算法整理

简答题整理

1、分治法适用的问题和求解思想

- **分治法适用的问题**
  - n取值相当大，直接求解往往非常困难，甚至无法求出。
  - 将n个输入分解成k个不同子集，得到k个不同的可独立求解的子问题。
  - 在求出子问题的解之后，能找到适当的方法把它们合并成整个问题的解。
- **分治法的思想**：将整个问题分成若干个小问题后分而治之。
  - 分(Divide)：子问题与原问题具有相同的特征，但规模更小。
  - 治(Conquer)：反复使用分治策略，直到可以直接求解子问题为止。（递归地求解）
  - 合(Combine)：将子问题的解组合成原问题的解。
- **具体问题**
  - 二分查找
  - 归并排序
  - 矩阵乘法（斯特拉森矩阵乘法）
  - 二维极大点

2、贪心法适用的问题和求解思想

- **贪心法适用的问题**：有这样一类问题：它有n个输入, 而它的解就是这n个输入的某个子集，这些子集必须满足某些事先给定的条件。
  - **约束条件**：必须满足的条件
  - **可行解**：满足约束条件的子集
  - **目标函数**：为了**衡量可行解的优劣**，预先给定衡量标准，以函数形式给出
  - **最优解**：**使目标函数取极值(极大值或极小值)的可行解**
- **贪心法的思想**：贪心方法是一种"只顾眼前"的分级处理方法。
  - **量度标准**：根据问题的性质，选择一个量度标准来评估每个输入的价值。
  - **选择输入**：按照量度标准，一次选择一个输入。
  - **构建解决方案**：如果当前选择的输入能与已有的部分解决方案结合，且满足约束条件，则将其加入到解决方案中；否则，放弃这个输入。
- **具体问题**
  - 带有期限的作业调度问题
  - 最小生成树问题
  - 背包问题（允许小数的）

3、动态规划法适用的问题和设计思想概要

- **动态规划法适用的问题特点**：

    - 有一类多阶段决策问题
        - 它可以被分成若干个阶段，每一个阶段都需要作出决策，i阶段的决策仅依赖于i阶段当前的状态
        - 当每个阶段的决策都确定后，问题得到一个决策序列，进而计算出一个解
    - 动态规划方法适用于解决求**最优解的多阶段决策问题**（也称为多阶段决策优化问题或组合优化问题）
        - 寻找最优解和最优决策序列（问题描述中一定存在目标函数）
- **动态规划法的设计思想概要**
    - 动态规划方法求解多阶段决策问题时：
        - 证明多阶段决策(最优)问题满足最优性原理
        - 设计递推关系式
        - 迭代实现程序
- **具体问题**
    - 多段图问题
    - 货郎担问题
    - 0/1背包问题
    - 可靠性设计问题
    - 最优二分检索树问题
    - 矩阵链乘积问题

4、回溯法适用的问题和设计思想

- **回溯法适用的问题特点**：
    - 方法适用于解决多阶段决策问题，也称为组合问题
        - 问题的解向量用元组来表示, 元素x_i通常取自于某个有穷集 S_i ,1≤i≤n, n表示问题规模
            - 固定长n-元组$(x_1, \ldots, x_n)$
            - 不定长k-元组$(x_1, \ldots, x_k)$, k≤n
        - 问题满足多米诺性质
        - 问题寻找满足约束条件的一个解或所有解（组合搜索）；有时则要寻找最优解（组合优化），此时问题中还存在目标函数
- **回溯法的设计思想**
    -
        - 针对问题**定义解空间树结构**：元组、显式约束条件、隐式约束条件。
        - 检验问题满足**多米诺性质。**
        - 以**深度优先方式搜索解空间树**, 在搜索过程中**使用限界函数（隐式约束条件）避免无效搜索。**
            - 首先根结点成为一个活结点, 同时也是当前的扩展结点。沿当前扩展结点向纵深方向移至一个新的活结点, 该活节点成为当前新的扩展结点。
            - 如果当前扩展结点不能再向纵深方向移动, 则其成为死结点。回溯至最近的一个活结点, 并使该活结点成为当前新的扩展结点。
        - 在解空间树中搜索, **直至找到所要求的解或解空间中已没有活结点时为止。**
- **具体问题**
    - n-皇后问题
    - 子集和数问题

- 图着色问题

- **影响回溯法求解效率的关键步骤**
  - 问题的多米诺性质是回溯法提高算法效率的关键（**如何合理地设置限界函数，在节省限界函数计算时间的同时，尽可能大地实现剪枝**）
  - **影响回溯法效率的因素**：
  - 生成下一个$X(k)$的时间
    - 生成一个结点的时间
  - 满足显式约束条件的$X(k)$的数目
    - 子结点的数量
  - 限界函数B_i的计算时间
    - 检验结点的时间
    - B_i能够大大减少生成的结点数，但在计算时间和减少程度上要进行折中
  - **对于所有的i，满足B_i的$X(k)$的数目**
    - **通过检验的结点数量**
    - **该因素会导致不同实例产生的结点数不同**

5、分支限界法适用的问题和设计思想

- **分支限界法适用的问题特点**：
  - 与回溯法类似，分支限界法同样适用于求解组合数较大的问题，特别是组合优化问题**(求最优解)。**
  - 分支限界法中，解向量的表达、显示约束条件、隐式约束条件、解空间和解空间树等概念均与回溯法相同
  - 两者主要区别在于E-结点(即扩展结点)处理方式不同

- **分支限界法的设计思想**
  - 针对问题定义解空间树结构：元组、显式约束条件、隐式约束条件。
  - 检验问题满足多米诺性质。
  - 假设当前寻找一个答案结点，按下列方式搜索解空间树：
    - 如果根结点T是答案结点，输出T，操作结束；否则令T是当前扩展结点E。
    - 生成E的所有儿子结点，判断每个儿子结点X：
      - 如果X是答案结点，输出到根的路径，操作结束；
      - 如果X满足约束函数B，则将X添加到活结点表中；否则舍弃X。
    - 从活结点表中选出下一个结点成为新的E-结点，重复上述操作。如果活结点表为空，则算法以失败结束。

- **具体问题**
  - 15谜问题
  - 带有期限的作业调度问题（从奖励变为惩罚）
  - 0/1背包问题

6、算法有哪些重要特性，算法分析主要包括哪两方面内容?

算法：一组有穷的规则，规定了解决某一特定问题的一系列运算。

算法的五个特性

- 确定性：每一种运算必须要有确切定义，无二义性。
- 能行性：运算都是基本运算，原理上能用纸和笔在有限时间完成。
- 输入：有0个或多个输入。在算法开始之前，从特定的对象集合中取值。
- 输出：一个或多个输出。这些输出和输入有特定关系。
- 有穷性：在执行了有穷步运算后终止。

算法分析主要包括两方面

- 事前分析(a priori analysis)
    - 确定每条语句的执行次数
    - 求出该算法的一个时间限界函数(关于问题规模n的函数)
- 事后测试(a posterior testing)
    - 作时空性能分布图

举例说明贪心算法中将目标函数直接作为度量标准不一定能得到最优解

背包问题： M=6; (p1,p2,p3)=(6,4,3); (w1,w2,w3)=(6,2,1);

取效益值最大的选了p1，此时背包已经无法再继续装载，(1,0,0)不是最优解，最优解是(0.5,1,1)

贪心方法设计求解的核心问题是什么？给出贪心方法解决背包问题的形式化描述，包括约束条件、可行解、目标函数和最优解

核心问题是选择能产生问题最优解的最优度量标准

贪心方法解决背包问题的形式化描述如下：

- **约束条件**：给定一组物品，每种物品都有自己的重量 (w_i) 和价值 (v_i)，以及一个最大承重为 (W) 的背包。目标是选择一些物品装入背包，使得总重量不超过 (W)。
- **可行解**：一个解决方案是物品的一个子集，其总重量不超过背包的最大承重 (W)。
- **目标函数**：对于背包问题，目标函数是最大化背包中物品的总价值，即 (\sum_{i=1}^{n} v_i x_i)，其中 (x_i) 是一个二元变量，表示物品 (i) 是否被选中（1 表示选中，0 表示未选中）。
- **最优解**：在所有可行解中，使目标函数达到最大值的解决方案。

贪心算法解决背包问题的步骤通常包括：

1. **初始化**：将所有物品按照单位重量的价值从大到小排序。
2. **选择**：依次考虑每个物品，如果当前背包的剩余容量可以容纳该物品，则将其加入背包。
3. **更新**：每选择一个物品后，更新背包的剩余容量。
4. **重复**：重复步骤2和3，直到背包装满或所有物品都已考虑完毕。

什么是最优性原理？举一个最优性原理不成立的例子

无论过程的初始状态或者初始决策是什么，其余的决策都必须相对于初始决策所产生的状态构成一个最优决策序列。

最优性原理不成立的例子：多段图问题（以乘法作为路径长度且出现负权边时）或 包含负长度环的任意两点间最短路径问题；

最优性原理成立的例子：流水线调度问题，货郎担问题；

回溯法和分支限界法的状态空间生成方式有何不同？简述两种方法的基本思想。

- **状态空间生成方式**
  - 回溯法: 当前的E-结点R 一旦生成一个新的儿子结点C, 这个C结点就变成一个新的E-结点, 当检测完了子树C后, R结点就再次成为E-结点, 生成下一个儿子结点。
  - 分支-限界方法：当前结点一旦成为E-结点，就一直处理到变成死结点为止。其生成的儿子结点加入到活结点表中，然后再从活结点表中选择下一个新的E-结点。
- **回溯法：**
  - 回溯法从根结点出发，按深度优先策略搜索解空间树。判断E-结点是否包含问题的解。如果不包含，则跳过以该结点为根的子树，逐层向其祖先结点回溯。否则就进入该子树，继续按深度优先的策略进行搜索。
- **分支限界法：**
  - 分支限界法从根结点出发，生成当前E-结点全部儿子之后，再选择新的活结点成为E-结点，重复上述过程。为提高算法效率，会使用限界函数和成本估计函数等进行剪枝。

（简述回溯法与分支限界法的主要区别）

解释下列术语的含义，回溯法运行过程中，状态空间树所有结点没有完全遍历到，但为什么没有丢解？

- **状态空间树**：基于解空间画成的树形状
- **显式约束**：限定每个x只从一个给定的集合上取值，可以与所求解问题的实例有关也可以无关
- **隐式约束**：规定问题的实例的解空间中的那些实际上满足规范函数的元组，描述了xi必须彼此相关的情况
- **问题状态**：解空间树中的每一个节点确定所求解问题的一个问题状态
- **解状态**：解状态是这样一些问题状态S，对于这些问题状态，由根到S的哪条路径确定了这解空间的一个元组。
- **答案状态**：答案状态是这样的一些解状态S，对于这些解状态而言，由根到S的这条路径确定了这问题的一个解（它满足隐式约束条件）
- **活结点**：如果已经生成一结点，而它的所有儿子结点还没有全部生成，则这个结点就叫活结点。
- **E结点**：当前正在生成其儿子节点的活结点
- **死结点**：不再进一步扩展或者其儿子结点已全部生成的结点
- **限界函数**：所求解的问题需要求一个是某一限界函数P（x1，x2……）取极大值（或取极小值或满足该限界函数）的向量，表示问题的约束条件或目标。
- 加限界的深度优先生成方法称为回溯法。回溯法在包含问题所有解的解空间树中, 按深度优先的策略, 从根结点出发搜索解空间树。搜索至解空间树的任一结点时, 先判断该结点是否肯定不包含问题的解。如果肯定不包含, 则跳过以该结点为根的子树, 逐层向其祖先结点回溯。所以没有遍历的结点都是没有解的，因此不会丢解。

给出LC分支-限界检索中活结点表使用的数据结构，并阐述LC分支-限界检索的检索策略。

随堂测试答案：堆。（因为指明了是LC分支限界）

LC分支限界检索的策略是这样的：

1. **初始化**：将根节点放入活结点表（优先队列）。

2. 循环：只要活结点表不为空，就继续执行以下步骤：
    - 从活结点表中选择**成本估计函数值最小/最大**的节点作为当前扩展节点（E-节点）。
    - 生成当前E-节点的所有子节点，并计算它们的成本估计函数值。
    - 将这些子节点按照成本估计函数值的大小插入到活结点表中。
    - 如果在任何时候找到了目标解，就结束搜索。
3. **终止**：当活结点表为空或找到目标解时，算法结束。

LC检索是否每次都能选出具有最小结点成本函数得结点作为下一个E结点？简述理由。

答：不能，要满足以下两点，LC检索才可以保证找到最小成本得答案结点（如果存在）

1.对于每一个结点X都有成本估计函数<=C(X)且对于答案结点有，C^(X)=C(X)

2.采用改进后的算法LC1,从活结点表中删除元素时判断是否为答案结点。

简述COOK定理。

COOK定理指出SAT问题是NP完全的，即任何NP问题都可以在多项式时间内归约为SAT问题。这表明，如果能在多项式时间内解决SAT问题，那么所有NP问题都能在多项式时间内解决，揭示了NP问题的内在联系。

简述P问题、NP问题、NPC问题，并解释P和NP之间，NP和NPC之间的关系。

1. P问题是所有可在多项式时间内用确定算法求解的判定问题的集合，即给定输入的大小为n，存在一个多项式p(n)，使得算法可以在O(p(n))时间内找到问题的解。
2. NP问题是所有可在多项式时间内用不确定算法求解的判定问题的集合，即能够用不确定算法在多项式时间里猜出一个解和验证一个解。
3. 一个问题是NPC问题，需要满足两个条件：1）问题L∈NP；2）任何NP问题都可以在多项式时间内归约为问题L。
4. P和NP之间的关系：所有P问题都是NP问题，即P ⊆ NP。换句话说，所有可以在多项式时间内用确定算法求解的问题，它们的解也可以在多项式时间内验证。但是，NP问题不一定都能在多项式时间内求解（即P不一定等于NP）。
5. NP和NPC之间的关系：NPC问题是NP问题中最难的一类，即所有NP问题都可以在多项式时间内归约为任何一个NPC问题。这意味着，如果能在多项式时间内解决一个NPC问题，那么所有NP问题都能在多项式时间内解决。因此，NPC问题是NP类中的核心问题，具有最高的复杂性。

简述什么是NP难问题，什么是NP完全问题，什么是可满足性问题。并阐明SAT问题是否为NP难问题，是否为NP完全问题

NP难问题和NP完全问题有什么区别?

请给出三个NP问题的例子。

NP问题：

- 0/1背包问题（NPC）
- 子集和问题（NPC）

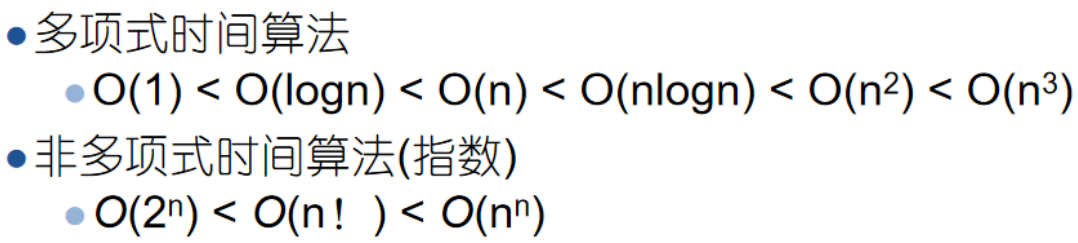
- 恰好覆盖问题（NPC）
- 旅行商问题（NPC）
- 图着色问题（NPC）

如何证明一个问题是NP-完全问题?

1. 证明问题L是NP问题：首先，需要证明问题L的解可以在多项式时间内验证，即给定一个候选解，我们可以在多项式时间内检查它是否是正确的解。
2. 已知一个 NPC-问题L', 证明 L'≤p L ： 证明一个已知的NPC问题L'可以多项式时间归约为问题L，即可以找到一个算法，能够将L'的任意实例转换为L的实例，并保持问题的解集不变，而这个归约过程需要在多项式时间内完成。

不同问题树结构的差异

时间复杂度整理



4 分治【存在疑问】

以比较为基础的检索算法的时间下界是O(logn);　【究竟是Ω还是O】
以比较为基础的分类算法的时间下界是Ω(nlogn);

4.1 二分查找

平均时间复杂度：$O(logn)$

4.2 归并排序

平均时间复杂度：$O(nlogn)$

4.3 斯特拉森矩阵乘法

$$T(n)=\begin{cases} O(1) & n=2 \\ 7T(n/2)+O(n^2) & n>2 \end{cases} \Leftarrow \boxed{T(n)=\begin{cases} \mathrm{b} & n=2 \\ 8T(n/2)+\mathrm{d}n^2 & n>2 \end{cases}}$$

时间复杂度：$O(n^{log_27})$ 或$O(n^{2.81})$

5 贪心

5.1 分数背包问题

时间复杂度：$O(n)$ （不考虑预排序的时间复杂度的情况下）

5.2 带有期限的作业调度问题

优化前：JS算法的时间复杂度：$O(n^2)$

优化后：算法的时间复杂度接近于$O(n)$

5.3 最小生成树问题

按照边来看：时间复杂度：$O(eloge)$

按照点来看：时间复杂度：$O(n^2logn)$

（连通无向图边的个数最多为n(n-1)/2）

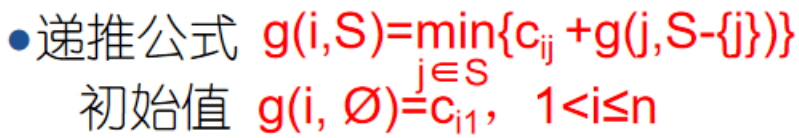
6 动态规划

6.1 多段图问题

向前处理法：COST(i, j)=min{ c(j, l)+ COST(i+1, l)}

- **时间复杂度：Θ(n+e)**
  - n是图G中节点个数。
  - e是图G中边的个数。
- **空间复杂度：Θ(n+e)**
  - **邻接表**的存储空间（多段图可以用邻接表存）
  - 除E外，还需要COST、D、P

6.2 货郎担问题【NPC】



时间复杂度：$\Theta(n^2 2^n)$

6.3 0/1背包问题【NPC】

$$f_i(X) = \max\{ f_{i-1}(X) , f_{i-1}(X-w_i)+p_i \}$$

$$f_0(X) = \begin{cases} -\infty & X<0 \\ 0 & X\ge 0 \end{cases}$$

二维数组：

- 时间复杂度：Θ(nM)
  - n是物品个数
  - M是背包容量
- 空间复杂度：Θ(nM)

一维数组：

- 时间复杂度：Θ(nM)
  - n是物品个数
  - M是背包容量
- 空间复杂度：
  - 只求问题最优解,空间复杂度：Θ(M)
  - 还需要知道问题的最优决策序列，空间复杂度：Θ(nM)

序偶对法：

- 时间复杂度：$O(min\{2^n, n\sum_{1\le j\le n} P_j, nM\})$
- 空间复杂度：$O(2^n)$

6.4 可靠性设计

6.5 最优二分检索树

优化前：

时间复杂度：$O(n^3)$

Knuth优化后：R(i,j-1)≤k≤ R(i+1,j)

时间复杂度：$O(n^2)$

7 回溯

与实际生成的结点个数有关

7.1 n-皇后问题

最坏情况下：$O(n!)$

7.2 子集和数问题【NPC】

最坏情况下：$O(2^n)$

7.3 图着色问题【NPC】

最坏情况下：$O(m^n)$ （m为颜色个数，n为顶点个数）

7.4 最大团问题

$O(n2^n)$

8 分支限界

8.1 15谜问题

8.2 带有期限的作业调度问题

8.3 0/1背包问题

随堂测试

2 导引

一. 单选题

1、(单选题, 5.0 分) 下列哪个选项不是算法必须满足的五个重要特性之一

A. 确定性

B. 能行性

C. 有穷性

D. 可读性

正确答案: D

五个特性：确定性；可行性；有穷性；输入；输出

2、(单选题, 5.0 分)分析算法时，需要构造数据集，下面说法错误的是

A. 需要构造出最好、最坏和有代表性的数据

B. 这是算法分析中非常重要且有创造性的工作

C. 通过构造出的数据集，能够更好的了解算法的性能

D. 数据集的规模越大，越有利于分析

正确答案: D

二. 判断题

1、(判断题, 5.0 分) 如果g(n) = Ω(f(n))， 则Ω(f(n))+Ω (g(n)) =Ω (f(n))。

A. 对

B. 错

正确答案: 对

2、(判断题, 5.0 分) 如果g(n) = Ω(f(n))， 则Ω(f(n))+Ω (g(n)) =Ω (g(n))。

A. 对

B. 错

正确答案: 对

3、(判断题, 5.0 分) Ω(cf(n)) = Ω(f(n))， 其中c是一个正的常数。

A. 对

B. 错

正确答案: 对

4、(判断题, 5.0 分) Θ(f(n))+Θ(g(n)) =Θ(min(f(n), g(n))

A. 对

B. 错

正确答案: 错

5、(判断题, 5.0 分) Θ(f(n))+Θ(g(n)) =Θ(max(f(n), g(n))

A. 对

B. 错

正确答案: 对

6、(判断题, 5.0 分) Θ(f(n))+Θ(g(n)) =Θ(f(n)+g(n))

A. 对

B. 错

正确答案: 对

三. 排序题

请按从小到大的顺序，排列下列算法的时间复杂度：

A. $O(n!)$

B. $O(n^3)$

C. $O(2^n)$

D. $O(n^2logn)$

正确答案：DBCA

4 分治

一. 单选题（共 5 题，25.0 分）

1、(单选题, 5.0 分) 以比较为基础的排序算法的时间下界是。

A. O(logn)

B. O(nlogn)

C. $\Omega(logn)$

D. $\Omega(nlogn)$

正确答案: D

2、(单选题, 5.0 分) 使用分治法设计算法，最重要的步骤是。

A. 最优性原理和最优量度标准

B. 子问题的划分和合并

C. 状态空间树和限界函数

D. 最优性原理和递推关系式

正确答案: B

3、(单选题, 5.0 分) 用分治法解决下列问题，时间复杂度最小的是。

A. 有序表查找问题

B. 排序问题

C. 斯特拉森矩阵乘法问题

D. 二维极大点问题

正确答案: A

A为O(logn)，B、D为O(nlogn)，C为$O(N^{log_27})$

4、(单选题, 5.0 分) 用斯特拉森矩阵乘法解决矩阵相乘问题，其算法的时间复杂性是。

A. O(n^log6)

B. O(n^log7)

C. O(n^log8)

D. O(n^log9)
正确答案: B

5、(单选题, 5.0 分) 用分治法解决有序表查找问题（给定X，确认X是否在表中），其时间复杂性是。

A. O(logn)

B. O(n)

C. O(nlogn)

D. O(n^2)
正确答案: A

二. 判断题（共 7 题，35.0 分）

1、(判断题, 5.0 分）通常情况下，分治法得到的子问题与原问题具有相同的类型。
A. 对
B. 错
正确答案: 对

2、(判断题, 5.0 分)二分查找算法是以比较为基础解决有序表查找问题最坏情况下的最优算法。

A. 对
B. 错
正确答案: 对

注意表述：“最坏情况下的最优算法”

3、(判断题, 5.0 分) 归并排序算法的时间复杂度是O(nlogn)。
A. 对
B. 错
正确答案: 对

4、(判断题, 5.0 分)斯特拉森矩阵乘法是针对矩阵加法运算过多的问题进行了改进。

A. 对
B. 错
正确答案: 错

主要创新在于减少了矩阵乘法运算的次数，而不是矩阵加法运算。原来的时间复杂度是O(n^3)，降到了O(n^2.81)

5、(判断题, 5.0 分) 对于有序表查找问题，最坏情况下，三分查找优于二分查找。

A. 对
B. 错
正确答案: 错

6、(判断题, 5.0 分) 分治法在任何情况下都优于蛮力法(直接求解)。
A. 对
B. 错
正确答案: 错

比如斯特拉森矩阵如果单纯的分治并没有降低时间复杂度

7、(判断题, 5.0 分) 利用分治策略解决二维极大点问题时，预排序后的算法时间复杂度能降到O(nlogn)。
A. 对
B. 错
正确答案: 对

5 贪心

一. 单选题（共 4 题，20.0 分）

1、(单选题, 5.0 分) 对于小数背包问题，设n=3, M=40, (p1,p2,p3)=(40,20,30), (w1,w2,w3)=(40,5,15)，下面哪个是解可以使得放进背包的物品效益和最大?
A.(1,1/2,1)

B. (1/2,1,1)

C. (1,1,1/2)

D. (0,1,1)

正确答案: B

按照p/w进行降序排列再求解

2、(单选题, 5.0 分)给定N个开区间(a,b)，从中选择尽可能多的开区间，使得这些开区间两两没有交集。应该如何设置贪心选择策略?

A. 按照a值从小到大考虑每个区间

B. 按照b值从小到大考虑每个区间

C. 按照a-b值从小到大考虑每个区间

D. A,B,C均可

正确答案: B

类似于活动安排问题，按照结束时间从早到晚进行排序。

3、(单选题, 5.0 分)对于作业排序问题，设J是4个作业的集合{J1,J2,J3,J4}，其中(p1,p2,p3,p4) = (100,10,15,20)，(d1,d2,d3,d4) = (2,1,2,3)，根据定理5.3下面哪个作业调度不可行?

A.J2, J1

B. J2, J1, J4

C. J1, J3

D. J3, J2, J4

正确答案: D

带有期限的作业调度问题的可行解判断，作业排列需要按照截止时间的递增次序进行排列才为可行解，A、B、C都满足，而D中不满足

4、(单选题, 5.0 分) 已知n表示图中点的个数，e表示图中边的个数，考虑对边的预排序，算法Kruskal的时间复杂度是:

A. O(nlogn)

B. O(n)

C. O(eloge)

D. O(e)

正确答案: C

二. 判断题（共 8 题，40.0 分）

1、(判断题, 5.0 分) 对于小数背包问题，贪心算法总能找到全局最优解。
A. 对
B. 错
正确答案: 对

2、(判断题, 5.0 分)对于小数背包问题，贪心算法的时间复杂度是Θ(n^2).

A. 对
B. 错
正确答案: 错

小数背包的贪心算法的时间复杂度是O(n)级别的（不考虑预排序）

3、(判断题, 5.0 分)求最小支撑树的算法Prim以及算法Kruskal，其本质都是贪心算法。

A. 对
B. 错
正确答案: 对

4、(判断题, 5.0 分)已知定理5.3：设J是k个作业的集合, б=i1,i2,...,ik是J中作业的一种排列, 它使得di1≤di2≤...≤dik。J是一个可行解, 当且仅当J中的作业可以按照б的次序而又不违反任何一个期限的情况来处理。当作业有不同处理时间时，该定理不成立。

A. 对
B. 错
正确答案: 错
答案解析：作业处理时间不同时，定理仍然成立，作业证明

5、(判断题, 5.0 分) 贪心法设计求解的核心问题是选择能产生问题最优解的量度标准
A. 对
B. 错
正确答案: 对

注意表述：**“能产生问题最优解”**的量度标准

6、(判断题, 5.0 分) 基于集合树解决带有期限的作业调度问题，能将算法的复杂度由O(n^3)降低到接近于O(n^2)

A. 对

B. 错

正确答案: 错

优化前为O(n^2)，优化后时间复杂度接近O(n)

7、(判断题, 5.0 分) 贪心法以目标函数为度量标准一定能够求出问题的最优解，如带有期限的作业调度问题。

A. 对

B. 错

正确答案: 错

前半句错误，后半句正确

8、(判断题, 5.0 分) 贪心法可以在多项式时间内求解出货郎担问题，目标函数就是问题的最优量度标准。

A. 对

B. 错

正确答案: 错

首先贪心法无法解决TSP问题，其次TSP是NPC问题

6 动态规划

一. 单选题（共 4 题，20.0 分）

1、(单选题, 5.0 分)使用动态规划策略设计算法，最重要的步骤是__.

A.最优性原理和最优量度标准

B.最优性原理和递推关系式

C.状态空间树和限界函数

D.子问题的划分和合并

正确答案: B

2、(单选题, 5.0 分)以下问题中最适合用动态规划方法求解的是__。

A.带有限期作业调度问题

B.8皇后问题

C.可靠性设计问题

D. 排序问题

正确答案: C

其中带有期限的作业调度问题为贪心或者分支限界法

3、(单选题, 5.0 分)以下问题中不适合用动态规划法求解的是__。

A. 0/1背包问题

B. 矩阵乘法问题

C. 构造最优二分检索树

D. 货郎担问题

正确答案: B

矩阵乘法问题一般用分治法解决，动态规划中的那个是矩阵链乘积问题（根据加括号影响乘法的顺序）

4、(单选题, 5.0 分) 关于动态规划方法，下面说法错误的是 __。

A. 若最优性原理对于所求解问题成立，则可以考虑采用动态规划方法解决该问题。

B. 最优性原理对所有的多阶段决策问题都成立。

C. 用向前处理法和向后处理法解决多段图问题，两者的时间复杂度相同。

D. 动态规划通过子问题的求解，有效利用已经得到的子问题结果，最大限度减少重复工作，以提高算法效率。

正确答案: B

B选项最优性原理并不适用于所有多阶段决策问题，只有当子问题的最优解能够保证整个问题的最优解时，最优性原理才成立。

二. 判断题（共 12 题，60.0 分）

1、(判断题, 5.0 分) 动态规划求解子问题的最优解时，求解过程并不关注子问题规模的大小，可在任意子问题规模之间变换。

A. 对

B. 错

正确答案: 错

动态规划问题不可以在任意子问题规模之间切换，一般是先求小规模的子问题再求较大规模的子问题。

2、(判断题, 5.0 分) 使用动态规划方法求解多阶段决策问题时，首先要确认问题满足最优性原理，然后设计递推关系式，算法实现优先考虑递归方式。

A. 对

B. 错

正确答案: 错

动态规划算法应该优先考虑迭代求解方式。

3、(判断题, 5.0 分) 使用动态规划法求解货郎担问题，时间复杂度为$\Theta(n^2 2^n)$

A. 对

B. 错

正确答案: 对

4、(判断题, 5.0 分)构造最优二分检索树是多阶段决策过程，可以使用动态规划法求解，并能在O(n^2)时间内构造出最优二分检索树。

A. 对

B. 错

正确答案: 对

使用Knuth优化后能在O(n^2)时间内构造出最优二分检索树

5、(判断题, 5.0 分) 动态规划法求解0/1背包问题，在最坏情况下，时间复杂度是指数阶的。

A. 对
B. 错
正确答案: 对

利用动态规划的确是O（n*w）的时间复杂度，但是要知道，n的确是输入规模的一部分，输入了n个重量与价值，但是w并不是输入规模，对于一个数W，需要m=log w的位数来表示。因此，m才是输入规模的一部分。所以O（n*w）=O（n2^m），所以是NPC问题。

6、(判断题, 5.0 分) 货郎担问题既可以用贪心法也可以用动态规划法来求最优解。

A. 对
B. 错
正确答案: 错

贪心法无法求解货郎担问题

7、(判断题, 5.0 分) 动态规划求解0/1背包问题时，最坏情况下就数量级而言，迭代法优于序偶对法。

A. 对
B. 错
正确答案: 错。0/1背包问题中，迭代法和序偶对法都可以用来求解，但没有一种方法在所有情况下都绝对优于另一种。通常，迭代法（也称为表格法）在空间复杂度上更高效，因为它可以通过滚动数组来优化。而序偶对法（也称为状态压缩法）在某些情况下可以减少计算量，但并不总是在数量级上优于迭代法。

8、(判断题, 5.0 分) 动态规划方法求解多阶段决策问题时，问题一定具备重叠子问题性质。
A. 对
B. 错
正确答案: 错。动态规划方法求解问题的一个关键特性是子问题的重叠，这意味着子问题的解会被重复利用。然而，并不是所有多阶段决策问题都具备重叠子问题性质。

9、(判断题, 5.0 分) 动态规划方法求解多阶段决策问题时，问题一定满足最优性原理。
A. 对
B. 错
正确答案: 对。存在前提“动态规划方法求解多阶段决策问题时”。

10、(判断题, 5.0 分) 动态规划求解多阶段决策问题时，重叠子问题规模越大数量越多，则算法迭代实现就越比递归实现的效率高。
A. 对
B. 错
正确答案: 对。在动态规划中，如果一个问题有大量的重叠子问题，那么迭代实现通常比递归实现更高效，因为迭代实现可以避免重复计算相同子问题的解，而递归实现可能会导致大量的重复计算。

11、(判断题, 5.0 分) 最优子结构性质是动态规划方法可以解决多阶段决策最优问题的判定依据。重叠子问题性质则是在递推关系式的算法实现环节起作用，对算法效率产生影响。
A. 对
B. 错
正确答案: 对。最优子结构性质（最优性原理）确保了动态规划方法可以应用于多阶段决策问题，而重叠子问题性质则影响算法的效率。通过存储子问题的解，动态规划可以避免重复计算，从而提高效率。

12、(判断题, 5.0 分) 矩阵链乘积问题不适用于动态规划方法求解。

A. 对
B. 错
正确答案: 错

8 回溯与分支限界

一. 判断题（共 5 题，25.0 分）

1、(判断题, 5.0 分) n皇后问题对于n>=4的每个值都有解。
A. 对
B. 错
正确答案: 对

2、(判断题, 5.0 分) 在最小成本的FIFO-分支限界算法中，使用成本估计函数并不能起到优化算法的作用。
A. 对
B. 错
正确答案: 错。可以通过成本估计函数和成本上界U进行比较，从而实现剪枝

3、(判断题, 5.0 分) 回溯法的解必须表示成一个n-元组（x1，x2，…，xn）。
A. 对
B. 错
正确答案: 错。既可以表示为定长的n-元组，也可以表示为不定长的k-元组。

4、(判断题, 5.0 分) 计算成本函数的工作量与解决原问题的工作量一样。
A. 对
B. 错
正确答案: 对（应该就是说并不会增加算法的最高时间复杂度）

5、(判断题, 5.0 分) FIFO分支限界法在生成动态树时，更接近于深度优先搜索。
A. 对
B. 错
正确答案: 错。FIFO分支限界法在生成动态树时，实际上更接近于广度优先搜索（BFS）。这是因为FIFO（先进先出）的队列结构会优先扩展最早生成的节点，这与BFS的搜索顺序相符。

二. 单选题（共 4 题，20.0 分）

1、(单选题, 5.0 分) 对于问题状态S，由根到S的那条路径确定了这个解空间中的一个元组，那么状态S称为___。
A. 解状态
B. 答案状态
C. 目标状态
D. 问题状态
正确答案: A

2、(单选题, 5.0 分) 与实例相关的状态空间树称为
A. 静态树
B. 动态树
C. 状态空间树
D. 查找树
正确答案: B

3、(单选题, 5.0 分)在状态空间树中，从根结点开始然后生成其它结点。如果已生成一个结点而它的所有儿子结点还没有全部生成，则这个结点一定是___。

A. 活结点
B. 死结点
C. E结点
D. 正在扩展的结点
正确答案: A

4、(单选题, 5.0 分) 下列哪个策略不能求解0/1背包问题?
A. 回溯法
B. 分支限界法
C. 动态规划法
D. 贪心法
正确答案: D。**贪心法**通常不能用来求解0/1背包问题，因为贪心法在每一步都做出局部最优选择，而0/1背包问题要求全局最优解。贪心法可能无法考虑到后续选择对当前选择的影响，因此可能无法找到最优解。相反，回溯法、分支限界法和动态规划法都能够找到0/1背包问题的最优解。

三. 填空题（共 4 题，25.0 分）

1、(填空题, 5.0 分) 用回溯法解决3个顶点的3种颜色的图着色问题，在最坏情况下将生成__个结点
正确答案：
(1) 40

n个顶点，m种颜色的图着色问题。m意味着m叉树，n代表有几层。

3个顶点，3种颜色，所以共有1+3+3*3+3*3*3=40个结点。

2、(填空题, 5.0 分) 在LC分支限界法中，用__数据结构存储活结点。
正确答案：
(1) 堆

已经指明为LC分支限界，所以用极大堆/极小堆优先队列来存储活结点。（FIFO分支限界法用队列）

3、(填空题, 5.0 分) 不考虑任何隐式约束条件，回溯算法硬处理解决n-皇后问题时，算法的时间复杂度是　　。
正确答案：
(1)O（n^n）

注意不考虑任何隐式约束条件，所以是O(n^n)，如果考虑了隐式约束条件，则为O(n!)

4、(填空题, 10.0 分) n=4时，子集和问题定长元组的状态空间树中，问题状态__个， 解状态__个
正确答案：
(1) 31
(2) 16

子集和问题的树结构在定长元组的情况下为完全2叉树，所以问题状态个数是所有结点总数即（2^6+2^5）=31个，解状态就是叶子结点个数即2^4=16个

9 NP

一. 单选题（共 4 题，20.0 分）

1、(单选题, 5.0 分) 关于P问题、NP问题、NP完全问题，下面说法正确的是：
A. P=NP
B. 有的NP问题无法约化为可满足性问题
C. NP完全问题都是NP问题
D. NP问题都是NP完全问题
正确答案: C

2、(单选题, 5.0 分) 下面说法错误的是：
A. 如果SAT问题存在多项式时间算法，则所有NP问题都可以在多项式时间内解决。
B. 非P问题都能转化为NP问题。
C. NP问题是可在多项式时间内用不确定算法验证的判定问题。
D. 一个优化问题通常可以表示为一个判定问题。

正确答案: B

非P问题指的是不能在多项式时间内解决的问题，而NP问题是指可以在多项式时间内验证其解的问题。并不是所有非P问题都属于NP类别，因为有些问题可能连验证解都无法在多项式时间内完成。

3、(单选题, 5.0 分) 如果存在NP-难问题L’，使得 L’ ≤p L，则L 是
A. P 问题
B. NP问题
C. NPC问题
D. NP-难问题
正确答案: D

4、(单选题, 5.0 分) 如果存在NPC-问题L’，使得L’ ≤p L, 且L∈NP, 则L是
A. P问题
B. NP问题
C. NPC问题
D. NP-难问题
正确答案: C

证明题

程序题

1. 分治法（二分查找）

   数组先递减后递增找到最小值

   > 力扣162.寻找峰值

```cpp
class Solution {
public:
    int findPeakElement(vector<int> &nums) {
        int left = -1, right = nums.size() - 1; // 开区间 (-1, n-1)
        while (left + 1 < right) { // 开区间不为空
            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[mid + 1]) right = mid; // 蓝色
            //本题将'>'改成'<'即可符合题意
            else left = mid; // 红色
        }
        return right;
    }
};
```

2. 动态规划（商店选址问题）

在一条呈直线的公路两旁有n个位置x1, x2,...xn可以开商店,在位置xi处开商店的预期收益是pi, i=1,2,... n.如果任何两个商店之间的距离必须至少为d，那么如何选择开设商店的位置使得总收益达到最大?

> 类似力扣198.打家劫舍
>
> 题目如下
>
> 你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。**
>
> 给定一个代表每个房屋存放金额的非负整数数组，计算你 **不触动警报装置的情况下** ，一夜之内能够偷窃到的最高金额。
>
> **示例 1：**
>
> ```
> 输入：[1,2,3,1]
> 输出：4
> 解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
>   偷窃到的最高金额 = 1 + 3 = 4 。
> ```
>
> **示例 2：**
>
> ```
> 输入：[2,7,9,3,1]
> 输出：12
> 解释：偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
>   偷窃到的最高金额 = 2 + 9 + 1 = 12 。
>
> ```
>
> **题解**

```cpp
class Solution {
public:
 int rob(vector<int> &nums) {
     int n = nums.size();
     vector<int> f(n + 2);
     for (int i = 0; i < n; ++i)
         f[i + 2] = max(f[i + 1], f[i] + nums[i]);
     return f[n + 1];
 }
};
```

将题解中的f数组初始条件设置一下，再将递推方程修改为**f[i]=max(f[i-1],f[i-d]+nums[i])**即可

力扣热题整理

动态规划

- 198.打家劫舍

> 你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。**
>
> 给定一个代表每个房屋存放金额的非负整数数组，计算你 **不触动警报装置的情况下** ，一夜之内能够偷窃到的最高金额。
>
> **示例 1：**
>
> ```
> 输入：[1,2,3,1]
> 输出：4
> 解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
>   偷窃到的最高金额 = 1 + 3 = 4 。
> ```
>
> **示例 2：**
>
> ```
> 输入：[2,7,9,3,1]
> 输出：12
> 解释：偷窃 1 号房屋 (金额 = 2)，偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
>   偷窃到的最高金额 = 2 + 9 + 1 = 12 。
> ```
>
> **提示：**
>
> - `1 <= nums.length <= 100`
> - `0 <= nums[i] <= 400`

题解

```cpp
class Solution {
public:
    int rob(vector<int>& nums) {
        int n=nums.size();
        for(int i=1;i<n;++i){
        if(i==1)
            nums[i]=max(nums[i],nums[i-1]);
        else nums[i]=max(nums[i-1],nums[i-2]+nums[i]);
        }

        return nums[n-1];
    }
};
```

- 279.完全平方数

> 给你一个整数 `n` ，返回 *和为 `n` 的完全平方数的最少数量* 。
>
> **完全平方数** 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，`1`、`4`、`9` 和 `16` 都是完全平方数，而 `3` 和 `11` 不是。
>
> **示例 1：**
>
> ```
> 输入：n = 12
> 输出：3
> 解释：12 = 4 + 4 + 4
> ```
>
> **示例 2：**
>
> ```
> 输入：n = 13
> 输出：2
> 解释：13 = 4 + 9
> ```
>
> **提示：**
>
> - `1 <= n <= 104`

题解

```cpp
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 0; i <= n; i++) { // 遍历背包
            for (int j = 1; j * j <= i; j++) { // 遍历物品
                dp[i] = min(dp[i - j * j] + 1, dp[i]);
            }
        }
        return dp[n];
    }
};
```

- 322.零钱兑换

> 给你一个整数数组 `coins` ，表示不同面额的硬币；以及一个整数 `amount` ，表示总金额。
>
> 计算并返回可以凑成总金额所需的 **最少的硬币个数** 。如果没有任何一种硬币组合能组成总金额，返回 `-1` 。
>
> 你可以认为每种硬币的数量是无限的。
>
> **示例 1：**
>
> ```
> 输入：coins = [1, 2, 5], amount = 11
> 输出：3
> 解释：11 = 5 + 5 + 1
> ```
>
> **示例 2：**
>
> ```
> 输入：coins = [2], amount = 3
> 输出：-1
> ```
>
> **示例 3：**
>
> ```
> 输入：coins = [1], amount = 0
> 输出：0
> ```
>
> **提示：**
>
> - `1 <= coins.length <= 12`
> - `1 <= coins[i] <= 231 - 1`
> - `0 <= amount <= 104`

题解

```cpp
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        int n=coins.size();
        sort(coins.begin(),coins.end());
        int dp[amount+1];
        memset(dp,0x3f,sizeof(dp));
        dp[0]=0;
        for(int i=1;i<=amount;++i){
            for(int j=0;j<n;++j){
                if(coins[j]>i)
                    break;
                dp[i]=min(dp[i],dp[i-coins[j]]+1);
            }
        }
        return dp[amount]==0x3f3f3f3f?-1:dp[amount];
    }
};
```

- 139.单词拆分

> 给你一个字符串 `s` 和一个字符串列表 `wordDict` 作为字典。如果可以利用字典中出现的一个或多个单词拼接出 `s` 则返回 `true`。
>
> **注意：**不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。
>
> **示例 1：**
>
> ```
> 输入: s = "leetcode", wordDict = ["leet", "code"]
> 输出: true
> 解释: 返回 true 因为 "leetcode" 可以由 "leet" 和 "code" 拼接成。
> ```
>
> **示例 2：**
>
> ```
> 输入: s = "applepenapple", wordDict = ["apple", "pen"]
> 输出: true
> 解释: 返回 true 因为 "applepenapple" 可以由 "apple" "pen" "apple" 拼接成。
>   注意，你可以重复使用字典中的单词。
> ```
>
> **示例 3：**
>
> ```
> 输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
> 输出: false
> ```
>
> **提示：**
>
> - `1 <= s.length <= 300`
> - `1 <= wordDict.length <= 1000`
> - `1 <= wordDict[i].length <= 20`
> - `s` 和 `wordDict[i]` 仅由小写英文字母组成
> - `wordDict` 中的所有字符串 **互不相同**

题解

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        auto wordDictSet = unordered_set <string> ();
        for (auto word: wordDict) {
            wordDictSet.insert(word);
        }

        auto dp = vector <bool> (s.size() + 1);
        dp[0] = true;
        for (int i = 1; i <= s.size(); ++i) {
            for (int j = 0; j < i; ++j) {
                if (dp[j] && wordDictSet.find(s.substr(j, i - j)) != wordDictSet.end()) {
```

```
                    dp[i] = true;
                    break;
                }
            }
        }

        return dp[s.size()];
    }
};

```

- 300.最长递增子序列

> 给你一个整数数组 `nums` ，找到其中最长严格递增子序列的长度。
>
> **子序列** 是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，`[3,6,2,7]` 是数组 `[0,3,1,6,2,2,7]` 的子序列
>
> **示例 1：**
>
> ```
> 输入：nums = [10,9,2,5,3,7,101,18]
> 输出：4
> 解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。
> ```
>
> **示例 2：**
>
> ```
> 输入：nums = [0,1,0,3,2,3]
> 输出：4
> ```
>
> **示例 3：**
>
> ```
> 输入：nums = [7,7,7,7,7,7,7]
> 输出：1
> ```
>
> **提示：**
>
> - `1 <= nums.length <= 2500`
> - `-104 <= nums[i] <= 104`

题解

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        int n=nums.size();
        vector<int> ans(n,1);
        for(int i=1;i<n;++i){
            for(int j=0;j<i;++j){
                if(nums[j]<nums[i])
                    ans[i]=max(ans[i],ans[j]+1);
            }
        }
        return *max_element(ans.begin(),ans.end());
```

```
    }
};
```

- 152.乘积最大子数组

> 给你一个整数数组 `nums` ，请你找出数组中乘积最大的非空连续
>
> 子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。
>
> 测试用例的答案是一个 **32-位** 整数。
>
> **示例 1:**
>
> ```
> 输入: nums = [2,3,-2,4]
> 输出: 6
> 解释: 子数组 [2,3] 有最大乘积 6。
> ```
>
> **示例 2:**
>
> ```
> 输入: nums = [-2,0,-1]
> 输出: 0
> 解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。
> ```
>
> **提示:**
>
> - `1 <= nums.length <= 2 * 104`
> - `-10 <= nums[i] <= 10`
> - `nums` 的任何前缀或后缀的乘积都 **保证** 是一个 **32-位** 整数

题解

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        vector <int> maxF(nums), minF(nums);
        for (int i = 1; i < nums.size(); ++i) {
            maxF[i] = max(maxF[i - 1] * nums[i], max(nums[i], minF[i - 1] * nums[i]));
            minF[i] = min(minF[i - 1] * nums[i], min(nums[i], maxF[i - 1] * nums[i]));
        }
        return *max_element(maxF.begin(), maxF.end());
    }
};
```

- 416.分割等和子集

> 给你一个 **只包含正整数** 的 **非空** 数组 `nums` 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
>
> **示例 1:**

```
输入：nums = [1,5,11,5]
输出：true
解释：数组可以分割成 [1, 5, 5] 和 [11] 。
```

示例 2：

```
输入：nums = [1,2,3,5]
输出：false
解释：数组不能分割成两个元素和相等的子集。
```

提示：

- `1 <= nums.length <= 200`
- `1 <= nums[i] <= 100`

题解

```cpp
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int n=nums.size();
        int sum=0;
        for(int num:nums){
            sum+=num;
        }
        if(sum%2)
            return false;
        int target=sum/2;
        int f[n+1][target+1];
        memset(f,0,sizeof(f));
        f[0][0]=0; //边界条件
        for(int i=0;i<n;++i){
            for(int c=0;c<=target;++c){
                if(c<nums[i]){
                    f[i+1][c]=f[i][c];
                }
                else
                    f[i+1][c]=max(f[i][c],f[i][c-nums[i]]+nums[i]); //0-1背包模板
            }
        }
        if(f[n][target]==target)//
            return true;
        else
            return false;
    }
};
```

- 32.最长有效括号

> 给你一个只包含 `'('` 和 `')'` 的字符串
>
> 找出最长有效（格式正确且连续）括号子串的长度。

示例 1：

```
输入：s = "(()"
输出：2
解释：最长有效括号子串是 "()"
```

示例 2：

```
输入：s = ")()())"
输出：4
解释：最长有效括号子串是 "()()"
```

示例 3：

```
输入：s = ""
输出：0
```

提示：

- `0 <= s.length <= 3 * 104`
- `s[i]` 为 `'('` 或 `')'`

题解 *由于本人用的是栈方法而非动态规划，贴一个题解链接*

32. 最长有效括号 - 力扣（LeetCode）

二分查找

- 74.搜索二维矩阵

给你一个满足下述两条属性的 `m x n` 整数矩阵：

- 每行中的整数从左到右按非严格递增顺序排列。
- 每行的第一个整数大于前一行的最后一个整数。

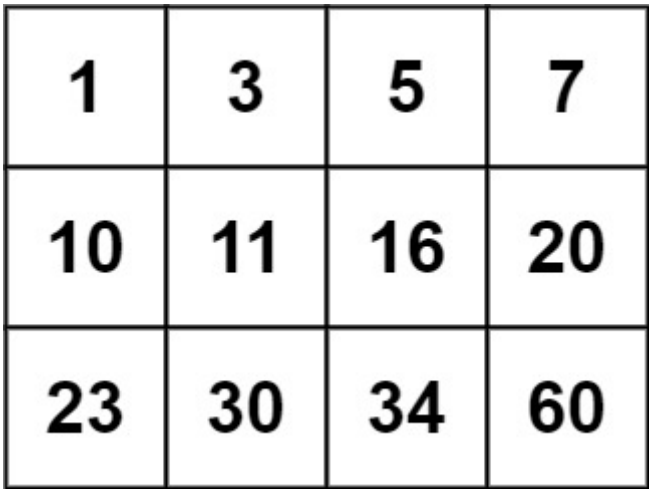
给你一个整数 `target` ，如果 `target` 在矩阵中，返回 `true` ；否则，返回 `false` 。

示例 1：

1	3	5	7
10	11	16	20
23	30	34	60

```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
输出：true
```

示例 2：




```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13
输出：false
```

提示：

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= m, n <= 100`
- `-104 <= matrix[i][j], target <= 104`

题解

```cpp
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m = matrix.size(), n = matrix[0].size();
        int left = -1, right = m * n;
        while (left + 1 < right) {
            int mid = left + (right - left) / 2;
            int x = matrix[mid / n][mid % n];
            if (x == target) {
                return true;
            }
            (x < target ? left : right) = mid;
        }
        return false;
    }
};
```

复杂度分析

时间复杂度：O(log(mn)) , 其中 m 和 n 分别为 matrix 的行数和列数。

空间复杂度：O(1)