《算法设计与分析》各章知识点总结

主要是交待复习的范围和重点，文档设计的具体知识点仅供参考，请自行看PPT和课本复习。

使用支持markdown格式的应用程序打开本文档。

第1章 算法概述

- 算法基本概念
  - 定义：由有限指令组成的有穷序列，具有输入、输出、确定性和有限性
  - 程序与算法的区别：程序允许无限循环，算法必须终止

特性	定义	详细说明
确定性	算法中每一条指令的含义明确，无歧义，相同输入必得相同结果。	- 步骤顺序和执行逻辑严格定义（如“若A>B则执行X”而非“可能执行X”） - 避免模糊描述（如“适当处理”）。
有限性（有穷性）	算法必须在有限步骤内终止，且每一步骤的执行时间可接受。	- 排除无限循环或无法终止的操作（如未定义终止条件的递归） - 实际应用中需满足合理时间约束（如分钟级而非年数）。

- 算法复杂度分析
  - 时间/空间复杂度的渐近表示法（O, Ω, Θ, o）【结合期中第一题复习】
  - 最坏情况、平均情况、最好情况的复杂度分析
- 主定理计算时间复杂度

使用主定理计算时间复杂度的例子

主定理（Master Theorem）概述

主定理用于解决形如 **T(n) = aT(n/b) + f(n)** 的递归式的时间复杂度，其中：

- **a ≥ 1**（子问题的数量）
- **b > 1**（问题规模的缩减因子）
- **f(n)** 是合并步骤的时间复杂度。

示例 1：归并排序的递归式

递归式：
$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

参数分析：

- **a = 2**（每次递归分成 2 个子问题）
- **b = 2**（问题规模缩小为原来的 1/2）
- **f(n) = O(n)**（合并步骤的时间复杂度）

计算 logba：
$log_2 2 = 1$

比较 f(n) 与 nlogba：

- nlogba = n1 = n
- f(n) = O(n) 与 nlogba 同阶

应用情况 2：
$T(n) = \Theta(n \log n)$

示例 2：Strassen 矩阵乘法

递归式：
$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$

参数分析：

- **a = 7**（分解为 7 个子问题）
- **b = 2**（矩阵规模缩小为 1/2）
- **f(n) = O(n^2)**（合并子矩阵的时间复杂度）

计算 logba：
$log_2 7 \approx 2.81$

比较 f(n) 与 nlogba：

- nlog27 ≈ n2.81
- f(n) = O(n^2) 的阶数 **小于** n2.81

应用情况 1：
$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$

示例 3：二分搜索

递归式：
$T(n) = T\left(\frac{n}{2}\right) + O(1)$

参数分析：

- **a = 1**（每次递归分成 1 个子问题）
- **b = 2**（问题规模缩小为 1/2）
- **f(n) = O(1)**（比较操作的时间）

计算 logba：
$$log_2 1 = 0$$

比较 f(n) 与 nlogba：

- nlog21 = n^0 = 1
- f(n) = O(1) 与 n^0 同阶

应用情况 2：
`T(n) = \Theta(\log n)`

示例 4：快速排序的最优情况

递归式：
`T(n) = 2T\left(\frac{n}{2}\right) + O(n)`

参数分析：

- **a = 2**（每次递归分成 2 个子问题）
- **b = 2**（问题规模缩小为 1/2）
- **f(n) = O(n)**（分区操作的时间）

比较 f(n) 与 nlogba：

- 与归并排序相同，属于 **情况 2**
- 时间复杂度为 **Θ(n log n)**。

示例 5：递归式 T(n) = 3T(n/4) + n^2

参数分析：

- **a = 3**, **b = 4**, **f(n) = n^2**
- **log43 ≈ 0.792**

比较 f(n) 与 n0.792：

- f(n) = n^2 的阶数 **大于** n0.792
- 需验证正则条件：3f(n/4) ≤ c f(n)
  `3\left(\frac{n}{4}\right)^2 = \frac{3n²}{16} ≤ c n²`
  取 c = 3/16 < 1，满足条件。

应用情况 3：
`T(n) = \Theta(n^2)`

总结

通过主定理，可以快速判断递归算法的时间复杂度。关键在于：

1. 确定 **a**, **b** 和 **f(n)**。

2. 计算 **logba**。
3. 比较 **f(n)** 与 **nlogba** 的阶数。
4. 选择对应的主定理情况并应用。

表格总结：

递归式	参数 (a, b)	log_b a	f(n)	主定理情况	时间复杂度
T(n) = 2T(n/2) + n	(2, 2)	1	O(n)	情况 2	Θ(n log n)
$T(n) = 7T(n/2) + n^2$	(7, 2)	~2.81	$O(n^2)$	情况 1	Θ(n2.81)
T(n) = T(n/2) + 1	(1, 2)	0	O(1)	情况 2	Θ(log n)
$T(n) = 3T(n/4) + n^2$	(3, 4)	~0.792	$O(n^2)$	情况 3	$\Theta(n^2)$

第2章 递归与分治策略

- 递归算法设计
  - 递归函数的定义与实现（阶乘、Fibonacci数列、Hanoi塔问题）
  - 递归工作栈的原理与空间开销
- 分治法基本思想
  - 分解→解决→合并三步骤
  - 分治法的适用条件：子问题独立且与原问题性质相同
- 经典分治算法
  - 二分搜索：时间复杂度 O(log n)
  - 合并排序：时间复杂度 O(n log n) 的稳定排序

分治法框架（通用模板）

分治法遵循 **"分而治之"** 的核心思想，其通用框架可分为以下三步：

```
// 伪代码框架
ResultType divideAndConquer(ProblemType problem) {
    // 1. 递归终止条件
    if (problem is small enough) {
        return base_case_solution(problem);
    }

    // 2. 分解原问题为子问题
    SubProblemType subProblems = split(problem);
    SubProblemType sub1 = subProblems[0];
    SubProblemType sub2 = subProblems[1];
    // ... 可能分解为多个子问题（如棋盘覆盖分解为4个子问题）
```

```
    // 3. 递归解决子问题
    ResultType res1 = divideAndConquer(sub1);
    ResultType res2 = divideAndConquer(sub2);
    // ...

    // 4. 合并子问题的解
    return merge(res1, res2, ...);
}
```

分治法具体步骤详解

1. 分解 (Divide)

- **目标**：将原问题划分为 **规模更小、结构相同** 的子问题
- 示例
  ：
  - 归并排序中将数组分为左右两半
  - 棋盘覆盖中将棋盘分为4个子棋盘

2. 解决 (Conquer)

- **递归终止条件**：当子问题足够小时直接求解（如单元素排序）
- **递归调用**：对每个子问题调用自身

3. 合并 (Combine)

- **合并策略**：将子问题的解合并为原问题的解
- **复杂度关键**：合并操作的效率直接影响整体时间复杂度

经典示例：归并排序

```
void mergeSort(int arr[], int left, int right) {
    if (left >= right) { // 终止条件：单元素区间已有序
        return;
    }

    // 1. 分解：将数组分为两半
    int mid = left + (right - left) / 2;

    // 2. 递归解决子问题
    mergeSort(arr, left, mid);      // 排序左半部分
    mergeSort(arr, mid + 1, right); // 排序右半部分
```

```
    // 3. 合并有序子数组
    merge(arr, left, mid, right);
}
```

分治法适用条件

1. **子问题独立性**：子问题之间无重叠（动态规划处理重叠子问题更优）
2. **合并效率**：合并操作的时间复杂度需低于暴力解法
3. **问题可分性**：原问题可分解为更小的相同结构问题

常见分治算法

算法	分解方式	合并操作	时间复杂度
归并排序	数组一分为二	合并两个有序数组	O(n log n)
快速排序	根据pivot划分区间	无显式合并	O(n log n)
二分查找	每次舍弃一半区间	无显式合并	O(log n)
大整数乘法	分解为较小整数乘法	公式重组	O(n^1.585)

框架总结

分治法通过 **递归分解→解决→合并** 的流程，将复杂问题简化为可管理的子问题，是算法设计中解决大规模问题的经典范式。实际应用中需重点关注 **分解策略** 和 **合并效率** 的优化。

第3章 动态规划

- 动态规划要素
  - 最优子结构性质：问题的最优解包含子问题的最优解
  - 重叠子问题性质：通过备忘录或自底向上避免重复计算
- 经典动态规划问题
  - 矩阵连乘：最小化乘法次数的括号化方案
  - 最长公共子序列（LCS）：二维状态转移方程
  - 0-1背包问题：状态表示与物品选择策略
  - 图像压缩：分段存储的优化策略

0-1背包问题

问题描述

给定一组物品，每个物品有重量 `w_i` 和价值 `v_i`，以及一个容量为 `C` 的背包。要求在不超过背包容量的前提下，选择物品装入背包，使得总价值最大。
特点：每个物品只能选择 0次或 1次（不可分割）。

动态规划算法设计

1. 状态定义

设 `dp[i][j]` 表示前 `i` 个物品在容量为 `j` 的背包中能获得的最大价值。

2. 状态转移方程

对于第 `i` 个物品（1 ≤ i ≤ n），决策为选或不选：

- **不选第 `i` 个物品**：`dp[i][j] = dp[i-1][j]`
- **选择第 `i` 个物品**（需满足 `j \geq w_i`）：
  `dp[i][j] = \max(dp[i-1][j], \ dp[i-1][j - w_i] + v_i)`

递推公式：

$$dp[i][j] = \begin{cases} dp[i-1][j] & j < w_i \\ \max(dp[i-1][j],\ dp[i-1][j-w_i] + v_i) & j \geq w_i \end{cases} \tag{1}$$

3. 初始化条件

- 容量为 0时，价值为 0：`dp[0][j] = 0 \quad (0 \leq j \leq C)`
- 无物品可选时，价值为 0：`dp[i][0] = 0 \quad (0 \leq i \leq n)`

伪代码实现

```
int knapsack(vector<int>& w, vector<int>& v, int C) {
    int n = w.size();
    vector<vector<int>> dp(n + 1, vector<int>(C + 1, 0));

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= C; ++j) {
            if (j < w[i-1]) {
                dp[i][j] = dp[i-1][j];
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i-1][j - w[i-1]] + v[i-1]);
            }
        }
    }
    return dp[n][C];
}
```

时间复杂度与空间复杂度

- **时间复杂度**：`O(nC)` （n为物品数量，C为背包容量）
- **空间复杂度**：`O(nC)`

空间优化策略

观察状态转移方程发现，当前状态 `dp[i][j]` 仅依赖于前一行的状态 `dp[i-1][...]` ，因此可将二维数组压缩为一维数组，**逆序更新**容量值：

```
int knapsack_optimized(vector<int>& w, vector<int>& v, int C) {
    int n = w.size();
    vector<int> dp(C + 1, 0);

    for (int i = 1; i <= n; ++i) {
        for (int j = C; j >= w[i-1]; --j) {  // 逆序更新
            dp[j] = max(dp[j], dp[j - w[i-1]] + v[i-1]);
        }
    }
    return dp[C];
}
```

示例分析

输入：

- 重量数组 `w = [2, 3, 4, 5]`
- 价值数组 `v = [3, 4, 5, 6]`
- 背包容量 `C = 8`

递推过程（优化后的一维数组）：

物品	容量j	dp[j]更新过程（初始为[0,0,0,0,0,0,0,0,0]）
i=1	j=8→2	dp[8]=max(0, dp[6]+3)=3 → ... dp[2]=3
i=2	j=8→3	dp[8]=max(3, dp[5]+4)=7 → ... dp[3]=4
i=3	j=8→4	dp[8]=max(7, dp[4]+5)=9 → ... dp[4]=5
i=4	j=8→5	dp[8]=max(9, dp[3]+6)=10 → 最终结果10

最大价值：10（选择物品1、2、4）

关键点总结

1. **最优子结构**：当前状态由前一个物品的状态转移而来。
2. **逆序更新**：避免覆盖前一状态的值（优化空间复杂度）。

3. **适用场景**：物品数量较少且背包容量可接受时效率高，若容量极大需结合其他优化（如贪心+剪枝）。

第4章 贪心算法

- 贪心选择性质
  - 局部最优选择可导致全局最优解（需证明正确性）
- 经典贪心问题
  - 活动安排问题：按结束时间排序选择相容活动
  - 哈夫曼编码：构造最优前缀码的二叉树策略
  - 单源最短路径（Dijkstra算法）：优先队列优化
  - 最小生成树（Prim/Kruskal算法）：边权贪心选择

参考贪心作业题

第5章 回溯法

- 回溯法框架
  - 解空间树（子集树、排列树）的深度优先搜索
  - 剪枝策略：约束函数（可行性剪枝）与限界函数（最优性剪枝）
- 经典回溯问题
  - N皇后问题：对角线冲突检测与状态回溯
  - 0-1背包问题：基于价值密度剪枝的搜索优化
  - 旅行售货员问题（TSP）：路径代价限界剪枝

8皇后问题的回溯法

问题描述

在8×8的棋盘上放置8个皇后，使得任意两个皇后不在同一行、同一列或同一对角线上，求所有可能的合法布局。

回溯法思想

1. **逐行放置**：每次在棋盘的第`i`行放置一个皇后。
2. **冲突检测**：检查当前列、左上方对角线和右上方对角线是否已有皇后。
3. **剪枝回溯**：若当前位置冲突，跳过该列；若所有列尝试失败，回溯到上一行调整位置。

关键步骤

1. **数据结构**：使用一维数组`queens`，其中`queens[i]`表示第`i`行的皇后所在的列。

2.

冲突检查：对于当前行i和列col，需满足：

- 列不重复：`queens[j] != col`（对所有`j < i`）
- 对角线不重复：`abs(queens[j] - col) != i - j`（对所有`j < i`）

算法实现（Python示例）

```python
def solve_n_queens(n):
    def backtrack(row, queens):
        # 终止条件：所有行已放置皇后
        if row == n:
            result.append(queens.copy())
            return
        # 遍历当前行的所有列
        for col in range(n):
            # 检查是否冲突
            if is_valid(row, col, queens):
                queens.append(col)         # 放置皇后
                backtrack(row + 1, queens) # 进入下一行
                queens.pop()               # 回溯，撤销选择

    def is_valid(row, col, queens):
        # 检查当前列是否与已有皇后冲突
        for r in range(row):
            c = queens[r]
            if c == col or abs(c - col) == row - r:
                return False
        return True

    result = []
    backtrack(0, [])
    return result

# 测试8皇后问题
solutions = solve_n_queens(8)
print(f"共有 {len(solutions)} 种解")
```

复杂度分析

- **时间复杂度**：最坏情况下为`O(n!)`，但实际通过剪枝优化后远小于阶乘复杂度。
- **空间复杂度**：`O(n)`，用于存储当前皇后的列位置。

示例解

一种合法解的棋盘布局如下（`Q`表示皇后）：

```
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

通过回溯法遍历所有可能的列，最终找到全部92种解（包括对称解）。

第6章 分支限界法

- 分支限界法特点
  - 广度优先搜索或最小代价优先策略（优先队列实现）
  - 活结点表的维护与扩展（FIFO队列 vs 优先队列）
- 经典问题应用
  - 单源最短路径：优先队列优化的Dijkstra算法
  - 任务分配问题：代价矩阵的上下界估算
  - 最大团问题：剪枝策略与优先级排序
  - 电路板排列：最小延展长度优先搜索

对比维度	队列式分支限界法	优先队列式分支限界法
数据结构	普通队列（FIFO）	优先队列（堆结构，按优先级排序）
节点扩展顺序	按先进先出（FIFO）原则扩展节点	按优先级（如目标函数值、耗费等）选择最优节点扩展
搜索策略	广度优先搜索（BFS）	最小耗费优先（或最大效益优先）
时间复杂度	较高（可能需遍历更多节点）	较低（优先处理潜力节点，剪枝效率高）
空间复杂度	较高（需存储所有活节点）	较低（优先队列动态管理高优先级节点，减少无效存储）
适用场景	寻找可行解或问题规模较小时	寻找最优解（如最短路径、最大价值问题）或大规模问题
解的质量	可能较早找到可行解，但不一定最优	更快收敛到最优解（优先扩展潜力节点）
实现复杂度	简单（仅需队列操作）	较复杂（需维护优先队列的堆结构）
典型应用例子	装载问题的广度优先搜索	单源最短路径、0-1背包问题
是否需要剪枝策略	需要（通过约束条件剪枝）	需要（结合限界函数和优先级剪枝）

注：表格综合对比了两种方法的核心差异，具体选择需根据问题特性（如解的最优性需求、规模、时间空间限制等）。

第7章 随机化算法

- 随机化算法分类
  - **数值概率算法**：近似解高概率正确（例：π值计算）
  - **舍伍德算法**：消除最坏情况，平均性能优化（例：快速排序随机化版本）
  - **拉斯维加斯算法**：结果必然正确，但可能无法给出解（例：随机化素数测试）
  - **蒙特卡罗算法**：结果可能错误，但错误概率可控（例：随机化近似计数）
- 应用场景
  - 随机化快速排序：平衡划分避免最坏时间复杂度
  - 随机化选择算法：线性时间选择中位数

以下是对四种随机化算法的对比表格，总结其核心特点、应用场景及差异：

算法类型	核心思想	结果正确性	时间性质	优缺点	应用实例
数值概率算法	通过随机采样计算近似解，正确性以高概率保证。	高概率正确（非严格正确）	确定时间（固定计算量）	优点：速度快，适用于近似解需求；缺点：结果存在微小误差。	计算π值、积分近似、大规模数据统计估计
舍伍德算法	通过随机化消除输入对性能的影响，优化平均性能。	结果必然正确	平均时间优化（消除最坏情况）	优点：稳定平均性能； 缺点：无法改进最优情况下的时间。	随机化快速排序（避免最坏时间复杂度）、随机化线性选择算法
拉斯维加斯算法	保证结果正确，但可能因随机选择导致无法在有限时间内得到解。	必然正确（若给出解）	不确定时间（可能无法终止）	优点：结果严格正确； 缺点：可能无法找到解。	随机化素数测试（如Miller-Rabin算法的确定性版本）、随机化回溯法解决N皇后问题
蒙特卡罗算法	允许结果存在可控的错误概率，通过增加计算量降低错误率。	可能错误（但错误概率可控制）	确定时间（固定计算量）	优点：高效解决复杂问题； 缺点：结果不严格正确。	近似计数问题（如随机采样估计集合大小）、NP难问题的近似解

对比要点总结：

1. **正确性**：
    - **拉斯维加斯算法**：唯一严格保证结果正确的算法（若给出解）。
    - **蒙特卡罗算法**：允许错误，但可通过重复运行降低错误率。
    - **数值概率算法**：结果近似正确（高概率）。
    - **舍伍德算法**：结果正确，但关注性能优化。
2. **时间性质**：
    - **拉斯维加斯算法**：可能无法终止（如搜索失败）。
    - **其他算法**：均在确定时间内完成。
3. **适用场景**：

- **近似解需求** → 数值概率算法、蒙特卡罗算法。
- **严格正确性需求** → 拉斯维加斯算法、舍伍德算法。
- **消除最坏情况** → 舍伍德算法。

知识点关联对比

- **分治 vs 动态规划**
  分治法的子问题独立，动态规划的子问题重叠且需要记忆化存储。
- **贪心 vs 动态规划**
  贪心算法无后效性，动态规划需考虑所有子问题的相互影响。
- **回溯 vs 分支限界**
  回溯法深度优先+剪枝，分支限界法广度优先+优先级扩展。
- **随机化算法应用**
  通过引入随机性优化确定性算法的最坏情况表现（如快速排序）。
