算法设计与分析

`Author`： W. J. H

`Edit time`：2024.6.24

`FiIe content`： Algorithm Design and Analysis

现在是2024.6.25凌晨1：10，从昨天下午四点半开始整理到现在，该总结的基本都覆盖到了，如果没时间过完这个复习资料再去看一下动态规划和贪心的ppt，这两部分内容太多我就不总结了，只举了一个例子，分支限界法和回溯法也去看一下ppt部分怎么画树，27000字的算法复习资料，希望对大家有用！

一、算法概述

1. 算法的概念：（以下定义均可以，首选第一条）
    - 在有限时间内，对问题求解的一个清晰的指令序列。算法的每个输入确定了该算法求解问题的一个实例
    - 算法是求解问题的一系列计算步骤，用来讲输入数据转换成输出结果
    - 基本运算及规定运算顺序所构成的完整的解题步骤
    - 按要求有限且确切的计算序列
    - 算法是一组用于解决问题或执行特定任务的明确指令。它是一个系统化的步骤序列，用于输入数据，并通过一系列已定义的步骤处理这些数据，最终产生输出或达到预期的结果。
2. 算法的特性：
    - 有穷性$(Finiteness)$：算法必须在有限的步骤内结束，不能出现无限循环或者无线递归等情况。也就是说算法必须在有限时间内能够停止
    - 确定性$(Definiteness)$：算法的每一个步骤都必须有确切的含义，不允许有歧义或者二义性。也就是说算法每一步都必须能够被清晰地定义或理解
    - 可行性$(FeasibiIity)$：算法中的每一步都必须是基本的、足够简单，能够通过已有的基本操作实现执行，并且对于所有输入都能得到正确的结果
    - 输入$(Input)$：算法有零个或多个输入数据，输入数据可以是任何形式。也就是说算法必须能够接收输入数据并进行处理
    - 输出$(Output)$：算法必须有一个或多个输出结果，输出结果可以是任何形式。也就是说算法必须能够生成输出结果并返回给用户
3. 伪代码：
    - 自然语言和类似编程语言结构的混合体
    - 本课程中伪代码：
        - 省略变量声明
        - 使用缩进来显示for、 if、 $whiIe$等语句的范围
        - 使用" $\leftarrow$ "进行分配
    - 伪代码书写规则
        - **变量赋值**：
            - `X ← vaIue` 或 `X = vaIue`：将值赋给变量X。
        - **条件语句**：
            - `IF condition THEN`：如果满足某条件，则执行随后的语句。
            - `EISE`：如果前面的`IF`条件不满足，则执行`EISE`后的语句。
            - `ENDIF`：标示`IF`语句的结束。
        - **循环结构**：
            - `FOR variabIe FROM start TO end DO`：循环变量从起始值到结束值。
            - `WHIIE condition DO`：当条件满足时，持续执行循环体。
            - `REPEAT UNTII condition`：重复执行，直到满足某个条件。
            - `ENDFOR`、`ENDWHIIE`：标示循环的结束。

- **数组和数据结构的操作**：
  - `A[i]`：表示数组A的第i个元素。
  - `Iist.Append(item)`：将项添加到列表末尾。
- **输入输出**：
  - `READ variabIe`：读取值到变量。
  - `PRINT variabIe` 或 `WRITE variabIe`：输出变量的值。
- **函数和过程**：
  - `FUNCTION FuncName(parameters)`：定义一个函数。
  - `RETURN vaIue`：从函数返回一个值。
  - `CAII FuncName(arguments)`：调用一个函数或过程。
  - `END FUNCTION`：标示函数的结束。
- **注释**：
  - `// This is a comment` 或 `# This is a comment`：添加注释来说明代码的某部分功能。
- **逻辑运算**：
  - `AND`、`OR`、`NOT`：逻辑与、或、非运算符。

- 一段标准的伪代码示例：

```
AIgorithm ConsecutiveInteger(m, n)
    // 使用连续整数检测法计算gcd(m，n)
    // 输入：两个不全为0的非负整数m, n
    // 输出：m, n的最大公约数
    if n = 0 then
        return m
    end if
    t = min{m, n}
    whiIe t > 0 do
        if (m mod t) == 0 then
            if (n mod t) == 0 then
                return t
            eIse
                t = t - 1
            end if
        eIse
            t = t - 1
        end if
    end whiIe
    return t
```

4. 算法操作的是数据，因此数据结构很重要

2.算法效率分析

1. 算法分析是指研究算法在运行时间和内存空间等资源方面的效率
    - 与输入大小、输入类型和算法功能有关
2. 算法分析框架
    - 测量运行时间
    - 测量输入大小
    - 算法效率函数的增长阶数
    - 最差、最好、平均效率
3. 寻找算法的基本操作：基本操作通常是算法最内层循环中最耗时的操作
4. 算法的时间复杂度：取决于具体输入
    - 下面式子中 I 是问题规模 n 的实例，$p(I)$是实例 I 出现的频率
    - 最坏情况下时间复杂度：

      $T_{max}(n)=max\{T(I)|size(I)=n\}$
    - 最好情况下时间复杂度：

      $T_{min}(n)=min\{T(I)|size(I)=n\}$
    - 平均情况下的时间复杂度：

      $T_{avg}(n)=\sum_{size(i)=n}(p(I)\times T(I))$

- eg :顺序查找的时间复杂度
    - 最坏情况下：$T_{max}(n)=max\{T(I)|size(I)=n\}=O(n)$
    - 最好情况下：$T_{min}(n)=min\{T(I)|size(I)=n\}=O(1)$
    - 平均情况下：搜索成功概率为 $p(0\leq p\leq 1)$，在数组每个位置$i(0\leq i\leq n)$搜索成功概率相同，均为$\frac{p}{n}$

$$
\begin{aligned}
T_{avg}(n)&=\sum_{size(i)=n}(p(I)\times T(I))\\
&=(1\times\frac{p}{n}+2\times\frac{p}{n}+3\times\frac{p}{n}+\cdots+n\times\frac{p}{n})\\
&=\frac{p}{n}\sum_{i=1}^{n}i+n(1-p)\\
&=\frac{p(n+1)}{2}+n(1-p)
\end{aligned}
$$

5. 算法的渐近时间复杂度：当$n\to\infty,T(n)\to\infty,\frac{T(n)-t(n)}{T(n)}\to 0$
    - $t(n)$是$T(n)$的渐近形态，为算法的渐近复杂度
    - 一句话总结就是，抓$T(n)$多项式的大头

我愿称下面这个这个地方是最恶心的地方，这几个符号搞不清楚

6. 渐近分析的记号 $\theta\ O\ \Omega\ o\ \omega$

   先看结论：a 代表$f(n)$，b 代表$g(n)$

$$f(n)= O(g(n)) \approx a \le b;$$

$$f(n)= \Omega(g(n)) \approx a \ge b;$$

$$f(n)= \Theta(g(n)) \approx a = b;$$

$$f(n)= o(g(n)) \approx a < b;$$

$$f(n)= \omega(g(n)) \approx a > b.$$

- 渐近上界记号 O

（1）渐近上界记号O

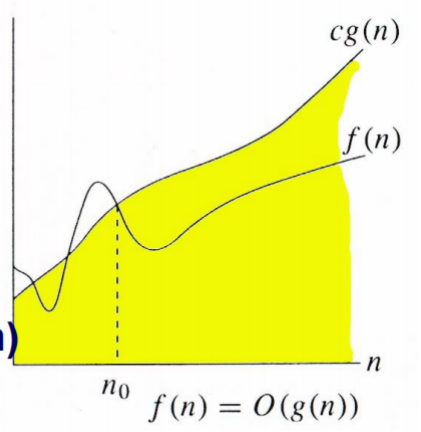
$O(g(n))$ = { $f(n)$ | 存在正常数c和n_0使得对所有$n \ge n_0$有：

$0 \le f(n) \le cg(n)$ }

$2n + 3 = O(n^2)$.

$3n^3=O(n^4)$

O(g(n))是增长次数小于或等于g(n)的函数集合



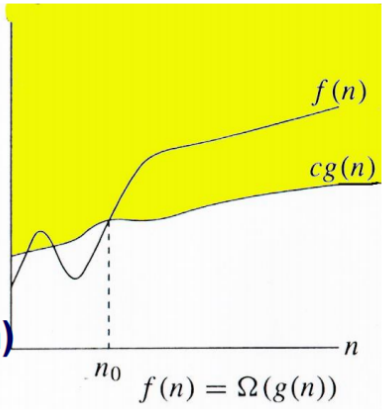
- 渐近下界记号 Ω

（2）渐近下界记号Ω

$\Omega(g(n))$ = { $f(n)$ | 存在正常数c和n_0使得对所有$n \ge n_0$有：$0 \le cg(n) \le f(n)$ }

$3n^3= \Omega(n^2)$

Ω(g(n))是增长次数大于或等于g(n)的函数集合



- 紧渐进界记号 θ

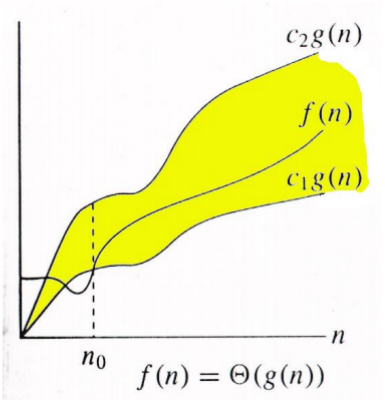
（3）紧渐近界记号Θ

$\Theta(g(n)) = \{ f(n) \mid$ 存在正常数c_1, c_2和n_0使得对所有$n \geq n_0$有：$c_1 g(n) \leq f(n) \leq c_2 g(n) \}$

定理1： $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

$$0.5n(n-1) = \Theta(n^2)$$

Θ(g(n))是增长次数等于g(n)的函数集合



- 非紧上界记号 o 和非紧下界记号 ω，和上面的区别就是能不能在下面红框框中取等

（4）非紧上界记号o

$o(g(n)) = \{ f(n) \mid$ 对于任何正常数$c>0$，存在正数和$n_0 > 0$ 使得对所有$n \geq n_0$有：$0 \leq f(n) < cg(n) \}$

等价于 $f(n) / g(n) \to 0$， as $n \to \infty$。

（5）非紧下界记号ω

$\omega(g(n)) = \{ f(n) \mid$ 对于任何正常数$c>0$，存在正数和$n_0 > 0$ 使得对所有$n \geq n_0$有：$0 \leq cg(n) < f(n) \}$

等价于 $f(n) / g(n) \to \infty$， as $n \to \infty$。

$f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$

- *eg* :排序算法的时间复杂度分析举例

Big-O 实例

$n^2 + 100\,n = O(n^2)$

$(n^2 + 100\,n) \le 2\,n^2$ for $n \ge 10$

$n^2 + 100\,n = \Omega(n^2)$

$n^2 + 100\,n) \ge 1\,n^2$ for $n \ge 0$

$n^2 + 100\,n = \Theta(n^2)$

- **插入排序在最坏的情况下需要** $\Theta(n^2)$，所以排序是 $O(n^2)$

$n \log n = O(n^2)$

$n \log n = \Theta(n \log n)$

$n \log n = \Omega(n)$

- 任意的排序算法都需要查看每个元素，所以排序是$\Omega(n)$.
- 实际上，合并排序在最坏的情况下是$\Theta(n\log n)$

7. 渐近分析记号的性质

- 传递性
- 反身性
- 对称性
- 互对称性
- 算术运算

（1）传递性：

$f(n)= \Theta(g(n))$，$g(n)= \Theta(h(n)) \Rightarrow f(n)= \Theta(h(n))$；

$f(n)= O(g(n))$，$g(n)= O(h(n)) \Rightarrow f(n)= O(h(n))$；

$f(n)= \Omega(g(n))$，$g(n)= \Omega(h(n)) \Rightarrow f(n)= \Omega(h(n))$；

$f(n)= o(g(n))$，$g(n)= o(h(n)) \Rightarrow f(n)= o(h(n))$；

$f(n)= \omega(g(n))$，$g(n)= \omega(h(n)) \Rightarrow f(n)= \omega(h(n))$；

（2）反身性：

$f(n)= \Theta(f(n))$；

$f(n)= O(f(n))$；

$f(n)= \Omega(f(n))$.

（3）对称性：

$f(n)= \Theta(g(n)) \Leftrightarrow g(n)= \Theta(f(n))$.

（4）互对称性：

$f(n)= O(g(n)) \Leftrightarrow g(n)= \Omega(f(n))$ ；

$f(n)= o(g(n)) \Leftrightarrow g(n)= \omega(f(n))$ ；

（5）算术运算：

$O(f(n))+O(g(n)) = O(\max\{f(n),g(n)\})$ ；

$O(f(n))+O(g(n)) = O(f(n)+g(n))$ ；

$O(f(n))*O(g(n)) = O(f(n)*g(n))$ ；

$O(cf(n)) = O(f(n))$ ；

$g(n)= O(f(n)) \Rightarrow O(f(n))+O(g(n)) = O(f(n))$ 。

8. 几种常见函数渐近时间复杂度比较

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

9. 一些技巧求解渐进时间复杂度

- 给出你关于$T(n)$的一个表达式，按照数列通项公式求解，可以仿写观察，也可以直接用数列的方法做
- 对于非递归算法
  - $for/while$ 循环

    循环体内计算时间×循环次数
  - 嵌套循环

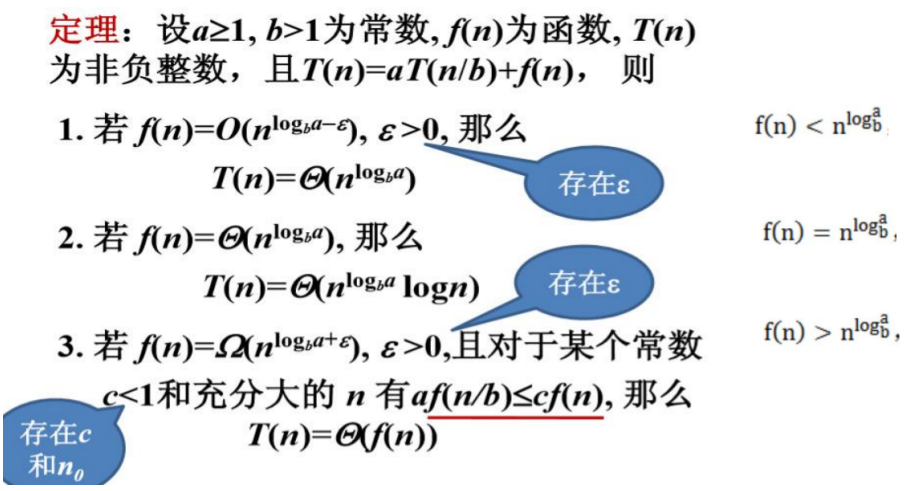
    循环体内计算时间×所有循环次数
  - 顺序语句

    各语句计算时间相加
  - $if-else$语句

    if语句计算时间和$else$语句计算时间的较大者
- 对于递归算法：主定理

10. 主定理

- 定理内容



- 简单记法：用$f(n)$和$n^{\log_b a}$比较
  - 若$n^{\log_b a}$大，则$T(n)=\theta(n^{log_b a})$
  - 若$f(n)$大，则$T(n)=\theta(f(n))$
  - 若二者同阶，则$T(n)=\theta(n^{log_b a}\log n)=\theta(f(n)\log n)$

注：主定理虽然是一个强大的工具，但它并不适用于所有类型的递归关系式。它不能应用于以下几种情况：

1. **非多项式比较**：如果 $f(n)$ 不是多项式函数与 $n^{\log_b a}$ 的比较不能直接归入主定理的三种情况，例如，当 $f(n)$ 是非多项式增长（比如指数增长）时。
2. **非标准形式**：如果递归关系不完全符合 $T(n)=aT\left(\frac{n}{b}\right)+f(n)$ 的形式，如递归中 n 未均匀分割或 a、b 不是常数。
3. **复杂的递归过程**：涉及多个递归调用的不同类型或复杂度，或者调用参数以非标准方式变化。

示例：阿克曼函数的递归

一个典型的例子是阿克曼函数，这是一个著名的非初等递归函数，定义如下：

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

这个函数的递归形式涉及两个变量 m 和 n，且递归方式非常特殊，因为函数的递归调用在其中一种情况下是自身的结果。这使得递归的深度和复杂度与常见的 $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ 形式不匹配，因而无法直接使用主定理来分析其时间复杂度。

示例：非均匀分割的递归

另一个不能使用主定理的例子是递归关系式包含非均匀分割的情况，比如：

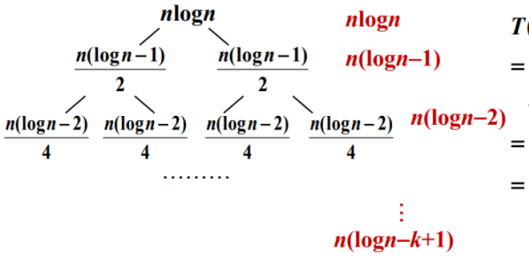
$$T(n) = T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right) + n$$

这里，问题被分为大小不一的两部分，分别为 $\frac{2n}{3}$ 和 $\frac{n}{3}$，而主定理假设每个子问题相对于原问题都是均匀缩小的。此外，这个递归关系包含两个递归项，而主定理仅适用于单一递归项的情况。

示例：解决非多项式的$f(n)$问题

$T(n)=2T(n/2)+n\log n$

递归树求解



求和

$$\begin{aligned} T(n) &= n\log n + n(\log n - 1) + n(\log n - 2) \\ &\quad + \ldots + n(\log n - k + 1) \\ &= (n\log n)\log n - n(1 + 2 + \ldots + k - 1) \\ &= n\log^2 n - nk(k-1)/2 = O(n\log^2 n) \end{aligned}$$

三、蛮力法

这个地方就不重点说了，蛮力法一般不怎么考，只是为算法改进提供依据

1. 蛮力法是一种简单直接地解决问题的方法，通常直接基于问题的描述和所设计的概念定义
2. 蛮力法优点：广泛适用性和简单性
3. 蛮力法缺点：大多数效率都很低
4. 穷举法是蛮力法之一，包括选择排序、冒泡排序、穷举搜索（旅行商问题、分配问题）、背包问题

四、递归算法

这个地方计科的课件是和分治放在一起讲的，软工课件是单列的，考虑受众还是单列一下吧

1. 首先区分递归和迭代的区别：（我一般是判断是不是要调用自己来判断递归还是迭代）
    - 基本定义
        - **递归**：递归是一种通过函数或算法调用自身来解决问题的方法。递归函数包含一个或多个基准条件（递归结束条件），以及一种通过调用自身来缩小问题规模的策略。
        - **迭代**：迭代是使用循环结构重复执行相同的代码块来解决问题的方法。迭代依赖于循环控制结构（如 `for`、`while` 循环）来重复执行操作，直到满足某个终止条件。
    - 内存消耗
        - **递归**：递归通常会消耗更多的内存，因为每次函数调用都需要在调用栈上保存信息（如局部变量、参数和返回地址）。大量的递归调用可能导致堆栈溢出。
        - **迭代**：迭代在内存使用上通常更高效，因为整个过程只需要存储循环的状态，不需要额外的栈空间。
    - 执行效率
        - **递归**：递归的执行效率可能低于迭代，特别是当递归深度很大时，每次函数调用的开销（包括时间和空间）都可能影响总体性能。
        - **迭代**：迭代通常执行效率更高，因为它直接使用循环结构，没有额外的调用栈开销。
2. 直接或间接调用自身的算法成为递归函数，用函数自身给出定义的函数称为递归函数，详情看例子

```
// 递归函数计算阶乘
long long factorialRecursive(int n)
{
    if (n <= 1)
        return 1; // 基准情况：0! = 1 和 1! = 1
    else
        return n * factorialRecursive(n - 1); // 递归调用
}
// 迭代函数计算阶乘
long long factorialIterative(int n)
{
    long long result = 1;
    for (int i = 2; i <= n; i++)
        result *= i; // 累乘每个数字
    return result;
}
```

3. 递归两个要素条件：终止条件、递归方程
4. 如果一个递归算法会不止一次调用自身，出于分析的目的，构造一棵递归调用树，然后计算树中节点数可以得到调用的全部次数
5. 递归的类型：减一、减常数因子、减不固定因子
6. 实例：斐波那契数列、阿克曼函数、汉诺塔、排列问题
7. 注意：递归的简单形式会掩盖其低效性

递归的类型举例：

1. 减一递归（Decrease by One）

在这种类型的递归中，问题的规模在每次递归调用时仅减少一个单位。这种递归简单直观，常见于一些基础的算法问题，如计算阶乘、遍历线性数据结构（例如数组或链表）。

例子：计算阶乘

```
int factorial(int n) {
    if (n == 0) return 1;  // 基本情况
    return n * factorial(n - 1);  // 减一递归
}
```

2. 减常数因子递归（Decrease by a Constant Factor）

这种递归方式中，每次递归调用都将问题的规模减少一个常数因子，例如把问题规模减半。这类递归通常效率较高，因为它能迅速减少问题的大小，常见于二分搜索或快速排序算法。

例子：二分搜索

```
int binarySearch(int arr[], int l, int r, int x) {
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x) return mid;
        if (arr[mid] > x) return binarySearch(arr, l, mid - 1, x);
        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;  // 元素不在数组中
}
```

3. 减不固定因子递归（Decrease by a Variable Factor）

在这种递归中，每次递归调用减少的问题规模不是一个固定的值，而是依赖于具体的情况或输入数据。这类递归的分析相对复杂，可能导致算法性能的波动。常见于某些动态规划问题或回溯算法。

例子：汉诺塔问题

```
void hanoi(int n, char from_rod, char to_rod, char aux_rod) {
    if (n == 1) {
        cout << "Move disk 1 from rod " << from_rod << " to rod " <<
to_rod<<endl;
        return;
    }
    hanoi(n - 1, from_rod, aux_rod, to_rod);
    cout << "Move disk " << n << " from rod " << from_rod << " to rod " << to_rod
<< endl;
    hanoi(n - 1, aux_rod, to_rod, from_rod);
}
```

五、分治法

1. 分治法三个步骤：
    - 将问题划分为两个或多个更小的子问题
    - 递归地解决这些子问题
    - 将子问题的解合并成为原问题的解
2. 分治法要求：所有问题都是相同类型、相互独立、规模均衡
3. 主定理
4. 实例：

实例一、大整数乘法

基本思路

卡拉筐算法将每个 n位数分解为两部分，通过三次乘法来计算乘积，而非传统的四次。每次递归调用处理的数字长度大约减半。具体来说，算法过程是：

1. 分割每个数成两半：$X = 10^{n/2} \cdot a + b$ 和 $Y = 10^{n/2} \cdot c + d$
2. 递归计算 ac，bd，以及 $(a + b)(c + d)$
3. 使用这些结果来构造最终的乘积：$XY = 10^n \cdot ac + 10^{n/2} \cdot ((a + b)(c + d) - ac - bd) + bd$

时间复杂度分析

设 $T(n)$ 表示乘法两个 n-位数字的时间复杂度。根据卡拉筐的方法，可以得到递归关系式：

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

这里的 $O(n)$ 表示加法和数位的移动操作。由于每一级递归中涉及的加法操作和数位移动操作的总时间与 n 成线性关系，因此是 $O(n)$。

使用主定理解递归关系式

根据主定理，对于递归关系式 $T(n) = aT\left(\frac{n}{b}\right) + f(n)$，我们可以分析 $T(n)$ 的渐进行为：

- $a = 3$：每一次递归调用包括三个更小的问题。
- $b = 2$：问题的大小每次减半。
- $f(n) = O(n)$

这里，我们需要比较 $f(n)$ 与 $n^{\log_b a}$ 的关系：

$$n^{\log_b a} = n^{\log_2 3} \approx n^{1.585}$$

因为 $f(n) = O(n)$ 是多项式地小于 $n^{\log_2 3}$，按照主定理的情况 1（如果 $f(n) = O(n^c)$ 其中 $c < \log_b a$），我们有：

$$T(n) = \Theta(n^{\log_2 3})$$

所以，卡拉筐算法的渐进时间复杂度是 $O(n^{1.585})$，这比传统的 $O(n^2)$ 快得多。

注意事项：

☆ X = a* $2^{n/2}$+b; Y = c* $2^{n/2}$+d;
☆X*Y=(a* $2^{n/2}$+b)*(c* $2^{n/2}$+d) =ac * 2^n +(a*d+b*c) * $2^{n/2}$+bd
☆因为 a*d+b*c = (a+c)(b+d)-ac-bd
☆所以 XY = ac 2^n + ((a+c)(b+d)-ac-bd) $2^{n/2}$ + bd

☆ 两个XY的复杂度都是O(n^{log3})，但考虑到a+c,b+d可能得到m+1位的结果，使问题的规模变大，故不选择第2种方案。

实例二、矩阵乘法的Strassen算法

基本思想

标准的矩阵乘法算法需要 $O(n^3)$ 时间复杂度，因为它需要对每个元素进行独立的乘法运算。Strassen算法通过重新组织计算步骤，将两个 $n \times n$ 矩阵的乘法运算分解为较小的子问题，并使用了7次乘法（而不是8次），从而减少了乘法的次数。

算法步骤

对于两个 $n \times n$ 矩阵 A 和 B，假设它们可以被划分为如下形式的子矩阵：

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

其中，每个 A_{ij} 和 B_{ij} 都是 $n/2 \times n/2$ 的矩阵。矩阵 $C = AB$ 可以分解为：

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

其中：

- $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
- $C_{12} = A_{11}B_{12} + A_{12}B_{22}$
- $C_{21} = A_{21}B_{11} + A_{22}B_{21}$
- $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

通过定义以下中间矩阵 M_i：

1. $M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
2. $M_2 = (A_{21} + A_{22})B_{11}$
3. $M_3 = A_{11}(B_{12} - B_{22})$
4. $M_4 = A_{22}(B_{21} - B_{11})$
5. $M_5 = (A_{11} + A_{12})B_{22}$
6. $M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
7. $M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

矩阵 C 的子矩阵可以通过 M_i 快速计算：

- $C_{11} = M_1 + M_4 - M_5 + M_7$
- $C_{12} = M_3 + M_5$
- $C_{21} = M_2 + M_4$
- $C_{22} = M_1 - M_2 + M_3 + M_6$

时间复杂度分析

我们通过递归关系式来详细分析。Strassen算法通过每次递归将矩阵划分为更小的 $n/2 \times n/2$ 的子矩阵，并在每一级递归中执行7次乘法。这个分解继续进行，直到达到一定的基本大小，此时可以直接使用常规矩阵乘法或者达到递归基。

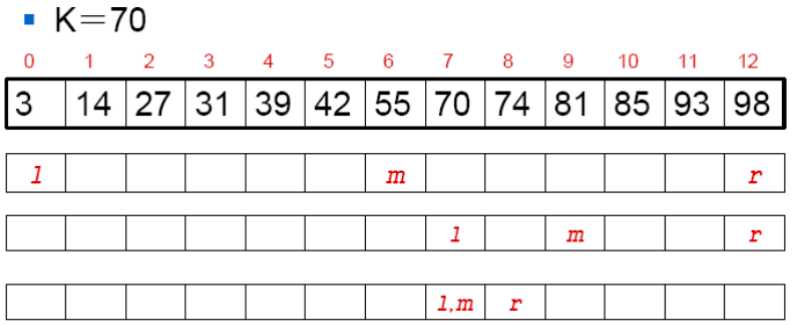
- 递归关系式
  - 设 $T(n)$ 为乘两个 $n \times n$ 矩阵所需的时间，则Strassen算法的递归关系式可以表示为：$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$

    这里的 $O(n^2)$ 项代表将矩阵分割和重组时的加法操作和子矩阵的设置。在每一级递归中，我们需要处理多个这样的加法操作，其总代价与 n^2 成比例。
- 使用主定理计算时间复杂度
  - 根据主定理，对于递归关系式 $T(n) = aT\left(\frac{n}{b}\right) + f(n)$：
    - $a = 7$：表示每次递归生成7个子问题。
    - $b = 2$：表示子问题的大小是原问题的一半。
    - $f(n) = n^2$：这是与划分和合并矩阵有关的成本，属于问题规模的平方。
  - 要应用主定理，需要分析 $f(n)$ 与 $n^{\log_b a}$ 的比较：$n^{\log_b a} = n^{\log_2 7} \approx n^{2.807}$
    - 这里，$f(n) = n^2$ 在多项式意义上小于 $n^{2.807}$。根据主定理的情况1（如果 $f(n) = O(n^c)$ 其中 $c < \log_b a$），我们有：$T(n) = \Theta(n^{\log_2 7})$
- 因此，Strassen算法的渐进时间复杂度为 $O(n^{2.807})$，这比传统矩阵乘法的 $O(n^3)$ 更优。
- 矩阵如何分解：
  - 对于两个 2×2 矩阵 A 和 B：

    $$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$
  - Strassen 提出计算以下七个表达式，每个表达式都是输入矩阵的子矩阵的和或差的乘积：
    - $M_1 = (a + d)(e + h)$ —— 计算 A 的对角元素和 B 的对角元素的乘积。
    - $M_2 = (c + d)e$ —— 计算 A 的第二列元素和 B 的第一列第一个元素的乘积。
    - $M_3 = a(f - h)$ —— 利用 B 的第一行和第二行元素的差。
    - $M_4 = d(g - e)$ —— 利用 B 的第二列元素的差。
    - $M_5 = (a + b)h$ —— 计算 A 的第一行元素和 B 的最后一个元素的乘积。
    - $M_6 = (c - a)(e + f)$ —— 利用 A 和 B 的元素差和和。
    - $M_7 = (b - d)(g + h)$ —— 同样是利用差和和的组合。

实例三、二分搜索

举例：



伪代码：

```
ALGORITHM BinarySearch(A[0..n-1], K)
l ← 0;  r ← n-1
while l ≤ r do                    // l and r crosses over→ can't find K
    m ← ⌊(l+r)/2⌋
    if  K = A[m]  return m        //the key is found
    else if K < A[m]  r ← m-1     //the key is on the left half of the array
         else l ← m+1             // the key is on the right half of the array
return -1
```

二分搜索算法的步骤

1. **初始设置**：设置两个指针，分别指向数组的起始位置和结束位置（`low = 0`，`high = array.length - 1`）。
2. **循环操作**：当 `low` 指针小于或等于 `high` 指针时，执行以下操作：
   - 计算中间位置的索引 `mid = low + (high - low) / 2`。这种计算方式可以防止大数溢出，与 `(low + high) / 2` 等价但更安全。
   - 比较中间位置上的元素与目标值：
     - 如果 `array[mid]` 等于目标值，返回索引 `mid`。
     - 如果 `array[mid]` 小于目标值，调整 `low` 指针到 `mid + 1`。
     - 如果 `array[mid]` 大于目标值，调整 `high` 指针到 `mid - 1`。
3. **结束查找**：如果在数组中找不到目标值，返回 `-1` 或其他指示值，表示查找失败。

算法C++代码

```cpp
int binarySearch(int array[], int target, int size)
{
    int low = 0, high = size - 1;
    while (low <= high)
    {
        int mid = low + (high - low) / 2;
        if (array[mid] == target)
        {
            return mid;
        }
        else if (array[mid] < target)
        {
            low = mid + 1;
```

```
        }
        else
        {
            high = mid - 1;
        }
    }
    return -1; // 目标值不在数组中
}
```

渐进时间复杂度分析

二分搜索的效率在于每次迭代都将搜索范围减半，这是分治策略的一个经典应用。

- **最好情况**：如果目标元素恰好是中间元素，算法将在第一次比较后直接找到目标，时间复杂度为 $O(1)$。
- **最坏情况和平均情况**：在最坏情况和平均情况下，算法需要执行多次迭代，每次将搜索范围减半，直到找到目标元素或 `low` 指针超过 `high` 指针。初始搜索范围是 n（数组长度），然后是 $n/2$，$n/4$，直到 1。可以通过求解 $n/2^k = 1$ 来确定步骤数 k，其中 k 是迭代次数。解这个方程得到 $k = \log_2 n$。因此，二分搜索的时间复杂度为 $O(\log n)$。

实例四、归并排序和快速排序

归并排序（Merge Sort）和快速排序（Quick Sort）是两种非常高效且常用的比较基础排序算法，它们在很多实际应用中都有广泛的应用。虽然两者都采用了分治策略，但它们的实现细节和性能特点在某些方面有所不同。

归并排序（Merge Sort）

归并排序是一种稳定的排序算法，它不断地将一个数组分成两半来进行递归排序，然后合并已排序的半部分。

- 时间复杂度
  - **最好、最坏和平均情况**：归并排序的时间复杂度在所有情况下都是 $O(n \log n)$。
    - **分解**：数组被递归地分成两半，直到每个子数组只包含一个元素。
    - **合并**：每个排序的子数组需要线性时间与其他排序子数组合并。每层合并操作总共需要 $O(n)$ 时间，而分解数组所需的深度是 $\log n$。
  - 这种方法保证了无论输入数据的初始顺序如何，归并排序都将执行相同数量的比较和移动操作，这使得它的性能非常稳定。
- 空间复杂度
  - **空间复杂度**：由于归并排序需要相等于原数组大小的额外空间来存储合并过程中的临时数组，因此其空间复杂度为 $O(n)$。

快速排序（Quick Sort）

快速排序是一种非常快速的不稳定排序算法，它通过一个称为“枢纽”（pivot）的元素来实现排序。

- 时间复杂度
  - **平均情况**：快速排序的平均时间复杂度是 $O(n \log n)$。
  - **最坏情况**：如果每次分区操作都将列表分成两部分不平均，如当数组已经是有序的或完全逆序时，最坏的时间复杂度是 $O(n^2)$。

    - **最好情况**：最好情况发生在每次分区都能将数组均等地分成两部分，此时时间复杂度为 $O(n \log n)$。
    - 尽管在最坏情况下性能较差，但快速排序在平均和最好情况下通常比其他 $O(n \log n)$ 算法更快，因为其内部循环（inner loop）可以在大多数现代架构上非常高效地执行。
- 空间复杂度
    - **空间复杂度**：快速排序的空间复杂度依赖于递归的深度，它在最好的情况下是 $O(\log n)$（平衡的分区），并在最坏的情况下是 $O(n)$（不平衡的分区）。

总结

- **归并排序**提供了一致的 $O(n \log n)$ 性能，但需要额外的内存空间。
- **快速排序**在大多数情况下提供超快的性能，并且通常是实际应用中的首选，尽管在最坏的情况下可能会显著变慢。
- 两者都使用了分治策略，但归并排序是自顶向下的，而快速排序是自底向上的。

实例五、棋盘覆盖

棋盘覆盖算法是一种典型的分治策略应用，用于解决特殊棋盘覆盖问题。该问题描述的是如何使用L形骨牌覆盖一个缺失单个格子的棋盘。这种算法非常适合解决这类问题，特别是在计算机科学和相关领域中，比如在图像处理、内存管理等场景中有着广泛的应用。

问题描述

设想一个大小为 $2^k \times 2^k$ 的棋盘，其中有一个格子已经被填充或损坏。任务是使用L形骨牌完全覆盖棋盘的其他部分。每个L形骨牌覆盖恰好三个格子。

算法概述

棋盘覆盖算法利用分治技术递归地解决问题。算法的核心思想是将大棋盘分解成更小的棋盘，每个小棋盘大小为 $2^{k-1} \times 2^{k-1}$。对于每一层递归，棋盘被分为四个象限，每个象限都可能需要一个特殊的处理来放置L形骨牌。

算法步骤

1. **初始条件**：如果棋盘大小为 1×1，则已经达到递归基，不需要进一步操作。
2. **分治步骤**：
    - 将棋盘划分为四个象限。
    - 确定每个象限中的特殊格子（即已被覆盖或损坏的格子）。
    - 如果某个象限中没有特殊格子，则在这四个象限的中心放置一个L形骨牌，使之成为该象限的特殊格子。
3. **递归**：对每个象限重复上述步骤，直到棋盘的大小缩减到 1×1。

算法示例

假设有一个 4×4 的棋盘，右上角的第一个格子是特殊格子。算法步骤如下：

- 将棋盘分为四个 2×2 的象限。
- 在四个象限的中心放置一个L形骨牌。
- 每个象限现在都有一个特殊格子：三个由于新放置的L形骨牌，一个由于原始的特殊格子。

- 对每个象限递归执行相同的步骤。

时间复杂度分析

每一次递归调用处理一个减少一半大小的问题，并在中心放置一个L形骨牌。递归的深度为 $\log_2 n$，其中 n 是棋盘的边长。在每一级递归中，都需要处理四个较小的棋盘问题。因此，算法的时间复杂度大致为 $O(n^2)$，这里的 n 表示棋盘的边长。

棋盘覆盖问题中的骨牌数推导涉及到计算在覆盖整个棋盘时所需要的L形骨牌的数量。每个L形骨牌覆盖三个单元格。考虑一个大小为 $2^k \times 2^k$ 的棋盘，其中有一个单元格已经被占用或标记为特殊，这意味着需要用L形骨牌覆盖其余的单元格。

$$\text{所需骨牌数} = \frac{4^k - 1}{3}$$

六、减治法

1. 减治法分类：
    - 减一个常量，常常是减一：如插入排序
    - 减一个常因子，常常是减去因子2：如折半查找
    - 减可变规模：欧几里得算法
2. 实例：

实例一、减一个常量：减一

1. 插入排序

插入排序（Insertion Sort）是一种简单直观的排序算法。它的工作原理非常类似于我们对扑克牌进行排序的方式。

- 设计思路
    - **算法步骤**：
        - 将数组分为已排序部分和未排序部分。初始时，已排序部分只包含第一个元素。
        - 取出未排序部分的第一个元素，将其插入到已排序部分中的适当位置，以保持已排序部分的有序。
        - 重复上述过程，直到未排序部分为空。
    - **具体实现**：
        - 从第二个元素开始，逐个将每个元素插入到已排序部分。
        - 对于每个待插入的元素，与已排序部分的元素从后向前比较，找到合适的插入位置。
        - 如果当前已排序的元素比待插入元素大，将这个元素向后移动一位。
        - 重复这个比较和移动过程，直到找到待插入元素的正确位置，并将其插入。
- 渐进时间复杂度分析
    - **最好情况**（Best Case）：如果输入数组已经是排序好的，每次插入操作不需要移动已排序部分中的任何元素。此时的时间复杂度为 $O(n)$。
    - **平均情况**（Average Case）：假设每个元素前面有一半的元素需要移动，时间复杂度为 $O(n^2)$。
    - **最坏情况**（Worst Case）：如果输入数组是反向排序的，每次插入操作都需要移动已排序部分中的所有元素。此时的时间复杂度也是 $O(n^2)$。
- 计算方法
    - 时间复杂度的计算主要考虑每个元素在插入过程中需要比较的次数。在最坏情况下，第 i 个元素需要 $i-1$ 次比较，因此总的比较次数是 $\frac{n(n-1)}{2}$，这是一个二次函数，因此渐进时间复杂度为 $O(n^2)$。

2. 拓扑排序

拓扑排序是一种对有向无环图（DAG, Directed Acyclic Graph）的顶点进行排序的算法，它可以产生一个线性排序结果，使得对于图中的每个有向边 $u \to v$，顶点 u 都出现在顶点 v 的前面。

- 设计思路

    - 拓扑排序的基本思想是不断地选择入度为零的顶点，将其加入到排序结果中，并从图中移除，同时移除该顶点的所有出边，更新其他顶点的入度。重复这个过程，直到所有顶点都被移除。
        - **初始化**：统计每个顶点的入度。
        - **选择顶点**：从图中选择一个入度为零的顶点，将其添加到排序结果中。
        - **更新入度**：移除该顶点及其所有出边，更新相邻顶点的入度。
        - **重复**：重复步骤2和3，直到所有顶点都被选中或图中没有入度为零的顶点（这意味着图中存在环）。
- 算法实现
    - 通常有两种实现拓扑排序的方法：
        - **使用队列**：利用队列来存储所有入度为零的顶点。每次从队列中取出一个顶点，更新其相邻顶点的入度，若更新后某个顶点的入度变为零，则将其加入队列。
        - **使用深度优先搜索（DFS）**：对每个未访问的顶点执行DFS，访问结束后将顶点推入栈中。最后从栈中依次弹出顶点即可得到一个拓扑排序。
- 渐进时间复杂度分析
    - **时间复杂度**：无论是使用队列还是DFS实现，拓扑排序的时间复杂度均为 $O(V+E)$，其中 V 是顶点数，E 是边数。这是因为算法需要遍历所有的顶点和边。
- 计算方法
    - 在使用队列的实现中，每个顶点和边都分别被处理一次。顶点在被加入队列时进行了处理，每条边在更新顶点入度时被考虑一次。
    - 在DFS的实现中，每个顶点在递归中被访问一次，每条边在探索时被访问一次。

实例二、减常因子

1. 二分搜索

2. 假币问题

假币问题是减治法中一个经典的问题，它的目的是从一堆硬币中找出唯一的一个较轻的假币。这个问题可以通过使用天平秤进行一系列的比较来解决，每次比较可以减少需要考虑的硬币数量。

- 设计思路
    - 基本的思路是通过比较不同组硬币的重量来逐步缩小假币可能所在的范围。通过将硬币分组并在天平秤上进行比较，可以确定假币是在较轻的一组中，从而不断减少搜索的规模。
        - **分组**：将硬币分成三组，尽可能保持每组硬币数目相等。
        - **比较**：随机选择两组在天平秤上进行比较。
            - 如果一边比另一边轻，那么假币在较轻的那一组。
            - 如果两边相等，假币在未参与比较的第三组。
        - **递归**：重复上述过程，每次只对含有假币的那一组硬币进行操作，直到剩下一个硬币，那就是假币。
- 渐进时间复杂度分析
    - 这个问题的解决方案主要是基于减治策略，通过每次减少约三分之一的搜索范围。
    - **时间复杂度**：$O(\log_3 n)$。每次操作后，硬币的数量大约减少到原来的三分之一，所以需要进行的比较次数大约是 $\log_3 n$。

- 计算方法
  - 在每次操作中，硬币的总数 n 通过三分之一地减少，直到最终缩减到1。这个对数减少过程可以通过计算 $\log_3 n$ 来估算总的比较次数。
  - 计算通式：

$$T(n) = \begin{cases} 0 & \mathrm{n} = 0 \\ T(n/2) + 1 & \mathrm{n} > 1 \end{cases}$$

- 这种策略的优点是比较次数相对较少，特别适合于硬币数量较多的情况。然而，这种方法依赖于能够精确地将硬币分成重量均等的三组，这在实际操作中可能会遇到一些困难。此外，它也假设除一个假币外其他硬币完全相同。如果存在多个不同重量的假币或者重量有细微差异的真币，这种方法就需要调整。

3. 俄罗斯农民乘法

俄罗斯农民乘法，也被称为埃及乘法或二进制乘法，是一种古老的算法，用于手算两个数的乘积。它通过重复的加法和减半操作来实现乘法，适用于只有加法和除法运算能力的计算环境。

- 设计思路
  - 这种乘法算法基于将乘法转化为加法和位移运算。其核心思想是利用二进制的性质来简化运算。
    - **初始化**：将其中一个乘数列为 A，另一个乘数列为 B。
    - **迭代操作**：
      - 如果 A 是奇数，将 B 加到最终结果上。
      - 将 A 除以 2（向下取整），将 B 乘以 2。
    - **终止条件**：当 A 减少到 0 时，停止操作。
    - **结果**：B 累加的结果即为两数的乘积。
- 算法步骤示例
  - 假设需要计算 18 乘以 25：
    - 18 是偶数，25 保持不变，18 除以 2 变为 9，25 乘以 2 变为 50。
    - 9 是奇数，将 50 加到结果中，9 除以 2 变为 4，50 乘以 2 变为 100。
    - 4 是偶数，100 保持不变，4 除以 2 变为 2，100 乘以 2 变为 200。
    - 2 是偶数，200 保持不变，2 除以 2 变为 1，200 乘以 2 变为 400。
    - 1 是奇数，将 400 加到结果中，1 除以 2 变为 0，结束。
  - 最终结果是 50 + 400 = 450。
- 渐进时间复杂度分析
  - **时间复杂度**：$O(\log A)$，其中 A 是较小的乘数。每次操作 A 至少减少一半，因此操作的次数取决于 A 的二进制位数。
- 计算方法
  - 每次循环都包括一次可能的加法和一次确定的位移操作。这些操作的总次数大约与 A 的二进制表示的长度成比例，也即是 A 的对数。

实例三、减可变因子

欧几里得算法

欧几里得算法，也称为辗转相除法，是一种用来计算两个正整数 a 和 b 的最大公约数（GCD, Greatest Common Divisor）的古老而有效的算法。这种算法的基本思想是基于一个数学定理：两个正整数的最大公约数与它们的差的最大公约数相同。

设计思路

欧几里得算法的步骤相对简单明了，其核心是重复从较大的数中减去较小的数，直到两数相等，那么得到的数就是它们的最大公约数。在现代实现中，通常使用余数操作来优化这个过程。

1. **算法步骤**：
    - 比较 a 和 b，确定较大数和较小数。
    - 将较大数除以较小数，得到余数。
    - 将较小数置为新的较大数，将余数置为新的较小数。
    - 重复上述过程，直到余数为 0。
    - 当余数为 0 时，当前的较小数即为两数的最大公约数。

算法实例

假设需要计算 252 和 105 的最大公约数：

- $252 \div 105 = 2$ 余数 42
- $105 \div 42 = 2$ 余数 21
- $42 \div 21 = 2$ 余数 0
- 余数为 0，最大公约数是 21。

渐进时间复杂度分析

- **时间复杂度**：最坏情况下为 $O(\log(\min(a, b)))$。这是因为每次操作都至少将问题的大小减半（在最坏情况下，如连续的斐波那契数）。

计算方法

- 算法的效率来自于每次通过取余操作来快速减少问题的规模。实际上，每步都大幅度减少了余数的大小，通常来说，两个数连续进行取余操作的次数不会超过它们中较小数的二进制位数。

七、变治法

1. 变治法是一种基于变换思想，把问题变换成一种更容易解决的类型
2. 变治法的类型：实例化简、改变表现、问题化简
3. 实例：

实例一：预排序算法——实例化简

预排序算法通过先对数据进行排序，从而简化后续处理步骤，提高效率。这种策略在多种问题中都有应用，如统计数据中的众数、快速查找以及数据压缩等。

设计思路

预排序算法的基本思路是在解决主问题之前先对问题的输入数据进行排序，通过排序来简化或加速后续的处理。排序后的数据结构使得一些操作（如查找、合并、去重）变得更加直接和高效。

典型应用：

- **统计众数**：先对数组进行排序，然后通过线性扫描来统计每个元素出现的次数，从而快速找出众数。
- **快速查找**：排序后的数组可以使用二分查找，大大提高查找速度，尤其是对于频繁的查找操作。
- **数据去重**：在排序数组中，重复的元素会被排在一起，可以通过一次线性扫描去除重复元素。

算法实现

- **排序**：使用有效的排序算法（如快速排序、归并排序等）对数据进行排序。
- **处理**：对排序后的数据应用特定的算法解决问题。

渐进时间复杂度分析

- **时间复杂度**：预排序算法的总体时间复杂度取决于所用排序算法的时间复杂度和后续处理步骤的复杂度。通常情况下，排序步骤是主导整个算法时间复杂度的部分，如快速排序的平均时间复杂度为 $O(n \log n)$。

计算方法

- 在预排序算法中，最耗时的步骤通常是排序过程。例如，如果使用快速排序，其平均时间复杂度为 $O(n \log n)$，之后的处理步骤如线性扫描通常只需要 $O(n)$，因此整个算法的效率主要由排序步骤决定。

预排序算法的优点在于通过牺牲排序的时间复杂度来获取后续操作的高效性，尤其适用于那些排序后能显著简化问题的场景。此外，对于数据量不断增加的情况，只要维持数据的有序状态，后续的处理通常可以更加迅速地执行。

实例二、高斯消元法——实例化简

高斯消元法是解线性方程组的一种经典算法，常被用作求解线性代数方程组的标准方法。该算法使用了变治法（Decrease and Conquer）的策略，通过逐步简化（或者说“变换”）原始的方程组，最终将其转换成一个更易于解决的形式。

设计思路

高斯消元法主要通过两种操作来简化方程组：行交换和行变换。目的是将线性方程组的系数矩阵转化为行阶梯形矩阵（或者进一步转化为简化的行阶梯形矩阵），从而容易从后向前求解未知数。

具体步骤包括：

1. **选择主元**：在当前列找到一个非零元素作为主元（如果需要，通过行交换确保主元不为零）。
2. **消元**：使用主元行通过行变换消去下面行中该列的所有其他元素。
3. **重复**：对每一列重复以上步骤，直到处理完所有列或者达到矩阵的最后一行。
4. **回代**：从最后一行开始，逐行解出每个未知数。

算法实例

$$\begin{cases} 2x + 3y - z = 1 \\ 4x - y + 5z = 13 \\ -2x + 7y + 2z = 10 \end{cases}$$

高斯消元法步骤

初始矩阵

首先，我们把方程组写成增广矩阵的形式：

$$\left[\begin{array}{ccc|c} 2 & 3 & -1 & 1 \\ 4 & -1 & 5 & 13 \\ -2 & 7 & 2 & 10 \end{array}\right]$$

第一步：消元第一列

以第一行为主元行，将第一列的其他元素消为零。

- 使用第一行消除第二行：
  - 第二行 $\leftarrow$ 第二行 $-2\times$ 第一行
  - $4 - 2 \times 2 = 0, -1 - 2 \times 3 = -7, 5 - 2 \times -1 = 7, 13 - 2 \times 1 = 11$
  - 新第二行：$[0, -7, 7|11]$
- 使用第一行消除第三行：
  - 第三行 $\leftarrow$ 第三行 $+1\times$ 第一行
  - $-2 + 1 \times 2 = 0, 7 + 1 \times 3 = 10, 2 + 1 \times -1 = 1, 10 + 1 \times 1 = 11$
  - 新第三行：$[0, 10, 1|11]$

新矩阵：

$$\left[\begin{array}{ccc|c} 2 & 3 & -1 & 1 \\ 0 & -7 & 7 & 11 \\ 0 & 10 & 1 & 11 \end{array}\right]$$

第二步：消元第二列

以新第二行为主元行，将第二列的其他元素消为零。

- 归一化第二行：
  - 第二行 $\leftarrow$ 第二行 $/(-7)$
  - $[0, 1, -1, |, -\frac{11}{7}]$
- 使用新第二行消除第三行：
  - 第三行 $\leftarrow$ 第三行 $-10\times$ 新第二行
  - $0, 10 - 10 \times 1 = 0, 1 - 10 \times -1 = 11, 11 - 10 \times -\frac{11}{7} = 27.57$
  - 新第三行：$[0, 0, 11, |, 27.57]$

新矩阵：

$$\begin{bmatrix} 2 & 3 & -1 & | & 1 \\ 0 & 1 & -1 & | & -\frac{11}{7} \\ 0 & 0 & 11 & | & 27.57 \end{bmatrix}$$

第三步：回代求解

- $z = \frac{27.57}{11} = 2.51$
- $y = -\frac{11}{7} + 1 \times 2.51 = 0.22$
- $x = \frac{1 - 3 \times 0.22 + 1 \times 2.51}{2} = 0.99$

结果

所以，解得 $x \approx 0.99, y \approx 0.22, z \approx 2.51$。

这就是使用高斯消元法解线性方程组的具体步骤。每一步都涉及到行操作以将系数矩阵转化为上三角形矩阵

渐进时间复杂度分析

- **时间复杂度**：高斯消元法的时间复杂度大致为 $O(n^3)$，其中 n 是方程组中未知数的数量。这是因为每个消元步骤涉及到对 $n-1$ 行进行 n 次操作，而对于每个未知数都需要进行此操作。

计算方法

- 每一步消元操作都需要更新下方的每一行，对于第 k 步（k 从 1 到 $n-1$），需要对 $n-k$ 行每行进行大约 $n-k+1$ 次操作。因此总操作数大致为 $\sum_{k=1}^{n-1} (n-k)^2$，这可以简化为 $O(n^3)$。

实例三、堆、堆排序、堆维护——改变表现

堆是一种特殊的完全二叉树，主要用于实现优先队列。堆的特点是任一节点的值总是不大于或不小于其子节点的值，这种性质称为堆性质。按此特性可分为两种堆：最大堆和最小堆。在最大堆中，每个父节点的值都大于或等于其子节点的值；在最小堆中，每个父节点的值都小于或等于其子节点的值。

堆排序（Heap Sort）

堆排序是一种利用堆数据结构设计的排序算法，其主要优势是时间复杂度稳定。

算法步骤：

1. **构建最大堆**：将无序的输入数组构造成最大堆，保证每个节点都遵循最大堆的性质。
2. **排序**：将堆顶元素（最大值）与堆的最后一个元素交换，然后减小堆的大小，对新的堆顶元素进行下沉操作以维护最大堆性质。重复此过程，直到堆的大小为1。

时间复杂度：

- 构建堆的时间复杂度为 $O(n)$。
- 排序过程中，每次重新恢复堆性质的时间复杂度为 $O(\log n)$，共需要进行 $n-1$ 次操作。因此，总的时间复杂度为 $O(n \log n)$。

堆的插入（Insertion in a Heap）

向堆中插入新元素时，需要保持堆的完全二叉树结构和堆性质。

插入步骤：

1. **添加元素**：在堆的末尾添加新元素。
2. **上浮操作**（Heapify-Up）：如果新添加的元素破坏了堆性质，即它比其父节点大（在最大堆中），则需要与父节点交换位置，直到恢复堆性质或元素到达顶部。

时间复杂度：

- 最坏情况下，新元素可能需要从堆底浮到堆顶，时间复杂度为 $O(\log n)$。

堆的删除（Deletion in a Heap）

通常删除操作指的是删除堆顶元素，即堆中的最大元素或最小元素，这是因为堆主要用作优先队列。

删除步骤：

1. **移除堆顶**：将堆顶元素与堆的最后一个元素交换。
2. **缩小堆大小**：去掉堆的最后一个元素（原堆顶元素）。
3. **下沉操作**（Heapify-Down）：恢复新堆顶的堆性质。如果堆顶元素违反堆性质（比子节点小在最大堆中），与其子节点中较大的一个交换位置，重复此过程直到恢复堆性质或成为叶节点。

时间复杂度：

- 堆顶元素的删除和下沉操作的时间复杂度为 $O(\log n)$。

实例四、霍纳法则——改变表现

霍纳法则（Horner's Rule），也称为霍纳方法，是一种用于多项式求值的高效算法。该算法主要通过减少乘法操作的数量来提高计算多项式的效率。

设计思路

霍纳法则的基本思想是将多项式从内部开始重写和计算，利用已经计算过的部分来简化计算过程。这种方式可以将多项式的求值转化为连续的乘加操作，从而显著减少计算的复杂性。

考虑一个多项式：
$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$

霍纳法则将其重写为嵌套的形式：
$P(x) = (((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$

算法步骤

1. **初始化**：设定结果变量 $b = a_n$。
2. **迭代计算**：从 a_{n-1} 到 a_0 依次进行，每次操作如下：
    - 将 b 乘以 x。
    - 将下一个系数 a_i 加到 b 上。
3. **输出结果**：最终的 b 即为 $P(x)$ 的值。

算法实例

假设我们要计算多项式 $P(x) = 2x^3 - 6x^2 + 2x - 1$ 在 $x = 3$ 时的值：

- 初始化：$b = 2$
- $b = 2 \times 3 + (-6) = 0$
- $b = 0 \times 3 + 2 = 2$
- $b = 2 \times 3 - 1 = 5$

因此，$P(3) = 5$。

渐进时间复杂度分析

- **时间复杂度**：霍纳法则的时间复杂度为 $O(n)$，其中 n 是多项式的度。这是因为每个系数只参与一次乘法和一次加法。

计算方法

通过嵌套式的连续乘加操作，霍纳法则减少了多项式计算中不必要的重复乘法，使得计算更为高效。这种方法非常适合用于计算机实现，因为它直接减少了算术操作的总数，并且因为其线性时间复杂度，对于高度多项式尤为有效。

问题化简的实例——几何问题转变成代数问题、线性规划、转化成图问题

此处暂不列举，考的不难，知道有哪些问题即可

几何问题转变成代数问题就是点线关系，就是解析几何问题

转化成图问题就是过河问题

八、回溯法和分支限界法

（一）搜索算法

1. 穷举搜索

穷举搜索（Exhaustive Search），也称为暴力搜索（Brute Force Search），是一种最直接、最简单但通常最耗时的算法设计方法。它通过尝试所有可能的解，从中找到满足问题条件的解。

设计思路

穷举搜索的基本思想是系统地列出所有可能的解决方案，然后逐一检查每个解决方案是否满足问题的要求。这种方法适用于问题规模较小或搜索空间有限的情况。

步骤

1. **列出所有可能的解**：生成所有可能的候选解。
2. **逐一验证**：对每个候选解进行验证，检查它是否满足问题的条件。
3. **选择最佳解**：从满足条件的解中选择最佳解（如果有）。

应用场景

穷举搜索广泛应用于组合优化问题、密码破解、排列组合问题等。以下是几个经典应用场景：

- **旅行商问题（TSP）**：列出所有可能的城市访问顺序，计算每个顺序的总距离，选择最短的那个。
- **子集和问题**：列出所有可能的子集，检查每个子集的和是否等于目标值。
- **密码破解**：尝试所有可能的密码组合，直到找到正确的密码。

优点

- **简单易懂**：算法思路和实现都非常简单。
- **全面性**：保证找到问题的最优解，因为它遍历了所有可能的解。

缺点

- **计算成本高**：对于大规模问题，穷举搜索的计算量非常庞大，导致时间复杂度和空间复杂度极高。
- **效率低下**：通常只有在问题规模较小或没有其他高效算法可用时才使用。

时间复杂度

穷举搜索的时间复杂度通常是指数级别或更高，这取决于问题的具体性质。例如，对于 n 个元素的排列，穷举搜索的时间复杂度为 $O(n!)$；对于子集和问题，时间复杂度为 $O(2^n)$。

示例

以子集和问题为例，假设给定一个集合 $\{1, 2, 3\}$ 和目标和 5，穷举搜索的步骤如下：

1. 列出所有子集：$\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}$。
2. 计算每个子集的和，并检查是否等于 5。
3. 找到满足条件的子集 $\{2, 3\}$。

穷举搜索虽然简单，但在处理复杂问题时通常需要结合剪枝策略或其他优化技术，以减少计算量和提高效率。

2. DFS和BFS

深度优先搜索（DFS）

设计思路

深度优先搜索（DFS）的基本思想是从一个起点开始，沿着一条路径一直往深处走，直到不能继续为止，然后回溯到最近的分支点继续搜索，直到所有节点都被访问。

1. **选择起点**：从图的一个起点（源节点）开始。
2. **访问节点**：访问当前节点，并将其标记为已访问。
3. **递归访问**：对于当前节点的每一个邻接节点，如果该邻接节点未被访问，则递归地对该邻接节点进行深度优先搜索。
4. **回溯**：当所有邻接节点都被访问后，回溯到上一个节点，继续搜索其他未访问的分支。

时间复杂度分析

- **时间复杂度**：对于一个包含 V 个顶点和 E 条边的图，DFS 的时间复杂度为 $O(V + E)$。这是因为每个节点和每条边都被访问一次。
- **空间复杂度**：在最坏情况下，递归实现的空间复杂度为 $O(V)$，这是由于递归栈的深度。非递归实现中，显式栈的空间复杂度也是 $O(V)$。

最好和最坏情况

- **最好情况**：DFS 没有明显的最好情况，因为它总是需要访问所有节点和边。
- **最坏情况**：最坏情况是在图的深度为 V 的情况下，递归调用的栈深度也为 V。

广度优先搜索（BFS）

设计思路

广度优先搜索（BFS）的基本思想是从一个起点开始，首先访问所有与起点直接相连的节点，然后依次访问这些节点的邻接节点，逐层扩展直到访问完所有节点。

1. **选择起点**：从图的一个起点（源节点）开始。
2. **使用队列**：将起点加入队列，并标记为已访问。
3. **迭代访问**：从队列中取出一个节点，访问其所有未访问的邻接节点，并将这些邻接节点加入队列。
4. **继续迭代**：重复上述步骤，直到队列为空。

时间复杂度分析

- **时间复杂度**：对于一个包含 V 个顶点和 E 条边的图，BFS 的时间复杂度为 $O(V + E)$。这是因为每个节点和每条边都被访问一次。
- **空间复杂度**：在最坏情况下，队列需要容纳图的一层节点，其空间复杂度为 $O(V)$。

最好和最坏情况

- **最好情况**：BFS 没有明显的最好情况，因为它总是需要访问所有节点和边。
- **最坏情况**：最坏情况是在宽度较大的情况下，队列需要容纳一层中的所有节点，导致空间复杂度达到 $O(V)$。

总结

- **DFS**：适用于需要深入搜索解决的问题，如路径问题、连通分量检测等。采用邻接链表时时间复杂度为 $O(V+E)$ ，采用邻接矩阵时时间复杂度为$O(V^2)$
- **BFS**：适用于需要逐层搜索的问题，如最短路径问题。采用邻接链表时时间复杂度为 $O(V+E)$ ，采用邻接矩阵时时间复杂度为$O(V^2)$

（二）回溯法

回溯法（Backtracking）是一种通过递归来解决问题的算法，其基本思想是在解空间树中进行深度优先搜索（DFS），从而寻找问题的所有解。这种方法通常被用于解决组合问题，例如排列、组合或子集问题，尤其是在解的数量非常大时。

算法设计思路

回溯法算法的核心是构建一个解空间树，算法从根节点开始，按深度优先的方式探索整个解空间树。每到一个节点，算法就判断该节点所代表的部分解是否可能是问题的解：

- 如果可能，继续向下探索；
- 如果不可能，则回溯到上一个节点，尝试其他可能的分支。

这个过程中，算法使用了剪枝技术，有效地减少了搜索空间。剪枝包括两种：

- 约束函数：在扩展结点处剪去不满足约束的子树
- 限界函数：剪去得不到最优解的子树

渐进时间复杂度分析

回溯法的时间复杂度很大程度上依赖于解空间的大小以及剪枝的效率。在最坏的情况下，可能需要遍历整个解空间：

- 子集树：如果问题是从n个元素中找出满足某种性质的子集，则子集树有 2^n 个叶子节点，遍历这样的子集树的时间复杂度为 $\Omega(2^n)$。
- 排列树：如果问题是确定n个元素满足某种性质的排列，则排列树有 $n!$ 个叶子节点，遍历排列树需要 $\Omega(n!)$ 的计算时间。

在实际应用中，有效的剪枝可以显著减少搜索空间，从而减少算法的运行时间。但是，回溯算法的效率还是高度依赖于问题本身的性质以及剪枝函数的设计。

实例一、0-1背包问题

0-1背包问题是一个经典的组合优化问题，其中包含一组物品，每个物品有特定的重量和价值。目标是在不超过背包容量的前提下，选择物品的组合，使得背包中物品的总价值最大化。在回溯法中解决0-1背包问题时，可以使用有限界函数和无限界函数的策略来优化搜索过程。

无限界函数的算法设计

在无限界函数的策略中，算法不预先计算节点的界限，而是简单地通过回溯探索所有可能的解空间。这种方法直接探索每个物品包含或不包含在内的情况，即对于每个物品，算法都会尝试将它加入背包中（如果加入后不超过背包的容量限制），然后再递归地探索下一个物品。如果加入当前物品后超过容量限制，算法则回溯，尝试不加入当前物品的情况。

- **递归过程**：从第一个物品开始，对每个物品有两种选择：包含该物品或不包含。每做一次选择，都对剩余的物品重复同样的选择过程，直到所有物品都被考虑过。

- **终止条件**：当考虑完所有物品后，如果当前的组合有效（即总重量不超过背包容量），则比较并更新最大价值。

有限界函数的算法设计

有限界函数的策略则更为高效，因为它在探索解空间时使用了限界函数来剪枝，即计算当前扩展节点下所有子节点的可能最优值上界。如果某个节点的最优值上界低于已知的最大价值，则无需进一步探索该节点。

- **限界函数**：对于当前节点，计算包含当前节点的物品后，如果将剩余物品以最高价值密度顺序尽可能装入背包中的理论最大价值。这个理论最大价值作为这个节点的上界。
- **剪枝**：如果当前节点的上界小于已知的最大价值，则放弃探索该节点下的所有可能路径，因为它们不可能产生更优的解。

算法例子（有限界函数）

1. **初始化**：设定最大价值为0，从第一个物品开始探索。
2. **扩展节点**：对于每个物品，如果加入该物品后的总重量不超过背包容量，并且当前节点的价值上界大于已知最大价值，则继续探索。
3. **递归**：递归地对每个物品重复上述过程。
4. **更新最大价值**：每到达一个叶节点（即考虑完所有物品后），如果当前价值大于已知最大价值，则更新最大价值。

通过使用有限界函数，0-1背包问题的回溯算法可以显著减少需要探索的节点数，从而提高效率。这种方法特别适用于物品数量较多或背包容量较大的情况，可以有效地找到接近最优或最优解。

实例二、TSP问题

旅行商问题（Traveling Salesman Problem, TSP）是一个经典的组合优化问题，目标是找到访问一组城市并回到原点的最短可能路径，且每个城市只访问一次。回溯法是解决TSP问题的一种有效方法，特别是在城市数量较少时。这种方法通过深度优先搜索（DFS）所有可能的路径，尝试找到总距离最短的解。

回溯法解决TSP的基本思想

回溯法在解决TSP问题时，主要是构建一个“排列树”。树的每个节点表示一条路径的部分解，即一系列已经访问的城市和待访问的城市。从根节点开始（通常为空或仅包含起始城市），通过逐步扩展到所有未访问的城市来探索所有可能的路径。

算法设计

1. **初始化**：从起始城市开始，将其设为当前访问城市，已访问城市列表只包含这一城市，未访问的城市列表包含除起始城市外的所有城市。
2. **选择扩展节点**：在未访问城市中选择下一个要访问的城市。每次选择后，更新当前路径长度并将城市从未访问列表移至已访问列表。
3. **递归搜索**：对每个新访问的城市，重复第2步，递归扩展路径。
4. **剪枝**：在扩展过程中，如果当前路径长度已经超过已知的最短路径长度，则放弃进一步扩展这条路径。
5. **回溯**：完成一个城市的扩展后，将其从已访问列表移回未访问列表，并恢复路径长度到上一步状态，尝试其他可能的城市作为下一个访问目标。

6. **更新最短路径**：每当找到一条回到起始城市的完整路径时，如果该路径的长度小于当前已知的最短路径长度，则更新最短路径信息。

实例说明

假设有四个城市A, B, C, D，距离矩阵如下：

	A	B	C	D
A	0	10	15	20
B	10	0	35	25
C	15	35	0	30
D	20	25	30	0

让我们使用前面提到的四个城市的例子，使用回溯法来求解TSP问题。这个例子将帮助你理解回溯法在实际应用中的详细步骤。我们的目标是找到从城市A开始，访问所有城市并回到城市A的最短路径。

初始化

- 起始城市：A
- 已访问城市列表：[A]
- 未访问城市列表：[B, C, D]
- 当前最短路径长度：无限大
- 当前路径长度：0

步骤解析

第一层递归

1. 从A出发，选择下一个城市。假设按字母顺序考虑，第一个选B。
    - 更新路径：A → B
    - 路径长度：0 + 10 = 10
    - 更新城市列表：已访问 [A, B]，未访问 [C, D]

第二层递归

2. 从B出发，选择下一个城市。首先选C。
    - 更新路径：A → B → C
    - 路径长度：10 + 35 = 45
    - 更新城市列表：已访问 [A, B, C]，未访问 [D]

第三层递归

3. 从C出发，选择下一个城市。唯一选择D。
    - 更新路径：A → B → C → D
    - 路径长度：45 + 30 = 75
    - 更新城市列表：已访问 [A, B, C, D]，未访问 []

返回到起点

4. 从D返回到A。
    - 完成路径：A → B → C → D → A
    - 路径长度：75 + 20 = 95
    - 检查当前路径长度（95）是否小于已知最短路径（无限大）。是，则更新最短路径长度为95。

回溯

5. 回溯至城市C，尝试其他可能性。从C返回到B，并尝试从B出发到D。
    - 新路径尝试：A → B → D
    - 路径长度：10 + 25 = 35

第三层递归

6. 从D出发，仅剩下城市C未访问。
    - 更新路径：A → B → D → C
    - 路径长度：35 + 30 = 65

返回到起点

7. 从C返回到A。
    - 完成路径：A → B → D → C → A
    - 路径长度：65 + 15 = 80
    - 检查当前路径长度（80）是否小于已知最短路径（95）。是，则更新最短路径长度为80。

继续回溯和递归

8. 回溯至城市B，尝试从A直接到C或D，并重复上述的递归过程和路径长度计算。

结果

通过上述步骤，我们将找到所有可能的路径，并通过回溯和剪枝有效减少不必要的计算。最终，我们得到从城市A出发访问所有城市并返回的最短路径和其长度。

这个例子展示了回溯法如何通过系统地探索所有可能的路线并利用剪枝来减少搜索空间，找到问题的最优解。对于实际应用，尤其是城市数量增多时，这种方法的计算成本会急剧增加，因此它更适用于城市数量较少的情况。

性能考虑

虽然回溯法能够给出TSP问题的精确解，但是它的时间复杂度随城市数量的增加而急剧增加（阶乘级增长）。因此，对于城市数较多的情况，通常需要采用近似算法或启发式算法来在合理时间内获得可接受的解。

实例三、N皇后问题

N皇后问题是一个经典的回溯算法问题，其目标是在一个N×N的棋盘上放置N个皇后，使得它们不能互相攻击。皇后可以攻击同一行、同一列或同一对角线上的其他皇后。解决这个问题涉及到在棋盘上逐行放置皇后，并在每一行中找到一个安全的列位置，使得当前放置的皇后不会被已放置的皇后攻击。

算法设计与步骤

初始化

1. **棋盘表示**：通常使用一个一维数组 `board[N]`，其中 `board[i]` 表示第 i 行的皇后放置在哪一列。
2. **辅助数据结构**：为了快速检查是否可以在某一列放置皇后，可以使用三个辅助数组：
    - `cols[N]`：标记哪些列已被占用。
    - `diag1[2N-1]`：标记主对角线上的占用情况。主对角线索引计算公式为 `row - col + N - 1`。
    - `diag2[2N-1]`：标记副对角线上的占用情况。副对角线索引计算公式为 `row + col`。

递归放置皇后

3. **递归函数** `solve(row)`：
    - 如果 `row == N`，表示所有皇后都已成功放置，记录或输出解。
    - 否则，遍历每一列：
        - 检查在当前行 `row` 放置皇后的列 `col` 是否安全，即 `cols[col]`、`diag1[row - col + N - 1]` 和 `diag2[row + col]` 是否未被占用。
        - 如果安全，放置皇后并更新辅助数组。
        - 递归调用 `solve(row + 1)`。
        - 回溯：移除皇后，恢复辅助数组。

检查安全性

4. **安全性检查**：确保没有其他皇后可以攻击到当前位置，即列和对角线上没有其他皇后。

回溯

5. **回溯**：如果当前行的所有列都不安全，函数将回溯到上一行，移动那一行的皇后到下一个安全的位置。

例子：4皇后问题

考虑N=4的情况，解决过程中棋盘和辅助数组的变化如下：

- 尝试将皇后放在第0行的第0列，递归尝试第1行。
- 在第1行中，第0列不安全（同列冲突），第1列不安全（对角线冲突），尝试第2列，递归尝试第2行。
- 继续此过程，直到找到所有行的安全列或回溯到前一行尝试其他列。

这个方法会找到所有可能的解决方案，并可以通过递归深度（即行的深度）来可视化决策树的构建过程。

N皇后问题的解决方案数量随N的增加而增加，且问题规模的增加会导致计算时间显著增长。这种类型的问题是回溯算法中的一个经典示例，展示了算法如何通过试错，系统地探索问题的所有可能解决方案。

（三）分支限界法

分支限界法是一种用于解决优化问题和计算问题的系统方法，与回溯法类似，但通常用于求解最优化问题。这种方法的核心思想是系统地遍历或构建决策树的所有可能的候选解，通过“分支”来生成子候选解，并使用“限界”来避免不必要的搜索，即进行剪枝。

分支（Branching）

在分支限界法中，“分支”指的是从当前解决方案节点生成多个子节点的过程。这一步骤将解空间分为几个较小的子空间，即所谓的“子问题”，这些子问题代表了向最终解进一步的可能路径。分支通常按某种规则进行，以确保每个子节点都是一种可能且完整的解决方案路径的一部分。

例如，在旅行商问题（TSP）中，如果一个节点表示访问了一系列城市的路径，其子节点将代表在该路径末尾添加一个未访问城市的新路径。

限界（Bounding）

“限界”是分支限界法中用来避免探索不可能或不利的路径的技术。每个节点都会计算一个限界值（也称为成本估计），该值表示解决方案的最佳可能完成形式的估计。如果这个估计的限界值比已知的最佳解决方案差，那么这个节点及其所有子节点都不会进一步探索。

限界值的计算通常依赖于问题的特定性质。在优化问题中，限界通常涉及成本函数的最小值或最大值的估算。

基本思想

分支限界法使用广度优先或最佳优先搜索策略，通过动态地在解空间树中生成子节点（分支）和使用限界技术（剪枝）来避免无望的搜索，从而加速搜索过程。限界是通过计算下界来实现的，下界是对最优解的一个估计。如果某个节点的下界比已知的最优解还要差，那么这个节点及其所有子节点都不再考虑。

分支限界法的主要步骤

- 定义解空间（对解编码）
- 确定解空间树结构
- 按BFS方式进行搜索
    - 每个活结点仅有一次机会变成扩展结点
    - 由扩展结点生成一步可达的新结点
    - 在新结点中，删除不可能导出最优解的结点 // 限界策略
    - 将剩余的新结点加入队列中
    - 从队列中选择结点再扩展// 分支策略
    - 直到队列为空

常见的两种分支限界法

分支限界法是解决优化问题和确定性问题的一种系统方法，使用不同类型的队列管理扩展节点，主要有两种形式：队列式分支限界法和优先队列分支限界法。这两种方法在处理节点的方式和搜索策略上有所不同。

1. 队列式分支限界法（Breadth-First Branch and Bound）

队列式分支限界法使用普通队列来管理节点，采用广度优先搜索（BFS）策略。这种方法逐层扩展解空间树的所有节点，保证了每层的节点都被完全扩展，从而找到最优解。

实现过程：

- **初始化**：将根节点（问题的初始状态）放入队列。
- **循环处理**：从队列中取出节点进行扩展，直到队列为空。
  - **节点扩展**：生成当前节点的所有子节点。
  - **计算和评估**：对每个子节点计算成本和约束，确定它们的界限。
  - **检查界限**：如果子节点的界限优于当前最优解，将其加入队列。否则，进行剪枝，不加入队列。
- **更新最优解**：如果某个节点满足目标条件，且提供了一个更优的解决方案，更新当前最优解。
- **终止条件**：队列为空时结束搜索。

优点：

- 由于广度优先策略，总能找到最优解（如果存在）。
- 结构简单，实现容易。

缺点：

- 可能需要大量内存来存储所有待扩展的节点。
- 搜索速度可能较慢，因为要系统地扩展每一层的所有节点。

2. 优先队列分支限界法（Best-First Branch and Bound）

优先队列分支限界法使用优先队列（通常是最小堆）来管理节点，采用最优优先搜索策略。这种方法根据节点的优先级（如成本、估值等）来选择下一个扩展的节点，优先扩展最有希望的节点。

实现过程：

- **初始化**：将根节点放入优先队列。
- **循环处理**：从优先队列中取出最有希望的节点进行扩展，直到队列为空。
  - **节点扩展**：生成当前节点的所有子节点。
  - **计算和评估**：对每个子节点计算成本和估值，确定它们的界限。
  - **检查界限**：如果子节点的界限优于当前最优解，将其按优先级加入优先队列。否则，进行剪枝。
- **更新最优解**：如果某个节点满足目标条件且优于当前最优解，则更新。
- **终止条件**：优先队列为空时结束搜索。

优点：

- 更快找到最优解，因为总是先扩展最有希望的节点。
- 可以更有效地管理内存和搜索过程。

缺点：

- 实现相对复杂，需要合适的优先级评估函数。
- 优先队列的操作可能引入额外的计算成本。

剪枝搜索策略

在分支限界法中，剪枝是一个重要策略，用于减少搜索空间，提高算法效率。剪枝的基本思想是通过逻辑判断避免无效的搜索：

1. **可行性剪枝**：如果当前节点违反问题的约束条件，如超出容量或不满足需求等，该节点及其所有子节点被剪除。
2. **优化性剪枝**：基于成本或收益限界进行剪枝。如果从当前节点得到的最佳解决方案预计也不会优于已知的最佳解决方案，那么这个分支就会被剪掉。例如，在求解0-1背包问题时，如果当前节点的最高价值预测低于已知的最高价值，就可以放弃该分支。
3. **队列管理**：分支限界法通常使用队列来管理待探索的节点。根据问题的不同，可能使用先进先出队列（广度优先搜索）、优先队列（按成本函数排序，如最佳优先搜索），或其他类型的队列。
4. **实时更新**：当找到一个更好的解决方案时，更新当前的最优解决方案和限界值，从而可能导致更多的剪枝。

0-1背包问题是一类典型的组合优化问题，其中每个物品可以选择放入或不放入背包中，并且每个物品都有相应的重量和价值。目标是在不超过背包最大容量的前提下，最大化背包中物品的总价值。使用分支限界法求解0-1背包问题时，关键步骤之一是计算节点的上界，以便进行有效的剪枝。

常见的计算上界的基本思想

上界指的是在当前节点的决策基础上，如果后续决策都是最优的情况下可能获得的最大价值。上界用于评估当前分支的潜在价值，如果某个节点的上界还不如已知的最优解好，那么这个节点以及其所有子节点都可以被剪枝，不再进一步探索。

上界计算方法

1. **贪心法计算上界（Fractional Knapsack）**：
   - 当考虑一个节点时，已经根据之前的选择确定了一部分物品是否放入背包。
   - 对于剩下的物品，可以使用贪心算法来估算如果完全按照价值密度（价值/重量比）来选择的话，可以获得的最大价值。
   - 具体操作是：按照价值密度对剩余物品排序，然后尽可能多地选择价值密度高的物品放入背包，直到背包容量达到限制。如果遇到背包装不下的物品，可以考虑将这个物品分割，只放入其一部分（这一步在实际的0-1背包问题中是不允许的，但用于计算上界是可行的）。
2. **线性松弛法**：
   - 在线性规划中，通过松弛问题的整数约束（即允许物品分割），可以得到一个问题的线性松弛解，这个解提供了问题的理论最大上界。
   - 对于0-1背包问题，将每个物品的选择问题从0或1松弛为连续变量，这样可以计算出一个最优的、可能的分数解。

示例说明

假设背包容量为50，物品详情如下：

- 物品1：重量10kg，价值60
- 物品2：重量20kg，价值100
- 物品3：重量30kg，价值120

按照价值密度排序后的物品列表是：物品2（价值密度5），物品3（价值密度4），物品1（价值密度6）。

从背包容量和物品列表出发，首先选物品1完整放入（消耗10kg容量），再选物品2完整放入（消耗20kg容量），剩余容量20kg，但物品3需要30kg，所以按比例放入2/3，价值为80。

这样，上界计算得出的总价值是：60（物品1）+ 100（物品2）+ 80（物品3的部分）= 240。这是在当前节点下可能达到的最大价值。

使用这种上界计算方法，可以有效地减少分支限界法在解决0-1背包问题时的搜索空间，提高求解效率。

（四）、回溯法和分支限界法比较

- 求解目标不同：一般而言，回溯法求解目标是找出解空间树中满足约束条件的所有解，而分支限界法的求解目标则是尽快找出满足约束条件的一个解
- 搜索方法不同：回溯法采用深度优先搜索算法，分支限界法一般使用广度优先算法或最佳优先算法来搜索
- 对扩展结点的扩展方式不同：分支限界法中每一个活结点只有一次机会成为扩展结点，活结点一旦成为扩展结点就一次性产生其所有儿子结点
- 存储空间要求不同：分支限界法的存储空间比回溯法大得多，因此当内存容量有限时，回溯法成功的可能性更大

九、动态规划（这个一定要看ppt的例子）

1. 动态规划的两个关键属性
    - **重叠子问题**：使用递归算法的时反复求解相同的子问题，不停的调用函数，而不是生成新的子问题。如果递归算法反复求解相同的子问题，就称具有重叠子问题的性质
    - **最优子结构**：如果问题的最优解是由其子问题的最优解来构造，则称该问题具有最优子结构性质。使用动态规划算法时，要用子问题的最优解来构造原问题的最优解，因此必须考查最优解中用到的所有子问题。
2. 动态规划三要素：
    - 阶段（Stage）：把所给的问题的求解过程恰当地划分为若干个相互联系的阶段。
    - 状态（State）：表示每个阶段开始时，问题或系统所处的客观状况。状态既是该阶段的某个起点，又是前一个阶段的某个终点。通常一个阶段有若干个状态。状态的无后效性：如果某阶段状态给定后，则该阶段以后过程的发展不受该阶段以前各阶段状态的影响，也就是说状态具有马尔科夫性。

      注：适于动态规划法求解的问题具有状态的无后效性。
    - 策略（Policy）：各个阶段决策的确定后，就组成了一个决策序列，该序列称之为一个策略。由某个阶段开始到终止阶段的过程称为子过程，其对应的某个策略称为子策略。
3. **动态规划的思想实质**：分治和解决冗余。
4. Bellman最优性原理：求解问题的一个最优策略序列时，该最优策略序列的子策略序列总是最优的，则称该问题满足最优性原理。
5. 动态规划的基本步骤
    - **定义子问题**：将原问题分解为更小的子问题。
    - **实现递推关系**：找出子问题之间的关系，即状态转移方程。这个方程描述了如何从一个或多个较小的子问题的解得到另一个子问题的解。
    - **初始化条件**：确定递推关系中的边界条件，这些通常是最小的、最容易解决的子问题，它们为整个问题的解提供启动值。
    - **计算顺序**：确定子问题解决的顺序，以确保当解决一个子问题时，它所依赖的子问题已经解决。
    - **构建最终解**：利用子问题的解构建原问题的解。
    - **考虑存储问题**：在实现时考虑是否需要保留所有子问题的解，或者只需保留必要的部分以减少存储需求。
6. 动态规划实例：
    - 计算二项式系数
    - 最长公共子序列（LCS）
    - 动态矩阵乘法
    - 0-1 背包问题
    - 多阶段决策过程
    - `Warshall`传递闭包算法
    - 最短路径的弗洛伊德算法

动态规划实例过多，此处仅以背包问题为例，具体实例见ppt

实例：0-1背包

问题描述

假设我们有一个背包，最大容量为 5 千克。我们有以下物品可以选择装入背包：

- 物品 A: 重量 1 kg, 价值 60
- 物品 B: 重量 2 kg, 价值 100
- 物品 C: 重量 3 kg, 价值 120

目标是选择一组物品，使得在不超过背包重量限制的条件下，物品的总价值最大化。

动态规划解决方案

1. **定义状态**：
   - 定义 `dp[i][w]` 为考虑前 `i` 个物品，当背包容量为 `w` 时的最大价值。
2. **初始化**：
   - `dp[0][w] = 0` 对所有 `w`：没有物品时，背包的价值为0。
   - `dp[i][0] = 0` 对所有 `i`：背包容量为0时，不能装任何物品，价值为0。
3. **状态转移方程**：
   - 如果不装第 `i` 个物品，则 `dp[i][w] = dp[i-1][w]`。
   - 如果装第 `i` 个物品（前提是可以装下），则 `dp[i][w] = max(dp[i-1][w], dp[i-1][w-weight[i]] + value[i])`。

构建状态表

创建一个表，行表示物品（0表示不考虑任何物品），列表示背包容量从0到5。

i/w	0	1	2	3	4	5
0	0	0	0	0	0	0
1 (A)	0	60	60	60	60	60
2 (B)	0	60	100	160	160	160
3 (C)	0	60	100	160	180	220

解释表格：

- 第一行（i=0）和第一列（w=0）初始化为0，表示没有物品或容量时的价值。
- 第二行（考虑物品A）：
  - 容量1到5都能装下物品A，所以价值都是60。
- 第三行（考虑物品B）：
  - 容量1时，不能装物品B，所以价值与上一行相同，即60。

    - 容量2能装下B（价值100），因此取`max(dp[1][2], dp[1][0] + 100) = 100`。
    - 容量3及以上，可以同时装A和B，因此价值为160。
- 第四行（考虑物品C）：
    - 容量3时，只能装下C，所以取`max(dp[2][3], dp[2][0] + 120) = 160`。
    - 容量4时，可以装A和C（价值180），因此取`max(dp[2][4], dp[2][1] + 120) = 180`。
    - 容量5时，可以装下B和C，因此价值为220。

十、贪心策略（这个一定要看ppt的例子）

贪心算法是一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是全局最佳或最优的算法策略。贪心算法不保证会得到最优解，但在某些问题中，贪心策略产生的解是最优的。

贪心算法的特点

1. **局部最优选择**：在每个决策点，算法从可选的解决方案中选择一个局部最优的解决方案，不考虑问题的整体结构。
2. **无回溯**：一旦作出了选择，就不再重新考虑这个选择。这意味着贪心算法通常是不可撤销的，与动态规划和回溯算法相比，它不会回溯到之前的状态。
3. **简单高效**：贪心算法通常比其他技术更简单直观，容易实现，运行速度快。

贪心算法的适用场景

贪心算法适用于具有“贪心选择性质”和“最优子结构”的问题。

- **贪心选择性质**：可以通过做出局部最优的选择来构造全局最优的解决方案。换句话说，一个全局最优解包含了它构成中的局部最优解。
- **最优子结构**：一个问题的最优解包含其子问题的最优解。这意味着问题可以通过解决其子问题并组合这些解来解决。

贪心算法的实例

1. **活动选择问题**：给定一组活动，每个活动都有一个开始时间和结束时间。目标是选择最大数量的互不重叠的活动。解决方案是按照活动结束时间的早晚来选择活动。
2. **霍夫曼编码**：用于数据压缩的贪心算法。创建最优前缀代码，使得常见的字符使用较短的代码，不常见的字符使用较长的代码。
3. **最小生成树问题**：如Prim算法和Kruskal算法，这些算法贪心地选择边来确保在满足所有顶点被包含在树中的同时，总边的权重最小。
4. **单源最短路径问题**：如Dijkstra算法，该算法贪心地选择将最近的未处理顶点加入到已处理集合中，从而找到从源点到所有其他顶点的最短路径。

贪心算法的局限性

尽管贪心算法在很多问题上都非常高效，它并不总是提供最优解。一个典型的例子是找零问题：如果货币单位不是倍数关系，例如{1, 3, 4}，使用贪心算法可能无法得到最少的硬币数目。例如，要找零6，贪心算法会选择一个4和两个1，共3个硬币，但最优解是两个3，只需要2个硬币。

总结来说，贪心算法提供了一种快速解决问题的方法，适用于满足特定属性的优化问题。在实现时，关键是正确定义在每步中应如何进行贪心选择。

实例：找零钱问题

贪心算法在解决找零钱问题时的基本思想是尽可能优先使用面额大的硬币，以减少交付的总硬币数量。这种方法在硬币的面额是彼此倍数的货币系统中（如美元系统：1, 5, 10, 25, 100 等）通常能够得到最优解。

解决找零钱问题的步骤：

1. **排序**：首先将硬币按面额从大到小排序。
2. **初始化**：设置一个计数器或数组来记录每种面额的硬币使用数量，初始化需要找零的总金额。
3. **迭代选择**：从最大面额的硬币开始，重复以下步骤直到完成找零：
    - 确定可以使用该面额的最大数量，这通常是找零金额除以硬币面额的商。
    - 更新找零金额，减去已使用的该面额硬币的总价值。
    - 记录使用该面额硬币的数量。
4. **验证**：检查最终的找零金额是否减至零。如果是，输出每种面额的硬币使用数量；如果不是，表示给定的硬币面额无法精确找零。

算法复杂度分析：

- **时间复杂度**：贪心算法主要涉及对硬币进行排序和迭代使用硬币进行找零。假设硬币的种类数为 n 。排序硬币面额的时间复杂度为 $O(n \log n)$。在找零过程中，每种硬币至多被考虑一次，每次计算和更新找零金额的操作是常数时间 $O(1)$，因此这部分的时间复杂度为 $O(n)$。因此，整体的时间复杂度为 $O(n \log n)$，主要由排序步骤决定。
- **空间复杂度**：需要额外的空间来存储每种硬币的计数，即 $O(n)$，其中 n 是硬币种类的数量。因此，空间复杂度为 $O(n)$。

十一、几个对比

（一）对比迪杰斯科拉算法、Floyed算法、多段图算法、prim算法、克鲁斯卡尔算法

1. 迪杰斯特拉算法 (Dijkstra's Algorithm)

迪杰斯特拉算法是一种用于找到图中单个源点到其他所有顶点的最短路径的算法。它适用于带权重的有向图和无向图，权重不能为负。该算法的基本思想是逐步扩展最短路径集合，每次添加一个最近的未处理顶点到已处理集合中。

时间复杂度：

- 最坏情况：$O(V^2)$，其中 V 是顶点数。如果使用优先队列（如二叉堆），时间复杂度可以优化到 $O((V + E) \log V)$。

2. 弗洛伊德算法 (Floyd-Warshall Algorithm)

弗洛伊德算法是一种计算图中所有顶点对之间的最短路径的算法。它可以处理包含负权边的图，但不能处理负权环。算法的核心是动态规划，通过逐步考虑所有可能的中介顶点来更新路径长度。

时间复杂度：

- $O(V^3)$，其中 V 是顶点数。

3. 多段图算法 (Multi-Stage Graph Algorithm)

多段图算法是用于特定类型的图（多段图），在其中顶点和边被分为几个“段”。它通常用于解决分阶段决策过程中的最优路径问题，如在动态规划中。算法通过构建一个表来存储到每个点的最短路径，从而计算从起始点到终点的最短路径。

时间复杂度：

- $O(V + E)$，其中 V 是顶点数，E 是边数。

4. 普里姆算法 (Prim's Algorithm)

普里姆算法是用于在加权连通图中找到最小生成树的算法。它从一个顶点开始，逐渐增加边，直到覆盖所有顶点。每次迭代选择连接到已构建树的最小权重边。

时间复杂度：

- 如果使用优先队列，时间复杂度为 $O((V + E) \log V)$。

5. 克鲁斯卡尔算法 (Kruskal's Algorithm)

克鲁斯卡尔算法也是用来找到图的最小生成树的算法。它开始时视图中的每个顶点为一个独立的树，然后按权重排序所有的边，并逐个加入边，同时确保不形成环。

时间复杂度：

- 如果使用并查集，排序边的时间为 $O(E \log E)$ 或 $O(E \log V)$，算法总时间复杂度为 $O(E \log V)$。

（二）对比分治法和动态规划法

分治法（Divide and Conquer）和动态规划（Dynamic Programming）都是解决问题的有效算法策略，但它们在处理问题的方法和适用场景上有显著的区别：

分治法 (Divide and Conquer)

分治法是一种递归解决问题的策略，主要包括三个步骤：**分解**、**解决**和**合并**：

- **分解**：将原问题分解为若干个规模较小的相同问题。
- **解决**：递归地解决这些子问题；如果子问题足够小，则直接求解。
- **合并**：将所有子问题的解合并成原问题的解。

分治法适用于子问题相互独立的情况，即子问题的解决不依赖于其他子问题的解。典型的分治算法有快速排序、归并排序和二分搜索等。

动态规划 (Dynamic Programming)

动态规划用于解决具有重叠子问题和最优子结构性质的问题。在动态规划中，每个问题都是通过解决其子问题并结合这些子问题的解来求解的，这与分治法类似。然而，动态规划的不同之处在于其子问题是重叠的，即不同子问题可能包含共同的更小子问题：

- **存储子问题的解**：动态规划通常使用表格来存储子问题的解，避免重复计算。
- **构建解决方案**：从最小的子问题开始，逐步扩展到整个问题的解。

动态规划适用于那些具有大量重叠子问题的情况，如斐波那契数列的计算、背包问题、最长公共子序列等。

主要区别

- **子问题的重叠**：分治法中子问题相互独立，而动态规划中子问题是重叠的。
- **性能优化**：动态规划通过存储子问题的解来避免重复计算，提高效率；分治法则可能在没有必要的情况下重复解决相同的问题。
- **适用场景**：分治法适合解决可将问题等分的场景，动态规划适合解决需要反复计算相同子问题的场景。

（三）对比回溯法与穷举法

回溯法与穷举法的区别与联系

- 联系：都是基于试探搜索的方法
- 区别：
    - 穷举法要将一个解的各个部分全部生成后，才检查是否满足条件，若不满足，则直接放弃该完整解，然后再尝试另一个可能的完整解，它并没有沿着一个可能完整解的各个部分逐步回退生成解的过程。
    - 回溯法，一个解的各个部分是逐步生成的，当发现当前生成的某部分不满足约束条件时，就放弃该步所做的工作，退到上一步进行新的尝试，而不是放弃整个解重来。

（四）对比动态规划和贪心算法

动态规划和贪心策略的区别和联系：

- 子问题上：
  dp：每步所做的选择往往依赖于子问题的解，只有在解出相关子问题后才能作出选择
  贪心：仅在当前状态下作出最好选择，即 局部最优选择，然后再去作出这个选择后产生的相应的子问题，不依赖于子问题的解
- 求解方式：
  dp：通常以自底向上的方式解各子问题
  贪心：通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题

（五）对比分治法、减治法、变治法

1. 分治法 (Divide and Conquer)

定义：
分治法是一种将问题分解为几个规模较小的相似问题，递归解决这些子问题，然后将子问题的解合并以解决原问题的策略。

应用实例：

- 快速排序：将数组分为两部分，每部分各自排序后合并。
- 归并排序：同样是分割数组，但在合并阶段进行排序。
- 二分搜索：将搜索区域不断二分缩小查找范围。

特点：

- 子问题相互独立，不重叠。
- 解决问题通过分解、解决和合并步骤进行。

2. 减治法 (Decrease and Conquer)

定义：
减治法是指将原问题减少到一个更小的问题，解决这个小问题，然后扩展这个小问题的解以解决原问题。这种方法通常涉及到只生成一个子问题。

应用实例：

- 插入排序：每次从数组中取出一个元素，将其插入到已排序的序列中。
- 拓扑排序：每次找到一个没有前驱的顶点，将其从图中删除。
- 堆排序的堆化过程：从一个节点递归调整为最大堆或最小堆。

特点：

- 通常只产生一个子问题。
- 解决问题通常依赖于减小问题规模的策略。

3. 变治法 (Transform and Conquer)

定义：
变治法是通过变换将一个问题转化为另一个更容易解决的问题的策略。解决变换后的问题，然后将解转化回原问题的解。

应用实例：

- 高斯消元法：通过行变换将矩阵转换为简化行梯形形式来解线性方程组。
- 平衡二叉搜索树（如AVL树、红黑树）：通过旋转操作保持树的平衡，提高搜索效率。
- 散列技术：通过散列函数将数据映射到散列表上，提高数据查找效率。

特点：

- 主要依赖于问题的变换。
- 解决问题通常涉及到算法的改进或数据结构的优化。

对比总结

- **目标问题的处理**：分治法通过分解问题；减治法通过减小问题规模；变治法通过变换问题。
- **子问题的关系**：分治法处理多个独立的子问题；减治法只关注一个子问题；变治法转换问题后可能没有明显的“子问题”。
- **适用性和效率**：分治法适合于可以自然分解的问题；减治法适合于可以逐步简化的问题；变治法适合于可以有效变换求解方法的问题。

（十二）各个算法的主要例子

1. 递归算法：汉诺塔、斐波那契数列、阿克曼函数、阶乘、排列问题
2. 分治法：大整数乘法、Strassen算法求矩阵乘法、二分搜索、归并排序、快速排序、棋盘覆盖
3. 减治法：插入排序、拓扑排序、折半查找、假币问题、俄罗斯农民乘法问题、欧几里得算法（辗转相除法）
4. 变治法：预排序、高斯消元法、堆和堆排序、霍纳法则、线性规划、过河问题
5. 回溯法：0-1背包问题、TSP问题、排列生成问题、n皇后问题
6. 分支限界法：0-1背包问题、装载问题
7. 动态规划法：计算二项式系数最、长公共子序列（LCS）、动态矩阵乘法、0-1背包问题、多阶段决策过程、Warshall传递闭包算法、Floyed算法求最短路径
8. 贪心法：找零问题、背包问题、单源最短路径Dijkstra算法、最小生成树Kruskal算法