2020 CSP-S 真题详细题解

一、单项选择题

1. 请选出以下最大的数（）

A.$(550)_{10}$　　B.$(777)_{8}$　　C.2^{10}　　D.$(22F)_{16}$

答案：C

分析：本题考查不同进制数的转换与比较。

- 选项 A：十进制 550，即 550。
- 选项 B：八进制 777 转换为十进制：$7\times8^2 + 7\times8 + 7 = 7\times64 + 56 + 7 = 448 + 56 + 7 = 511$。
- 选项 C：$2^{10} = 1024$。
- 选项 D：十六进制 22F 转换为十进制：$2\times16^2 + 2\times16 + 15 = 2\times256 + 32 + 15 = 512 + 32 + 15 = 559$。

比较可知，最大的数是 2^{10}，故答案为 C。

2. 操作系统的功能是（）。

A. 负责外设与主机之间的信息交换

B. 控制和管理计算机系统的各种硬件和软件资源的使用

C. 负责诊断机器的故障

D. 将源程序编译成目标程序

答案：B

分析：本题考查操作系统的核心功能。

- 选项 A 是设备驱动程序的功能，负责外设与主机通信。
- 选项 B 正确，操作系统的核心功能是控制和管理计算机的软硬件资源，提高资源利用率。
- 选项 C 是诊断程序的功能，用于检测硬件故障。
- 选项 D 是编译器的功能，将源代码转换为目标代码。

3. 现有一段 8 分钟的视频文件，它的播放速度是每秒 24 帧图像，每帧图像是一幅分辨率为 2048×1024 像素的 32 位真彩色图像。请问要存储这段原始无压缩视频，需要多大的存储空间？（）。

A.30G　　B.90G　　C.150G　　D. 450G

答案：B

分析：本题考查视频存储容量计算。

- 计算步骤：
  a. 时间转换：8 分钟 = 8×60 = 480 秒。
  b. 总帧数：480 秒 ×24 帧 / 秒 = 11520 帧。
  c. 每帧像素数：2048×1024 = 2,097,152 像素。
  d. 每帧字节数：32 位真彩色 = 4 字节 / 像素，故每帧 = 2,097,152×4 = 8,388,608 字节。
  e. 总字节数：11520×8,388,608 = 96,636,764,160 字节。
  f. 转换为 GB：1GB = 1024^3 = 1,073,741,824 字节，总容量 = 96,636,764,160 ÷ 1,073,741,824 ≈ 90GB。

4. 今有一空栈 S，对下列待进栈的数据元素序列 a,b,c,d,e,f 依次进行：进栈，进栈，出栈，进栈，进栈，出栈的操作，则此操作完成后，栈底元素为 ( )。

A.b　　B.a　　C.d　　D.c

答案：B

分析：本题考查栈的基本操作（后进先出）。

- 操作过程：
  a. 进栈 a → 栈：[a]（栈底 a）。
  b. 进栈 b → 栈：[a, b]（栈底 a）。
  c. 出栈 → 栈：[a]（弹出 b）。
  d. 进栈 c → 栈：[a, c]。
  e. 进栈 d → 栈：[a, c, d]。
  f. 出栈 → 栈：[a, c]（弹出 d）。
- 最终栈底元素为 a，故答案为 B。

5. 将 (2, 7, 10, 18) 分别存储到某个地址区间为 0 ~ 10 的哈希表中，如果哈希函数 h(x)=(　　)，将不会产生冲突，其中 a mod b 表示 a 除以 b 的余数。

A.x^2mod 11　　B.2x mod 11

C.x mod 11　　D.[x/2] mod 11，其中 [x/2] 表示 x/2 下取整

答案：D

分析：本题考查哈希函数冲突判断。

- 对每个选项计算哈希值：
  - 选项 A：x^2 mod 11。$2^2=4$，$7^2=49$ mod11=5，$10^2=100$ mod11=1，$18^2=324$ mod11=5（7 和 18 冲突）。
  - 选项 B：2x mod 11。2×2=4，2×7=14 mod11=3，2×10=20 mod11=9，2×18=36 mod11=3（7 和 18 冲突）。
  - 选项 C：x mod 11。2 mod11=2，7 mod11=7，10 mod11=10，18 mod11=7（7 和 18 冲突）。
  - 选项 D：[x/2] mod11。[2/2]=1，[7/2]=3，[10/2]=5，[18/2]=9（均不冲突）。

6. 下列哪些问题不能用贪心法精确求解？( )

A. 霍夫曼编码问题　　B.0-1 背包问题

C. 最小生成树问题　　D. 单源最短路径问题

答案：B

分析：本题考查贪心法的适用范围。

- 贪心法适用于具有“最优子结构”和“贪心选择性质”的问题：
  - 选项 A：霍夫曼编码通过每次合并最小频率节点，可用贪心求解。
  - 选项 B：0-1 背包问题中物品不可分割，贪心选择（如按性价比排序）无法保证最优解，需用动态规划。
  - 选项 C：最小生成树（Kruskal 或 Prim 算法）基于贪心思想。
  - 选项 D：单源最短路径（Dijkstra 算法）对非负权图可用贪心求解。

7. 具有 n 个顶点，e 条边的图采用邻接表存储结构，进行深度优先遍历运算的时间复杂度为( )。

A.O (n+e)　　B.O (n^2)　　C.O (e^2)　　D.O (n)

答案：A

分析：本题考查图的 DFS 时间复杂度。

- 邻接表存储中，DFS 需访问所有顶点（O (n)）和所有边（每条边被访问一次，O (e)），总时间复杂度为 O (n+e)。

8. 二分图是指能将顶点划分成两个部分，每一部分内的顶点间没有边相连的简单无向图。那么，24 个顶点的二分图至多有（）条边。

A.144　　B.100　　C.48　　D.122

答案：A

分析：本题考查二分图的最大边数。

- 二分图边数最多时，两部分顶点数应尽量接近（乘积最大）。24 个顶点分为 12 和 12 时，边数 = 12×12 = 144。

9. 广度优先搜索时，一定需要用到的数据结构是（）。

A. 栈　　B. 二叉树　　C. 队列　　D. 哈希表

答案：C

分析：本题考查 BFS 的实现机制。

- BFS 通过队列实现“逐层访问”，DFS 通过栈实现“深度优先”，故答案为 C。

10. 一个班学生分组做游戏，如果每组三人就多两人，每组五人就多三人，每组七人就多四人，问这个班的学生人数 n 在以下哪个区间？已知 n<60。()。

A.30<n<40　　B.40<n<50　　C.50<n<60　　D.20<n<30

答案：C

分析：本题考查中国剩余定理的应用。

- 条件转化：
  - $n \equiv 2 \bmod 3$，
  - $n \equiv 3 \bmod 5$，
  - $n \equiv 4 \bmod 7$。
- 试算满足条件的数：
  - 满足 $n \equiv 3 \bmod 5$ 的数：3,8,13,18,23,28,33,38,43,48,53,58。
  - 筛选出 $n \equiv 2 \bmod 3$ 的数：8,23,38,53。
  - 筛选出 $n \equiv 4 \bmod 7$ 的数：53（53÷7=7 余 4）。
- 53 位于 50<n<60 区间，故答案为 C。

11. 小明想通过走楼梯来锻炼身体，假设从第 1 层走到第 2 层消耗 10 卡热量，接着从第 2 层走到第 3 层消耗 20 卡热量，再从第 3 层走到第 4 层消耗 30 卡热量，依此类推，从

第 k 层走到第 k+1 层消耗 10k 卡热量 (k>1)。如果小明想从 1 层开始，通过连续向上爬楼梯消耗 1000 卡热量，至少要爬到第几层楼？(　　)。

A.14　　B.16　　C.15　　D.13

答案：C

分析：本题考查等差数列求和。

- 消耗热量总和为 10×(1+2+...+k) = 10×k (k+1)/2 = 5k (k+1)。
- 令 5k (k+1) ≥ 1000 → k (k+1) ≥ 200。
- 计算得 k=14 时，14×15=210 ≥ 200，此时爬到第 15 层，故答案为 C。

12. 表达式 a*(b+c)-d 的后缀表达形式为 (　　)。

A.abc*+d-　　B.-+*abcd*　　*C.abcd*+-　　D.abc+*d-

答案：D

分析：本题考查表达式的后缀表示（逆波兰式）。

- 后缀表达式规则：运算符在操作数之后，按运算顺序排列。
- 步骤：
  a. 先算 b+c → 后缀为 bc+。
  b. 再算 a*(b+c) → 后缀为 abc+*。
  c. 最后算减法 → 后缀为 abc+*d-。

13. 从一个 4×4 的棋盘中选取不在同一行也不在同一列上的两个方格，共有 ( ) 种方法。

A.60　　B.72　　C.86　　D.64

答案：B

分析：本题考查组合计数。

- 步骤：
  a. 第一个方格：4×4=16 种选择。
  b. 第二个方格：排除同一行和同一列，剩 3×3=9 种选择。
  c. 去重（不考虑顺序）：16×9÷2=72 种。

14. 对一个 n 个顶点、m 条边的带权有向简单图用 Dijkstra 算法计算单源最短路时，如果不使用堆或其它优先队列进行优化，则其时间复杂度为 ( )。

A.O ((m+n²) log n)　　B.O (mn+n³)　　C.O ((m+n) log n)　　D.O (n²)

答案：D

分析：本题考查 Dijkstra 算法的时间复杂度。

- 不使用堆优化时，每次需遍历所有顶点找最小距离（O (n)），共需 n 次，总时间复杂度为 O (n²)。

15. 1948 年，( ) 将热力学中的熵引入信息通信领域，标志着信息论研究的开端。

A. 欧拉 (Leonhard Euler)　　B. 冯·诺伊曼 (John von Neumann)

C. 克劳德·香农 (Claude Shannon)　　D. 图灵 (Alan Turing)

答案：C

分析：本题考查信息论的创始人。

- 克劳德·香农于 1948 年发表《通信的数学理论》，将熵引入信息论，标志着信息论的诞生。

二、阅读程序题

1.

```
01  #include <iostream>
02  using namespace std;
03
04  int n;
05  int d[1000];
06
07  int main() {
08    cin >> n;
09    for (int i = 0; i < n; ++i)
10      cin >> d[i];
11    int ans = -1;
12    for (int i = 0; i < n; ++i)
13      for (int j = 0; j < n; ++j)
14        if (d[i] < d[j])
15          ans = max(ans, d[i] + d[j] - (d[i] & d[j]));
16      cout << ans;
17      return 0;
18  }
```

假设输入的 n 和 d[i]都是不超过 10000 的正整数，完成下面的判断题和单选题：

● 判断题

1)n 必须小于 1000,否则程序可能会发生运行错误。( )

2)输出一定大于等于 0。( )

3)若将第 13 行的“j=0”改为“j=i+1”,程序输出可能会改变。( )

4)将第 14 行的“d[i]<d[j]”改为“d[i]!=d[j]”,程序输出不会改变。( )

● 单选题

5)若输入 n 为 100,且输出为 127,则输入的 d[i]中不可能有( )。

A. 127　　B. 126　　C. 128　　D. 125

6)若输出的数大于 0,则下面说法正确的是( )。

A. 若输出为偶数，则输入的 d[i]中最多有两个偶数

B. 若输出为奇数，则输入的 d[i]中至少有两个奇数

C. 若输出为偶数，则输入的 d[i]中至少有两个偶数

D. 若输出为奇数，则输入的 d[i]中最多有两个奇数

该程序用于求解**数组中所有满足 d[i] < d[j]的数对的 d[i] | d[j]（等价于** d[i] + d[j] - (d[i] & d[j]）**）的最大值**。核心逻辑基于二进制位运算的性质：a + b - (a & b) = a | b（位或运算），通过两层循环遍历所有数对，筛选出 d[i] < d[j]的情况并计算位或结果，最终保留最大值。

```
#include <iostream>
using namespace std;

int n;                    // 存储输入的元素个数
int d[1000];              // 存储输入的整数数组（大小固定为 1000）

int main() {
   cin >> n;              // 读取元素个数 n
   for (int i = 0; i < n; ++i)
      cin >> d[i];        // 读取 n 个整数到数组 d 中
   int ans = -1;          // 初始化结果变量为-1（无有效元素对时的默认值）
   // 双层循环遍历所有可能的(i,j)元素对
   for (int i = 0; i < n; ++i)
13       for (int j = 0; j < n; ++j)
         if (d[i] < d[j])   // 筛选出 d[i]小于 d[j]的元素对
            // 计算 d[i] + d[j] - (d[i] & d[j])，并更新最大值
            ans = max(ans, d[i] + d[j] - (d[i] & d[j]));
   cout << ans;           // 输出结果
   return 0;
}
```

输入：n=3，d = [1, 2, 3]

执行步骤：

读入 n=3 和数组 d = [1, 2, 3]，初始化 ans = -1（存储最终结果）。

外层循环 i=0（此时 d[i] = 1）：

内层循环 j=0：d[j] = 1，判断 d[i] == d[j]，不成立，跳过。

内层循环 j=1：d[j] = 2， d[i] < d[j]成立。

计算表达式：d[i] + d[j] - (d[i] & d[j]) = 1 + 2 - (1 & 2)。

其中 1 & 2（二进制 01 & 10）结果为 0，因此表达式值为 3 - 0 = 3。

更新 ans = max(-1, 3) = 3。

内层循环 j=2：d[j] = 3，判断 d[i] < d[j]（成立。

计算表达式：1 + 3 - (1 & 3)。
其中 1 & 3（二进制 01 & 11）结果为 1，因此表达式值为 4 - 1 = 3。
更新 ans = max(3, 3) = 3（无变化）。

外层循环 i=1（此时 d[i] = 2）：

内层循环 j=0：d[j] = 1，判断 d[i] < d[j] 不成立，跳过。
内层循环 j=1：d[j] = 2，判断 d[i] < d[j] 不成立，跳过。

内层循环 j=2：d[j] = 3，判断 d[i] < d[j]（2 < 3）？成立。
计算表达式：2 + 3 - (2 & 3)。
其中 2 & 3（二进制 10 & 11）结果为 2，因此表达式值为 5 - 2 = 3。
更新 ans = max(3, 3) = 3（无变化）。

外层循环 i=2（此时 d[i] = 3）：

- 判断题

1. n 必须小于 1000，否则程序可能会发生运行错误。( )

答案：×

分析：n=1000 也可以

2. 输出一定大于等于 0。( )

答案：×

分析：n=1 时，输出-1，特例。

3. 若将第 13 行的“j=0”改为“j=i+1”，程序输出可能会改变。( )

答案：√

分析：原程序遍历所有 j（包括 j < i 的情况），只要 d[i] < d[j]就计算结果。修改后 j 从 i+1 开始，仅考虑 j > i 的情况，可能漏掉 j < i 且 d[j] > d[i]的有效数对（如 i=1, j=0 且 d[1]=2, d[0]=3），导致最大值变小，输出可能改变。

4. 将第 14 行的“d [i]<d [j]”改为“d [i]!=d [j]”，程序输出不会改变。( )

答案：√

分析：原条件 d[i] < d[j]仅计算 i、j 满足 d[i] < d[j]的对。若改为 d[i] != d[j]，会额外包含 d[i] > d[j]的对。而按位或运算满足交换律（d[i] &d[j] = d[j] & d[i]），因此 d[i]+d[j]-d[i]&d[j] 与 d[j]+d[i]-d[j]&d[i]的结果相同。因此最大值不会改变，该说法正确。

- 单选题

5) 若输入 n 为 100，且输出为 127，则输入的 d [i] 中不可能有 ( )。

A.127　　B.126　　C.128　　D.125

答案：C

分析：输出 127 表示数组中所有满足 d[i] < d[j]的对的 d[i] + d[j] - (d[i] & d[j])最大值为 127。

128 的二进制为 10000000。若数组中存在 128，取任意小于 128 的数 x（x < 128，二进制最高位为 0），则：
x & 128 = 0（因 x 最高位为 0，128 最高位为 1，按位与结果为 0），
因此 x + 128 - (x & 128) = x + 128 - 0 = x + 128。
由于 x ≥ 1（正整数），结果至少为 1 + 128 = 129，这比 127 大，与“输出为 127”矛盾。
因此，数组中不可能有 128，选 C。

6) 若输出的数大于 0，则下面说法正确的是（）。

A. 若输出为偶数，则输入的 d [i] 中最多有两个偶数

B. 若输出为奇数，则输入的 d [i] 中至少有两个奇数

C. 若输出为偶数，则输入的 d [i] 中至少有两个偶数

D. 若输出为奇数，则输入的 d [i] 中最多有两个奇数

答案：C
输出为 d[i] + d[j] - (d[i] & d[j])（a < b，且结果 > 0），分析其奇偶性与 d[i]、d[j]的关系：

一个数的奇偶性由二进制末位决定（0 为偶，1 为奇）。

对末位而言：

若 d[i]末位为 0（偶），d[j]末位为 0（偶）：0+0 - (0&0) = 0-0=0（结果末位为 0，偶数）。
若 d[i]末位为 0，d[j]末位为 1（奇）：0+1 - (0&1)=1-0=1（结果末位为 1，奇数）。
若 d[i]末位为 1，d[j]末位为 0：1+0 - (1&0)=1-0=1（结果末位为 1，奇数）。
若 d[i]末位为 1，d[j]末位为 1：1+1 - (1&1)=2-1=1（结果末位为 1，奇数）。
由此可知：
结果为偶数 仅当 d[i]和 d[j]均为偶数（末位都是 0），即数组中至少有两个偶数（C 正确）。
结果为奇数时，d[i]和 d[j]可仅一个为奇数（如 d[i]=2（偶）、d[j]=3（奇），结果为 2+3-(2&3)=5-2=3（奇）），因此 B 错误。
A 错误：输出为偶数时，数组可包含多个偶数（如 3 个偶数 2,4,6，其对的计算结果可能仍为偶数）。
D 错误：输出为奇数时，数组可包含多个奇数（如 3 个奇数 1,3,5，其对的计算结果可能仍为奇数）。

因此选 C。

2.

```
01  #include <iostream>
02  #include <cstdlib>
03  using namespace std;
04
```

```
05  int n;
06  int d[10000];
07
08  int find(int L, int R, int k) {
09      int x = rand() % (R - L + 1) + L;
10      swap(d[L], d[x]);
11      int a = L + 1, b = R;
12      while (a < b) {
13          while (a < b && d[a] < d[L])
14              ++a;
15          while (a < b && d[b] >= d[L])
16              --b;
17          swap(d[a], d[b]);
18      }
19      if (d[a] < d[L])
20          ++a;
21      if (a - L == k)
22          return d[L];
23      if (a - L < k)
24          return find(a, R, k - (a - L));
25      return find(L + 1, a - 1, k);
26  }
27
28  int main() {
29      int k;
30      cin >> n;
31      cin >> k;
32      for (int i = 0; i < n; ++i)
33          cin >> d[i];
34      cout << find(0, n - 1, k);
35      return 0;
36  }
```

假设输入的 n,k 和 d[i]都是不超过 10000 的正整数，且 k 不超过 n,并假设 rand()函数产生的是均匀的随机数，完成下面的判断题和单选题：

●判断题

1)第 9 行的“x”的数值范围是 L+1 到 R,即[L+1,R]。( )

2)将第 19 行的“d[a]”改 为“d[b]”,程序不会发生运行错误。( )

●单选题

3)(2.5 分)当输入的 d[i]是严格单调递增序列时，第 17 行的“swap”平均执行次数是( )。

A.O(n log n)　　B.O(n)　　C.O(log n)　　D.O(n^2)

4)(2.5 分)当输入的 d[i]是严格单调递减序列时，第 17 行的“swap”平均执行次数是( )。

A.O(n^2)　　B.O(n)　　C.O(n log n)　　D.O(log n)

5)(2.5 分)若输入的 d[i]为 i，此程序①平均的时间复杂度和②最坏情况下的时间复杂度分别是()。

A.O(n),O(n^2)　　B.O(n),O(n log n)

C.O(n log n),O(n^2)　　D.O(n log n),O(n log n)

6)(2.5 分) 若 输 入 的 d[i]都为同一个数，此程序平均的时间复杂度是 ()。

A.O(n)　　B.O(logn)　　C.O(n logn)　　D.O(n^2)

假设输入的 n,k 和 d [i] 都是不超过 10000 的正整数，且 k 不超过 n，并假设 rand () 函数产生的是均匀的随机数，完成下面的判断题和单选题：

该程序实现了**快速选择（Quickselect）算法**，用于在无序数组中高效查找第 k 小的元素。其核心思想与快速排序（Quicksort）类似：通过随机选择 pivot（基准元素）进行分区，将数组分为“小于 pivot”和“大于等于 pivot”两部分，然后根据 pivot 的位置递归处理目标所在的子区间，最终找到第 k 小的元素。与完全排序后查找（时间复杂度 O (n log n)）相比，快速选择算法平均时间复杂度为 O (n)，适用于大规模数据的查找场景。

```
#include <iostream>
#include <cstdlib>
using namespace std;

int n;              // 数组长度
int d[10000];    // 存储数据的数组

// 快速选择算法：在 d[L..R]中查找第 k 小的元素
int find(int L, int R, int k) {
    // 随机选择主元
    int x = rand() % (R - L + 1) + L;   // x ∈ [L, R]
    swap(d[L], d[x]);   // 将主元交换到首位

    int a = L + 1, b = R;
    // 分区操作：将小于主元的放左侧，大于等于主元的放右侧
    while (a < b) {
        while (a < b && d[a] < d[L]) ++a; // 从左向右找第一个≥主元的元素
        while (a < b && d[b] >= d[L])   --b;   // 从右向左找第一个<主元的元素
        swap(d[a], d[b]);//交换，将<d[L]的都移动到左侧，将>=d[L]的都移动到右侧
    }
    // 调整分区点
    if (d[a] < d[L])
        ++a;
//完成后，[L,R]分成三部分，[L]、[L+1,a-1]、[a,R],值分别是 d[L]、<d[L]、>=d[L]

    //情况 1:a-L==k,最左侧的 d[L]就是答案，举例 k=3,L=3,a=6,数字是 5,2,4,6..
    if (a - L== k)
```

```
            return d[L];   //第 3 小的数是 5

//情况 2:a-L<k,在右侧区间[a,R]找，参数 3 要减去[L,a-1]的元素个数，因为下次从下标[a]开始找
    if (a - L< k)
            return find(a, R, k - (a-L));   // 在右分区查找

//情况 3:a-L>k,仕左侧区间[L+1,a-1]找，参数 3 还是 k
    return find(L + 1, a - 1, k);   // 在左分区查找
}
int main() {
    int k;
    cin >> n >> k;
    for (int i = 0; i < n; ++i)
            cin >> d[i];
    cout << find(0, n - 1, k);
    return 0;
}
```

示例 1：

输入：n=5, k=3，数组 d = [3, 1, 4, 2, 5]（目标：第 3 小元素，预期结果为 3）

执行步骤：

main 函数读入数据，调用 find(0, 4, 3)（查找 [0..4] 区间的第 3 小元素）。

第一次调用 find(0,4,3)：

随机选择主元：假设 x=0（即主元为 d[0]=3，无需交换）。

初始化 a=1，b=4。

分区循环：

a=1：d[1]=1 < 3 → a 增至 2；d[2]=4 ≥ 3 → 左指针停在 a=2。

b=4：d[4]=5 ≥ 3 → b 减至 3；d[3]=2 < 3 → 右指针停在 b=3。

交换 d[2]和 d[3] → 数组变为[3,1,2,4,5]。

此时 a=2 < b=3，继续循环：

a=2：d[2]=2 < 3 → a 增至 3，此时 a=b=3，循环结束。

调整 a：d[3]=4 ≥ 3 → 无需调整，a=3。

左分区大小 left_size = a - L = 3 - 0 = 3（左分区为[0..2]，包含 3 个元素：3、1、2）。

因 left_size = k=3，直接返回 d[L]=d[0]=3。（注意这里没有像排序的时候，将 主元进行交换）

输出结果 3，程序结束。

示例 2：含重复元素的数组

输入：n=6, k=4，数组 d = [5, 3, 3, 1, 2, 4]（目标：第 4 小元素，预期结果为 3）

执行步骤：

main 函数读入数据，调用 find(0,5,4)（查找 [0..5] 区间的第 4 小元素）。

第一次调用 find(0,5,4)：
随机选择主元：假设 x=2（d[2]=3），交换 d[0]和 d[2] → 数组变为[3,3,5,1,2,4]（主元 d[0]=3）。
初始化 a=1，b=5。
分区循环：
a=1：d[1]=3 < 3？否（3 不小于 3）→ 左指针停在 a=1。
b=5：d[5]=4 ≥ 3 → b 减至 4；d[4]=2 < 3 → 右指针停在 b=4。
交换 d[1]和 d[4] → 数组变为[3,2,5,1,3,4]。
此时 a=1 < b=4，继续循环：
a=1：d[1]=2 < 3 → a 增至 2；d[2]=5 ≥ 3 → 左指针停在 a=2。
b=4：d[4]=3 ≥ 3 → b 减至 3；d[3]=1 < 3 → 右指针停在 b=3。
交换 d[2]和 d[3] → 数组变为[3,2,1,5,3,4]。
此时 a=2 < b=3，继续循环：
a=2：d[2]=1 < 3 → a 增至 3，此时 a=b=3，循环结束。（第一轮）

调整 a：d[3]=5 ≥ 3 → 无需调整，a=3。
左分区大小 left_size = 3 - 0 = 3（左分区为[0..2]，包含 3 个元素：3、2、1）。
因 left_size=3 < k=4，需在右分区[3..5]查找第 4-3=1 小元素，调用 find(3,5,1)。

调用 find(3,5,1)：

随机选择主元：假设 x=3（d[3]=5），无需交换。
初始化 a=4，b=5。
分区循环：
a=4：d[4]=3 < 5 → a 增至 5，此时 a=b=5，循环结束。
调整 a：d[5]=4 < 5 → a 增至 6。
左分区大小 left_size = 6 - 3 = 3（左分区为[3..5]，包含 3 个元素：5、3、4）。
因 left_size=3 > k=1，需在左分区[4..5]查找第 1 小元素，调用 find(4,5,1)。

调用 find(4,5,1)：

随机选择主元：假设 x=4（d[4]=3），无需交换。
初始化 a=5，b=5 → 循环不执行。
调整 a：d[5]=4 ≥ 3 → 无需调整，a=5。
左分区大小 left_size = 5 - 4 = 1（左分区为[4..4]，包含 1 个元素：3）。
因 left_size=1 = k=1，返回 d[4]=3。

最终输出结果 3，程序结束。

●判断题

1. 第 9 行的“x”的数值范围是 L+1 到 R，即 [L+1,R]。( )

答案：×

分析：分析：第 9 行代码为 x = rand() % (R - L + 1) + L。其中 rand() % (R - L + 1) 生成的随机数范围是 0 到 (R-L)（包含两端），加上 L 后，x 的范围是 [L, R]（包含 L），而非 [L+1, R]。因此该说法错误。

2. 将第 19 行的“d [a]”改为“d [b]”，程序不会发生运行错误。( )

答案：√

分析：第 12-18 行的循环结束时，a >= b；由于循环条件是 a < b，退出循环时必然 a == b（否则会继续循环）。因此 d[a]和 d[b]指向同一个元素，修改后判断条件不变，程序逻辑不受影响，不会发生运行错误。

●单选题

3) 当输入的 d [i] 是严格单调递增序列时，第 17 行的“swap”平均执行次数是 ( )。

A.O (n log n)　　B.O (n)　　C.O (log n)　　D.O (n²)

答案：B (官方答案：ABCD 都可以，有歧义)

分析：快速选择算法在均匀随机 主元 选择下，平均时间复杂度为 O (n)，swap 操作次数与元素比较次数同阶，故为 O (n)。

4) 当输入的 d [i] 是严格单调递减序列时，第 17 行的“swap”平均执行次数是 ( )。

A. O (n²)　　B.O (n)　　C.O (n log n)　　D.O (log n)

答案：B

分析：17 行的 swap 发生在分区过程中。随机选择 主元 时，分区平均是平衡的（期望分区点为中位数）。每次分区交换次数与分区大小成线性关系，总交换次数的期望为 O(n) + O(n/2) + O(n/4) + ... = O(n)。因此，平均执行次数为 O(n)。

$$1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{2^n}+\ldots$$

等比级数前n项的和S_n公式为：

$$S_n = a \cdot \frac{1-r^n}{1-r}$$

代入$a=1$，$r=\frac{1}{2}$：

$$S_n = 1 \cdot \frac{1-(\frac{1}{2})^n}{1-\frac{1}{2}} = 2 \cdot \left[1-\left(\frac{1}{2}\right)^n\right]$$

当n趋向于无穷大时，$(\frac{1}{2})^n$趋向于 0（因为$\frac{1}{2}$的无穷次幂无限接近 0）：

$$\lim_{n\to\infty} S_n = 2 \cdot (1-0) = 2$$

5) 输入的 d [i] 为 i，此程序①平均的时间复杂度和②最坏情况下的时间复杂度分别是 ()。

A.O (n),O (n²)　　B.O (n),O (n log n)

C.O (n log n),O (n²)　　D.O (n log n),O (n log n)

答案：A

平均情况：随机选择 主元 使每次分区后问题规模减半，总操作次数为 O (n)（n + n/2 + n/4 + ... ≈ 2n）。

最坏情况：若每次随机选择的 主元 都是当前区间的最小元素（概率极低），则每次分区仅减少一个元素，总操作次数为 O (n²)（n + (n-1) + ... + 1 ≈ n²/2）。

因此答案为 O (n) 和 O (n²)。

假设 k=1,最坏情况下，每次分治，选择的 X 对应的比较基准 d[L]都是[L,R]区间内的最大值，那么每次分治数据规模只减少 1,每次分治都会对[L,R]区间内所有数据遍历一遍，所以时间复杂度是 O(n²)

1. 若输入的 d [i] 都为同一个数，此程序平均的时间复杂度是 ()。

A.O (n) B.O (log n) C.O (nlogn) D.O (n^2)

答案：D

分析：当所有元素相等时，分区操作中 d[a] < d[L]恒为假，a 始终为 L+1。因此每次递归仅能将问题规模减少 1（处理区间为 [L+1, R]），总操作次数为 O (n^2)（n + (n-1) + … + 1 ≈ $n^2/2$）。即使随机选择 pivot，也无法改变这一结果，故平均时间复杂度为 O (n^2)。

3.

```
#include <iostream>
#include <queue>
using namespace std;

const int maxl = 20000000;

class Map {
    struct item {
        string key; int value;
    } d[maxl];
    int cnt;
  public:
    int find(string x) {
        for (int i = 0; i < cnt; ++i)
            if (d[i].key == x)
                return d[i].value;
        return -1;
    }
    static int end() { return -1; }
    void insert(string k, int v) {
        d[cnt].key = k; d[cnt++].value = v;
    }
} s[2];

class Queue {
    string q[maxl];
    int head, tail;
  public:
    void pop() { ++head; }
    string front() { return q[head + 1]; }
    bool empty() { return head == tail; }
    void push(string x) { q[++tail] = x;  }
} q[2];

string st0, st1;
int m;

string LtoR(string s, int L, int R) {
    string t = s;
    char tmp = t[L];
    for (int i = L; i < R; ++i)
        t[i] = t[i + 1];
    t[R] = tmp;
    return t;
}

string RtoL(string s, int L, int R) {
    string t = s;
    char tmp = t[R];
    for (int i = R; i > L; --i)
```

```
51          t[i] = t[i - 1];
52      t[L] = tmp;
53      return t;
54  }
55
56  bool check(string st , int p, int step) {
57      if (s[p].find(st) != s[p].end())
58          return false;
59      ++step;
60      if (s[p ^ 1].find(st) == s[p].end()) {
61          s[p].insert(st, step);
62          q[p].push(st);
63          return false;
64      }
65      cout << s[p ^ 1].find(st) + step << endl;
66      return true;
67  }
68
69  int main() {
70      cin >> st0 >> st1;
71      int len = st0.length();
72      if (len != st1.length()) {
73          cout << -1 << endl;
74          return 0;
75      }
76      if (st0 == st1) {
77          cout << 0 << endl;
78          return 0;
79      }
80      cin >> m;
81      s[0].insert(st0, 0); s[1].insert(st1, 0);
82      q[0].push(st0); q[1].push(st1);
83      for (int p = 0;
84              !(q[0].empty() && q[1].empty());
85              p ^= 1) {
86          string st = q[p].front(); q[p].pop();
87          int step = s[p].find(st);
88          if ((p == 0 &&
89                  (check(LtoR(st, m, len - 1), p, step) ||
90                   check(RtoL(st, 0, m), p, step)))
91                  ||
92                  (p == 1 &&
93                   (check(LtoR(st, 0, m), p, step) ||
94                    check(RtoL(st, m, len - 1), p, step))))
95              return 0;
96      }
97      cout << -1 << endl;
98      return 0;
99  }
```

●判断题
1)输出可能为 0。（ ）
2)若输入的两个字符串长度均为 101 时，则 m=0 时的输出与 m=100 时的输出是一样的。（ ）
3)若两个字符串(长度均为 n,则最坏情况下，此程序的时间复杂度为 O(n!)。（ ）
●单选题
4)(2.5 分)若输入的第一个字符串长度由 100 个不同的字符构成，第二个字符串是第一个字符串的倒序，输入的 m 为 0,则输出为( )。
A.49 B.50 C.100 D.-1
5)(4 分)已知当输入为“0123\n3210\n1”时输出为 4,当输入为“012345\n543210\n1”时输出为 14,当输入为“01234567\n76543210\n1”时输出为 28,则当输入为“0123456789ab\nba9876543210\n1”输出为( )。其中“\n”为换行符。
A.56 B.84 C.102 D.68

6)(4 分)若两个字符串的长度均为 n,且 0<m<n-1,且两个字符串的构成相同(即任何一个字符在两个字符串中出现的次数均相同),则下列说法正确的是( )。提示：考虑输入与输出有多少对字符前后顺序不一样。
A.若 n、m 均为奇数，则输出可能小于 0。
B.若 n、m 均为偶数，则输出可能小于 0。
C.若 n 为奇数、m 为偶数，则输出可能小于 0。
D.若 n 为偶数、m 为奇数，则输出可能小于 0。

程序含义

该程序通过**双向广度优先搜索（BFS）** 求解两个字符串之间的最少变换步骤。变换规则由 `LtoR` 和 `RtoL` 函数定义，分别实现特定范围内的字符移动。程序从初始字符串（正向）和目标字符串（反向）同时开始搜索，当两个方向的搜索相遇时，得到最少步骤数；若无法相遇则输出 `-1`。

核心知识点

1. **双向 BFS**：从起点和终点同时搜索，减少搜索空间，提高效率（时间复杂度优于单向 BFS）。
2. **状态变换**：`LtoR` 和 `RtoL` 实现特定范围内的字符移动，是问题的核心操作。
3. **状态判重**：通过 `Map` 类记录已访问的字符串状态，避免重复处理（关键优化）。
4. **队列管理**：`Queue` 类用于存储待处理的状态，确保 BFS 按层次扩展（保证步骤数最少）。

```
#include <iostream>

#include <queue>

using namespace std;


const int maxl = 20000000;  // 最大存储容量
```

```
// Map 类：存储字符串与对应步数的映射（模拟哈希表）
class Map {
    struct item {
        string key;   // 字符串键
        int value;    // 对应的步数
    } d[maxl];        // 存储数组
    int cnt;          // 当前存储的元素数量
  public:
    // 查找字符串 x 对应的步数，未找到返回-1
    int find(string x) {
        for (int i = 0; i < cnt; ++i)
            if (d[i].key == x)
                return d[i].value;
        return -1;
    }
    // 静态方法：返回-1（表示未找到）
    static int end() { return -1; }
    // 插入键值对
    void insert(string k, int v) {
        d[cnt].key = k;
        d[cnt++].value = v;
    }
} s[2];   // s[0]用于正向搜索，s[1]用于反向搜索

// Queue 类：简单队列实现
```

```
class Queue {
    string q[maxl];   // 队列数组
    int head, tail;   // 头指针和尾指针
  public:
    void pop() { ++head; }                    // 出队（头指针后移）
    string front() { return q[head + 1]; }   // 获取队首元素
    bool empty() { return head == tail; }    // 判断队列是否为空
    void push(string x) { q[++tail] = x; }   // 入队（尾指针后移）
} q[2];   // q[0]正向队列，q[1]反向队列

string st0, st1;   // 初始字符串和目标字符串
int m;             // 操作参数

// 将 s 中位置 L 的字符移动到位置 R，中间（L 到 R-1）的字符依次左移一位。
//根据 BFS 方向（正向 / 反向）生成符合规则的下一步字符串（如正向搜索时执行 LtoR(st, m, len-1)）。
string LtoR(string s, int L, int R) {
    string t = s;
    char tmp = t[L];   // 保存 L 位置的字符
    for (int i = L; i < R; ++i)
        t[i] = t[i + 1];   // 中间字符左移
    t[R] = tmp;   // L 位置的字符放到 R 位置
    return t;
}
```

若 s="01234"，L=1，R=3：

步骤 1：保存 t[1]='1'；

步骤 2：i=1 时 t[1]=t[2]='2'，i=2 时 t[2]=t[3]='3'（中间字符左移）；

步骤 3：t[3]='1'（原 L 位置字符放到 R 位置）；

结果：t="02314"。

// 将 s 中位置 R 的字符移动到位置 L，中间（L+1 到 R）的字符依次右移一位。

```
string RtoL(string s, int L, int R) {
    string t = s;
    char tmp = t[R];   // 保存 R 位置的字符
    for (int i = R; i > L; --i)
        t[i] = t[i - 1];   // 中间字符右移
    t[L] = tmp;   // R 位置的字符放到 L 位置
    return t;
}
```

// check 是双向 BFS 的核心判断函数，用于检查当前生成的字符串

//是否为 “两个方向搜索的交点”，并更新 BFS 状态。

```
bool check(string st, int p, int step) {
    // 若当前方向已访问过该字符串，无需处理
    if (s[p].find(st) != s[p].end())
        return false;
    ++step;   // 步数加 1（当前操作的步数）
```

```
        // 若对面方向未访问过，记录状态并加入队列
        if (s[p ^ 1].find(st) == s[p].end()) {
            s[p].insert(st, step);
            q[p].push(st);
            return false;
        }

        // 若对面方向已访问，输出总步数并返回找到解
        cout << s[p ^ 1].find(st) + step << endl;
        return true;
}

int main() {
        cin >> st0 >> st1;
        int len = st0.length();

        // 若长度不同，直接输出-1（无法转换）
        if (len != st1.length()) {
            cout << -1 << endl;
            return 0;
        }

        // 若两字符串相同，输出 0
        if (st0 == st1) {
```

```
        cout << 0 << endl;
        return 0;
}

cin >> m;

// 初始化：正向和反向分别插入初始状态，步数为 0
s[0].insert(st0, 0);
s[1].insert(st1, 0);
q[0].push(st0);
q[1].push(st1);

// 双向 BFS 循环：p=0（正向）和 p=1（反向）交替搜索
for (int p = 0; !(q[0].empty() && q[1].empty()); p ^= 1) {
        string st = q[p].front();   // 获取当前方向的队首字符串
        q[p].pop();                 // 出队
        int step = s[p].find(st);   // 获取当前步数

        // 根据当前方向（p）执行对应的操作，并检查是否找到解
        if ((p == 0 &&   // 正向操作：LtoR(m, len-1) 和 RtoL(0, m)
                (check(LtoR(st, m, len - 1), p, step) ||
                 check(RtoL(st, 0, m), p, step)))
                ||
                (p == 1 &&   // 反向操作：LtoR(0, m) 和 RtoL(m, len-1)
                 (check(LtoR(st, 0, m), p, step) ||
```

```
                    check(RtoL(st, m, len - 1), p, step))))
                return 0;   // 找到解，退出程序
        }

        // 若队列空仍未找到解，输出-1
        cout << -1 << endl;
        return 0;
}
```

双向BFS复杂度：

双向BFS就是从起点和终点两个状态同时出发，直到某个状态同时被访问到两次，那么起点 状态->被访问两次的状态->终点状态就是起点状态到终点状态的解。那具体会有多快呢？加入每一次搜索都有n个新的状态（比如迷宫问题就是上下左右四个状态），从起点到目标路径长为m，那就要搜索$n+n^2+n^3+.....+n^m$ 个状态，状态数就是n^m数量级的，若是双向广搜呢？在路径中点就能相遇，状态数是$2*n^{m/2+1}$数量级的，比起朴素的BFS，枚举状态数少了很多，所以对于起点状态和终点状态已知的情况，可以使用双向BFS来优化程序时间复杂度。

●判断题

1. 输出可能为 0。( )

答案：√

分析：当 st0 == st1 时，程序直接输出 0，故可能为 0。

2. 若输入的两个字符串长度均为 101 时，则 m=0 时的输出与 m=100 时的输出是一样的。( )

答案：×

分析：m=0 时，LtoR 操作范围是 0 到 100，RtoL 是 0 到 0；m=100 时，LtoR 是 100 到 100，RtoL 是 100 到 100，操作不同，转换路径不同，输出可能不同。

3. 若两个字符串（长度均为 n，则最坏情况下，此程序的时间复杂度为 O (n!)。( )

答案：√

分析：字符串的可能状态数为 n!（全排列），最坏情况下需遍历所有状态，时间复杂度 O (n!)。

●单选题

4) 若输入的第一个字符串长度由 100 个不同的字符构成，第二个字符串是第一个字符串的倒序，输入的 m 为 0，则输出为 ( )。

A.49B.50C.100D.-1

答案：B

分析：m=0 时，LtoR 操作将第一个字符移到末尾。从正序到倒序需 50 步（每次移动第一个字符到末尾，100 步循环一次，倒序为 50 步）。

1. 已知当输入为“0123\n3210\n1”时输出为 4，当输入为“012345\n543210\n1”时输出为 14，当输入为“01234567\n76543210\n1”时输出为 28，则当输入为“0123456789ab\nba9876543210\n1”输出为（ ）。

A.56B.84C.102D.68

答案：A

分析：规律：长度 4→4，6→14，8→28。差值为 10（14-4）、14（28-14），推测每次增加 4，长度 12 时为 28+18+10=56（或其他递增规律），故答案为 A。

2. 若两个字符串的长度均为 n，且 0<m<n-1，且两个字符串的构成相同 (即任何一个字符在两个字符串中出现的次数均相同)，则下列说法正确的是（ ）。

A. 若 n、m 均为奇数，则输出可能小于 0。

B. 若 n、m 均为偶数，则输出可能小于 0。

C. 若 n 为奇数、m 为偶数，则输出可能小于 0。

D. 若 n 为偶数、m 为奇数，则输出可能小于 0。

答案：D

分析：程序输出 -1 仅当无解，否则为非负步数。但根据提示，当 n 为偶数、m 为奇数时，可能因逆序对数为奇数导致无法转换，输出 -1（小于 0），故选项 D 正确。

三、完善程序题

1. (分数背包)

(分数背包)小 S 有 n 块蛋糕，编号从 1 到 n。第 i 块蛋糕的价值是 w_i，体积是 v_i；。他有一个大小为 B 的盒子来装这些蛋糕，也就是说装入盒子的蛋糕的体积总和不能超过 B。
他打算选择一些蛋糕装入盒子，他希望盒子里装的蛋糕的价值之和尽量大。
为了使盒子里的蛋糕价值之和更大，他可以任意切割蛋糕。具体来说，他可以选择一个 a(0<a<1)，并将一块价值是 w，体积为 v 的蛋糕切割成两块，其中一块的价值是 a×w，体积是 a×v，另一块的价值是(1-a)×w，体积是(1-a)×v。他可以重复无限次切割操作。

现要求编程输出最大可能的价值，以分数的形式输出。
比如 n=3,B=8,三块蛋糕的价值分别是 4、4、2,体积分别是 5、3、2。那么最优的方案就是将体积为 5 的蛋糕切成两份，一份体积是 3,价值是 2.4,另一份体积是 2,价值是 1.6,然后把体积是 3 的那部分和后两块蛋糕打包进盒子。最优的价值之和是 8.4,故程序输出 42/5。
输入的数据范围为：1≤n≤1000,1≤B≤105;1≤w;,vi≤100。
提示：将所有的蛋糕按照性价比 w_i/v_i 从大到小排序后进行贪心选择。试补全程序。

```
01  #include <cstdio>
02  using namespace std;
03
04  const int maxn = 1005;
05
06  int n, B, w[maxn], v[maxn];
07
08  int gcd(int u, int v) {
09      if (v == 0)
10          return u;
11      return gcd(v, u % v);
12  }
13
14  void print(int w, int v) {
15      int d = gcd(w, v);
16      w = w / d;
17      v = v / d;
18      if (v == 1)
19          printf("%d\n", w);
20      else
21          printf("%d/%d\n" w, v);
22  }
23  void swap(int &x, int &y) {
24      int t = x; x = y; y = t;
25  }
26
27  int main() {
28      scanf("%d %d" &n, &B);
29      for (int i = 1; i <= n; i ++) {
30          scanf("%d %d", &w[i], &v[i]);
31      }
32      for (int i = 1; i < n; i ++)
33          for (int j = 1; j < n; j ++)
34              if (①) {
35                  swap(w[j], w[j + 1]);
36                  swap(v[j], v[j + 1]);
37              }
38      int curV, curW;
39      if  (②) {
40          ③
41      } else {
42          print(B * w[1] , v[1]);
43          return 0;
44      }
```

```
45      for (int i = 2; i <= n; i ++)
46          if (curV + v[i] <= B) {
47              curV += v[i];
48              curW += w[i];
49          } else {
50              print (④);
51              return 0;
52          }
53      print(⑤);
54      return 0;
55  }
```

1)①处应填( )

A.w[j]/v[j]<w[j+1]/v[j+1]　　B.w[j]/v[j]>w[j+1]/v[j+1]

C.v[j]*w[j+1]<v[j+1]*w[j]　　D.w[j]*v[j+1]<w[j+1]*v[j]

2)②处应填( )

A.w[1]<=B　　B.v[1]<=B　　C.w[1]>=B　　D.v[1]>=B

3)③处应填( )

A.print(v[1],w[1]);return 0;　　B.curV=0;curW=0;

C.print(w[1],v[1]);return 0;　　D.curV=v[1];curW=w[1];

4)④处应填( )

A.curW* v[i] + curV * w[i], v[i]

B.(curW - w[i])* v[i] + (B - curV) * w[i], v[i]

C.curW + v[i], w[i]

D.curW* v[i] + (B - curV) * w[i], v[i]

5)⑤处应填( )

A.curW,curV

B.curW,1

C.curV,curW

D.curV,1

该程序解决**分数背包问题**：在盒子容量有限的情况下，通过切割蛋糕（可取部分体积），使装入的蛋糕总价值最大。核心策略是**贪心算法**：按蛋糕的性价比（价值 / 体积）从高到低排序，优先装入性价比高的蛋糕，直到盒子装满。最终以最简分数形式输出最大价值。

核心知识点

1. **贪心算法**：分数背包问题的最优解可通过贪心策略获得 —— 优先选择性价比最高的物品，因物品可分割，无需考虑整体选择的约束。
2. **性价比排序**：比较性价比时，为避免浮点数精度误差，采用交叉相乘（`w[j] * v[j+1] < w[j+1] * v[j]`）判断 `w[j]/v[j] < w[j+1]/v[j+1]`。
3. **分数化简**：通过最大公约数（GCD）将结果化为最简分数，确保输出格式正确。
4. **部分装入处理**：当蛋糕无法完全装入时，计算可装入部分的价值（按体积比例分配价值）。

1. ①处应填（）

A.w [j]/v [j]<w [j+1]/v [j+1]　　B.w [j]/v [j]>w [j+1]/v [j+1]

C.v [j]*w [j+1]<v [j+1]*w [j]　　D.w [j]*v [j+1]<w [j+1]*v [j]

答案：D

分析：需按性价比 w/v 从大到小排序，为避免浮点数误差，比较 w [j]*v [j+1] < w [j+1]*v [j]（等价于 w [j]/v [j] < w [j+1]/v [j+1]）。

2. ②处应填（）

A.w [1]<=B　　B.v [1]<=B　　C.w [1]>=B　　D.v [1]>=B

答案：B

分析：else 分支处理第一个蛋糕体积超过 B 的情况（仅取部分），故 if 条件应为 v [1] <= B（可完整装入第一个蛋糕）。

3. ③处应填（）

A.print (v [1],w [1]);return 0;　　B.curV=0;curW=0;

C.print (w [1],v [1]);return 0;　　D.curV=v [1];curW=w [1];

答案：D

分析：当第一个蛋糕可完全装入时，需初始化当前装入的体积 curV 为 v[1]，价值 curW 为 w[1]，以便后续累加其他蛋糕的体积和价值。选项 D 正确。

4. ④处应填（）

A.curW* v [i] + curV * w [i], v [i]

B.(curW - w [i])* v [i] + (B - curV) * w [i], v [i]

C.curW + v [i], w [i]

D.curW* v [i] + (B - curV) * w [i], v [i]

答案：D

分析：当第 i 个蛋糕无法完全装入时，可装入的体积为 B - curV，对应价值为(B - curV) * w[i] / v[i]。总价值为 curW + (B - curV)*w[i]/v[i]，化为分数形式为(curW * v[i] + (B - curV) * w[i]) / v[i]，故选项 D 正确。

5. ⑤处应填（）

A.curW,curV　　B.curW,1　　C.curV,curW　　D.curV,1

答案：B

分析：若所有蛋糕均能完全装入（总容积≤B），总价值为 curW，是整数（分母为 1），故输出 curW, 1。选项 B 正确。

```cpp
#include <cstdio>
using namespace std;

const int maxn = 1005;   // 最大蛋糕数量

int n, B;                    // n：蛋糕数量；B：盒子容量
int w[maxn], v[maxn];    // w[i]：第 i 块蛋糕的价值；v[i]：第 i 块蛋糕的体积

// 计算最大公约数（用于化简分数）
int gcd(int u, int v) {
    if (v == 0)
        return u;
    return gcd(v, u % v);
}

// 以最简分数形式输出价值（分子为 w，分母为 v）
void print(int w, int v) {
    int d = gcd(w, v);   // 计算最大公约数
    w /= d;                  // 化简分子
    v /= d;                  // 化简分母
    if (v == 1)              // 若分母为 1,w/v 是整数，直接输出整数
        printf("%d\n", w);
    else                         // 否则输出分数形式
        printf("%d/%d\n", w, v);   // 原程序此处有笔误，修正为逗号
}

// 交换两个整数（用于排序）
void swap(int &x, int &y) {
    int t = x; x = y; y = t;
}

int main() {
    scanf("%d %d", &n, &B);   // 输入蛋糕数量和盒子容量
        // 输入每块蛋糕的价值和体积
    for (int i = 1; i <= n; i ++) {
        scanf("%d %d", &w[i], &v[i]);
```

```
    }
    // 按性价比（w/v）从大到小排序（冒泡排序）

    for (int i = 1; i < n; i ++)
        for (int j = 1; j < n; j ++)
            if (①) {  // 排序条件：j 的性价比低于 j+1，需要交换 D
w[j]/v[j] < w[j+1]/v[j+1]，，采用交叉相乘（w[j] * v[j+1] < w[j+1] * v[j]）判断
                swap(w[j], w[j + 1]);
                swap(v[j], v[j + 1]);
            }
    int curV, curW;  // curV：当前装入的总体积；curW：当前装入的总价值
    // 判断第一个蛋糕是否能完全装入盒子
    if (②) {  // v [1]<=B 如果第一个也就是最大的这个蛋糕 能完全装入
        ③  //curV=v [1];curW=w [1]; 初始化当前体积和价值为第一个蛋糕的体积和价值
    } else {  // 若第一个蛋糕体积超过盒子容量，只能装一部分
        print(B * w[1], v[1]);  // 价值为 (B / v[1]) * w[1]，即 B*w[1]/v[1]
        return 0;
    }
    // 依次装入后续蛋糕
    for (int i = 2; i <= n; i ++)
        if (curV + v[i] <= B) {  // 若当前蛋糕能完全装入
            curV += v[i];          // 累加体积
            curW += w[i];          // 累加价值
        } else {  // 若当前蛋糕不能完全装入，装一部分
            print(④);  // curW* v [i] + (B - curV) * w [i], v [i]  总价值为已有价值 + 部分蛋糕的价值
            return 0;
        }
    // 所有蛋糕都能完全装入，输出总价值
    print(⑤);
    return 0;
}
```

6. (最优子序列)

2.(最优子序列)取 m=16,给出长度为 n 的整数序列 $a_1, a_2, \cdots, a_n (0 \leqslant a_i < 2^m)$。对于一个二进制数 x,定义其分值 w(x)为 x+popcnt(x),其中 popcnt(x)表示 x 二进制表示中 1 的个数。对于一个子序列 $b_1, b_2, \cdots, b_k$,定义其子序列分值 S 为 $w(b_1 \oplus b_2)+w(b_2 \oplus b_3)+w(b_3 \oplus b_4)+\cdots+w(b_{k-1} \oplus b_k)$。其中⊕表示按位异或。对于空子序列，规定其子序列分值为 0。求一个子序列使得其子序列分值最大，输出这个最大值。
输入第一行包含一个整数 n($1 \leqslant n \leqslant 40000$)。接下来一行包含 n 个整数 $a_1, a_2, \cdots, a_n$
提示：考虑优化朴素的动态规划算法，将前 m/2 位和后 m/2 位分开计算。
Max[x][y]表示当前的子序列下一个位置的高 8 位是 x、最后一个位置的低 8 位是 y 时的最大价值。
试补全程序。

```
01  #include <iostream>
02
03  using namespace std;
04
05  typedef long long LL;
06
07  const int MAXN = 40000, M = 16, B = M >> 1, MS = (1 << B) - 1;
08  const LL INF = 1000000000000000LL;
09  LL Max[MS + 4][MS + 4];
10
11  int w(int x)
12  {
13      int s = x;
14      while(x)
15      {
16          ①;
17          s++;
18      }
19      return s;
20  }
21
22  void to_max(LL &x, LL y)
23  {
24      if(x < y)
25          x = y;
26  }
27
28  int main()
29  {
30      int n;
31      LL ans = 0;
32      cin >> n;
33      for(int x = 0; x <= MS; x++)
34          for(int y = 0; y <= MS; y++)
35              Max[x][y] = -INF;
36      for(int i = 1; i <= n ; i++)
37      {
38          LL a;
39          cin >> a;
40          int x = ② , y = a & MS;
41          LL v = ③;
```

```
42          for(int z = 0; z < = MS; z++)
43              to_max(v, ④);
44          for(int z = 0; z < = MS; z++)
45              ⑤;
46          to_max(ans , v);
47      }
48      cout << ans << endl;
49      return 0;
50  }
```

1)①处应填( )

A. x>>=1	B. x^=x&(x^(x+1))	C. x-=x|-x	D. x^=x&(x^(x-1))

2)②处应填( )

A. (a&MS)<<B	B. a>>B	C. a&(1<<B)	D. a&(MS<<B)

3)③处应填( )

A. -INF	B. Max[y][x]	C. 0	D. Max[x][y]

4)④处应填( )

A. Max[x][z]+w(y^z)	B. Max[x][z]+w(a^z)
C. Max[x][z]+w(x^(z<<B))	D. Max[x][z]+w(x^z)

5)⑤处应填( )

A. to_max(Max[y][z],v+w(a^(z<<B)))
B. to_max(Max[z][y],v+w((x^z)<<B))
C. to_max(Max[z][y],v+w(a^(z<<B)))
D. to_max(Max[x][z],v+w(y^z))

该程序用于求解**最优子序列分值问题**：给定长度为 n 的整数序列，每个整数为 16 位二进制数，定义子序列分值为相邻元素异或后的 w(x)（x + 二进制 1 的个数）之和，目标是找到分值最大的子序列。程序通过动态规划优化，将 16 位二进制数拆分为高 8 位和低 8 位，用 Max[x][y] 记录状态，降低时间复杂度，高效求解最大值。

核心知识点

1. **动态规划状态设计**：Max[x][y]表示子序列最后一个元素的低 8 位为 y，且下一个元素的高 8 位为 x 时的最大分值，通过拆分 16 位为前后 8 位，将状态空间从 2^16×2^16 降至 2^8×2^8，优化效率。
2. **分值计算（w 函数）**：w(x) = x + popcnt(x)，其中 popcnt(x)通过循环清除二进制最后一个 1 实现计数（x &= x-1 的等价操作）。
3. **状态转移**：对于每个元素，先计算以其为结尾的最大分值（衔接之前的状态），再更新后续可能的状态，确保覆盖所有可能的子序列扩展。
4. **异或运算性质**：相邻元素的异或结果由高 8 位和低 8 位分别异或组成，拆分处理可简化计算。

```
#include <iostream>


using namespace std;


typedef long long LL;
```

```
const int MAXN = 40000, M = 16, B = M >> 1, MS = (1 << B) - 1; // M=16, B=8, MS=255
const LL INF = 1000000000000000LL; // 无穷大值
LL Max[MS + 4][MS + 4]; // 状态数组，Max[x][y] 表示下一个位置高 8 位为 x、最后一个位置低 8 位为 y 时的最大价值

int w(int x)
{
    int s = x;
    while(x)
    {
        x^=x&(x^(x-1)); // 清除最低设置位，计算 popcnt
        s++;
    }
    return s; // 返回 x + popcnt(x)
}

void to_max(LL &x, LL y)
{
    if(x < y)
        x = y; // 更新 x 为最大值
}

int main()
{
    int n;
    LL ans = 0;
    cin >> n;
    for(int x = 0; x <= MS; x++)
        for(int y = 0; y <= MS; y++)
            Max[x][y] = -INF; // 初始化 Max 数组为负无穷
    for(int i = 1; i <= n ; i++)
```

```
    {
        LL a;
        cin >> a;
        int x = a >> B; // 提取高 8 位
        int y = a & MS; // 提取低 8 位
        LL v = 0; // 初始化当前子序列分值为 0（空子序列或单元素）
        for(int z = 0; z <= MS; z++)
            to_max(v, Max[x][z] + w(y^z)); // 更新 v，考虑所有可能的前一个低 8 位 z
        for(int z = 0; z <= MS; z++)
            to_max(Max[z][y], v + w((x^z) << B)); // 更新 Max 数组，考虑所有可能的下一个高 8 位 z
        to_max(ans , v); // 更新全局答案
    }
    cout << ans << endl;
    return 0;
}
```

①处应填（）

A.x>>=1　　B.x^=x&(x^(x+1))　　C.x-=x|-x　　D.x^=x&(x^(x-1))

答案：D

w(x)需要计算 popcnt(x)（x 的二进制中 1 的个数）。核心是循环清除 x 中最后一个 1，计数直到 x 为 0。

选项 D：x ^= x & (x ^ (x - 1))等价于 x &= x - 1（清除最后一个 1）。例如 x=5（101），x-1=100，x ^ (x-1)=001，x & 001=001，x ^= 001 后 x=100，成功清除最后一个 1。
其他选项：A（右移）无法计数 1 的个数；B（基于 x+1）用于进位场景，不适合清除最后一个 1；C（x-=x|-x）逻辑错误。
故 D 正确。

2. ②处应填（）

A. (a&MS)<<B　　B.a>>B　　C.a&(1<<B)　　D.a&(MS<<B)

答案：B
分析：需要提取 16 位整数 a 的高 8 位。

M=16，B=8（前 8 位与后 8 位的分割点），高 8 位可通过右移 8 位得到，即 a >> B。

选项 A（低 8 位左移）、C（仅第 8 位）、D（高 8 位掩码）均错误。
故 B 正确。

3. ③处应填（ ）

A.-INF　　B.Max [y][x]　　C.0　　D.Max [x][y]

答案：C

分析：v 表示以当前元素 a 为结尾的子序列的最大分值。

初始状态：子序列仅包含 a 时，无相邻元素，分值为 0。
选项 A（负无穷）、B（Max [y][x]）、D（Max [x][y]）均不符合初始状态定义。
故 C 正确。

4. ④处应填（ ）

A.Max [x][z]+w (y^z)　　B.Max [x][z]+w (a^z)

C.Max [x][z]+w (x^(z<<B))　　D.Max [x][z]+w (x^z)

答案：A

分析：需衔接之前的状态 Max[x][z]（最后一个元素低 8 位为 z，下一个高 8 位为 x），当前元素低 8 位为 y，两者异或的低 8 位为 y ^ z，对应分值 w(y ^ z)。

前一个元素高 8 位为 x，当前元素高 8 位也为 x（②处的 x），故高 8 位异或为 0，仅需考虑低 8 位异或。
选项 B（a ^ z）、C（x ^ (z<<B)）、D（x ^ z）均错误，未正确计算低 8 位异或。
故 A 正确。

1. ⑤处应填（ ）

A.to_max (Max [y][z],v+w (a^(z<<B)))

B.to_max (Max [z][y],v+w ((x^z)<<B))

C.to_max (Max [z][y],v+w (a^(z<<B)))

D.to_max (Max [x][z],v+w (y^z))

答案：B

分析：更新 Max 数组，对于每个 z（高 8 位），计算 v + w((x^z)<<B)，其中 w((x^z)<<B) 对应高 8 位异或的分值，并更新 Max[z][y]。