1. 算法

我们在很多课上都学习了算法。
算法是一组有限的、定义明确的、可由计算机实现的指令，通常用于解决一类问题或者执行某种计算。
算法分析是研究算法的计算复杂性的过程。
算法设计是开发用于解决问题的算法的过程。

1.1 算法分析

1.1 时间复杂度

如果我们使用执行时间来衡量算法性能，其具有局限性。
因为执行时间依赖于具体输入和系统环境。

所以我们使用大O符号衡量算法效率。
大O符号不关心算法在特定输入数据或特定计算机硬件上的具体执行时间，而是关注算法在输入规模变化时的增长趋势。
这种方法通过分析算法的时间复杂度来衡量其效率，时间复杂度描述了算法的执行时间如何随着输入规模的增加而变化。

使用大O符号来描述算法的时间复杂度时，需要注意两点：

1. 可以忽略乘法常数，因为它们不影响算法的增长趋势。
2. 忽略非主导项，因为但当n增大时，主导项（如n）的增长速度远远超过非主导项（如-1），因此非主导项对整体复杂度的影响变得微不足道。

我们还需要知道如果一个算法的执行时间与输入规模无关，即无论输入规模如何变化，执行时间始终保持不变，那么这个算法被称为具有常数时间复杂度，用符号$O(1)$表示。
因为在计算机中，数组的元素是连续存储的，每个元素的内存地址可以通过公式计算得出：地址=数组起始地址+索引×元素大小
这个计算过程是固定的，不依赖于数组的长度。
因此，无论数组有多长，访问任意一个元素的时间都是相同的。

1.1.2 空间复杂度

空间复杂度是指算法在运行过程中所占用的内存空间的大小。
它衡量的是算法在执行过程中需要多少额外的存储空间，包括输入数据本身占用的空间以及算法运行时额外需要的空间。
大O符号不仅可以用于描述时间复杂度，也可以用于描述空间复杂度。
例如，如果一个算法的空间复杂度是 $O(n)$，这意味着算法占用的内存空间与输入规模n成线性关系。
如果空间复杂度是$O(1)$，则表示算法占用的内存空间是固定的，不随输入规模变化。
这里顺便复习一下等比数列求和公式：$Sn = a_1(1-q^n)/(1-q)$，$(q \neq 1)$。
所以$a_0 + a_1 + a_2 + a_3 + \ldots + a_{(n-1)} + a_n = a^{n+1} - 1/a - 1$。

1.1.3 具体分析

我们现在使用大O符号来确定不同类型算法的时间复杂度。

1. Repetition（重复结构）
   重复结构通常指的是循环结构，如for循环或while循环，其中算法重复执行一系列操作固定次数或直到满足某个条件。
   如果循环运行固定次数（例如，for (int i = 0; i < n; i++)），则时间复杂度通常是$O(n)$。
   如果循环次数与输入规模的对数相关（例如，while (n > 1) { n = n/2; }），则时间复杂度可能是$O(logn)$。
   如果循环嵌套（例如，双重循环），则时间复杂度是各个循环次数的乘积（例如，for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { ... } } 的时间复杂度是 $O(n^2)$）。
2. Sequence（序列结构）
   序列通常指的是一系列操作，每个操作都独立于其他操作，并且通常按顺序执行。
   如果序列中的每个操作都是常数时间（$O(1)$），并且有n个这样的操作，则总的时间复杂度是$O(n)$。
   如果操作的数量或复杂度随输入规模增加而增加，则总的时间复杂度可能会更高。
3. Selection（选择结构）
   选择结构通常指的是if-else语句或switch-case语句，用于基于条件选择不同的执行路径。
   选择结构本身通常不增加时间复杂度，因为它们只是决定执行哪一部分代码。
   总的时间复杂度取决于被选择执行的代码块的复杂度。
   如果所有可能的执行路径都有相同的时间复杂度，则整个选择结构的时间复杂度就是这些路径中最大的一个。
4. Logarithm（对数结构）
   对数时间复杂度通常出现在分治算法中，如二分搜索或归并排序。
   在每次迭代中，问题规模减半（例如，二分搜索每次将搜索范围减半），则时间复杂度是$O(logn)$。
   对数时间复杂度表明算法的效率非常高，尤其是在处理大规模数据时。

1.1.3.1 Repetition（重复结构）

下面给出一些例子。
例1：

```
for (i = 1; i <= n; i++) {
    k = k + 5;
}
```

这里的算法复杂度为$O(n)$。

```
public class PerformanceTest {
    public static void main(String[] args) {
        getTime(1000000);
        getTime(10000000);
        getTime(100000000);
        getTime(1000000000);
    }
    public static void getTime(long n) {
        long startTime = System.currentTimeMillis();
        long k = 0;
        for (int i = 1; i <= n; i++) {
            k = k + 5;
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Execution time for n = " + n
                + " is " + (endTime - startTime) + " milliseconds");
    }
}
```

我们简单测试一下这个线性算法的性能。
执行时间如下：
对于$n = 1,000,000$，执行时间为6毫秒。
对于$n = 10,000,000$，执行时间为61毫秒。
对于$n = 100,000,000$，执行时间为610毫秒。
对于$n = 1,000,000,000$，执行时间为6048毫秒。
从输出结果可以看出，执行时间与输入规模n成线性关系。具体来说：
当n增加 10 倍时，执行时间也大约增加10倍。

例2：

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++) {
        k = k + i + j;
    }
}
```

这是一个嵌套循环，外层和内层都是n次，所以算法复杂度为$O(n^2)$。
当然我们也可以计算一下，过程如下：
$T(n) = c + 2c + 3c + 4c + \ldots + nc = cn(n+1)/2 = (c/2)n^2 + (c/2)n = O(n^2)$

例3：

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= 20; j++) {
        k = k + i + j;
    }
}
```

我们如果稍微修改以下这个嵌套循环，现在内层是20次，那么相乘的结果是$20n$次，所以我们忽略这里的常数，算法复杂度为$O(n)$。

1.1.3.2 Sequence（序列）

```
for (i = 1; i <= n; i++) {
    k = k + 5;
}
for (i = 1; i <= n; i++) {
    for (j = 1; j <= 20; j++) {
        k = k + i + j;
    }
}
```

我们现在这里是前面两个代码放在了一起，从上到下依次执行，所以$T(n) = c * 10 + 20 * c * n = O(n)$

1.1.3.3 Selection（选择结构）

```
if (list.contains(e))
    System.out.println(e);
else
    for (Object t: list)
        System.out.println(t);
```

我们计算时间复杂度计算最坏情况，所以算法复杂度为$T(n) = O(n) + O(n) = O(n)$。

1.1.3.4 Logarithm（对数结构）

我们先看一个算法。

```
result = 1;
for (int i = 1; i <= n; i++)
    result *= a;
```

这个算法计算了a^n，其时间复杂度为$O(n)$。

我们现在可以假设$n = 2^k$，即$k = log_2 n$。
因此我们获得一个改进的算法，

```
result = a;
for (int i = 1; i <= k; i++)
    result = result * result;
```

这个新的算法，$T(n) = k = logn = O(logn)$，这就是对数时间。

所以我们回到经典的二分查找上。

```
public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;
    while (high >= low) {
        int mid = (low + high) / 2;
        if (key < list[mid])
            high = mid - 1;
        else if (key == list[mid])
            return mid;
        else
            low = mid + 1;
    }
    return -1 - low;
}
```

在每次迭代中，搜索范围的大小从n减少到$n/2$。
经过k次迭代后，搜索范围将减少到1（即找到目标元素或确定目标元素不存在）。
因此，二分搜索的时间复杂度是$O(log_2n)$，其中$k = log_2n$表示迭代次数。
所以对数时间算法（如二分搜索）非常高效，因为它们随着输入规模的增加而缓慢增长。
这种算法的效率在于，即使输入规模显著增加，所需的额外时间也相对较小。
因此，对数时间算法在处理大规模数据时特别有用，因为它们可以在合理的时间内完成任务，即使数据量很大。

1.1.3.5 二次算法（quadratic algorithm）

我们再看一下选择排序。

```
public static void selectionSort(double[] list) {
    for (int i = 0; i < list.length; i++) {
        // Find the minimum in the list[i..list.length-1]
        double currentMin = list[i];
        int currentMinIndex = i;
        for (int j = i + 1; j < list.length; j++) {
            if (currentMin > list[j]) {
                currentMin = list[j];
                currentMinIndex = j;
            }
        }
        // Swap list[i] with list[currentMinIndex] if necessary
        if (currentMinIndex != i) {
            list[currentMinIndex] = list[i];
            list[i] = currentMin;
        }
    }
}
```

选择排序的总比较次数是$(n-1)+(n-2)+\ldots+1$，这是一个等差数列求和问题，其和为 $n(n-1)/2$。
因此，选择排序的比较次数是$O(n^2)$。
除了比较操作外，选择排序还需要执行一些其他操作，如赋值和额外的比较（例如，检查 currentMinIndex != i）。
这些操作的数量可以表示为一个常数c，因为它们在每次迭代中都是固定的。
由于每次迭代都需要执行c次其他操作，而迭代次数为n，因此总的其他操作次数为cn，而这一部分会忽略，所以选择排序的总时间复杂度为$T(n) = n(n-1)/2 + cn = O(n^2)$。

对于这种时间复杂度为$O(n^2)$的算法,我们称其为二次算法。
二次算法随着问题规模的增加而快速增长。
如果你将输入规模翻倍（例如，从 n 增加到 2n），二次算法的执行时间将增加四倍。
嵌套循环的算法通常是二次的。

我们再看一个排序算法——插入排序。
对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。
详细步骤如下：
假设我们有一个数组 list，插入排序的算法步骤如下：

1. 从第一个元素开始，该元素可以认为已经被排序。
2. 取出第二个元素，在已经排序的第一个元素中从后向前扫描。
3. 如果第一个元素大于当前元素，将第一个元素移到下一个位置。
4. 重复步骤3，直到找到已排序的元素小于或者等于当前元素的位置。
5. 将当前元素插入到该位置后。
6. 重复步骤2~5，直到数组的最后一个元素已经被扫描完。

代码如下。

```
public static void insertionSort(int[] list) {
    for (int i = 1; i < list.length; i++) {
        int currentElement = list[i];
        int k;
        for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
            list[k + 1] = list[k];
        }
        // Insert the current element into list[k + 1]
        list[k + 1] = currentElement;
    }
}
```

其时间复杂度分析如下：
插入每个元素时，最多需要$n-1$次比较，和$n-1$次交换，因此总的比较和移动次数为：
$2 * ((n-1)+(n-2)+\ldots+2+1) = n(n-1)$。
因此插入排序的时间复杂度是$O(n^2)$。

1.1.3.6 多项式时间复杂度（Polynomial Time Complexity）

如果一个算法的运行时间可以被输入规模n的某个多项式表达式上界限制，即 $T(n) = O(n^k)$，其中k是某个正常数，则称该算法具有多项式时间复杂度。
多项式时间复杂度的概念引出了计算复杂性理论中的几个重要复杂度类：

1.P类（Polynomial Time）：
P类是可以在确定性图灵机上在多项式时间内解决的决策问题的复杂度类。
如果一个问题属于P类，那么存在一个算法可以在多项式时间内确定性地解决它。

2.NP类（Nondeterministic Polynomial Time）：
NP类是可以在非确定性图灵机上在多项式时间内解决的决策问题的复杂度类。
如果一个问题属于NP类，那么对于它的任意一个给定解，都可以在多项式时间内进行验证。

求解一个问题意味着找到问题的解，这通常需要进行搜索或计算。
验证一个问题的给定解意味着检查解是否正确。

NP问题是指那些给定一个解可以在多项式时间内验证其正确性的问题。
NP难问题（NP-hard）是指那些至少和NP中最难的问题一样难的问题。
一个问题既是NP问题又是NP难问题，则称其为NP完全问题（NP-complete）。
P类是NP类的子集，即所有可以在多项式时间内确定性地解决的问题都可以在多项式时间内进行验证。
然而，NP类是否等于P类，这便是P=NP问题，它是计算复杂性理论中最著名的未解决问题之一，它询问是否可以在多项式时间内解决所有NP问题。
NP完全问题有：

1. 布尔可满足性问题（Boolean satisfiability problem, SAT）：
   给定一个布尔表达式，判断是否存在一种变量赋值使得整个表达式为真。
2. 背包问题（Knapsack problem）：
   给定一组物品，每个物品有其重量和价值，确定在不超过背包最大承重的情况下，如何选择物品以使总价值最大。
3. 哈密顿路径问题（Hamiltonian path problem）：
   判断图中是否存在一条路径，该路径恰好访问每个顶点一次。
4. 旅行商问题（Traveling salesman problem）：
   给定一组城市和每对城市之间的距离，找到一条最短的可能路径，该路径恰好访问每个城市一次并返回起点。
5. 图着色问题（Graph coloring problem）：
   给定一个图，使用最少的颜色对图的顶点进行着色，使得没有两个相邻的顶点具有相同的颜色。
6. 子图同构问题（Subgraph isomorphism problem）：
   判断一个图是否是另一个图的子图。
7. 子集和问题（Subset sum problem）：
   给定一组整数和一个目标值，判断是否存在一个整数子集，其和恰好等于目标值。
8. 团问题（Clique problem）：
   在图中找到一个最大的完全子图（团），即图中的每个顶点都与其他所有顶点相连。
9. 顶点覆盖问题（Vertex cover problem）：
   找到图中的最小顶点集，使得图中的每条边至少与该集合中的一个顶点相关联。
10. 独立集问题（Independent set problem）：
    找到一个图中最大的顶点集，使得集合中的任意两个顶点都不相邻。
11. 支配集问题（Dominating set problem）：
    找到一个图中的最小顶点集，使得图中的每个顶点要么属于该集合，要么与该集合中的至少一个顶点相邻。

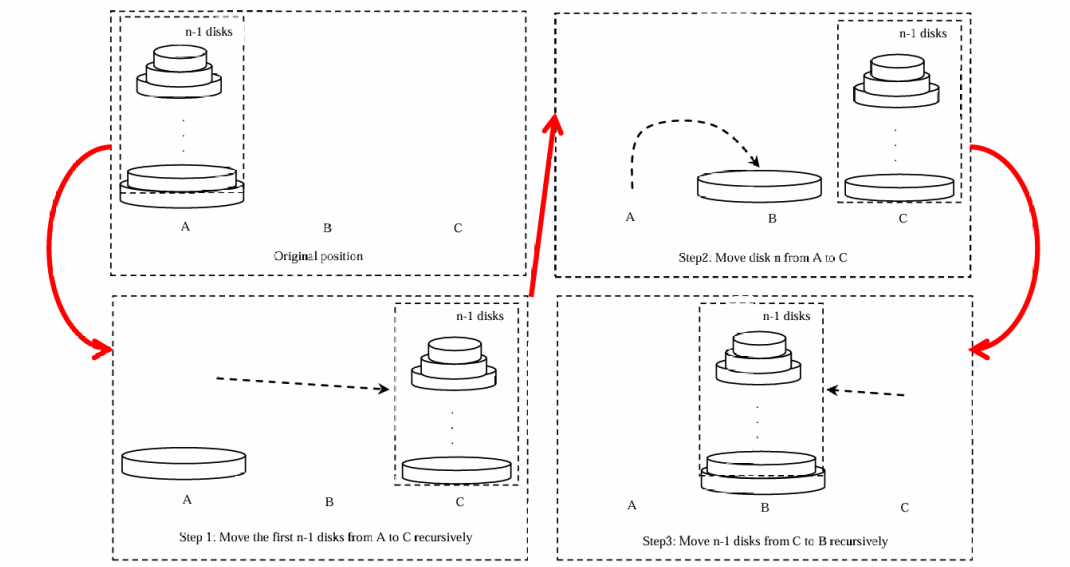
1.1.3.6.1 3-SAT问题

3-SAT是布尔可满足性问题（SAT）的一个特例，其中公式以合取范式（conjunctive normal form，CNF）给出，并且每个子句（clause）限制为最多三个文字（literals）。
在3-SAT中，公式的形式如下：
$(l_1 \vee l_2 \vee l_3) \wedge (l_4 \vee l_5 \vee l_6) \wedge \cdots \wedge (l_{n-2} \vee l_{n-1} \vee l_n)$
其中，$\vee$表示逻辑或（OR），$\wedge$表示逻辑与（AND）。
3-SAT是NP完全问题，这意味着它不仅自身是NP问题，而且NP类中的所有问题都可以在多项式时间内归约到3-SAT问题上。

1.1.3.7 指数算法（exponential algorithm）

汉诺塔问题是一个经典的数学和计算机科学问题，它涉及将一组盘子从一个塔移动到另一个塔，遵循特定的规则：
有n个盘子，标记为$1, 2, 3, \ldots, n$，其中盘子1是最小的，盘子n是最大的。
有三个塔，标记为A, B, 和C。
所有盘子最初都放置在塔A上，且按照从大到小的顺序堆叠（最大的盘子在底部，最小的盘子在顶部）。
每次只能移动一个盘子，且必须是塔顶的盘子。且任何时候，较大的盘子不能放在较小的盘子上面。
目标是将所有盘子从塔A移动到塔C（或另一个指定的塔），同时遵守上述规则。

汉诺塔问题的时间复杂度是指数级的，具体来说，解决 n 个盘子的汉诺塔问题所需的最小移动次数为$2^n - 1$。这是因为：
我们先将$n-1$个盘子从起始塔移动到辅助塔。
再将第n个盘子从起始塔移动到目标塔。
最后将$n-1$个盘子从辅助塔移动到目标塔。
这种递归解决方案导致移动次数呈指数增长。
$T(n) = T(n-1) + 1 + T(n-1) = 2T(n-1) + 1 = 2(2T(n-2) + 1) + 1 = 2(2(2T(n-3) + 1) + 1) + 1 = 2^{n-1}T(1) + 2^{n-2} + \ldots + 2 + 1$



代码如下。

```
import java.util.Scanner;

public class TowersOfHanoi {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter number of disks: ");
        int n = input.nextInt();
        System.out.println("The moves are:");
        moveDisks(n, 'A', 'B', 'C');
    }

    public static void moveDisks(int n, char fromTower, char toTower, char auxTower) {
        if (n == 1) { // Stopping condition
            System.out.println("Move disk " + n + " from " + fromTower + " to " + toTower);
        } else {
            moveDisks(n - 1, fromTower, auxTower, toTower);
            System.out.println("Move disk " + n + " from " + fromTower + " to " + toTower);
            moveDisks(n - 1, auxTower, toTower, fromTower);
        }
    }
}
```

汉诺塔问题的解决方案具有$O(2^n)$的时间复杂度，这被称为指数算法。
指数算法在处理大规模输入时效率非常低，因为它们的执行时间增长得非常快。
如果每秒移动一个盘子，移动32个盘子需要的时间是2^32秒，也就是136年。
而移动64个盘子需要的时间是2^64秒，也就是大约5850亿年。

1.1.3.8 递归算法

著名的斐波拉契数列就是递归算法的一个最好示例。

```
public static int fib(int index) {
    if (index == 0) // 基本情况：斐波那契数列的第0项是0
        return 0;
    else if (index == 1) // 基本情况：斐波那契数列的第1项是1
        return 1;
    else  // 递归情况：计算第index项
        return fib(index - 1) + fib(index - 2);
}
```

我们尝试下面的代码。

```
import java.util.Scanner;
public class Fibonacci2 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the Fibonacci index: ");
        int n = input.nextInt();
        System.out.println("fib(" + n + ") = " + fib(n));
        System.out.println("steps: " + steps);
    }
    static int steps = 0;
    public static int fib(int index) {
        steps++;
        if (index == 0) // 基本情况
            return 0;
        else if (index == 1) // 基本情况
```

```
            return 1;
        else // 递归情况
            return fib(index - 1) + fib(index - 2);
    }
}
```

现在输出结果的同时会输出递归调用了多少词，我们可以发现随着输入索引的增加，所需的递归调用次数呈指数增长。

$$T(n) = T(n-1) + T(n-2) + c = T(n-2) + T(n-3) + c + T(n-2) + c \geq 2T(n-2) + 2c \geq 2(2T(n-4) + 2c) + 2c \geq 2^2T(n-2^2) + 2$$

当然我们以前就知道，这里可以使用动态编程（Dynamic Programming）的思想去优化该算法。

```
public static int fib(int n) {
    if (n == 0)
        return 0;
    else if (n == 1 || n == 2)
        return 1;
    int f0 = 0;  // For fib(0)
    int f1 = 1;  // For fib(1)
    int f2 = 1;  // For fib(2)
    for (int i = 3; i <= n; i++) {
        f0 = f1;
        f1 = f2;
        f2 = f0 + f1;
    }
    return f2;
}
```

所以现在不会重复计算已计算的数列对象，因此现在的时间复杂度是$O(n)$，算法得到了大大的提升。

```
import java.util.Scanner;

public class Fibonacci3 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the Fibonacci index: ");
        int n = input.nextInt();
        System.out.println("fib(" + n + ") = " + fib(n));
        System.out.println("steps: " + steps);
    }
    static int steps = 0;
    public static int fib(int n) {
        if (n == 0)
            return 0;
        else if (n == 1 || n == 2)
            return 1;
        int f0 = 0; // For fib(0)
        int f1 = 1; // For fib(1)
        int f2 = 1; // For fib(2)
        steps = 3;
        for (int i = 3; i <= n; i++) {
            steps++;
            f0 = f1;
            f1 = f2;
            f2 = f0 + f1;
        }
        return f2;
    }
}
```

我们可以用这个代码去比较前后现在的运行次数的差异。

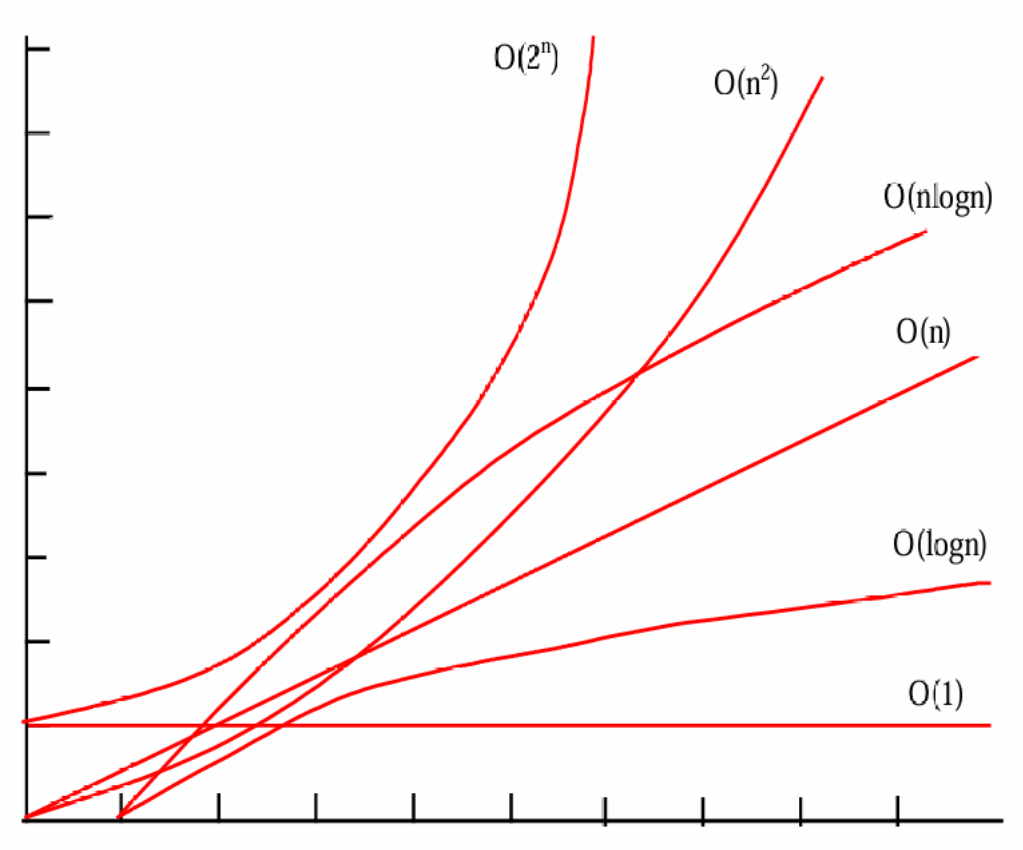
1.1.3.9 通过递推关系计算复杂度

递归关系是分析算法复杂度的有用工具。
如下图所示。

Recurrence Relation	Result	Example
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n\log n)$	Merge sort
$T(n) = 2T(n/2) + O(n\log n)$	$T(n) = O(n\log^2 n)$	
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort, insertion sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Towers of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

1.1.3.10 时间复杂度的比较。

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



1.2 算法设计

算法设计的步骤如下：

1. 问题定义（Problem definition）。
2. 开发模型（Development of a model）。
3. 算法规范（Specification of the algorithm）。
4. 设计算法（Designing an algorithm）。
5. 检查算法的正确性（Checking the correctness of the algorithm）。
6. 算法分析（Analysis of algorithm）。
7. 算法实现（Implementation of algorithm）。
8. 程序测试（Program testing）。
9. 文档编写（Documentation preparation）。

1.2.1 算法设计技术（Algorithm Techniques）

1. 暴力搜索（Brute-force or exhaustive search）：
   尝试每一种可能的解决方案，以找到最优解。
2. 分治法（Divide and conquer）：
   分治法是一种将问题分解为更小的子问题，递归地解决这些子问题，然后将结果合并以得到原始问题的解的方法。例如归并排序（Merge sort）。

3. 动态规划（Dynamic Programming）：
   它通过存储已经计算过的子问题的解（称为记忆化或缓存），避免重复计算，从而提高效率。例如非递归斐波那契数列（Non-recursive Fibonacci）。
4. 贪心算法（Greedy Algorithms）：
   贪心算法是一种在每一步选择中都采取当前状态下最优（最有利）的选择，从而希望导致结果是全局最优的算法策略。例如旅行商问题（Traveling Salesman Problem, TSP），目标是找到访问每个城市一次并返回起点的最短路径。用贪心策略我们就会在每一步选择最近的未访问城市作为下一个访问目标，但这种策略不保证找到最优解，却可以在合理的步骤内找到一个可行解。
5. 回溯算法（Backtracking）：
   回溯算法是一种通过递归地尝试所有可能的候选解，并在确定某个候选解不可能是最终解时回退到上一步的方法。例如解决约束满足问题（CSP），如八皇后问题、数独等。它不保证找到最优解，但可以找到所有可能的解。

1.2.1.1 动态编程与递归的对比

动态规划的核心思想是每个子问题只计算一次，并将结果存储起来。
这些结果可以被存储（通常在数组中），以便在需要时重复使用，从而避免重复计算相同的子问题。
因此其具有避免冗余计算和提升效率的优点。

1.2.2 最大公约数（Greatest Common Divisor, GCD）

我们先看一个版本的代码。

```
public static int gcd(int m, int n) {
    int gcd = 1;
    for (int k = 2; k <= m && k <= n; k++) {
        if (m % k == 0 && n % k == 0)
            gcd = k;
    }
    return gcd;
}
```

这个版本的算法通过暴力搜索的方法来找到两个数的最大公约数，其时间复杂度是 $O(n)$。

完整代码如下。

```
import java.util.Scanner;

public class GCD {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the numbers: ");
        int n1 = input.nextInt();
        int n2 = input.nextInt();
        System.out.println("gcd(" + n1 + "," + n2 + ") = " + gcd(n1, n2));
    }

    public static int gcd(int m, int n) {
        int gcd = 1;
        for (int k = 2; k <= m && k <= n; k++) {
            if (m % k == 0 && n % k == 0)
                gcd = k;
        }
        return gcd;
    }
}
```

我们再看一个版本的代码。

```
public static int gcd(int m, int n) {
    int gcd = 1;
    for (int k = n; k >= 1; k--) {
        if (m % k == 0 && n % k == 0) {
            gcd = k;
            break;
        }
    }
    return gcd;
}
```

这个版本从n开始，递减到1。在找到最大公约数后立即终止，避免了不必要的计算。然而其时间复杂度还是 $O(n)$。
完整代码如下。

```
import java.util.Scanner;

public class GCD2 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the numbers: ");
        int n1 = input.nextInt();
        int n2 = input.nextInt();
        System.out.println("gcd(" + n1 + "," + n2 + ") = " + gcd(n1, n2));
    }
```

```
    public static int gcd(int m, int n) {
        int gcd = 1;
        for (int k = n; k >= 1; k--) {
            if (m % k == 0 && n % k == 0) {
                gcd = k;
                break;
            }
        }
        return gcd;
    }
}
```

下面再展示一个版本。

```
public static int gcd(int m, int n) {
    int gcd = 1;
    if (m == n) return m;
    for (int k = n / 2; k >= 1; k--) {
        if (m % k == 0 && n % k == 0) {
            gcd = k;
            break;
        }
    }
    return gcd;
}
```

这个版本变量k从$n/2$开始，一直递减到1。这个优化可以减少一些不必要的计算，特别是在m和n相等的情况下。然而其时间复杂度还是$O(n)$。

1.2.2.1 欧几里得算法（Euclid's Algorithm）

欧几里得算法是由古希腊数学家欧几里得（Euclid）在公元前300年左右发现的。
该算法的步骤如下：
设$gcd(m,n)$表示整数m和n的最大公约数。
如果m能被n整除（即m），则n是m和n的最大公约数。
否则，m和n的最大公约数等于n和$m\%n$（m 除以n的余数）的最大公约数。

数学证明如下：
如果p是m和n的公约数，那么p也必须是$m\%n$（即r）的公约数。
这是因为如果$m = n * k + r$，其中k和r是整数，那么m/p可以表示为 $(n/p) * k + (r/p)$，其中 (n/p)和 (r/p)也必须是整数。

代码如下。

```
public static int gcd(int m, int n) {
    if (m % n == 0)
        return n;
    else
        return gcd(n, m % n);
}
```

其算法复杂度为$O(logn)$。
证明如下：
假设$m \geq n$，我们可以证明$m\%n < m/2$，如下：
如果$n \leq m/2$，那么$m\%n < m/2$，因为m除以n的余数总是小于n。
如果$n > m/2$，那么$m\%n = m - n < m/2$。
因此，$m\%n < m/2$。
由于$m\%n < m/2$和$n\%(m\%n) < n/2$，每次递归调用后，传递给 gcd 方法的参数大约减半。
这种减半的特性导致算法的时间复杂度为$O(logn)$。

完整代码如下。

```
import java.util.Scanner;

public class GCD4 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the numbers: ");
        int n1 = input.nextInt();
        int n2 = input.nextInt();
        System.out.println("gcd(" + n1 + "," + n2 + ") = " + gcd(n1, n2));
    }

    public static int gcd(int m, int n) {
        if (m % n == 0)
            return n;
        else
            return gcd(n, m % n);
    }
}
```

1.2.3 寻找素数（Finding Prime Numbers）

1. 暴力搜索（Brute-force）：
   这种方法涉及检查从2到$n-1$的每个数字，看它是否能整除n。
2. 检查到$\sqrt{n}$的可能除数：
   这种方法利用了素数的一个性质：如果n不是素数，那么它必定有一个不大于$\sqrt{n}$的因子。
   因此，只需要检查到$\sqrt{n}$的除数即可，这大大减少了需要检查的除数数量。
   解决方法如下。
3. 检查到$\sqrt{n}$的可能素数除数：
   这种方法进一步优化了第二种方法，只检查到$\sqrt{n}$的素数作为除数。
   这意味着，首先需要生成一个小于$\sqrt{n}$的素数列表，然后使用这个列表来检查n是否为素数。
   这种方法比前两种方法更高效，因为它减少了除法运算的次数，并且只考虑了素数除数。

```
import java.util.Scanner;

public class PrimeNumbers {
    public static void main(String[] args) {
       Scanner input = new Scanner(System.in);
       System.out.print("Find all prime numbers <= n, enter n: ");
       int n = input.nextInt();
       final int NUMBER_PER_LINE = 10; // Display 10 per line
       int count = 0; // Count the number of prime numbers
       int number = 2; // A number to be tested for primeness
       System.out.println("The prime numbers are:");
       // Repeatedly find prime numbers
       while (number <= n) {
           // Assume the number is prime
           boolean isPrime = true; // Is the current number prime?
           // Closest if number is prime
           for (int divisor = 2; divisor <= (int)(Math.sqrt(number)); divisor++) {
               if (number % divisor == 0) { // If true, number is not prime
                   isPrime = false; // Set isPrime to false
                   break; // Exit the for loop
               }
           }
           // Print the prime number and increase the count
           if (isPrime) {
               count++; // Increase the count
               if (count % NUMBER_PER_LINE == 0) {
                   // Print the number and advance to the new line
                   System.out.printf("%7d\n", number);
               }
              else
               System.out.printf("%7d", number);
           }
           // Check if the next number is prime
           number++;
       }
       System.out.println("\n" + count + " prime(s) less than or equal to " + n);
   }
}
```

完整代码如下。

```
import java.util.Scanner;

public class Primes {
    public static void main(String[] args) {
       Scanner input = new Scanner(System.in);
       System.out.print("Find all prime numbers <= n, enter n: ");
       int n = input.nextInt();
       final int NUMBER_PER_LINE = 10; // Display 10 per line
       int count = 0; // Count the number of prime numbers
       int number = 2; // A number to be tested for primeness
       System.out.println("The prime numbers are:");
       // Repeatedly find prime numbers
       while (number <= n) {
           // Assume the number is prime
           boolean isPrime = true; // Is the current number prime?
           // Closest if number is prime
           for (int divisor = 2; divisor <= (int) (Math.sqrt(number)); divisor++) {
               if (number % divisor == 0) { // If true, number is not prime
                   isPrime = false; // Set isPrime to false
                   break; // Exit the for loop
               }
           }
           // Print the prime number and increase the count
           if (isPrime) {
               count++; // Increase the count
               if (count % NUMBER_PER_LINE == 0) {
                   // Print the number and advance to the new line
```

```
                    System.out.printf("%7d\n", number);
                } else {
                    System.out.printf("%7d ", number);
                }
            }
            // Check if the next number is prime
            number++;
        }
        System.out.println("\n" + count + " prime(s) less than or equal to " + n);
    }
}
```

在这个算法中每次循环迭代都需要计算 Math,sqrt(number)，这倒是了大量的重复计算，从而降低了效率。
因此可以改进为以下代码。

```
int squareRoot = (int)(Math.sqrt(number));
for (int divisor = 2; divisor <= squareRoot; divisor++) {
    if (number % divisor == 0) {
        // 如果 number 能被 divisor 整除，则 number 不是素数
        isPrime = false;
        break; // 退出循环
    }
}
```

实际上，不需要为每个数字单独计算 Math.sqrt(number)。这是因为在一定范围内，很多数字的平方根是相同的。
例如，对于所有在36和48之间的数字（包括36和48），它们的 (int)(Math.sqrt(number)) 都是6。
我们只需要检查完全平方数，如$4, 9, 16, 25, 36, 49$等，因为这些数的平方根是整数。
这意味着我们只需要计算这些完全平方数的平方根一次，然后在循环中使用这个值。
改进代码如下。

```
int squareRoot = 1;
// Repeatedly find prime numbers
while (number <= n) {
    // Assume the number is prime
    boolean isPrime = true; // Is the current number prime?
    if (squareRoot * squareRoot < number) squareRoot++;
    // Test if number is prime
    for (int divisor = 2; divisor <= squareRoot; divisor++) {
        if (number % divisor == 0) { // If true, number is not prime
            isPrime = false; // Set isPrime to false
            break; // Exit the for loop
        }
    }
    // Print the prime number and increase the count
    if (isPrime) {
        count++; // Increase the count
        if (count % NUMBER_PER_LINE == 0) {
            // Print the number and advance to the new line
            System.out.printf("%7d\n", number);
        } else {
            System.out.printf("%7d ", number);
         }
     }
    // Check if the next number is prime
    number++;
}
System.out.println("\n" + count + " prime(s) less than or equal to " + n);
```

对于每个数字i，算法需要在循环中花费$\sqrt{i}$步来检查i是否是素数。
为了找出所有小于或等于n的素数，算法需要对每个数字i（从2到n）执行上述步骤。
因此，总的步骤数是$\sqrt{2} + \sqrt{3} + \sqrt{4} + \ldots + \sqrt{n}$。
这个总和可以被简化为小于或等于$n\sqrt{n}$的形式，因为每个项$\sqrt{i}$都小于或等于
$\sqrt{n}$，并且有n个这样的项。
因此时间复杂度是$O(n\sqrt{n})$。

我们现在看另一种方法。
如果i不是素数，那么存在一个素数p使得$i = pq$且$p \leq q$。
$\pi(i)$表示小于或等于i的素数的数量。
例如，$\pi(2) = 1$（2是素数），$\pi(3) = 2$（2和3是素数），$\pi(6) = 3$（2,3,5是素数），$\pi(20) = 8$（小于或等于20的素数有8个）。
已证明$\pi(i)$大约等于$i/log(i)$。
小于或等于$\sqrt{i}$的素数数量是 π($\sqrt{i}$)。

$$\frac{\sqrt{i}}{\log\sqrt{i}} = \frac{2\sqrt{i}}{\log i}$$

此外，素数在数列中相对均匀分布。

寻找所有小于或等于n的素数的复杂度可以表示为：

$$\frac{2\sqrt{2}}{\log 2}+\frac{2\sqrt{3}}{\log 3}+\frac{2\sqrt{4}}{\log 4}+\frac{2\sqrt{5}}{\log 5}+\frac{2\sqrt{6}}{\log 6}+\frac{2\sqrt{7}}{\log 7}+\frac{2\sqrt{8}}{\log 8}+\ \ldots\ +\frac{2\sqrt{n}}{\log n}$$

因为：

$$\frac{\sqrt{i}}{\log i}<\frac{\sqrt{n}}{\log n}\ \text{for}\ i<n\ \text{and}\ n\geq 16,$$

所以上述求和可以化简为：

$$\frac{2\sqrt{2}}{\log 2}+\frac{2\sqrt{3}}{\log 3}+\frac{2\sqrt{4}}{\log 4}+\frac{2\sqrt{5}}{\log 5}+\frac{2\sqrt{6}}{\log 6}+\frac{2\sqrt{7}}{\log 7}+\frac{2\sqrt{8}}{\log 8}+\ \ldots\ +\frac{2\sqrt{n}}{\log n}<\frac{2n\sqrt{n}}{\log n}$$

因此，该算法的复杂度是$O(n\sqrt{n}/logn)$。

1.2.3.1 埃拉托斯特筛法（Sieve of Eratosthenes）

埃拉托斯特筛法是一种古老的算法，由古希腊数学家埃拉托斯特（Eratosthenes）在公元前276年至194年之间发明。
算法步骤如下：
使用一个名为 primes 的布尔数组，长度为 n，初始时所有值都设为 true，表示假设所有数都是素数。
从最小的素数2开始，将2的所有倍数（即非素数）在数组中标记为 false。
接着处理下一个未被标记为 false 的数3，将其所有倍数标记为 false。
重复上述过程，直到处理到$\sqrt{n}$。
如图所示。

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
initial	×	×	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$k=2$	×	×	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
$k=3$	×	×	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F
$k=5$	×	×	Ⓣ	Ⓣ	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	Ⓣ	F	F	F	Ⓣ	F	F	F	F

代码如下。

```
import java.util.Scanner;

public class SieveOfEratosthenes {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Find all prime numbers <= n, enter n: ");
        int n = input.nextInt();
        boolean[] primes = new boolean[n + 1]; // Prime number sieve
        for (int i = 0; i < primes.length; i++) {
           primes[i] = true;
       }
       for (int k = 2; k <= n / k; k++) {
           if (primes[k]) {
               for (int i = k; i <= n / k; i++) {
                   primes[k * i] = false; // k * i is not prime
               }
           }
       }
       final int NUMBER_PER_LINE = 10; // Display 10 per line
       int count = 0; // Count the number of prime numbers found so far
       for (int i = 2; i < primes.length; i++) {
           if (primes[i]) {
                count++;
                if (count % 10 == 0)
                    System.out.printf("%7d\n", i);
                else
                    System.out.printf("%7d ", i);
            }
       }
       System.out.println("\n" + count + " prime(s) less than or equal to " + n);
   }
}
```

总的设置次数可以表示为：
$\frac{n}{2}-2+1+\frac{n}{3}-3+1+\frac{n}{5}-5+1+\frac{n}{7}-7+1+\frac{n}{11}-11+1+\ldots$这个序列可以简化为小于$O(n\pi(n))$，其中$\pi(n)$是小于或等于$n$的素数的数量。
进一步简化为$O(n^2/logn)$，因为$\pi(n)$大约等于$n/log(n)$。
这个上界非常松散，实际的时间复杂度要好得多。
该算法在处理较小的n时特别有效，因为它需要的内存空间相对较小。

1.2.4 “最近点对”问题（Closest-pair problem）

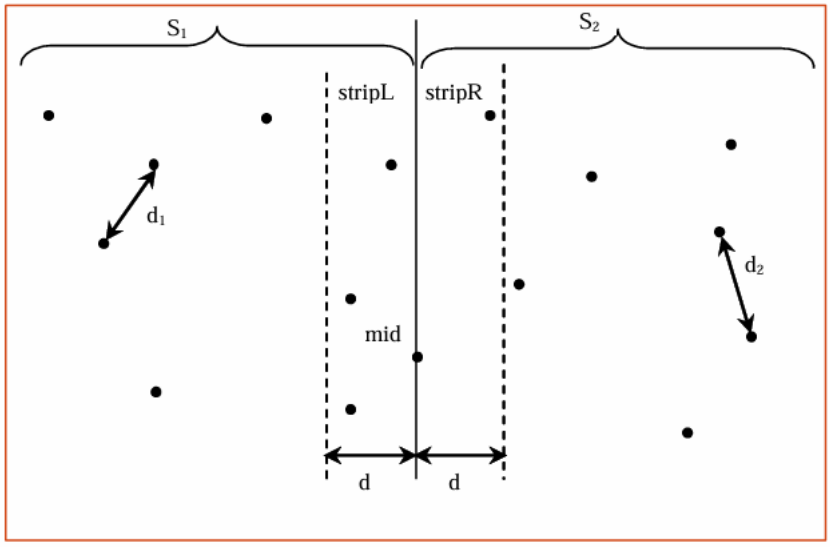
给定一组点，最近点对问题的目标是找出这组点中彼此之间距离最近的两个点。

1. 暴力算法：计算所有点对之间的距离，并找到距离最小的一对点。这种方法的时间复杂度是$O(n^2)$，因为需要计算$n(n-1)/2$次距离（对于n个点）。
2. 分治法：首先，按照x坐标的递增顺序对所有点进行排序。
   递归地将点集分为两半，直到每个子集只包含一个点或两个点。
   对于每个子集，计算子集中点之间的距离。由于子集较小，这一步相对高效。
   在合并子集时，需要检查跨子集的点对，因为最近的点对可能跨越子集边界。

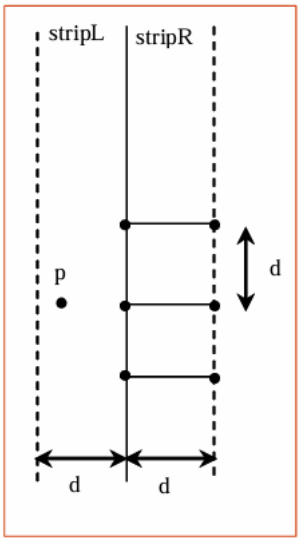
细节如下：

1. 如果存在x坐标相同的点，则按照y坐标进行排序。
   排序后得到一个点的列表S。
2. 然后使用排序列表的中点将列表S分成两个子集S_1和S_2，大小相等。设S_1的中点的中点为 mid。
   接着递归地在S_1和S_2中找到最近的点对。设d_1和d_2分别表示两个子集中最近点对的距离。
   然后计算$d = min(d_1, d_2)$，即两个子集中最近点对距离的最小值。
3. 最后我们还需要检查跨越子集的点对，找到S_1中的一个点和S_2中的一个点，它们的y坐标在范围$[mid_x - d, mid_x + d]$内，计算它们之间的距离，记为d_3。这里的mid_x是S_1中点的x坐标。
   最近点对是距离为$min(d_1, d_2, d_3)$的点对。

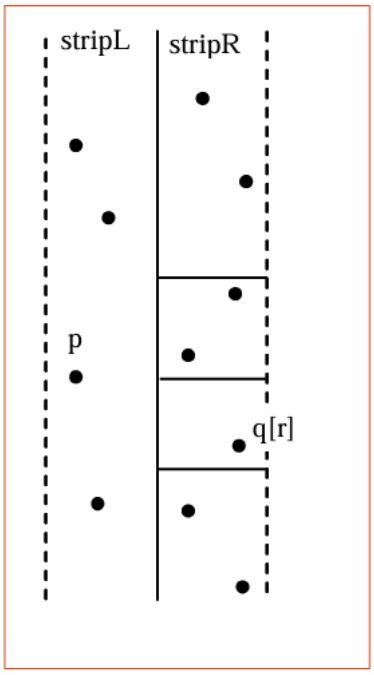
步骤1是排序所以时间复杂度是$O(nlogn)$，步骤3是检查跨越子集的点对所以时间复杂度是$O(n)$
其中$d = min(d_1, d_2)$，其中d_1是S_1中最近点对的距离，d_2是S_2中最近点对的距离。
条带L：包含S_1中所有点的条带，这些点的x坐标小于中点的x坐标减去最小距离d。
条带R：包含S_2中所有点的条带，这些点的x坐标大于中点的x坐标加上最小距离d。



对于每个点p，根据其x坐标与中点mid的x坐标的差值，将其分配到条带L或条带R中。
如果p在集合S_1中且$mid.x - p.x \leq d$，则将p添加到条带L。
如果p在集合S_2中且$p.x - mid.x \leq d$，则将p添加到条带R。



而在条带中找到最近点对的细节如下：
计算两个子集中最近点对的最小距离d。
初始化索引r为0，用于遍历条带R中的点。
遍历条带L中的每个点p，检查条带R中与p的y坐标差不超过d的点$q[r]$。
如果找到更近的点对，则更新最小距离d和当前最近点对。



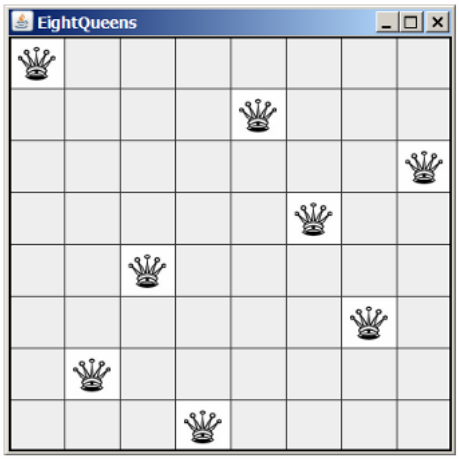
所以这个算法的递归关系为：$T(n) = 2T(n/2) + O(n)$，因此这种算法的时间复杂度是 $O(nlogn)$。
完整代码如下。

```
import java.util.*;

public class ClosestPair {
    // Each row in points represents a point
    private double[][] points;
    Point p1, p2;

    public static void main(String[] args) {
       double[][] points = new double[500][2];
       for (int i = 0; i < points.length; i++) {
           points[i][0] = Math.random() * 100;
           points[i][1] = Math.random() * 100;
       }
       ClosestPair closestPair = new ClosestPair(points);
       System.out.println("shortest distance is " + closestPair.getMinimumDistance());
       System.out.print("(" + closestPair.p1.x + ", " + closestPair.p1.y + ") to ");
       System.out.println("(" + closestPair.p2.x + ", " + closestPair.p2.y + ")");
   }

    public ClosestPair(double[][] points) {
       this.points = points;
    }

    public double getMinimumDistance() {
        Point[] pointsOrderedOnX = new Point[points.length];
        for (int i = 0; i < pointsOrderedOnX.length; i++)
            pointsOrderedOnX[i] = new Point(points[i][0], points[i][1]);
        Arrays.sort(pointsOrderedOnX);
        // Locate the identical points if exists
        if (checkIdentical(pointsOrderedOnX))
            return 0; // The distance between the identical points is 0
        Point[] pointsOnY = pointsOrderedOnX.clone();
        Arrays.sort(pointsOnY);
        return distance(pointsOnX, 0, pointsOnX.length- 1, pointsOnY);
    }

    public boolean checkIdentical(Point[] pointsOrderedOnX) {
        for (int i = 0; i < pointsOrderedX.length-1; i++) {
            if (pointsOrderedX[i].compareTo(pointsOrderedX[i + 1]) == 0) {
                p1 = pointsOrderedX[i];
                p2 = pointsOrderedX[i + 1];
                return true;
            }
        }
        return false;
    }
    /** Compute the distance between two points p1 and p2 */
    public static double distance(Point p1, Point p2) {
        return distance(p1.x, p1.y, p2.x, p2.y);
    }
    /** Compute the distance between two points (x1, y1) and (x2, y2) */
    public static double distance(
         double x1, double y1, double x2, double y2) {
```

```
        return Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
    }
    static class Point implements Comparable<Point> {
        double x;
        double y;
       Point(double x, double y) {
            this.x = x;
            this.y = y;
       }
       public int compareTo(Point p2) {
            if (this.x < p2.x)
                return -1;
            else if (this.x == p2.x) {
                // Secondary order on y-coordinates
                if (this.y < p2.y)
                    return -1;
                else if (this.y == p2.y)
                    return 0;
                else
                    return 1;
            } else return 1;
       }
       public double distance(Point[] pointsOrderedOnX, int low, int high,
                Point[] pointsOrderedOnY) {
            if (low >= high) // Zero or one point in the set
                return Double.MAX_VALUE;
            else if (low + 1 == high) {
                p1 = pointsOrderedOnX[low];
                p2 = pointsOrderedOnX[high];
                return distance(pointsOrderedOnX[low], pointsOnX[high]);
            }
            int mid = (low + high) / 2;
            Point[] pointsOrderedYL = new Point[mid - low + 1];
            Point[] pointsOrderedYR = new Point[high - mid];
            int j1 = 0; int j2 = 0;
            for (int i = 0; i < pointsOnY.length; i++) {
                if (pointsOnY[i].compareTo(pointsOrderedOnX[mid]) <= 0)
                    pointsOrderedYL[j1++] = pointsOnY[i];
                else
                    pointsOrderedYR[j2++] = pointsOnY[i];
            }
            // Recursively find the distance of the closest pair in the left
            // half and the right half
            double d1 = distance(pointsOrderedOnX, low, mid, pointsOrderedYL);
            double d2 = distance(pointsOrderedX, mid + 1, high, pointsOrderedYR);
            double d = Math.min(d1, d2);
            // stripL: the points in pointsOrderedYL within the strip d
            int count = 0;
            for (int i = 0; i < pointsOnYL.length; i++)
                if (pointsOnYL[i].x >= pointsOrderedOnX[mid].x - d)
                    count++;
            Point[] stripL = new Point[count];
            count = 0;
            for (int i = 0; i < pointsOnYL.length; i++)
                if (pointsOnYL[i].x >= pointsOrderedX[mid].x - d)
                   stripL[count++] = pointsOnYL[i];
           // stripR: the points in pointsOnYR within the strip d
            count = 1;
            for (int i = 1; i < pointsOnYR.length; i++)
                if (pointsOnYR[i].x <= pointsOrderedOnX[mid].x + d)
                   stripR[count++] = pointsOnYR[i];
            // Find the closest pair for a point in stripL and
            // a point in stripR
            double d3 = d;
            int j = 0;
            for (int i = 1; i < stripL.length; i++) {
                while (j < stripR.length && stripR[j].y - stripL[i].y <= d)
                    j++;
                // Compare a point in stripL with points in stripR
                int k = j; // Start from r1 up in stripR
                while(k < stripR.length && stripR[k].y <= stripL[i].y + d) {
                    if (d3 > distance(stripL[i], stripR[k])) {
                        d3 = distance(stripL[i], stripR[k]);
                        p1 = stripL[i];
                        p2 = stripR[k];
                     }
                    k++;
                }
            }
            return Math.min(d, d3);
       }
    }
}
```

1.2.5 八皇后问题（Eight Queens Problem）

八皇后问题（Eight Queens Problem）是一个经典的约束满足问题，目标是在8x8的国际象棋棋盘上放置8个皇后，使得没有任何两个皇后可以相互攻击。皇后可以攻击同一行、同一列或对角线上的任何棋子。



递归方法是一种解决八皇后问题的有效方法。
算法步骤如下：

1. 算法从棋盘的第一行（即第0行）开始放置皇后，这里k是当前考虑的行的索引。
2. 对于当前行的每个可能的列（从第0列到第7列），算法依次检查是否可以在该列放置一个皇后。j表示列的索引，范围从0到7。
3. 对于每个可能的列j，算法需要检查在该列放置皇后是否会导致攻击。
4. 如果在某一行成功放置了一个皇后，算法会继续在下一行搜索放置下一个皇后。
5. 如果当前行是最后一行，并且成功放置了皇后，那么找到了一个解决方案。
6. 如果在某一行放置皇后不成功（即当前放置的皇后会攻击到其他皇后），算法会回溯到上一行，并尝试在上一行的下一列放置皇后。
7. 如果算法回溯到第一行，并且无法在这一行找到新的放置位置，那么说明没有解决方案可以找到。
   这种算法被称为回溯法（backtracking）。

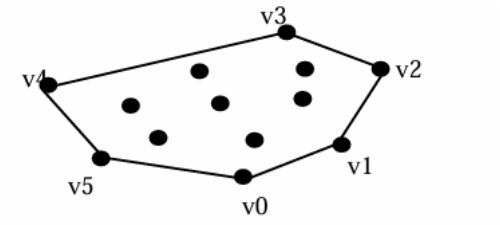
代码如下。

```
public class NQueens {
    static final int SIZE = 8;
    private int[] queens = new int[SIZE];

    public static void main(String[] args) {
       NQueens nq = new NQueens();
       nq.search();
        print(nq.queens);
    }

    private static void print(int[] queens) {
        System.out.print("[");
        for(int q : queens) {
           System.out.print(q + " ");
       }
       System.out.println("]");
    }

    public NQueens() {
    }

    /** Search for a solution */
    private boolean search() {
        // k - 1 indicates the number of queens placed so far
        // We are looking for a position in the kth row to place a queen
        int k = 0;
        while (k >= 0 && k < SIZE) {
            // Find a position to place a queen in the kth row
            int j = findPosition(k);
            if (j < 0) {
                queens[k] = -1;
                k--; // back track to the previous row
            } else {
                queens[k] = j;
                k++;
            }
        }
        if (k == -1)
            return false; // No solution
        else
            return true; // A solution is found
    }

    public int findPosition(int k) {
        int start = queens[k] + 1; // Search for a new placement
```

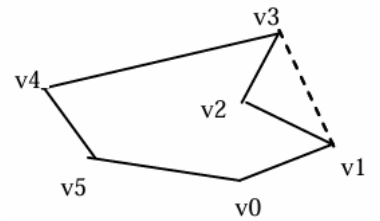
```
        for (int j = start; j < SIZE; j++) {
            if (isValid(k, j))
                break;
        }
        return j; // (k, j) is the place to put the queen now
    }

    /** Return true if a queen can be placed at (row, column) */
    public boolean isValid(int row, int column) {
        for (int i = 1; i <= row; i++)
           if (queens[row - i] == column // Check column
               || queens[row - i] == column - i // Check upleft diagonal
               || queens[row - i] == column + i) // Check upright diagonal
               return false; // There is a conflict
       }
       return true; // No conflict
    }
}
```

1.2.6 凸包（Convex Hull）

给定一组点，凸包是能够包围所有这些点的最小凸多边形（凸多边形）。
凸多边形是指连接多边形的任意两个顶点的直线段都在多边形内部的多边形。
下图是一个凸多边形的例子。



下图是一个非凸多边形的例子。

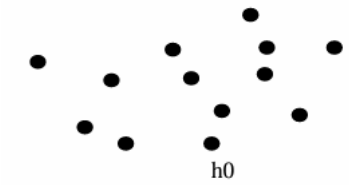


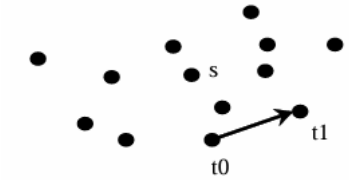
凸包的应用包含模式识别、图像处理、游戏编程。

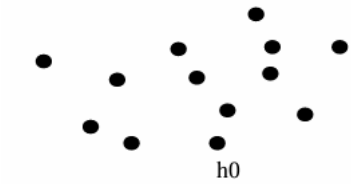
1.2.6.1 礼品包装算法（Gift-Wrapping Algorithm）

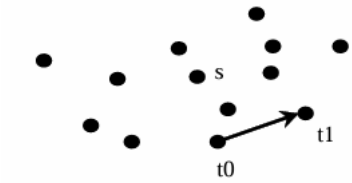
该算法用于计算一组点的凸包（Convex Hull）。
步骤如下：

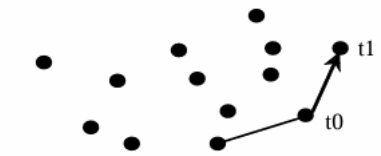
1. 初始化：
   创建一个空列表H，该列表最终将包含构成凸包的所有点。
2. 给定一组点S，将这些点标记为$s_0, s_1, \ldots, s_k$。
   选择最靠右且最低的点h_0。这个点将成为凸包的第一个顶点。
   将h_0添加到列表H。
   设t_0为h_0。
3. 选择一个尚未处理的点t_1。对于集合S中的每个点p，如果p在从t_0到t_1的直线的右侧，则设t_1为p。
   完成此步骤后，集合S中不应再有位于从t_0到t_1的直线右侧的点。
4. 如果t_1是h_0，则算法完成。
5. 设t_0为t_1，然后返回步骤3。

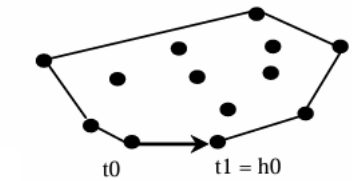












其中判断点相对于直线的方向的方法如下。

- 如果$(x_1 - x_0)\times(y_2 - y_0) - (x_2 - x_0)\times(y_1 - y_0) > 0$，则点$P_2$在直线的左侧。

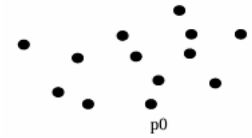
- 如果$(x_1-x_0)\times(y_2-y_0)-(x_2-x_0)\times(y_1-y_0)=0$，则点$P_2$在直线上。
- 如果$(x_1-x_0)\times(y_2-y_0)-(x_2-x_0)\times(y_1-y_0)<0$，则点$P_2$在直线的右侧。

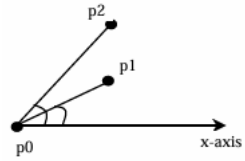
凸包逐步扩展。正确性由步骤3后没有点位于从t_0到t_1的直线右侧这一事实支持。
这确保了S中任意两个点构成的线段都位于多边形内部。
在步骤3中找到最右且最低的点可以在$O(n)$时间内完成。
判断一个点是在直线的左侧、右侧还是线上可以在$O(1)$时间内确定。
步骤3被重复执行h次，其中h是凸包的大小（即凸包中点的数量）。
在步骤3中，算法遍历集合S中的每个点。
因此，算法的总时间复杂度是$O(hn)$，其中n是输入点的数量。
在最坏情况下，h等于n（即所有点都在凸包上）。
因此，该算法的最坏情况下的时间复杂度是$O(n^2)$。

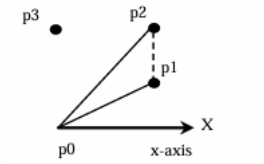
1.2.6.2 Graham 扫描算法

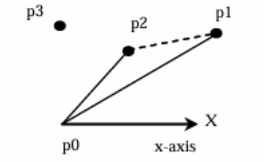
步骤如下：

1. 从点集S中选择最靠右且最低的点p_0，并将其命名为p_0。
2. 将点集S中的点按与p_0连线的角度进行排序，将p_0作为中心点。
   如果有两个点与p_0的角度相同，则丢弃距离p_0更远的点。
   排序后的点集S为$p_0,p_1,p_2,\ldots,p_{n-1}$。
3. 将p_0,p_1,p_2推入栈H。
4. 初始化$i=3$。
   当$i<n$时，执行以下操作：
   令t_1和t_2为栈H中的前两个元素。
   如果p_i在从t_2到t_1的直线左侧，则将p_i推入栈H并将i加1。
   否则，从栈H中弹出顶部元素。
5. 栈H中的点形成凸包。









步骤1可以在$O(n)$时间内完成，步骤2排序可以在$O(nlogn)$时间内完成，步骤3可以在$O(1)$时间内完成，步骤4可以在$O(n)$时间内完成。
因此Graham 扫描算法的总时间复杂度为$O(nlogn)$。

2. 练习

2.1 基础练习

1.大O符号的主要目的是什么?
A.测量实际执行时间。
B.估计最坏情况下的内存使用量。
C.描述算法随着输入规模增加的增长速率。
D.计算在一台机器上的平均执行时间。

答案是C。

2.在使用大O符号时，哪些因素可以被忽略?
A.输入规模。
B.主导项。
C.递归调用次数。

D.常数乘数和低阶项。

答案是D。

3.哪种算法具有对数时间复杂度?
A.Linear search（线性搜索）。
B.Binary search（二分搜索）。
C.Insertion sort（插入排序）。
D.Selection sort（选择排序）。

答案是B。

4.递归斐波那契算法与其动态规划版本的性能对比?
A.它们具有相同的效率。
B.递归归版本更快。
C.动态规划避免重复计算并且更快。
D.递归归版本使用更少的内存并且更好。

答案是C。

5.下列代码的时间复杂度是多少?

```
for (int i = 0; i < n; i++) {
    System.out.print('*');
}
```

答案是$O(n)$。

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        System.out.print('*');
    }
}
```

答案是$O(n^2)$。

```
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.print('*');
        }
    }
}
```

答案是$O(n^3)$。

```
for (int k = 0; k < 10; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.print('*');
        }
    }
}
```

答案是$O(n^2)$。

5.下列代码最糟糕情况下的时间复杂度是多少?

```
for (int i = 1; i < list.length; i++) {
    if (list[i] > list[i + 1]) {
        // 交换 list[i] 和 list[i + 1]
        int temp = list[i];
        list[i] = list[i + 1];
        list[i + 1] = temp;
    }
}
```

答案是$O(n^2)$，这是一个冒泡排序。

2.2 进阶练习

如何使用分治法（Divide-and-Conquer）来找出列表中的最大数?
示例代码如下。

```
public class DivideAndConquerMax {
    public static int findMax(int[] list) {
        return findMaxRecursive(list, 0, list.length - 1);
    }

    private static int findMaxRecursive(int[] list, int left, int right) {
        if (left == right) {
```

```
            return list[left]; // 基本情况：只有一个元素
        } else if (left + 1 == right) {
            return Math.max(list[left], list[right]); // 两个元素，直接比较
        } else {
            int mid = (left + right) / 2;
            int maxLeft = findMaxRecursive(list, left, mid);
            int maxRight = findMaxRecursive(list, mid + 1, right);
            return Math.max(maxLeft, maxRight); // 比较左右两部分的最大值
        }
    }

    public static void main(String[] args) {
        int[] list = {178, 33, 4, 2, -3, 5};
        System.out.println("The largest number is: " + findMax(list));
    }
}
```