一、选择题（每题 2 分，10 题，共 20 分）

1、关于记号 *O* 正确的定义是（ ）。

A、*O*(g(n)) = {f(n) |对任意正数 c 存在 n0 使得对所有n≥n0 有 0≤f(n)<cg(n)}

B、O(g(n)) = {f(n) |对任意正数 c 存在 n0 使得对所有n≥n0 有 0≤cg(n)< f(n)}

C、O(g(n)) = {f(n) |存在正数 c 和 n0 使得对所有n≥n0 有 0≤f(n)≤cg(n)}

D、*O*(g(n)) = {f(n) |存在正数 c 和 n0 使得对所有n≥n0 有 0≤cg(n) ≤f(n)}

2、以下算法思想中，有可能得到的解为不正确解的是（ ）。

A、动态规划 B、贪心 C、蒙特卡罗 D、拉斯维加斯

分析：

***蒙特卡罗（Monte Carlo）算法***：蒙特卡罗（Monte Carlo）算法用于求问题的准确解。用蒙特卡罗算法能求得问题的一个解，***但这个解未必是正确的***。

***拉斯维加斯（Las Vegas）算法***：拉斯维加斯（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，那么这个解肯定是正确的。**但是有时候用拉斯维加斯算法可能找不到解。**

**二、填空题（每空 2 分，15 空，共 30 分）**

1、算法是由若干条指令组成的有穷序列，且要满足有输入、有输出、确定性 、有效性和有限性性质。算法的效率一般可以从时间复杂度、空间复杂度两个方面衡量。

2、假设算法A 理论上的时间复杂度T(n)=3n，现在有两台同类型计算机M1和M2，M2的计算速度是M1的a倍。若在M1和M2上分别测试算法A，则在相同时间内，M1和M2能够求解的问题规模n1和n2的关系为n1 : n2 = 1 : a。若算法B理论上的时间复杂度为T(n)=n2， 若在M1和M2上分别测试算法B，则M1和M2求解相同规模问题所耗费的时间t1和t2的关系为t1:t2 = 1:。

3、在随机化算法中，能够平滑不同输入实例之间差异的算法是 舍伍德 算法。

*分析：*

*使用Sherwood处理后的算法能够平滑不同输入实例的执行时间。 sherwood是一种概率算法思想，使得算法的复杂度不依赖于实例，而是依赖于概率。*

4、采用分治法解决问题时，算法时间复杂度分析典型的递推方程为

*T* (*n*) *O*(1)



*aT* (*n* / *b*) *f* (*n*)

*n* 1

*n* 1

，对于某一问题，若 *f*(*n*) = *O*(1)且 *a* = 1，则求解该问题

的时间复杂度为 *O*(logn) ；若*f*(*n*)=*O*(*x*)且*a*=x,*b*>y，则求解该问题的时间复杂度为 *O*(n2) ；若*f*(*n*) = *O****(****n*)，则求解该问题的时间复杂度为 n ；若已知*f*(*n*)=*O****(****x*)，且*a*,*b*均大于1，若希望将算法时间复杂度降低为不超线性情况，则a和b应满足的条件为： a<=b。

三、算法设计与分析题（3 题，每题 10 分，共 30 分）

1、已知一箱苹果有 *n* 个，重量分别为 *w*1, *w*2, … , *w*n，若要求计算这 *n* 个苹果的最小重量差

min *h* *h*

*i j*

1*i**n*

1*j**n*

，请为该问题设计一个有效算法，并对所设计的算法进行时间复杂性分析。

要求：（1）给出算法基本思想描述。

1. 给出算法的伪代码描述。
2. 对所设计算法进行时间复杂性分析。需要给出步骤和结果，包括：确定问题规模参数，算法基本操作，影响基本操作执行次数因素等。

说明：不要求代码实现

（1）基本思想：

自顶向下分析，diff[i]记录0~i的最小数对之差，max[i]记录0~i的最大数，则状态递归方程为：

diff[i+1]=Math.min(max[i]-array[i+1]，diff[i] )

（2）

float getMinDiff(a)

输入：苹果数组 a[1-n]

返回：最小重量差 minDiff

sort(a)//按苹果重量非递减排序minDiff ← MAXDIFF

for i←1 to n-1

min ← a[i] – a[i-1]

if min < minDiff

minDiff ← min

（3）

在不考虑排序情况下：

**问题规模参数**为输入数组中元素个数； **算法基本操作**为 min ← a[i] – a[i-1]；

**影响基本操作执行次数的因素**只取决于问题规模 n； 基本操作执行次数为 n-1 次；**时间复杂度为 *O*(n)**。

排序可以在 *O*(nlogn)时间复杂度下完成，因此**算法时间复杂度为取决于排序时间复杂度，为**

***O*(nlogn)**。

2、给定 n 种物品和一个背包。物品 *i* 的重量为 *wi*，体积为 *bi*，其价值为 *vi*，背包容量为 *c*，容积为 *d*。应如何选择装入背包的物品，使得装入背包中物品的总价值最大？

（1）基本思想：

利用动态规划求最优值的方法。假设用dp[N][V]来存储中间状态值,dp[i][j]表示前i件物品能装入容量为j的背包中的物品价值总和的最大值(注意是最大值),则我们最终只需求知dp[i=N][j=V]的值，即为题目所求。

现在考虑动态规划数组dp[i][j]的状态转移方程：

假设我们已经求出前i-1件物品装入容量j的背包的价值总和最大值为dp[i-1][j],固定容量j的值不变，则对第i件物品的装法讨论如下：

首先第i件物品的重量weight[i]必须小于等于容量j才行，即

1、若weight[i]>j,则第i件物品肯定不能装入容量为j的背包，此时dp[i][j]=dp[i-1][j]

2、若weight[i]<=j,则首先明确的是这件物品是可以装入容量为j的背包的，那么如果我们将该物品装入，则有

dp[i][j]=dp[i-1][j-weight[i]]+value[i]

随之而来的问题是我们要判断第i件物品装到容量为j的背包后,背包内的总价值是否是最大？其实很好判断，即如果装了第i件物品后的总价值dp[i-1][j-weight[i]]+value[i]>没装之前的总价值最大值dp[i-1][j]，则肯是最大的；反之则说明第i件物品不必装入容量为j的背包(装了之后总价值反而变小，那么肯定就不需要装嘛)

故，状态转移方程如下：

dp[i][j] = (dp[i-1][j] > (dp[i-1][j-weight[i]]+value[i]))? dp[i-1][j]:(dp[i-1][j-weight[i]]+value[i])

注意：这里的前i件物品是给定次序的

b)求出背包中装入物品的编号

这里我们采用逆推的思路来处理，如果对于dp[i][j]>dp[i-1][j]，则说明第i个物品肯定被放入了背包，此时我们再考察dp[i-1][j-weight[i]]的编号就可以了。

**（2）java代码实现：**

/\*\*

\* 0-1背包问题

\* @param V 背包容量

\* @param N 物品种类

\* @param weight 物品重量

\* @param value 物品价值

\* @return

\*/

public static int ZeroOnePack2(int V,int N,int[] weight,int[] value){ //动态规划

int[] dp = new int[V+1];

for(int i=1;i<N+1;i++){ //逆序实现

for(int j=V;j>=weight[i-1];j--){

dp[j] = Math.max(dp[j-weight[i-1]]+value[i-1],dp[j]);

}

}

return dp[V];

}

四、算法实现题（1 题，共 20 分）

子集树—装载问题

输入： int[]w={21,10,5};

Load\_Cargo problem=new Load\_Cargo(w,34);

输出：

最大装载量：31

应装入的货物：1 2

算法思想：

进行子集树的搜索，首先会判断是否到达叶结点，如果到达叶结点就判断是否更新最优解。

之后判断是否进行左子树的搜素,左子树为将当前的集装箱装上轮船。

然后判断是否进行右子树的搜索,这里有一个剪枝策略。

进入左子树/右子树之后调用本身进行递归。

剪枝策略：

将剩余的集装箱都装上轮船求得的载重量小于当前最优载重量,那就没有必要搜素右子树。

java实现代码：

class Load\_Cargo{//装载问题

int[] w;//各货物的重量

int wMax;//最大载重量

int nowWei=0;//当前载重量

int bestWei=0;//最优重量

int restCargo=0;//剩余货物重量

int cargo\_len=0;//货物数目

int []x;//当前解向量

int []bestX;//最优解

Load\_Cargo(int[] w,int wMax){

this.w=w;

this.wMax=wMax;

cargo\_len=w.length;

x=new int[cargo\_len];

bestX=new int[cargo\_len];

int i;

for(i=0;i<cargo\_len;i++){

restCargo+=w[i];

}

}

void solve(){

BackTrace(0);

System.out.println("最大装载量："+bestWei);

System.out.print("应装入的货物：");

for(int i=0;i<cargo\_len;i++){

if(bestX[i]==1){

System.out.print((i+1)+" ");

}

}

System.out.println();

}

**void BackTrace(int t){**

**if(t<cargo\_len){//如果没有抵达叶子节点**

**restCargo-=w[t];//剩余货物 减去 当前货物重量**

**//遍历左子树**

**if(nowWei+w[t]<=wMax){//如果当前货物能放入**

**nowWei+=w[t];**

**x[t]=1;**

**BackTrace(t+1);**

**nowWei-=w[t];**

**}**

**//遍历右子树**

**if(nowWei+restCargo>bestWei){//如果当前重量 加上 剩余货物 比最优解重**

**x[t]=0;**

**BackTrace(t+1);**

**}**

**restCargo+=w[t];//恢复 剩余货物重量**

**}else{//叶子节点，回溯完毕**

**int a;**

**a=0;**

**if(nowWei>bestWei){//当前重量大于剩余重量**

**//更新最优解向量**

**bestX=x.clone();**

**bestWei=nowWei;**

**}**

**}**

}

}