算法设计与分析

希君生羽翼
一化北溟鱼
向更远的远方
加油
我的朋友

目录

第 1 章 算法模拟卷 A

1.1 判断题

1. 分治必须用递归实现 ( )
2. 问题的最优子结构性质是指问题的最优解包含子问题的最优解 ( )
3. 动态规划适合求解动态不确定性问题 ( )
4. 扩展节点的选择影响分支限界法 ( )
5. 问题 A 的上限是 $O(n^2)$，若 A 可以在 $O(n)$ 的时间内转化为 B 问题，则问题 B 的上限还是 $O(n^2)$ ( )

1.2 选择题

1. 若 $f(n) = O(g(n))$，则 $f(n)$ 的阶____ $g(n)$ 的阶

   A. 不大于　　B. 不小于　　C. 等价　　D. 逼近

2. 回溯法搜索有 n 个结点的排列树最坏情况下的时间复杂度 ____

   A. $n!$　　B. n^n　　C. 2^{n+1}　　D. $2^{n+1} - 1$

3. 单源最短路径迪杰斯特拉算法采用了____ 策略

   A. 贪心　　B. 动态规划　　C. 分治　　D. 回溯

4. 为提高回溯法和分支限界法的效率，通常可以采用界限函数剪____

   A. 不包含问题解的子树　　B. 不满足约束条件的子树

   C. 不满足问题最优解的子树　　D. 优先级较小的子树

5. 证明一个问题 Q 是 NP 完全问题的过程是，先证明 Q 属于 NP，再使用一个 NP 完全问题 L，结合条件____

   A. L 在多项式时间转化为 Q　　B. Q 在多项式时间转化为 L

   C. L 属于 Q　　D. Q 属于 L

1.3 分析题

1. 写出分治、动态规划、贪心、回溯算法的策略
2. 旅行商问题，n 个点，从点 i 出发的最短路为 $\min(i)$，试证明：最小回路路径和 $\geq \sum \min(i)$；

   设回路路径有前缀路径 $x[1:k]$，则路径和不小于

   $$\sum x[1:k] + \sum \min(x[k+1, n])$$

3. 现在有一个序列 $W[1:n]$，包含 n 个不相同的正整数，给定一个正整数 m，求出 W 的所有和为 m 的子集。（使用回溯法）

   (a). 定义问题的解向量

   (b). 写出显/隐约束

   (c). 给定 $W[4389], m(11)$，画出问题的解空间树
4. 将一个正整数分解解为若干个正整数，并使这些正整数之积最大。例如 $10 = 2+2+3+3$。

   (a). 写出贪心策略

   (b). 证明策略的贪心选择性质

1.4 算法设计题

1. 有序的数列 $A[0:n-1]$，数据不重复，设计一个高效的分治算法求解唯一满足 $A[i] = i$ 的位置，并分析算法时间复杂度
2. 利用动态规划思想，设计一个算法（不超过 $\Theta(n^2)$），求一个正整数序列的最长递增子序列。

   (a). 写出递归式

   (b). 写出算法
3. 目前拥有 1 元、3 元、9 元、27 元的纸币若干，请你设计一个算法使得花最少的张数凑

得总金额 m。

(a). 假设 $m = 35$，请设计并写上一种贪心算法使得满足原问题的要求，并写出输出结果。

(b). 请说明这个贪心算法得出来的解是最优解。

4. 旅行商问题，

(a). 给定邻接矩阵

$$\begin{pmatrix} \infty & 15 & 26 & 6 & 18 \\ & \infty & 14 & 3 & 5 \\ & & \infty & 7 & 11 \\ & & & \infty & 2 \\ & & & & \infty \end{pmatrix}$$

给出图的搜索树。

(b). 写出回溯法求解旅行商问题的算法框图。

第 2 章 算法模拟卷 A 答案

2.1 判断题

1. F

   解析：分治除了可以用递归实现，还可以用循环迭代实现。

2. T

3. F

   解析：动态规划通常用于求解具有重叠子问题和最优子结构性质的优化问题，而不是特定于动态不确定性的问题。

4. T

5. F

   解析：如果转化过程增加了额外的复杂度，例如需要遍历额外的数据结构或进行其他操作，那么问题 B 的复杂度可能会超过 $O(n^2)$。

2.2 选择题

1. A

   解析：由于 $f(n) = O(g(n))$，这意味着 $f(n)$ 的增长速度不会超过 $g(n)$ 的增长速度。因此，$f(n)$ 的阶不会超过 $g(n)$ 的阶。

2. A

   解析：如果为全排列，在这种情况下，每个结点都有 n 个可能的选择，并且排列树的深度为 n。因此，最坏情况下的时间复杂度为 $O(n!)$。

3. A

4. C

解析：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。

5. A

定理：设 L 是 NP 完全的，则若 L$\propto$ pL1，且 L1$\in$NP，则 L1 是 NP 完全的。

2.3 分析题

1. 分治策略：将一个复杂的问题分解为两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，最终通过子问题的解来解决原来的问题。

   动态规划：通过把原问题分解为相互重叠的子问题，并分别解决子问题，然后将子问题的解组合起来得到原问题的解。

   贪心算法：在每一步都采取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好或最优的算法。

   回溯算法：一种通过探索所有可能的解来解决问题的算法。当发现当前路径无法得到有效解时，回溯算法会“回退”到上一个状态，并尝试其他的路径。

2. 考虑回路路径 $x[1:k]$，其中 $k<n$，则路径 $x[1:k]$ 的路径和至少是 $\sum x[1:k]$。由于回路需要经过所有的点，那么从点 $x[k+1]$ 开始，至少需要经过剩余的 $n-k$ 个点。根据性质，从点 $x[k+1]$ 出发的最短路径至少是 $\min(x[k+1],x[k+2],,x[n])$。

   因此，路径 $x[1:k]$ 的路径和至少 $\geq x[1:k]+min(x[k+1],x[k+2],,x[n])$。由于 $x[1:k]$ 是任意路径的一部分，所以上述不等式对于任意回路路径 $x[1:k]$ 都成立。因此，整个回路路径的路径和至少大于等于 $\sum \min(i)$。

3. (a). 解向量 $x=(x_1,x_2,\cdots,x_n),x_i\in 01$ 表示第 i 个元素是否选择。

   (b). 显约束：选择元素之和为 m

   隐约束：由于 W 中的元素是不相同的正整数，因此一个子集中不能出现重复的元素。

   (c). 由较大元素开始画解空间树思路相对会更清晰

4. (a). 如果 $x=1$，只能分为 1；$x=2$，分为 2；$x=3$，分为 3

对 $x \geq 4$. 如果 $x=1(mod3)$，将 x 分成 $\frac{x-4}{3}$ 个 3，2 个 2；如果 $x=2(mod3)$，将 x 分成 $\frac{x-2}{3}$ 个 3，1 个 2；将 x 分成 $\frac{x-2}{3}$ 个 3。

(b). 如果分为大于等于 5 的数，例如 $5=2+3$，但 52×3，对于大于等于 6 的数 x，分为 $[x/2]$ 个 2 就有 $x<2^{[x/2]}$，显然分为 2 和 3 可以让乘积更大.$4=2+2$ 效果相同。于是为了让乘积更大可以直接分为 2 和 3，但是 2 和 3 该如何取舍。$6=2+2+2=3+3$，但是 $2\times2\times2<3\times3$, 可以知道我们应该让尽量多，但是如果再 $mod3$ 余 1 等情况下如果让 3 尽量多就会浪费 1，我们取出一个 $3, 4=2+2$，就完成了贪心策略。

2.4 算法设计题

1. 考虑二分搜索将数组 A 分成两半，分别处理左半部分和右半部分。如果 $A[mid]==mid$，那么 mid 就是我们要找的位置。否则，如果 $A[mid]>mid$，说明目标位置在左半部分，我们递归地在左半部分查找。如果 $A[mid]<mid$，说明目标位置在右半部分，我们递归地在右半部分查找。

Listing 1 算法

```
int Search(int A[], int left, int right) {
if (left > right)  return -1; // 没有找到满足条件的位置
int mid = (left + right) / 2;
if (A[mid] == mid) return mid; // 找到满足条件的位置
else if (A[mid] > mid) return Search(A, left, mid - 1); // 在左半部分查找
else return Search(A, mid + 1, right); // 在右半部分查找
}
```

算法时间复杂度为 $O(logn)$

2. 定义一个数组 L，其中 $L[i]$ 表示以第 i 个元素结尾的最大长度。

   $L[i] = 1$ 如果不存在 j，满足 $arr[i] > arr[j]$

   $L[i] = 1 + \max(L[j])$ 如果存在 j，满足 $arr[i] > arr[j]$

Listing 2 算法

```
int maxaddarr(vector<int>& arr) {
    int n = arr.size();
    vector<int> len(n, 1);  // 初始化数组
    for (int i = 1; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            if (arr[j] < arr[i]) {
                len[i] = max(len[i], len[j] + 1);
            }
        }
    }
// 返回最大长度
    int maxLen = 0;
    for (int i = 0; i < n; ++i) {
        maxLen = max(maxLen, len[i]);
    }
    return maxLen;
}//复杂度 O(n^2)
```

3. (a). 贪心算法的思想是每次选择当前可用的最大面额的纸币，以便尽量减少所需的纸币张数。

$35 = 27 + 3 + 3 + 1 + 1$

Listing 3 算法

```
int findMin(int m) {
    int cash[4] = {27, 9, 3, 1};
    int ans = 0;
    for (int i = 0; i < 4;) {
        while (m >= cash[i]) {
            m -= cash[i];
            ans++;
        }
    }
    return ans;
}
```

(b). 当我们选择当前可用的最大面额纸币时，我们最大程度地减少了剩余金额，从而最小化了总纸币数量。这种贪心选择的方式确保了每次选择都是局部最优解。该问题具有最优子结构性质，即全局最优解可以通过局部最优解得到。这是因为，当我们确定了第一张纸币的面额后，剩余的子问题是相同的，即求解剩余金额的最少纸币张数。因此，通过递归地应用贪心选择，我们可以得到整个问题的最优解。

4. (a). 图略

(b). 思路：写出回溯法求解旅行商问题的算法框图。

旅行商问题的解空间是一棵排列树。初始的时候 $x = [1, 2, 3, n]$.。在递归算法中，当 $i == n$ 时，当前遍历到了排列树的叶子节点的上一层节点 (即 $n - 1$ 层)，这个时候

要判断第 $n-1$ 个点与第 n 个点以及第 n 个点与第 1 个点之间是否形成一条回路，如果形成一条回路，则可以判断当前回路的 $cost$ 是否小于目前最优值，进而判断是否更新最优值和最优解。当 $i < n$ 的时候，还没有遍历到判断 n 个顶点是否形成回路。这个时候能够遍历当前节点的条件是当前节点 i 与上一节点 $i-1$ 连通 (即从第一个节点一直到第 i 个节点形成了一条路径)，并且这条路径的长度小于当前最优值，否则不遍历当前节点。

第 3 章 算法模拟卷 B

3.1 判断题

1. 若 $f(n)=\Theta(g(n))$，则 $f(n)=\Omega(g(n))$ (　)

2. 动态规划算法与分治法都采用自底向上的计算方式 (　)

3. 回溯法和分支限界法都是在问题解空间树上搜索问题解的算法 (　)

4. 求最小生成树的 Prim 算法使用的设计策略是贪心策略 (　)

5. 若问题 A 是一个 NP 问题，则 A 也是一个 P 类问题 (　)

3.2 选择题

1. 已知 $T(n)=\begin{cases}1, n\leq 1\\ 2T(\frac{n}{2})+c, n>1\end{cases}$，则 $T(n)$ 的渐进复杂度为____

A. $\Theta(\log n)$　　B. $\Theta(n)$　　C. $\Theta(n\log n)$　　D. $\Theta(n^2)$

2. 在对问题的解空间树进行搜索时，一个活结点最多有一次机会成为扩展结点的是____

A. 回溯法　　B. 分支限界法

C. 回溯法和分支限界法　　D. 回溯法求解子集树问题

3. 可用动态规划算法求解的问题应该具备____性质

A. 递归子结构和贪心选择　　B. 最优子结构和最优选择

C. 子集树和排列树　　D. 最优子结构和重叠子问题

4. 下列哪一种算法的设计思想是采用了分治策略____

A. 归并排序　　B. 单源点最短路径

C. 最优二叉搜索树构造　　D. 哈夫曼树构造

5. 若 A$\propto_{\tau(n)}$B，且问题 B 的计算时间上界为 $T(n)$，则问题 A 的一个计算时间上界为____

A. $O(T(n))$　　B. $O(\tau(n))$　　C. $T(n)-O(\tau(n))$　　D. $T(n)+O(\tau(n))$

3.3 分析题

1. 在求解 0/1 背包问题的动态规划算法中，为了解决物品重量为实数和背包容量很大的问题，可仅存储全部跳跃点。设包容量为 15，每个物品的重量为 $(2,2,6,5,4)$，每个物品的价值为 $(6,3,5,4,6)$

   (a). 请根据动态规划算法，给出每一步所求得的跳跃点序列

   (b). 根据所求出的跳跃点序列，给出其最优解和最优解的值

2. 若用回溯法求解问题：对于一个栈的输入序列 $(1,2,\cdots,n)$，求所能得到的不同输出序列的个数，请：

   (a). 定义问题的解向量形式

   (b). 写出结点显约束和隐约束条件

   (c). 对于输入序列 $(1,2,3)$，画出解空间树

3. 已知团问题属于 NPC，请证明定点覆盖问题属于 NPC

4. 下面为一个求解顶点覆盖的近似算法：

Listing 4 求解顶点覆盖

```
Vset appCover(Graph G){
    Cset=∅; E=G.e;
    while (E!=∅){
        从 E 中取出一条边 (u,v);
        cset=cset∪ {u,v};
        从 E 中删去与 u 和 v 相关联的所有边;
    }
    return cset
}
```

请证明算法 appCover 的性能比为 2

3.4 算法设计题

1. 有 n 个孩子和 m 块饼干，每个孩子有一个饥饿度 C_i，每个饼干都有一个大小 B_j。每个孩子只能吃最多一块饼干，且只有饼干的大小大于孩子的饥饿度时，这个孩子才能吃饱。使用贪心策略求解最多有多少孩子可以吃饱

2. 给定 n 个整数（可能是负整数）组成的序列 $a_1, a_2, \cdots, a_n$，求该序列形如 $\sum_{k=i}^{j} a_k$ 字段和的最大值。当所有整数均为负整数时定义其最大字段和为 0。试设计时间复杂度和空间复杂度均为 $O(n)$ 的动态规划算法

3. 给定实数数组 $A[1...n]$（可能含有负数），要找到它的一个子数组 $A[i...j]$，使得 $A[i...j]$ 中各个元素的乘积最大

   (a). 对于 i 和 j 的不同取值，子数组 $A[1...n]$ 有多少个？如果使用穷举方法求解此问题，则时间复杂度至少为多少?

   (b). 若用分治法求解此问题，可设计一个递归算法 MaxProduct(l, h) 求数组 $A[l...h]$ 中的子数组元素乘积的最大值。请简述分治法求解此问题的基本思路或过程。并给出算法的时间复杂度

   (c). 假定用 $M(k)$ 表示以 $A[k]$ 为结尾的子数组中元素乘积的最大值，用 $m(k)$ 表示以 $A[k]$ 为结尾的子数组中元素乘积的最小值，那么 $M(k)$ 和 $m(k)$ 可以由 $M(k-1)$ 和 $m(k-1)$ 递推求得。请给出 $M(k)$ 和 $m(k)$ 的递归表达式，并根据此表达式给出一种求解此问题的动态规划算法基本思路或过程。并给出算法的时间复杂度

4. 有 n 个不同的正数存储于数组 $w[1...n]$ 中，现给定一个正数 c，要求寻找这 n 个数的子集，使得该子集的元素和最大，且不超过 c。请设计并实现用回溯法解此问题的算法，要求使用剪枝函数避免无效搜索

第 4 章 算法模拟卷 B 答案

4.1 判断题

1. T
2. F

   解析：分治法则采用自顶向下
3. T

   解析：回溯法和分支限界法都是在问题解空间树上搜索问题解的算法，它们在搜索策略和终止条件上有所不同
4. T

   解析：prim 算法基于贪心，我们每次总是选出一个离生成树距离最小的点去加入生成树，最后实现最小生成树
5. F

   解析：若问题 A 是一个 P 问题，则 A 也是一个 NP 类问题，P∈NP

4.2 选择题

1. B

   解析：非递归项为常数，由公式 $T(n) = n^{\log_2 2} = n$。
2. B

   解析：回溯法同一个节点可以多次成为扩展结点。
3. D

   解析：动态规划算法通常用于解决具有最优子结构和重叠子问题性质的问题。最优子结构意味着问题的最优解可以通过子问题的最优解来求解，而重叠子问题意味着问题的解

空间中存在重叠的子问题，这些子问题可以通过存储已解决的子问题的解来避免重复计算，提高效率。

4. A

解析：单源点最短路径设计思想是贪心算法；最优二叉搜索树构造为动态规划；哈夫曼树构造为贪心。

5. D

4.3 分析题

1. (a). 初始化二维数组 $dp[][], dp[i][j]$ 表示考虑前 i 个物品，重量不超过 j 的情况下的最大价值。

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][p-wi] + vi)$$

用表 p[i] 储存 dp 的全部跳跃点

$P[6] = (0,0)$

$P[5] = (0,0), (4,6)$

$P[4] = (0,0), (4,6), (9,10)$

$P[3] = (0,0), (4,6), (9,10), (15,15)$

$P[2] = (0,0), (2,3), (4,6), (6,9), (9,10), (10,11), (11,13), (12,14), (15,15)$

$P[1] = (0,0), (2,6), (4,9), (6,12), (8,15), (11,16), (12,17), (13,19), (14,20)$

(b). 最优解选第 $1,2,3,5$ 件物品，总价值为 20

2. (a). 解向量是一个排列，表示输出序列的排列方式，其中元素取自输入序列 $(1,2,...,n)$。

(b). 显约束：生成输出序列时，不能改变输入序列中元素的相对顺序，只能在输入序列中选择元素组成输出序列。隐约束：如果当前栈中元素为 x，那么后续的输出序列中，如果还有元素 $y(y > x)$，那么 y 必须在 x 之后出现。

(c). 图略

3. 第一步：给定一个顶点集，可以在多项式时间内验证是否是一个顶点覆盖，只需检查图中的每条边，看它们的两个端点是否至少有一个在给定的顶点集中。因此，定点覆盖问题属于 NP。

   第二步：将团问题归约到定点覆盖问题, 给定一个图 G 和一个正整数 k，我们要判断 G 中是否存在一个团（即完全子图），使得团中至少有 k 个顶点。我们可以构造一个新的图 G'，其中 G' 的顶点集等于 G 的顶点集，边集等于 G 的边集加上一些额外的边。具体构造如下：

   (a) 对于图 G 中的每个顶点 v，我们在 G' 中添加两个顶点 v_1 和 v_2。

   (b) 对于图 G 中的每条边 (u, v)，我们在 G' 中添加一条边 (u_1, v_2) 和 (u_2, v_1)。

   构造完 G' 后，我们将定点覆盖问题的目标定为 k，即找到一个最小的顶点集，使得 G' 中的每条边都至少有一个端点在这个顶点集中。

   第三步：如果存在一个大小至少为 k 的团 C，那么取 C 中的任意一个顶点 v，在 G' 中对应的两个顶点 v_1 和 v_2 必然都包含在顶点覆盖中，因为对于任意一条边 (u, v)（其中 u 是 C 中的另一个顶点），至少有一个端点在 C 中，所以对应的两个端点在顶点覆盖中。反之，如果存在一个大小至少为 k 的顶点覆盖 S，那么考虑 S 中的任意一个顶点 v_1（或 v_2），根据 G' 的构造方式可知，v_1（或 v_2）对应的原图中的顶点 v 必然与 S 中的其他顶点都相连，因此 $\{v, S - v_1\}$(或 $\{v, S - v_2\}$) 构成一个团。

   综上可以知道定点覆盖问题属于 NPC

4. 为了说明 appCover 返回顶点覆盖的规模至多为最优覆盖的两倍，设 A 为 appCover 中第 4 行选出的边集合。为了覆盖 A 中所有的边，任意一个顶点覆盖，特别是最优覆盖 $Cset^*$，都必须至少包含 A 中每条边的一个端点。如果一条边一旦在第 4 行中被选中，那么在第 6 行就会从 E' 中删除所有与其端点关联的边，因此，A 中不存在两条具有共同的端点边。从而 A 中不存在由 $Cset^*$ 中的同一顶点所覆盖两条边，意味着对于 $Cset^*$ 中每一个顶点，A 只有一条其关联边，由此得到最优顶点覆盖的规模下界：$|Cset^*||A|$。第 4 行的

每一次执行都会挑选出一条边，其两个端点都不在 $Cset$ 中，因此，所返回顶点覆盖的规模上界（实际上是一个精确的上界）为：$|C| = 2|A|$。结合两者有 $|C| = 2|A| \leq 2|Cset^*|$，因此成立。

4.4 算法设计题

Listing 5 算法

```
1.  int maxChildrenFed(vector<int>& children, vector<int>& cookies) {
    sort(children.begin(), children.end());
    sort(cookies.begin(), cookies.end());

    int i = 0;
    int j = 0;
    int count = 0;

    while (i < children.size() && j < cookies.size()) {
        if (cookies[j] >= children[i]) {
            count++;
            i++;
        }
        j++;
    }

    return count;
}
```

Listing 6 算法

```
2. int maxSubarraySum(const vector<int>& nums) {
    int n = nums.size();
    int maxSum = 0; // 最大字段和
    int currentSum = 0; // 当前子段和

    for (int i = 0; i < n; ++i) {
        currentSum = max(0, currentSum + nums[i]); // 更新当前子段和
        maxSum = max(maxSum, currentSum); // 更新最大字段和
    }

    return maxSum;
}
```

3. (a). $\frac{n(n+1)}{2}$;$O(n^2)$

Listing 7 算法第一部分

(b).
```
// 辅助函数，用于计算跨越中间的最大子数组乘积
int maxCrossingProduct(vector<int>& A, int l, int m, int h) {
    int leftMax = INT_MIN;
    int leftMin = INT_MAX;
    int product = 1;
    for (int i = m; i >= l; --i) {
        product *= A[i];
        leftMax = max(leftMax, product);
        leftMin = min(leftMin, product);
    }

    int rightMax = INT_MIN;
    int rightMin = INT_MAX;
    product = 1;
    for (int i = m + 1; i <= h; ++i) {
        product *= A[i];
        rightMax = max(rightMax, product);
        rightMin = min(rightMin, product);
    }

    return max(leftMin * rightMin, max(leftMax * rightMax, max(leftMax, rightMax)));
}
```

Listing 8 算法第二部分

```
// 分治法求解最大子数组乘积的递归函数
int MaxProduct(vector<int>& A, int l, int h) {
    if (l == h) // 基本情况：只有一个元素
        return A[l];

    int m = l + (h - l) / 2; // 计算中间位置

    // 分别计算左右部分的最大子数组乘积
    int leftMaxProduct = MaxProduct(A, l, m);
    int rightMaxProduct = MaxProduct(A, m + 1, h);

    // 计算跨越中间的最大子数组乘积
    int crossingMaxProduct = maxCrossingProduct(A, l, m, h);

    // 返回三者中的最大值
    return max(crossingMaxProduct, max(leftMaxProduct, rightMaxProduct));
```

(c).

$$M(k) = \max\{A[k], M(k1)A[k], m(k1)A[k]\}$$

$$m(k) = \min\{A[k], M(k1)A[k], m(k1)A[k]\}$$

只需要循环遍历一次数组 A，因此时间复杂度为 $O(n)$

Listing 9 算法第二部分

```
4. void backtrackSubsetSum(const vector<int>& w, int c, int index, vector<int>& curSubset,
int curSum, int& maxSum, vector<int>& maxSubset) {
    if (curSum > c) { // 剪枝
        return;
    }
    if (curSum > maxSum) { // 更新最大和和对应的子集
        maxSum = curSum;
        maxSubset = curSubset;
    }

    if (index >= w.size()) { // 到达叶子节点
        return;
    }

    // 选择当前元素
    curSubset.push_back(w[index]);
    backtrackSubsetSum(w, c, index + 1, curSubset, curSum + w[index], maxSum, maxSubset)
    curSubset.pop_back(); // 回溯

    // 不选择当前元素
    backtrackSubsetSum(w, c, index + 1, curSubset, curSum, maxSum, maxSubset);
}
```