（　）贪婪技术所做的每一步选择所产生的部分解，不一定是可行性的。

×

贪婪技术的核心：所做的每一步选择都必须满足：(1) 可行的，即它必须满足问题的约束；(2) 局部最优，即它是当前步骤中所有可行性选择中最佳的局部选择；(2) 不可取消，即一旦做出，在算法的后续步骤就无法改变了。

（√）一个正确的算法，对于每一个合法输入，都会在有限的时间内输出一个满足要求的结果。

×

“正确”一词的含义在通常的用法中有很大差别，大体可分为以下 4 个层次：a. 程序不含语法错误；b. 程序对于几组输入数据能够得出满足规格说明要求的结果；c. 程序对于精心选择的典型、苛刻而带有刁难性的几组输入数据能够得出满足规格说明要求的结果；d. 程序对于一切合法的输入数据都能产生满足规格说明要求的结果。显然，达到第 d 层意义下的正确是极为困难的，所有不同输入数据的数量大得惊人，逐一验证的方法是不现实的。对于大型软件需要进行专业测试，而一般情况下，通常以第 c 层意义的正确性作为衡量一个程序是否合格的标准。

（　）在动态规划中，各个阶段所确定的策略就构成一个策略序列，通常称为一个决策。

×

决策和策略：决策是指某阶段状态给定以后，从该状态演变到下一个阶段某状态的选择；由每段的决策组成的决策函数序列就称为全过程策略，简称策略。

（　）通常来说，算法的最坏情况的时间复杂性比平均情况的时间复杂性容易计算。

√

而，在很多情况下，各种输入数据集出现的概率难以确定，算法的平均时间复杂度也就难以确定。因此，另一种更可行也更常用的办法是讨论算法在最坏情况下的时间复杂度，即分析最坏情况以估算算法执行时间的一个上界。例如，上述起泡排序的最坏情况为 a 中

（　）回溯法用深度优先法搜索状态空间树。

√

针对所要做的选择构造一棵所谓的状态空间树，树的每一层节点代表了对解的每一个分量所做的选择；用 DFS（深度优先法）搜索状态空间树；在状态空间树中的任一个节点，满足一定条件的情况下，搜索回溯。

（ ）快速排序算法的平均时间复杂度是 O(nlogn)，使用随机化快速排序算法可以将平均时间复杂度降得更低。

×

算法：在指针 high 减 1 和 low 增 1 的同时进行“起泡”操作，即在相邻两个记录处于“逆序”时进行互换，同时在算法中附设两个布尔型变量分别指示指针 low 和 high 在从两端向中间的移动过程中是否进行过交换记录的操作，若指针 low 在从低端向中间的移动过程中没有进行交换记录的操作，则不再需要对低端子表进行排序；类似地，若指针 high 在从高端向中间的移动过程中没有进行交换记录的操作，则不再需要对高端子表进行排序。显然，如此“划分”将进一步改善快速排序的平均性能。

期望运行时间：Θ(nlgn)；但与输入无关，最坏情况出现在随机数生成结果恰好使数组有序----概率极小。

（ ）P 类和 NP 类问题的关系用 $p \subset NP$ 来表示是错误的。

×

P 中所有问题均属于 NP。

NP 完全问题比其它所有 NP 问题都要难。

×

NP 完全问题至少同其它所有 NP 问题一样难

（√）若 P2 多项式时间转化为(polynomially transforms to) P1，则 P2 至少与 P1 一样难。

×

P1 多项式时间转化为 P2，则 P2 至少与 P1 一样难；总是可以在可比时间内用 P2 之 Algo2 求解 P1.

（ ）一个完全多项式近似方案是一个近似方案$\{A_\varepsilon\}$，其中每一个算法 A_ε 在输入实例 I 的规模的多项式时间内运行。

×

一个多项式近似方案(PAS)是一个近似方案{A_ε},其中每一个算法 A_ε 在输入实例 I 的规模的多项式时间内运行;一个完全多项式近似方案(FPAS)是一个近似方案{A_ε},其中每个算法 A_ε 在以输入实例的规模和 $1/\varepsilon$ 两者的多项式时间内运行.

（ ）基于比较的寻找数组 A[1...n]中最大值元素问题的下界是 $\Omega(n/3)$。

×

n-1

（ ）Las Vegas 算法只要给出解就是正确的。

√

Las Vegas 总是要么给出正确的解，要么告知无解。

（ ）若近似算法 A 求解某极小化问题一实例的解为 s_a，且已知该问题的最优解为 $s_a/3$，则该近似算法的性能比为 3。

×

对于一个 f 最小化的问题，可以用$r(s_a)=\frac{f(s_a)}{f(s*)}$，作为 s_a 的精确度量。对于问题的所有实例，它们可能的 $r(s_a)$的最佳（也就是最低）上界，被称为该算法的性能比，计作 R_A。
性能比是一个来指出近似算法质量的主要指标，我们需要那些 R_A 尽量接近 1 的近似算法。

（ ）$O(f(n))+O(g(n)) = O(\min\{f(n),g(n)\})$ 。

（ ）若 $f(n)=\Omega(g(n))$， $g(n)=\Omega(h(n))$ ，则 $f(n)=\Omega(h(n))$。

（ ）若 $f(n)=O(g(n))$， 则 $g(n)=\Omega(f(n))$ 。

× √ √

$O(f(n))+O(g(n)) = O(\max\{f(n),g(n)\})$
$O(f(n))+O(g(n)) = O(f(n)+g(n))$
$O(f(n))*O(g(n)) = O(f(n)*g(n))$
$O(cf(n)) = O(f(n))$
$f(n)= O(g(n))$， $g(n)= O(h(n)) \Rightarrow f(n)= O(h(n))$;
$f(n)= \Omega(g(n))$， $g(n)= \Omega(h(n)) \Rightarrow f(n)= \Omega(h(n))$;

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

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

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

$f(n)= \Theta(f(n))$；

$f(n)= O(f(n))$；

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

1、二叉查找树属于减治策略的三个变种中哪一个的应用？什么情况下二叉查找树表现出最差的效率？此时的查找和插入算法的复杂性如何？（本题 3 分）

减治策略有 3 个主要的变种，包括减常量、减常数因子和减可变规模。(1) 二叉查找树属于减可变规模变种的应用。(2) 当先后插入的关键字有序时，构成的二叉查找树蜕变为单支树，树的深度等于 n，此时二叉查找树表现出最差的效率，(3) 查找和插入算法的时间效率都属于 Θ(n)。

3、构造 AVL 数和 2-3 树的主要目的是什么？它们各自有什么样的查找和插入的效率？（本题 2 分）

(1)当先后插入的关键字有序时，构成的二叉查找树蜕变为单支树，树的深度等于 n，此时二叉查找树表现出最差的效率，为了解决这一问题，可以构造 AVL 树或 2-3 树，使树的深度减小。一棵 AVL 树要求它的每个节点的左右子树的高度差不能超过 1。2-3 树和 2-3-4 树允许一棵查找树的单个节点不止包含一个元素。(2) AVL 树在最差情况下，查找和插入操作的效率属于 Θ(lgn)。2-3 树无论在最差还是平均情况下，查找和插入的效率都属于 Θ(log n)。

4、写出 0/1 背包问题的一个多项式等价(Polynomially equivalent) 的判定问题，并说明为什么它们是多项式等价的。（本题 3 分）

0/1 背包问题：从 M 件物品中，取出若干件放在空间为 W 的背包里，给出一个能获得最大价值的方案。每件物品的体积为 W1，W2……Wn，与之相对应的价值为 P1,P2……Pn。

判定问题 I：从 M 件物品中，取出若干件放在空间为 W 的背包里，是否存在一个方案，所获价值≥P^*？ 。每件物品的体积为 W1，W2……Wn，与之相对应的价值为 P1,P2……Pn。

若判定问题 I 存在多项式时间的解法，则反复调用该算法就可以在多项式时间内解决 0/1 背包的优化问题。因而这个判定问题与原问题多项式等价。

2、何谓伪多项式算法？如何将一 Monte Carlo 算法转化为 Las Vegas 算法（本题 2 分）

伪多项式时间算法是一种算法，它在 L 值的多项式时间内运行，其中 L 是输入实例中的最大数值。

Las Vegas 算法不会得到不正确的解。一旦用拉斯维加斯算法找到一个解，这个解就一定是正确解。但有时用拉斯维加斯算法找不到解。

Monte Carlo 算法每次都能得到问题的解，但不保证所得解的准确性

转化：可以在 Monte Carlo 算法给出的解上加一个验证算法，如果正确就得到解，如果错误就不能生成问题的解，这样 Monte Carlo 算法便转化为了 Las Vegas 算法。

5、下面问题是否属于 NP 问题？为什么？（本题 2 分）

给定图 G=(N,A)中的两个点 p、q，整数 c 和 t，图 G 中每条边的长度 c_{ij} 及遍历这条边的时间 t_{ij}，问图 G 中是否存在一条由 p 到 q 的路径，使得其长度大于 C，且遍历时间小于 t?

这个问题属于 NP 问题。因为若给出该问题的一个解，可以在多项式时间内检验这个解的正确性。如给出一条由 p 到 q 的路径，可以在多项式时间内计算出它的长度及遍历时间，然后分别与 C 和 t 进行比较，从而可以判断这个解的对错。

三、A[1...n]为一整数序列，A 中的整数 a 如果在 A 中的出现次数多于$\lfloor n/2 \rfloor$，那么 a 称为多数元素。例如，在序列 1，3，2，3，3，4，3 中，3 是多数元素，因其出现 4 次，大于$\lfloor 7/2 \rfloor$。求 A 的多数元素问题的蛮力算法复杂性如何？设计一具有变治思想的算法，提高蛮力算法的效率，写出伪代码并分析其时间复杂性。（本题 5 分）

蛮力算法时间复杂度：$O(n^2)$；空间复杂度：O(1)

```
/**
*对长度为 len 的数组 A，找出它的多数元素
*/
majority(A[], len) {
    HeapSort(A);  //   对数组 A 中元素按从小到大顺序进行堆排序
    i = 0;
    flag = false;
    while i < len {
        tmp = A[i]
        count = 1;
        while i + count < len && A[i+count] == tmp {
            count++;
        }
        if count > len/2 {
            print(tmp+“是多数元素”)
            flag = true;
            break;
        }
         i + = count;
    }
    if flag == false
        print(“没有多数元素”)
}
```

该算法的时间复杂性：O(nlgn)

三、写出一求解下列问题的分治算法，推导其时间复杂性并与蛮力法相比较。（本题 5 分）

给定互不相等的 n 个数的一个序列 a_1，a_2，…，a_n，若其中某两个数 a_i 和 a_j 的关系为：$a_i > a_j$，且 $i<j$，则称 a_i 和 a_j 是逆序的。要求计算该序列中的逆序个数，即，具有逆序关系的元素对的总数目。

```
/**
*求解 n 个数的一个序列，具有逆序关系的元素对的总数目
*/
count = 0;      //逆序元素对的全局计数变量
mergeInvertedPairs(A,low,mid,high) {
    i = low;
    j = mid+1;
    k = low;
    tmp[n];   //用于归并排序的辅助数组
    while i <= mid && j <= high {
        if (A[i] > A[j]) {
            tmp[k] = A[j++];
            count += (mid-i+1);      //相比归并排序，就多了这一条语句
        } else {
            tmp[k] = A[i++];
        }
        k++;
    }
    while i <= mid {
        tmp[k++] = A[i++];
    }
    while j <= high {
        tmp[k++] = A[j++];
    }
    for (j = low; j <= high; j++) {
        A[j] = tmp[j];
    }
}
findInvertedPairs(A[], low, high) {
    if (low < high) {
        mid = (low+high) / 2;
        findInvertedPairs(A,low,mid);
        findInvertedPairs(A,mid+1,high);
        mergeInvertedPairs(A,low,mid,high);
    }
}
```

[动态规划]