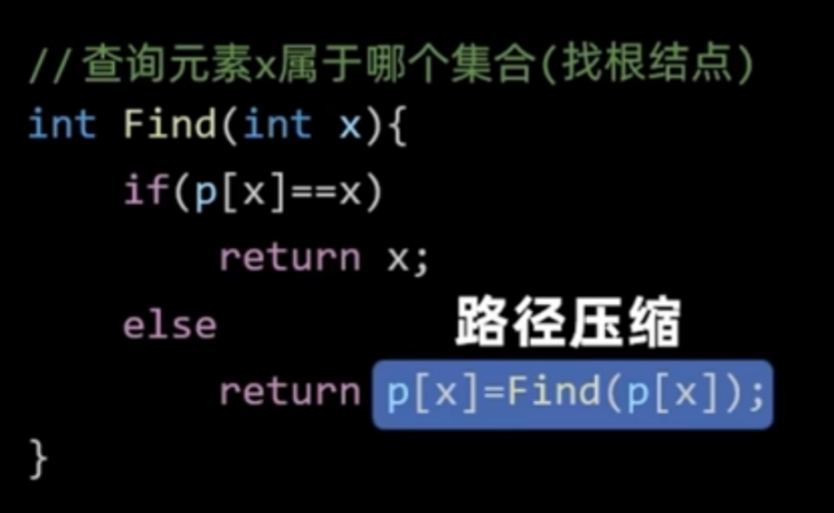
}

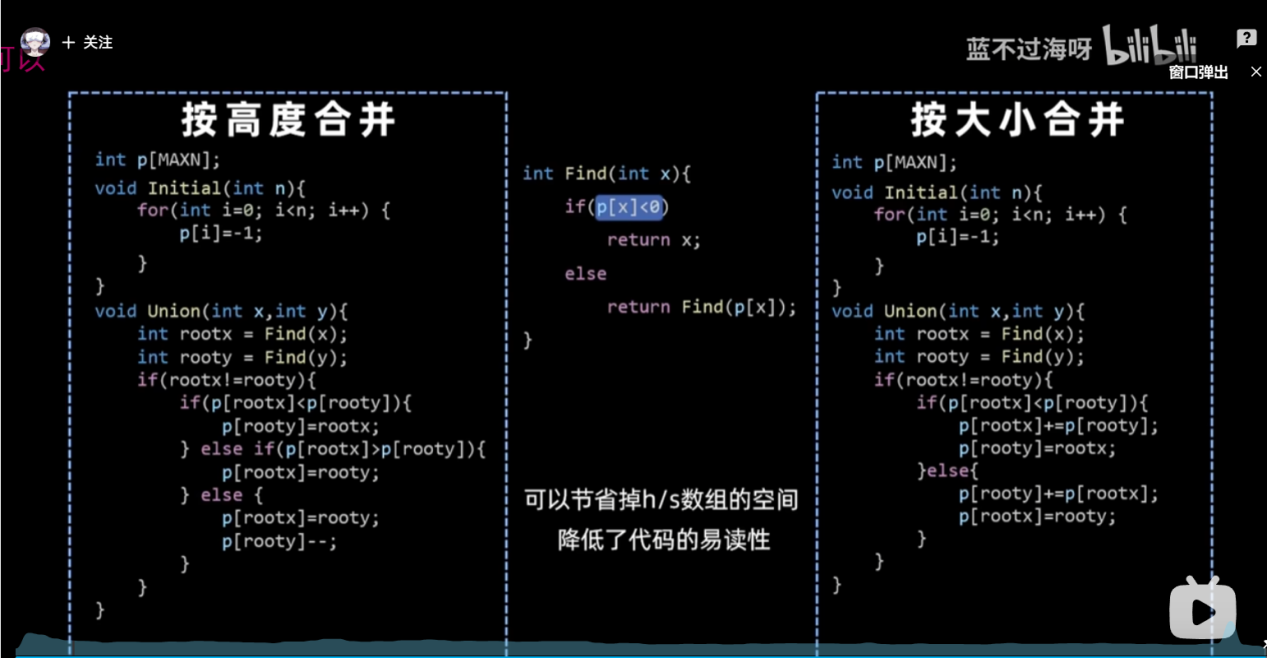
}

}

外层执行t=log2(n)次，内层执行2\*\*t次，时间复杂度为2\*\*(log2(n))=n

并查集





AVL树的旋转，

rotateLeft(G): Let x be the right child of G. Make G the new left child of x.

rotateRight(G): Let x be the left child of G. Make G the new right child of x.

左旋（rotateLeft(G)）：令 x 为 G 的右孩子节点。将 G 变成 x 的新的左孩子节点。

右旋（rotateRight(G)）：令 x 为 G 的左孩子节点。将 G 变成 x 的新的右孩子节点。

private Node rotateRight(Node h) {

// assert (h != null) && isRed(h.left);

Node x = h.left;

h.left = x.right;

x.right = h;

return x;

}

// make a right-leaning link lean to the leftprivate Node rotateLeft(Node h) {

// assert (h != null) && isRed(h.right);

Node x = h.right;

h.right = x.left;

x.left = h;

return x;

}

红黑树

### **1. **每个节点要么是红色，要么是黑色****

这是红黑树的基本分类规则。节点的颜色用于维护树的平衡性。

### **2. **根节点是黑色****

根节点必须是黑色，这是为了保证树的平衡性。如果根节点是红色，可能会导致一些复杂的平衡问题。

### **3. **叶子节点（空节点）是黑色****

叶子节点是指那些没有子节点的节点（即空节点或 NULL）。这些节点必须是黑色。这确保了树的底部是平衡的。

### **4. **红色节点的两个子节点都是黑色****

换句话说，不能有两个连续的红色节点。如果一个节点是红色，那么它的两个子节点必须都是黑色。这条性质确保了树的平衡性，防止树退化成链表。

### **5. **从任意节点到其每个叶子的所有路径都包含相同数量的黑色节点****

这条性质确保了从根到叶子的路径长度大致相等，从而保证了树的平衡性。具体来说，从根到任意叶子的路径上，黑色节点的数量是相同的。

注意插入的节点默认是红色的

完全二叉树的性质

leftChild(k)=k∗2=k∗2

rightChild(k)=k∗2+1=k∗2+1

parent(k) =k/2=k/2