“知乎杯”2018 CCF 大学生
计算机系统与程序设计竞赛

CCF CCSP 2018

时间：2018 年 10 月 25 日 09:00 ~ 21:00

题目名称	绝地求生	贪心算法	卷积	分组加密器	内存分配器
题目类型	传统型	传统型	传统型	传统型	传统型
输入	标准输入	标准输入	标准输入	标准输入	标准输入
输出	标准输出	标准输出	标准输出	标准输出	标准输出
每个测试点时限	4.0 秒	2.0 秒	3.0 秒	4.0 秒	30.0 秒
内存限制	512 MB	512 MB	1.5 GB	512 MB	256 MB
子任务数目	5	4	0	5	0
测试点是否等分	否	否	是	否	是

绝地求生（battleground）

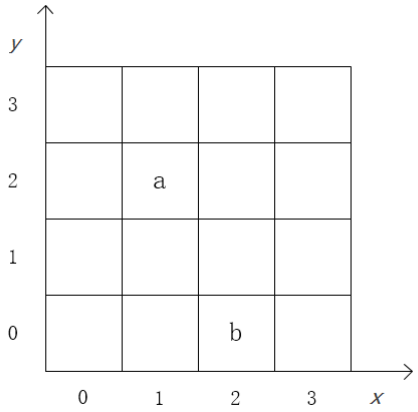
【题目描述】

《绝地求生》是一款战术竞技型射击类沙盒游戏，玩家需要在游戏地图上收集各种资源，并在不断缩小的安全区域内对抗其他玩家，让自己生存到最后。

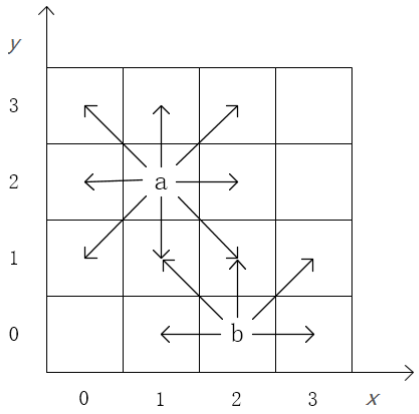
本题简化了游戏规则，需要你计算出最终的游戏结果，简化版规则如下。

【游戏规则】

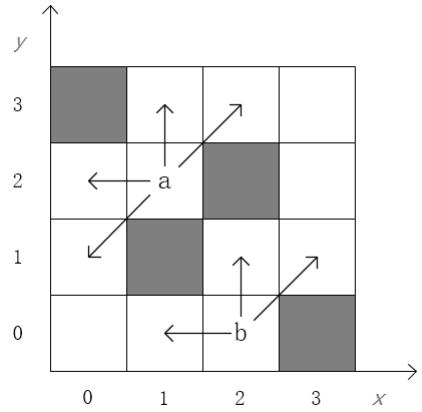
游戏地图是 $n \times n$ 的正方形棋盘，由 1×1 的方格组成，每个玩家用一个 1×1 的方格表示。



在不超出棋盘边界的情况下，玩家可以向八个方向（上、下、左、右、左上、左下，右上、右下）移动，进入周围的格子，一次移动称为一步。下图示意性地给出了玩家 a 和玩家 b 可能的移动方向，由于玩家 b 位于棋盘的边缘，因此可能的移动方向仅有 5 种。



棋盘中可能有**障碍物**，障碍物也是 1×1 的方格，玩家不能进入障碍物的方格，也不能穿越两个斜向相邻障碍物方格的间隙。



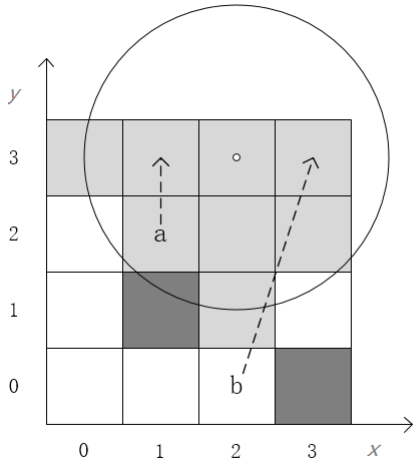
不同玩家之间**互不影响**，他们可以出现在一个方格里面。

【游戏流程】

游戏开始时，会给出棋盘的大小、玩家数量、障碍物数量、每个障碍物的位置、每个玩家的初始位置。所有的玩家在游戏开始时，都会被赋予相等的“生命值”。一次游戏分为多个回合，在游戏开始时，会给出本次游戏的回合数目。

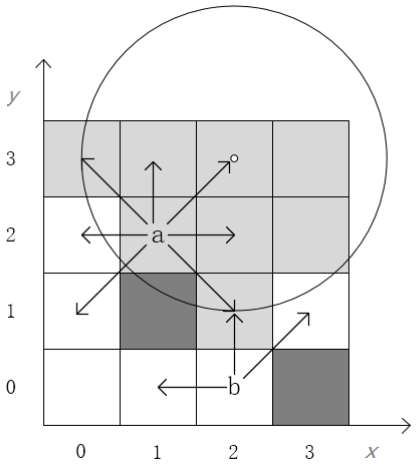
每回合开始时，都会给出每个玩家的目标位置。在这一回合内，玩家需要从上一回合结束时的位置（对于第一回合则为初始位置）移动到这一回合的目标位置，移动的步数不限。如果玩家在这一回合的起始位置和某一障碍物重合，那么假定在这一回合内，该障碍物对于**该玩家**是失效的。

下图中给出了某一回合开始时，玩家 a 的位置 $(1,2)$ 和目标位置 $(1,3)$，以及玩家 b 的位置 $(2,0)$ 和目标位置 $(3,3)$。



每一回合内，都会出现大小、位置都固定不变的一个圆形的安全区域，直到本回合结束。安全区域的圆心位于方格中心，如果某个方格的中心到圆心的**直线距离**小于或等于安全区域的半径，那么这个方格就是**安全的**。从不安全的方格移动一步到其他位置会被**扣除 1 点生命值**。安全的方格内的障碍物将会**在本回合失效**，允许玩家通过。所有玩家的目标位置保证是安全的。

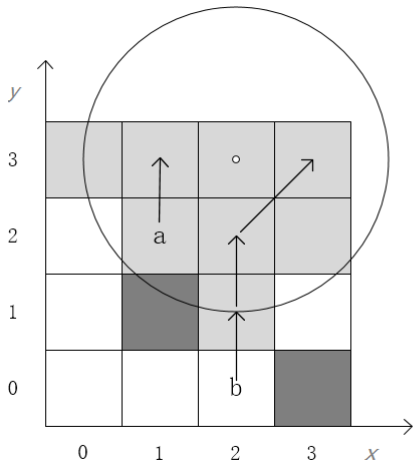
图中圆心的方格坐标是 $(2,3)$，半径为 2，浅灰色的方格是安全的方格，安全的方格内的障碍物会在本回合失效。



你的任务是，为每一位玩家找到**生命值扣除最少**的移动路线。若对于某位玩家，任何到达本回合目标位置的移动路线都会导致生命值扣除至 0，则称该玩家**死亡**。死亡的玩家不参与之后回合的游戏。

下图展示生命值扣除最少的移动路径，在此过程中，玩家 `a` 不被扣除生命值，玩家 `b` 被扣除 1 点生命值。

所有回合结束后，你需要输出所有玩家剩余的生命值，已经死亡的玩家输出 `0`。



【输入格式】

从标准输入读入数据。

输入第一行包括五个整数：n、m、e、f、h，表示棋盘为 $n \times n$ 大小，一共有 m 个玩家，棋盘上有 e 个障碍物，游戏一共有 f 个回合，玩家的初始生命值是 h。$1 \le n \le 400$，$1 \le m \le 10^5$，$0 \le e \le n \times n$，$1 \le f \le 10$，$0 < h \le 2.5n$。

接下来有 e 行，每行包含两个整数 p、q，(p, q) 即为该障碍物所在方格的坐标，$0 \le p, q < n$。

随后有 m 行，每行包含两个整数 i、j，(i, j) 即为该玩家初始所在方格的坐标，$0 \le i, j < n$。

随后有 f 个回合的数据，每个回合的数据有 $m+1$ 行。其中，包含一行安全区信息以及 m 行玩家移动目标。安全区信息包括三个整数，a、b 和 r，(a, b) 表示安全区圆心所在方格的坐标，r 表示安全区半径。玩家移动目标包含两个整数，u 和 v，(u, v) 表示玩家移动目标的方格坐标。即使玩家已经死亡，也会提供移动目标，但是并不需要进行计算。$0 \le a, b, u, v < n$，$0 < r \le 200$。保证每个玩家给出的目标坐标一定在安全区域以内。保证在任意回合，对于任意玩家，都存在一条到达本回合目标位置的移动路线。

所有输入都是整数。

【输出格式】

输出到标准输出。

输出 m 行，每行包含一个整数：z，表示该玩家的最终生命值。

m 个玩家的输出顺序与输入顺序相同。

【样例输入】

```
4 2 4 1 1
0 3
1 1
2 2
3 0
1 2
2 0
2 3 2
1 3
3 3
```

【样例输出】

```
1
0
```

【子任务】

共有五个子任务，每个子任务均包含一个或多个测试点。若你的程序对一个子任务的全部测试点，都能给出正确的输出，则得到该子任务的满分，否则该子任务得 0 分。本题满分共计 100 分。

- 子任务一（19 分）：输入数据保证无障碍物，每回合开始玩家都在安全区中；
- 子任务二（23 分）：输入数据保证无障碍物，每回合开始玩家不一定在安全区中，$1 \le n \le 10$，$1 \le m \le 20$；
- 子任务三（17 分）：输入数据保证无障碍物，每回合开始玩家不一定在安全区中，$10 < n \le 400$，$20 < m \le 10^5$；
- 子任务四（25 分）：输入数据保证有障碍物，$1 \le n \le 10$，$1 \le m \le 20$；
- 子任务五（16 分）：输入数据保证有障碍物，$10 < n \le 400$，$20 < m \le 10^5$。

贪心算法（greedy）

【题目描述】

点独立集是图论中的概念。一个点独立集是一个图中一些两两不相邻的顶点的集合，亦即一个由顶点组成的集合 S，使得 S 中任两个顶点之间没有边。顿顿设计了一个在无向图上求解点独立集的算法，希望你可以帮他实现一下。

算法框架

1. 对于给定的无向图，不断地使用“归约规则”缩减图的规模，直至无法继续使用为止。
2. 当无法使用归约规则时，每次“贪心”地选取一个顶点直接从图中删去，直至能继续使用归约规则或图为空。
3. 反复迭代上述步骤，直至图为空。

归约规则

每成功地执行一次规约规则，会将一个顶点选入答案中，选入的顶点按下面的规则唯一确定：

1. 若图中有顶点度为 0，则将其中编号最小的选入答案中，并把它从图中删去；
2. 否则若图中有顶点度为 1，则将其中编号最小的选入答案中，并把它和它唯一的邻接顶点从图中删去；
3. 否则不能成功执行规约规则。

贪心策略

当图中不存在度小于 2 的顶点时，需要从图中贪心地删去一个顶点，被删去的顶点按下面的策略唯一确定：

1. 若图中度最大的顶点唯一，则把它从图中删去；
2. 否则，在上述顶点中，选择这样的顶点，使得删去它之后，图中剩余的度为 1 的顶点最多。若这样的顶点唯一，则把它从图中删去；
3. 否则，在这样的顶点中，选择编号最大的那个从图中删去。

【输入格式】

从标准输入读入数据。

输入第一行包含用空格分隔的两个正整数 n 和 m，表示图中有 n 个顶点、m 条无向边，顶点编号从 1 到 n。

接下来 m 行，每行包含用空格分隔的两个正整数 u 和 v，表示编号为 u 和 v 的两个顶点间有一条无向边。输入数据保证所有的顶点编号（u、v 和 w）均为 $[1,n]$ 范围内的正整数，保证 $u \neq v$ 且同一条边不会出现多次。

【输出格式】

输出到标准输出。

按求解顺序输出该点独立集（即每成功地执行归约规则一次就输出一个被选入的顶点），每行输出一个顶点编号。

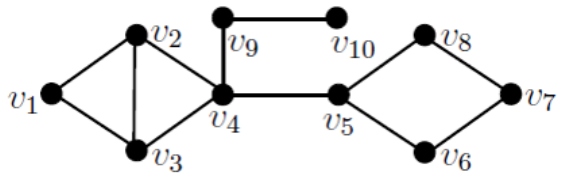
【样例输入】

```
10 12
1 2
2 3
3 1
2 4
3 4
4 9
4 5
9 10
5 8
8 7
7 6
6 5
```

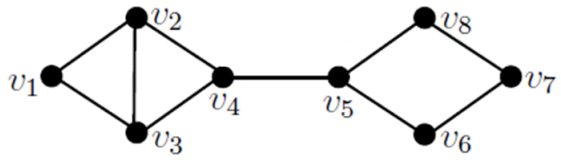
【样例输出】

```
10
6
8
1
4
```

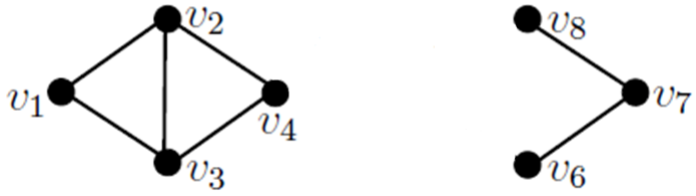
【样例解释】



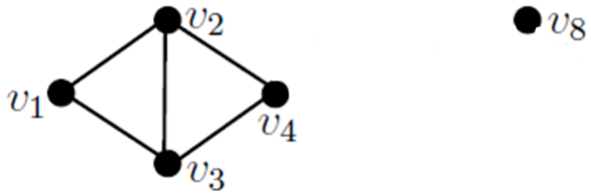
输出 v_{10}，删去 v_{10}、v_9。



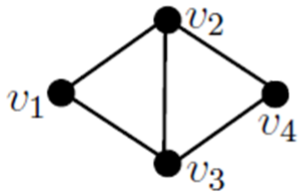
删去 v_5。



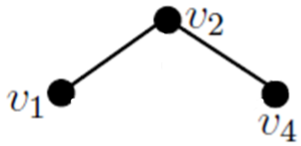
输出 v_6，删去 v_6、v_7。



输出 v_8，删去 v_8。



删去 v_3。



输出 v_1，删除 v_1、v_2。

输出 v_4，删除 v_4。

【子任务】

共有四个子任务，每个子任务均包含一个或多个测试点。若你的程序对一个子任务的全部测试点，都能给出正确的输出，则得该子任务的满分，否则该子任务得 0 分。本题满分共计 100 分。

子任务一（16 分）：输入数据保证 $n \le 2,000$ 且 $m \le 10^5$，且图中没有环。

子任务二（17 分）：输入数据保证 $n \le 2,000$ 且 $m \le 10^5$，且对于任意三个不同顶点 u、v 和 w，如果 u 和 v 之间有边且 v 和 w 之间有边，则 u 和 w 之间有边。

子任务三（28 分）：输入数据保证 $n \le 2,000$ 且 $m \le 10^5$。

子任务四（39 分）：输入数据保证 $n \le 10^5$ 且 $m \le 5 \times 10^5$。

【提示】

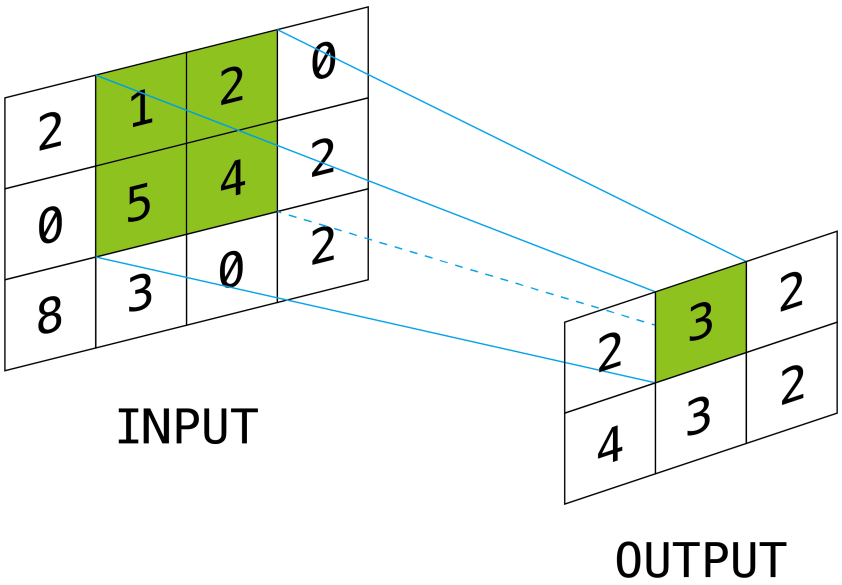
本题输入输出数据量较大，请选择合理方式进行读写。

卷积（convolution）

本题仅支持 C、C++ 和 Java 语言。

【题目描述】

卷积（convolution），是一种被广泛应用于信号处理、图像处理、深度学习等领域的一种基本运算。在本题中，我们要计算一个特殊的 $N \times N$ 的二维卷积。其过程是：给定一个大小为 $(H+N-1) \times (W+N-1)$ 的矩阵，经过计算后得到一个 $H \times W$ 的矩阵，该矩阵的每一个元素都是原矩阵以该点为左上角的 $N \times N$ 的正方形中的元素的平均值。



(1+2+5+4)÷4=3

【输入/输出格式】

输入为大小为 $(H+N-1) \times (W+N-1)$ 的内存中的单精度浮点矩阵，矩阵中的每个元素都在 $[0,1]$ 之间，均为独立等概率随机生成。

该矩阵以一个 $(H+N-1) \times (W+N-1)$ 的数组的形式给出。保证 $5000 \le W \le 10000$，$5000 \le H \le 10000$，$2 \le N \le 10$。

为了方便运算，W 和 H 都保证是 8 的整数倍。

请你将结果的 $H \times W$ 的矩阵输出到一个长度为 $H \times W$ 的一维数组中。你的答案和标准答案之间的差的绝对值在 10^{-5} 以内被认为正确。

请不要在程序中向标准输出流输出任何信息，否则程序会被判为 0 分。

【评分方式】

本题共有 10 个测试点，每个测试点 10 分。

设 t_i 为选手程序运行第 i 个测试点的时间；c_i 为选手程序计算第 i 个测试点的正确性，1 为正确，0 为不正确；T_i 为本次比赛所有选手的程序中，能够正确计算第 i 个测试点的最短用时。选手在第 i 个测试点的分数 s_i，由如下公式给出：

$$s_i = c_i \times \left(1 + 9 \times \frac{T_i}{t_i}\right)$$

选手本题的得分是各测试点得分之和，满分共计 100 分。

对于 10% 的数据，$N = 3$；对于 20% 的数据，$N = 5$；对于 30% 的数据，$N = 7$；对于 40% 的数据，$N = 10$。

【解题框架】

本题我们提供两种解题框架，建议使用 **C/C++ 解题框架**。

C/C++ 解题框架

你需要在 `solve.c` 中实现 `solve` 函数。

具体地，在所给的框架中，包含以下文件和文件夹：

- `Makefile`：用于编译，生成可执行文件，并执行测试。运行 `make` 来编译并执行测试。
- `solve.c` | `solve.cpp`：你所需要填写的代码文件，提交时仅能提交该文件。
  - 你需要实现 `solve` 函数。`output` 是一个已经分配好的长度为 $H \times W$ 的数组。请将计算出来的矩阵放在 `output` 中。
  - 你可以在该文件中添加你可能要用到的变量、函数和其他代码。
- `simd.h`：CPU 的 SIMD 指令的封装。
- `solve.h`：`solve` 函数声明。
- `main.c` | `main.cpp`：该文件负责评测。该文件中包含了 `solve_naive` 函数。该函数提供了一个可用于参考的 `solve` 函数的实现。

Java 解题框架

你需要在 `src/Solver.java` 中实现 `solve` 函数。

具体地，在所给的框架中，包含以下文件和文件夹：

- `Makefile`：用于编译，并执行测试。运行 `make` 来编译并执行测试。
- `src/Solver.java`：你所需要填写的代码文件，提交时仅能提交该文件。
  - 你需要实现 `solve` 函数。`output` 是一个已经分配好的长度为 $H \times W$ 的数组。请将计算出来的矩阵放在 `output` 中。
  - 你可以在该文件中添加你可能要用到的变量、函数和其他代码。

- `src/Tester.java`：该文件负责评测。该文件中包含了 `solveNaive` 函数。该函数提供了一个可用于参考的 `solve` 函数的实现。
- `src/Main.java`：主函数。

【提示】

- 选手可以使用多线程来加速程序的性能。
- 选手可以使用 CPU 的向量指令来加速程序性能，`simd.h` 文件里包含了选手可能使用的一些向量指令的封装。

分组加密器（encryption）

【题目背景】

在密码学中，分组加密（block cipher），又称分块加密或块密码，是一种对称密钥算法。它将明文分成多个等长的模块（block），使用确定的算法和对称密钥对每组分别加密解密。分组加密是极其重要的加密协议组成，其中典型的如 DES 和 AES 作为美国政府核定的标准加密算法，应用领域从电子邮件加密到银行交易转帐，非常广泛。（摘自维基百科）

【格式描述】

在学习了密码学之后，顿顿发现分组加密算法大都是由一些基本的操作组成，比如置换、S 盒等等。再加上一些简单的逻辑控制语句，就可以清晰地描述出该算法的内部结构。基于此，顿顿设计了一种简单的语言，用来描述分组加密算法的逻辑结构。

该语言的结构大致如下：

变量声明部分

`BEGIN`

加密算法部分

`END`

一、变量声明部分

该语言仅有两种变量，分别为**二进制串变量**和**循环控制变量**。在此处声明的变量均为**二进制串变量**，格式为：变量名 (长度)，例如：

```
state(64)
key(10)
tmp(5)
beta(20)
```

在上面的例子中，我们声明了四个**二进制串变量**——`state`、`key`、`tmp` 和 `beta`，长度分别为 64、10、5 和 20。这里我们约定**长度**为 $[1, 64]$ 范围内的整数，二进制串的**变量名**由 2 到 10 个小写英文字母组成，且要求**变量名**互不相同。

在变量声明部分，每行声明一个**二进制串变量**，其中第一行和第二行固定为 `state` 和 `key`，第三行开始可以声明其它**二进制串变量**。`state` 在加密开始时存储明文，加密结束时存储密文。`key` 在加密开始时存储密钥。其余二进制串变量在加密开始时默认每一位均为 0。

循环控制变量为整数类型，变量名仅为一个小写英文字母，因此总共只有 26 个**循环控制变量**，不需要声明即可在加密算法部分直接使用。

二、加密算法部分

在这一部分顿顿设计了三种句法，分别为**赋值**、**循环**和**分组**，下面逐一进行介绍。

1. 赋值（异或、置换、S 盒）

二进制串变量 = 表达式

赋值语句将**表达式**的运算结果（二进制串）存入相应的**二进制串变量**，且运算结果需要与变量长度相同。简单地说，**表达式**就是若干个二进制串进行**异或**、**置换**和 S 盒三种运算产生的算式，运算结果仍为一个二进制串。其严格定义如下所示：

```
表达式 ::= 二进制串变量 | 二进制串常量
表达式 ::= 表达式 + 表达式    //异或运算
表达式 ::= 置换表 (表达式)     //使用某个置换表对表达式的结果做置换运算
表达式 ::= S 盒 (表达式)       //使用某个 S 盒对表达式的结果做 S 盒运算
置换表 ::= P[常数] | P[循环控制变量]    //常数、控制变量用来指明使用哪一个置换表
S 盒 ::= S[常数] | S[循环控制变量]      //常数、控制变量用来指明使用哪一个 S 盒
```

在上面的定义中，`::=` 表示“定义为”，`|` 表示“或”的意思。下面则是一些表达式的示例：

```
state
"10100010"
state + "10100010"
P[0](state)
S[2](state + "10100010")
key + P[k](state) + S[1](state + "10100010")
```

这里顿顿使用双引号括起的 01 串来表示一个**二进制串常量**，并用 `+` 表示**异或**运算，即两个等长的二进制串逐位异或。下面详细介绍一下**置换**和 S 盒运算。

置换运算简单来说就是用输入二进制串中的若干位，拼出一个新的二进制串，其中输入二进制串的每一位既可多次使用也可以不用。如果输入二进制串长度为 a，输出二进制串长度为 b，那么对应的置换表就是一个长度为 b 的数组，其中每一个元素取值范围 $[0,a)$，表示输出二进制串中每一位的来源。假设第 i 个元素取值为 j，其含义为输出二进制串的第 i 位取自输入二进制串的第 j 位。输入、输出二进制串的长度同样满足 $1 \le a,b \le 64$。

```
输入二进制串：abcde
置换表：  {0, 1, 2, 3, 4, 3, 2, 1}
输出二进制串：abcdedcb
```

```
输入二进制串：abcde
置换表： {1, 1, 1}
输出二进制串：bbb
```

S 盒实际上就是一个查找表，对于每一个输入给出一个相应的输出。如果输入二进制串长度为 c，输出二进制串长度为 d，那么对应的 S 盒就是一个长度为 2^c 的数组，每一个元素是一个长度为 d 的二进制串对应的十进制整数。这里二进制串转化成十进制数时默认左边为高位、右边为低位，例如 `"1000"` 对应十进制整数 8。假设第 i 个元素取值为 j，其含义为当 i 对应的二进制串做为输入时，输出为 j 对应的二进制串。因为 S 盒长度较大，这里约定输入、输出二进制串的长度满足 $1 \le c, d \le 8$。

```
输入二进制串长度：3
输出二进制串长度：2
S 盒：{0, 1, 2, 3, 3, 2, 1, 0}
输入          输出
0   000     0   00
1   001     1   01
2   010     2   10
3   011     3   11
4   100     3   11
5   101     2   10
6   110     1   01
7   111     0   00
```

一个加密算法可能会使用多个置换表和 S 盒，因此需要用 P、S 后面方括号里的常数或循环控制变量来指明具体使用哪一个。

2. 循环（LOOP）

```
LOOP 循环控制变量 初值 终值
...
ENDLOOP
```

如上所示，循环控制变量从小到大取遍 [初值 , 终值] 范围内的所有整数值。每取一个值，便按顺序执行一遍 LOOP 到相应 ENDLOOP 间的所有语句。循环中可以继续嵌套循环，但外层使用的循环控制变量内层不得重复使用。

由于循环结构的存在，一条语句可能会被执行很多次，这里我们约定每条语句执行的次数总和不会超过 10000。这个约定只是为了方便估计程序的运行时间，下面的几个例子中给出了相应的执行次数分析。

初值和**终值**也只能是 $[0, 10000]$ 范围内的**常数**，不能用循环控制变量代替，并且要求**初值**不能大于**终值**。此外，每一个**循环控制变量**都只能在相应的循环中使用。

```
//置换表 P[0]: {1, 2, 3, 0}
//tmp 初始为"0001"
LOOP i 1 3
    tmp = P[0](tmp)
ENDLOOP
//此时 tmp 为"1000"
//三条语句，循环 3 次，所以语句执行总次数为 9。

//置换表 P[0]: {1, 2, 3, 0}
//置换表 P[1]: {3, 0, 1, 2}
LOOP i 0 1
    LOOP j 1 999
        tmp = P[i](tmp)
    ENDLOOP
ENDLOOP
//循环前后 tmp 不变
//中间三条语句共循环 1998 次，首尾两条语句循环两次，总计 5998 次。

LOOP a 10000 10000
ENDLOOP
//一个循环什么都不做也是可以的
//两条语句，循环 1 次，所以语句执行总次数为 2。

//错误的例子
LOOP i 0 1
    tmp = P[i](tmp)
ENDLOOP
tmp = P[i](tmp) //循环外部不能再使用 i
```

3. 分组（SPLIT/MERGE） 为了对二进制串进行更为细致的操作，顿顿使用 SPLIT 和 MERGE 来实现二进制串的拆分与合并。

```
SPLIT(二进制串变量, 常数)
... 二进制串变量 [常数] ...
... 二进制串变量 [循环控制变量] ...
```

```
MERGE(二进制串变量)

SPLIT(state,2)
... state[0] ...
... state[1] ...
... state[i] ...
MERGE(state)
```

SPLIT 和 MERGE 总是成对出现。SPLIT 把二进制串等分为若干份，在上面的例子中，state 被切成两半。在相应的 MERGE 语句之前，将使用 state[0] 和 state[1] 来表示 state 的左右两半。更一般地，如果将 state 分成 64 段，将使用 state[0] 到 state[63] 来表示其中每一部分（方括号中也可以使用**循环控制变量**）。为了保证可以等分，顿顿要求 SPLIT 语句中的**常数**必须能整除**二进制串变量**的长度且大于 1。而 MERGE 则是将拆分出来的若干段 state[...] 再重新合并回 state。

需要注意的是，在拆分期间 state[...] 将作为新的**二进制串变量**取代 state 存在，即在 MERGE(state) 之前都不能直接使用 state。并且，state[...] 作为**二进制串变量**仍然可以继续进行拆分，下面是两个简单的例子。

```
state(8)
key(4)
BEGIN
SPLIT(state,2)
    state[0] = state[0] + "1100"
    state[1] = state[1] + key
MERGE(state)
state = state + "01010101"  //MERGE 后才可以继续使用 state
END
//明文的左边四位异或"1100"，右边四位与密钥异或，
//最后整体异或"01010101" 得到密文。

//置换表 P[0]: {1, 2, 3, 0}
//置换表 P[1]: {3, 0, 1, 2}
state(64)
key(32)
BEGIN
SPLIT(state,8)
LOOP i 0 7
    SPLIT(state[i],2)
```

```
        LOOP j 0 1
            state[i][j] = P[j](state[i][j])
        ENDLOOP
        MERGE(state[i])
    ENDLOOP
    MERGE(state)
    END
    //把明文分成八块，每一块中的左四位使用 P[0] 进行置换，
    //右四位使用 P[1] 进行置换。
```

三、一些格式上的约定

该语言严格区分大小写，**保留字**均为大写字母，而变量名均为小写字母。这里**保留字**指的是有特定用途的字符串，包括 BEGIN、END、P、S、LOOP、ENDLOOP、SPLIT 和 MERGE。

变量声明部分每行声明一个变量，**加密算法部分**每行一条语句，再加上 BEGIN 和 END 代码总共不超过 50 行。

循环语句的各部分（LOOP、循环控制变量、初值 和 终值）之间至少有一个空格分隔，其余地方可以不用空格分隔。

在有具体含义的字母串（保留字、变量名）和数字串（常量）内部，不能插入空格 _ 和制表符 \t；对于**二进制串常量**，双引号和 01 串同样视作一个整体不可分隔。除此之外，可以在代码中的任意地方添加空格和制表符，但要求每行不得超过 10^2 个字符。

【任务描述】

任务 A

试实现一个分组加密器，可以把顿顿的代码转化为相应的加密程序。即根据输入的置换表、S 盒、代码，对若干组明文和密钥进行加密，输出相应的密文。保证输入的代码满足上述所有要求。

任务 B

```
//置换表 P[0]: {12, 1, 9, 2, 0, 11, 7, 3,
//             4, 15, 8, 5, 14, 13, 10, 6}
//置换表 P[1]: {1, 2, 3, 4, 5, 6, 7, 8, 9,
//             10, 11, 12, 13, 14, 15, 0}
//S 盒 S[0]: {12, 5, 6, 11, 9, 0, 10, 13,
//          3, 14, 15, 8, 4, 7, 1, 2}
```

```
state(32)
key(12)
dkey(16)
tmp(32)
BEGIN
SPLIT(dkey,4)
SPLIT(key,3)
dkey[0] = key[0]
dkey[1] = key[1]
dkey[2] = key[2]
dkey[3] = key[0] + key[1] + key[2]
MERGE(key)
MERGE(dkey)  //将密钥扩展到 16 位
LOOP i 1 16
    tmp = state
    SPLIT(state,2)
    SPLIT(tmp,2)
        state[0] = tmp[1]
        tmp[1] = tmp[1] + dkey
        SPLIT(tmp[1],4)
        LOOP j 0 3
            tmp[1][j] = S[0](tmp[1][j])
        ENDLOOP
        MERGE(tmp[1])
        state[1] = tmp[0] + P[0](tmp[1])
    MERGE(state)
    MERGE(tmp)
    dkey = P[1](dkey)  //密钥左移一位
ENDLOOP
tmp = state
SPLIT(state,2)
SPLIT(tmp,2)
    state[0] = tmp[1]
    state[1] = tmp[0]
MERGE(state)
MERGE(tmp)
END
```

如上所示，顿顿设计了一个简化版的 DES 算法，可以用 12 位的密钥对 32 位的明文进行加密。但一个显而易见的问题就是，密钥实在是太短了，简单地穷举所有可能密钥就可以快速将其破解。为了在不更改加密算法的前提下提高破解难度，顿顿决定在加密时使用两个密钥：先使用第一个密钥对明文进行加密，接着用另一个密钥对得到的密文再次加密，进而获得最终的密文。只是简单地进行两次加密，密钥空间的大小就从 2^{12} 增加至 2^{24}，穷举密钥的代价大大增加，顿顿对此非常满意。

试破解改进后的加密算法：输入一组明密文对，如果能对明文连续两次加密得到密文，则把这两次加密时使用的密钥依次输出出来。保证不会有多于一组可能的密钥对。

【输入格式】

从标准输入读入数据。

输入第一行包含一个字符串 TASKA 或 TASKB，表明该测试点是任务 A 还是任务 B。

对于任务 A，接下来一行包含用空格分隔的两个整数 n 和 m，表示有 n 个置换表和 m 个 S 盒，保证 $0 \le n, m \le 10$，然后依次输入这些置换表和 S 盒。

每个置换表占两行，第一行包含用空格分隔的两个正整数 a 和 b，分别表示该置换表对应输入、输出二进制串的长度，保证 $1 \le a, b \le 64$；第二行 b 个用空格分隔的整数，其中每个整数都在 $[0, a)$ 范围内，表示该置换表的内容。

每个 S 盒占两行，第一行包含用空格分隔的两个正整数 c 和 d，分别表示该 S 盒对应输入、输出二进制串的长度，保证 $1 \le c, d \le 8$；第二行 2^c 个用空格分隔的整数，其中每个整数都在 $[0, 2^d)$ 范围内，表示该 S 盒的内容。

从第 $3 + 2n + 2m$ 行开始，输入顿顿的代码。END 位于代码最后一行，可以借此判读是否读入了全部代码。

接下来一行包含一个正整数 k，表示有 k 组数据需要加密，保证 $k \le 10$。

在最后的 $2k$ 行里，每两行包含一组加密数据，其中第一行为明文、第二行为密钥，皆以 01 串的形式给出。

对于任务 B，输入的第二行和第三行各包含一个 32 位的 01 串，分别表示明文和密文。

【输出格式】

输出到标准输出。

对于任务 A，输出 k 行，每行一个 01 串表示相应的密文。

对于任务 B，第一行输出一个字符串 YES 或 NO。如果输出 NO，则说明输入的明文无法经过两次加密得到密文；否则在接下来两行各输出一个 12 位的 01 串，分别表示第一、二次加密时使用的密钥。

【样例 1 输入】

```
TASKA
0 0
    state (  10  )
        key(1)
         BEGIN
           END
1
0000000000
1
```

【样例 1 输出】

```
0000000000
```

【样例 1 解释】

实际上这个“加密算法”什么也没有做，所以将明文原样输出即可。

【样例 2 输入】

```
TASKA
1 1
4 4
1 2 3 0
4 4
0 1 2 3 3 2 1 0 12 13 14 15 15 14 13 12
state(4)
key(4)
tmp(4)
BEGIN
tmp = state
LOOP i 1 3
    tmp = S[0](P[0](tmp)) + key
ENDLOOP
state = state + tmp + "1111"
END
1
```

```
0101
1100
```

【样例 2 输出】

```
1010
```

【样例 2 解释】

将明文取反。

【样例 3】

见题目目录下的 ***3.in*** 与 ***3.ans***。

【样例 4】

见题目目录下的 ***4.in*** 与 ***4.ans***。

【样例 4 解释】

该样例使用了任务 B 中的代码。

【样例 5 输入】

```
TASKB
10110001100000011100011100011010
10110000100001011000010011000100
```

【样例 5 输出】

```
YES
110001011111
100001111100
```

【样例 6 输入】

```
TASKB
10110001100000011100011100011010
10110000100001011000010011000101
```

【样例 6 输出】

```
NO
```

【子任务】

本题共有五个子任务，每个子任务有一个或多个测试点。若你的程序对一个子任务的全部测试点，都能给出正确的输出，则得该子任务的满分，否则该子任务得 0 分。本题满分共计 100 分。

- 子任务一（14 分）：所有测试点保证任务类型是任务 A，代码中不包含分组和循环句法，且 $n = m = 0$、$k = 1$；
- 子任务二（18 分）：所有测试点保证任务类型是任务 A，代码中仅不包含分组句法，且 $n, m \leq 1$、$k = 1$；
- 子任务三（20 分）：所有测试点保证任务类型是任务 A，代码中仅不包含循环句法，且 $n, m \leq 10$、$k = 1$；
- 子任务四（27 分）：所有测试点保证任务类型是任务 A，代码中包含所有句法，且 $n, m \leq 10$、$k \leq 10$；
- 子任务五（21 分）：所有测试点保证任务类型是任务 B。

【提示】

顿顿把题目中的一些关键点整理了出来，提供在下载文件中，仅供大家参考。

内存分配器（malloc）

本题仅支持 C、C++ 和 Java 语言。

【题目背景】

动态内存分配器（dynamic memory allocator），又称为堆内存分配器，开发者通过动态内存分配（例如 `malloc`）来让程序在运行时得到虚拟内存，动态内存分配器会管理一个虚拟内存区域，称为堆（heap）。分配器以块（block）为单位来维护堆，可以进行分配或者释放操作。常见的分配器有两种基本风格：

- 显式分配器：需要开发者显式地释放任何已经分配的块。例如 C 语言中的 `malloc` 和 `free`。
- 隐式分配器：开发者只负责分配块，但是不负责手动释放已经分配的块。这就要求当分配器检测到一个已分配块不再被程序使用时，就释放这个块。这类分配器也被称为垃圾收集器，例如 Java 中的垃圾收集机制。

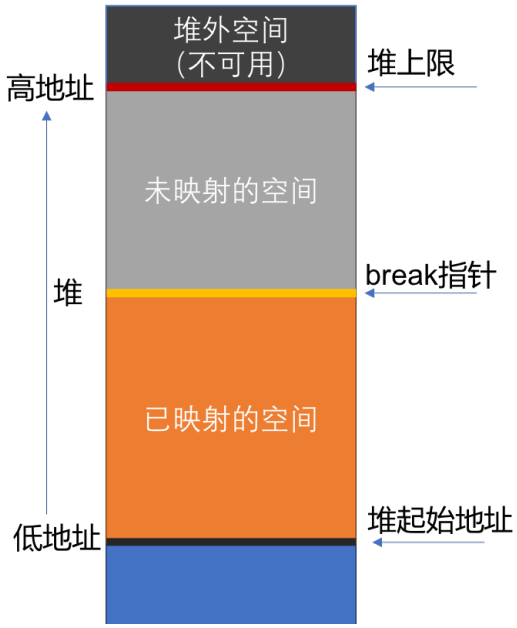
【题目描述】

本题需要你在我们模拟的“内存”系统下，借助我们提供的一些辅助函数，从我们提供的“堆内存”空间中请求地址空间，设计与实现一个显式动态内存分配器，即实现自己版本的 `malloc`、`free`、`realloc` 函数，从而被最终测试程序调用。

本题中你只需要考虑从我们提供的“堆”区域分配申请与管理内存，堆自低地址向高地址增长，本题中你可以通过我们提供的 `mem_sbrk` 函数调用来增加实际可用的堆大小。并且为了简化问题，本题中的堆区域只会向上扩展，不会缩小（即 `mem_sbrk` 的参数 **incr** 只能为非负整数）。堆的大小不超过 100×2^{20} 字节（100 MB）。

你可能还需要一些将用于管理“堆”中内存的数据信息存放到我们提供的“堆”区域中，本题的内存空间限制为 256 MB，其中我们用于评测的框架会使用的内存不超过 240 MB，因此我们建议你不要在解题文件中构造较大的全局 `array` 或者 `tree`，但可以将相应用于管理“堆”中内存的数据合理安排并存放在我们提供的“堆”区域中。

其中对堆区域的管理示意如下图：



我们维护了一个 break “指针”，这个“指针”指向堆空间的某个地址。从堆的起始位置到 break 之间的地址空间为**映射好的**，可以供进程访问的；而 break 之上的，是**未映射的**地址空间，如果访问这段空间则程序会报错。**注意**，在实现过程中你不可以直接操作 break “指针”，只能通过我们提供的辅助函数进行操作。

因此要增加一个进程实际的可用堆大小，就需要将 break 指针向高地址移动。在本题中，当现有可用堆大小不足以满足当前 malloc 调用的需求时，你可以通过调用 mem_sbrk(int incr) 将 break 指针从当前位置向上移动 **incr** 指定的增量，从而增加可用堆的大小。当然，本题中可用的堆空间并不是无限的，当你申请的空间超过我们的上限时，则会失败。这就要求你需要合理安排好堆内存的使用。

此外为了让你更方便的理解相关知识与解决这一问题，我们还提供了一系列与内存分配器相关的学习资料供你参考，其中既包括基本原理的阐述，也包括了一些相关的研究论文等等。

~~（当然我们并不保证这些资料都能很好的用到。）~~

【任务目标】

（以 **C** 版本为例进行说明，**C++**、**Java** 版本要求类似。）

你需要做的是完成在 solve.c 中的以下几个函数：

```
int my_init(void);
void *my_malloc(size_t size);
```

```
void my_free(void *ptr);
void *my_realloc(void *ptr, size_t size);
```

上述函数的要求如下：

- my_init：在这里执行所有你需要的初始化操作，例如分配初始的堆区域。若成功则返回 0；否则返回 −1。
  - 注意在这里不允许调用 utils.h 中的 mem_init 函数。
  - 测试程序对每个测点进行一次测试时，都会调用你的 my_init 函数，进行初始化操作。
- my_malloc：在系统中分配大小至少为 size 的连续可用的空间，若分配成功，则返回一个指向这段可用空间起始地址的指针；否则返回一个 NULL 指针。（Java 版本中返回值为具体的地址值）
- my_free:释放 ptr 指向的内存区域（ptr 保证是通过 my_malloc 或 my_realloc 分配得到的），没有返回值。my_free(NULL) 什么都不做。
- my_realloc：重新分配给定的内存区域，成功时，返回指向新分配内存的指针；失败时，返回空指针。ptr 所指向的区域必须是之前为 my_malloc 或 my_realloc() 所分配，并且仍未被释放。根据传入参数的不同，该函数有不同的表现：
  - ptr 为 NULL 时，等同于 my_malloc(size)。
  - size 为 0 时，等同于 my_free(ptr)，返回值应为 NULL。
  - ptr 不为 NULL 且 size 不为 0 时，你可以选择就地扩张或收缩 ptr 所指向的已分配的内存区域（如果可行的话），内存的内容在新旧大小中的较小者范围内保持不变；若扩张范围，则内存空间新增部分的内容是未初始化的。你也可以选择直接新分配一个大小为 size 字节的新内存块，并从原分配的内存区域中复制大小等于新旧大小中较小者的数据到新分配的内存区域中，然后释放原分配的内存区域；同样地，内存空间新增部分的内容是未初始化的。
  - 注意：调用成功后分配的地址空间不能和其他时刻 my_malloc 或者 my_realloc 分配的空间有重叠部分，并且不能超出整个“堆”空间。

utils.c 中模拟了一个简单的内存系统。你函数分配的所有堆空间应当通过这个系统申请。具体的，当你需要更大的堆空间时，你的函数应当调用下列函数进行操作：

- void *mem_sbrk(int incr)：将堆向上扩展 incr 个字节（注意本题中 incr 只能为非负整数）。如果成功则返回一个指向分配的新的堆空间的首地址的指针（即返回 break 指针在该次调用前的值）；如果堆空间达到了上限，则会失败，此

时返回 `(void*) -1`。注意：你最终通过这个方法获取到的堆空间总大小将影响到你的得分。

在这个系统中，你的函数还可以调用下列函数来得到相应的信息：

- `void *mem_heap_lo()`：返回指向堆的第一个字节的指针。
- `void *mem_heap_hi()`：返回指向当前可用堆的最后一个字节的指针。
- `size_t mem_heapsize()`：返回当前的堆大小。
- `utils.c` 中的上述 4 个函数在评测时的代码和你看到的相同，你实现中调用这些函数的运行时间都会计入你的程序耗费时间。

注意事项：

- `my_malloc` 和 `my_realloc` 返回的地址必须是 8 字节对齐的，即地址值模 8 余数为 0。
- `my_malloc` 和 `my_realloc` 调用成功后分配的地址空间不能和其他时刻 `my_malloc` 或者 `my_realloc` 分配的空间有重叠部分，并且不能超出整个“堆”空间。
- 我们需要你实现的 `my_malloc` 和 `my_realloc` 能够尽可能保证每次都能分配成功，在评测过程中需要你实现的内存分配器对测试文件中每一个请求都能成功分配并返回。当然你也必须处理失败的情况，即返回 `NULL` 指针的情况。

【解题框架】

本题我们提供三种语言解题框架，建议使用 C 解题框架。

提醒

- 下发的框架和最终实际测评的框架并不完全相同，但是基本功能是一致的，你最终只需要提交你所需要填写的代码文件。你可以在解题过程中修改其他文件用于调试，但请在最终提交前确认你的解题代码在原始版本下能否编译运行成功。
- 下发框架中包含了 `solve.c/solve.cpp/Solve.java` 简单实现的解题代码，旨在让你熟悉框架，不代表你需要使用这些代码或参考其中的实现。这些代码并不能通过所有的测试点，你应该编写自己的代码。
- 下发框架中包含了一些测试文件，这些测试文件并不是最终用于评测的文件，但是最终评测的文件与你能看到的测试文件基本是同类型的（同类型是指：申请、释放内存的指令序列模式类似等）。因此建议你在本地通过所有文件的正确性测试之后再提交代码。
- 你需要在评测网站上提交你的 `solve.c/solve.cpp/Solve.java`。网站不能提交多个工程文件，也不要提交其他框架代码。只有你提交到网站的文件会作为评

分的依据，本地的任何测试结果均不作为评分依据。

C C++ 解题框架

你需要在 solve.c | solve.cpp 中实现 my_init()，my_malloc()，my_free()，my_realloc() 函数。

具体地，在所给的框架中，包含以下文件和文件夹：

- Makefile：用于编译，生成可执行文件 ***tester***。
- tester.c | tester.cpp：测试程序的入口。运行时会读取相应的测试点，按照操作指令调用你实现的函数进行评测。
- utils.c | utils.cpp, h：一些工具函数。
- solve.h：my_init()，my_malloc()，my_free()，my_realloc() 函数声明。
- solve.c | solve.cpp：你所需要填写的代码文件，提交时仅能提交该文件。其中：
  - 你需要实现 my_init()，my_malloc()，my_free()，my_realloc() 函数。
  - 你可以在该文件中添加你可能要用到的变量、函数和其他代码。
- short1, short2：用于测试你程序的两个较小的测试样例，便于你调试。
- data：一些你可以用于本地测试的测试文件。

在答题时，有以下**注意事项**：

- 提交的代码运行后务必不能输出任何额外信息，即你最终提交的 solve.c | solve.cpp 文件中不应该包含 printf 类似的输出。否则该次提交得分为 0。

测试程序运行方法：

你可以在下发框架目录下，在终端中执行 make 命令编译代码。

编译成功后，直接执行 ./tester，即可进行完整测试，并得到相应的输出，可选的参数如下：

- -f <testfile>：进行一个特定的测试，输出正确性、时间等信息。
- -c <testfile>：进行一次特定的测试，仅评测正确性，用来检测正确性很方便。
- -v：对每一个测试点进行测试时输出额外信息。
- -h：输出命令行参数。

Java 解题框架

你需要在 Solve.java 中实现 myInit()，myMalloc()，myFree()，myRealloc() 函数。

具体地，在所给的框架中，包含以下一些文件/文件夹：

- src/Makefile：用于编译，生成 ***.class*** 文件。

- src/MyMain.java：测试程序的入口。运行时会调用 Tester 类，读取相应的测试点，按照操作指令调用你实现的函数进行测评。
- src/MyUtils.java：一些工具函数。
- src/Tester.java：Tester 类，用于读取相应的测试点并进行测评。
- src/Solve.java：你所需要填写的代码文件，提交时仅能修改该文件。其中：
  - 你需要实现 myInit()，myMalloc()，myFree()，myRealloc() 函数。
  - 你可以在该文件中添加你可能要用到的变量、函数和其他代码。
- Readme.pdf：针对 Java 框架的额外说明，请务必阅读。
- src/short1 src/short2：用于测试你程序的两个较小的测试样例，便于你调试。
- data：一些你可以用于本地测试的测试文件。

在答题中，有以下注意事项：

- 提交的代码运行后务必不能输出任何额外信息，即你最终提交的 Solve.java 文件中不应该包含 println 类似的输出。

测试程序运行方法：

你可以在下发框架的 src 目录下，在终端中执行 make 命令编译代码。

编译成功后，直接执行 java MyMain，即可进行完整测试，并得到相应的输出，可选的参数如下：

- -f <testfile>：进行一次特定的测试，输出正确性、时间等信息。
- -c <testfile>：进行一次特定的测试，仅评测正确性，用来检测正确性很方便。
- -v：对每一个测试点进行测试时输出额外信息。
- -h：输出命令行参数。

【输入格式】

本题的测试程序会运行一系列的测试点，按照其中的操作指令，调用你自己实现的动态内存分配器执行相应的操作，并进行正确性检验以及相应性能指标的统计。

测试点的输入是文本格式，前两行格式如下：

```
[num_ids]          /* number of request id's */
[num_ops]          /* number of operations */
```

从第 3 行开始每行对应着一个具体的操作指令，分别对应 malloc、free 和 realloc 中的一种操作，格式如下：

```
a [id] [size]   /* ptr_[id] = malloc([size]) */
r [id] [size]   /* realloc(ptr_[id], [size]) */
f [id]          /* free(ptr_[id]) */
```

其中 `size` 为在 $[0, 2^{26}]$ 之间的整数。

一个非常简单的测试点示例如下：

```
3
8
a 0 512
a 1 128
r 0 640
a 2 128
f 1
r 0 768
f 0
f 2
```

其中有些测试点中会出现 `f -1`，表示实际为执行 `free(NULL)`。以及测试点中可能出现下述形式的片段：

```
a 34 1234
a 33
a 23
a 30
f 49
a 38
a 17
a 35
a 31
a 43 600
```

这种形式实际可以理解成下述形式：

```
a 34 1234
a 33 1234
a 23 1234
```

```
a 30 1234
f 49
a 38 1234
a 17 1234
a 35 1234
a 31 1234
a 43 600
```

~~（其实你不需要过多研究测试点的格式，因为你需要编写的内存分配器不需要处理输入数据。）~~

【输出格式】

你正确实现内存分配器后，编译并运行测试程序，会输出相应的测试结果，其中包括正确性、堆利用率以及耗费时间等指标信息。对于某个测试点，如果你的程序未能通过正确性检验，则并不会输出利用率等信息。其中堆利用率的输出为 $(0, 1)$ 之间的小数，时间的输出为以秒为单位的小数，两者都保留到小数点后 6 位。

堆利用率

堆利用率是测试程序调用你实现的 `my_malloc`、`my_realloc` 等函数的过程中请求分配的“内存”累计总量的最大值与你的分配器最终所使用的堆大小之比。你需要尽量减少堆内碎片化程度从而让堆利用率提高。

例如对于下面的测试文件：

```
3
8
a 0 512
a 1 128
r 0 640
a 2 128
f 1
r 0 768
f 0
f 2
```

测试程序向你的程序请求分配的“内存”累计总量最大值为：896 Bytes（512 + 128 + (640 − 512) + 128）。

假如你实现的分配器通过调用 `mem_sbrk` 函数申请的堆大小为：1024 Bytes，则堆利用率此时为：$\frac{896}{1024} = 0.875$。

耗费时间

耗费时间是指测试程序读取一个测试文件的指令序列后，按照其中的操作指令，依次调用你实现的动态内存分配器执行所有相应的操作所用的总时间，用 t 表示。

样例输出

一个可能的测试运行结果如下：

```
Results for your malloc:
  id     valid      util      secs filename
   0       yes  0.857824  0.028609 1.in
   1       yes  0.761353  0.000757 2.in
   2       no          -         - 3.in
```

上述输出表示：你实现的代码可以通过 `1.in` 的测试，其中堆利用率为 0.857824，耗费时间为 0.028609 秒（`2.in` 测试文件类似）。但是你的代码并不能通过 `3.in` 的测试。

【评分方式】

本题的分数分为两部分：正确性分数和性能分数。

正确性分数满分为 20 分，包括 20 个测试点。若选手的程序能够在给定的时间、空间限制下正确处理**全部**测试点，则得 20 分，否则得 0 分。

这里的时间限制即评测网站上显示的时间限制，与前文所述的**耗费时间**不同，这里的时间限制**包括**我们提供的框架运行的时间，而**耗费时间**不包括。对于每个测试点，我们保证我们的框架运行总时长不超过 4 s，因此可以认为只要你的函数每个测试点的运行时长不超过 6 s，就能保证不会超过时间限制。

性能分数满分为 80 分，包括 20 个测试点，每个测试点的满分是 4 分。若正确性分数为 0 分，那么性能分数也为 0 分；否则每个测试点的分数计算方式如下：

对于第 i 个测试点，设 u_i 为选手程序的堆利用率，t_i 为选手程序的耗费时间，U_i 为所有选手程序运行此测试点堆利用率的最大值，T_i 为所有选手程序运行此测试点耗费时间的最小值，并取：

$$\alpha_i = \frac{1 - U_i}{1 - u_i}$$

$$\beta_i = \frac{T_i}{t_i}$$

那么选手在该测试点的性能得分 s_i 为：

$$s_i = 4 \times (0.7\alpha_i + 0.3\beta_i)$$

选手的性能分数为各测试点的性能得分之和，本题的得分为正确性分数和性能分数之和，共计满分 100 分。

【阅读材料】

本题为了还提供了一些关于动态内存分配的阅读材料，你可以在下发文件夹中找到相应的文件，有选择性的进行学习。

这里对一些阅读材料进行简要的说明。

- `basic` 文件夹：该文件夹下主要提供了关于动态内存分配基本原理的一些资料，其中包括一些经典的综述论文、相关教材章节以及相关的 Slides 等。你可以通过学习该文件下的资料，掌握动态内存分配的基本模型与原理，从而能更轻松的解决本题。
- `advanced` 文件夹：该文件夹下主要提供了一些进阶的阅读材料，其中既包括例如实际环境中采用的 `tcmalloc`、`ptmalloc`、`jemalloc` 等算法与实现分析，也有诸如对针对实时系统进行优化的内存分配算法 `TLSF` 的介绍。你通过阅读学习其中的一些材料可能能够得到一些启发，从而更好更高效的解决本题。