**江苏科技大学**

**课程实验报告**

课 程： 操作系统原理与安全

课 题： 实验三·存储管理

学 院： 计算机学院

姓 名： 陈四贵

班 级： 1822107101

学 号： 182210710119

开发语言： Java

目 录

[一、 实验目的 1](#_Toc41664494)

[二、 实验内容 1](#_Toc41664495)

[三、 流程图 2](#_Toc41664496)

[四、 实验要求 3](#_Toc41664497)

[五、 实验代码及其说明 3](#_Toc41664498)

[1．总体结构 3](#_Toc41664499)

[2．分支结构 3](#_Toc41664500)

[⑴ 静态全局数据成员 3](#_Toc41664501)

[核心思想 3](#_Toc41664502)

[附·源代码 4](#_Toc41664503)

[⑵ init()函数 4](#_Toc41664504)

[核心思想 4](#_Toc41664505)

[附·源代码 4](#_Toc41664506)

[⑶ OPT()函数（最佳页面置换法） 4](#_Toc41664507)

[核心思想 4](#_Toc41664508)

[附·源代码 4](#_Toc41664509)

[⑷ FIFO()函数（先进先出法） 6](#_Toc41664510)

[核心思想 6](#_Toc41664511)

[附·源代码 6](#_Toc41664512)

[⑸ LRU()函数（最近最少使用算法） 8](#_Toc41664513)

[核心思想 8](#_Toc41664514)

[附·源代码 8](#_Toc41664515)

[⑹ LFU()函数（最少访问页面算法） 9](#_Toc41664516)

[核心思想 9](#_Toc41664517)

[附·源代码 9](#_Toc41664518)

[⑺ printStatus()函数 11](#_Toc41664519)

[说明 11](#_Toc41664520)

[附·源代码 11](#_Toc41664521)

[⑻ 主函数 12](#_Toc41664522)

[说明 12](#_Toc41664523)

[附·源代码 12](#_Toc41664524)

[六、 运行结果 13](#_Toc41664525)

[七、 实验分析与结论 16](#_Toc41664526)

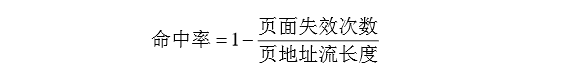
[1.实验结果展示 16](#_Toc41664527)

[2.结论 17](#_Toc41664528)

# 实验目的

存储管理的主要功能之一是合理地分配空间，而请求页式管理是一种常用的虚拟 存储管理技术。本实验的目的是通过请求页式管理中页面置换算法模拟设计，了解虚拟存储技术的特点，掌握请求页式存储管理的页面置换算法。

# 实验内容

（1）通过计算不同算法的命中率比较算法的优劣。同时也考虑了用户内存容量对命中率的影响。

页面失效次数为每次访问相应指令时，该指令所对应的页不在内存中的次数。

在本实验中，假定页面大小为1k，用户虚存容量为32k，用户内存容量为4页到 32 页。

（2）produce\_addstream 通过随机数产生一个指令序列，共320条指令。

A、 指令的地址按下述原则生成：

1） 50%的指令是顺序执行的

2）25%的指令是均匀分布在前地址部分

3） 25%的指令是均匀分布在后地址部分

B、具体的实施方法是：

1） 在[0，319]的指令地址之间随机选取一起点 m；

2） 顺序执行一条指令，即执行地址为 m+1 的指令；

3） 在前地址[0，m+1]中随机选取一条指令并执行，该指令的地 址为 m’；

4） 顺序执行一条指令，地址为 m’+1 的指令

5） 在后地址[m’+2，319]中随机选取一条指令并执行；

6） 重复上述步骤 1）~5），直到执行 320 次指令

C、将指令序列变换称为页地址流

（3）计算并输出下属算法在不同内存容量下的命中率。

1）先进先出的算法（FIFO）；

2）最近最少使用算法（LRU）；

3）最佳淘汰算法（OPT）；

4）最少访问页面算法（LFR）； 其中 3）和 4）为选择内容

在用户虚存中，按每k存放10条指令排列虚存地址，即320条指令在虚存中的存放方式为：

第0条~第9条指令为第0页（对应虚存地址为[0，9]）；

第10条~第19条指令为第1页（对应虚存地址为[10，19]）；

……

第310条~第319条指令为第31页（对应虚存地址为[310，319]）；

按以上方式,用户指令可组成32页。

# 流程图

# 实验要求

1.运行结果

a．终端先显示：

Start memory management.

Producing address flow, wait for while, please.

b．地址流、地址页号流生成后，终端显示：

There are algorithms in the program

1、 Optimization algorithm

2、 First in first out algorithm

3、 Least recently used algorithm

4、 Least frequently used algorithm Select an algorithm number, please.

用户输入适当淘汰算法的号码，并按回车，若是第一次选择，输出相应的地址页号流。然后输出该算法分别计算的用户内存从 2k~32k 时的命中率，若输入的号码不再 1~4 中，则显示：

There is not the algorithm in the program,并重复 b。

c． 输出结果后，终端显示 “Do you try again with anther algorithm(y/n)”。若键 入 y 则重复 b,否则结束。

2.运行结果讨论与总结

a.比较各种算法的命中率；

b.分析当用户容量增加时对命中率的影响。

# 实验代码及其说明

## 1．总体结构

整个实验采用面向对象的编程技术，按功能将各个部分模块化，分为主函数（流程控制）、init函数（初始化）、OPT函数（最佳置换法）、FIFO函数（先进先出法）、LRU函数（最近最少使用算法）、LFU函数（最少访问页面算法）、printStatus函数（状态打印函数）。

## 2．分支结构

### ⑴ 静态全局数据成员

核心思想： 为减少内存开销，提高程序可读性，将各个页面置换算法都会使用到的数据作为全局数据使用，在每个算法开始前初始化这些数据即可。

附·源代码：

1. **static** **int** Aimed;   //命中次数
2. **static** **int** Error;   //失效次数
3. **static** **int** Msize;   //用户内存容量
4. **static** **int** num;  //记录请求次数
5. **static** **double** Effective;  //命中率
6. **static** **int**[] instruction=**new** **int**[320];  //模拟连续执行的代码指令

### ⑵ init()函数

核心思想：本函数主要作用是初始化代码指令，模拟虚拟内存中指令的读取。但由于指令地址分布实际上是物理地址的分布要求，在虚拟内存中这些指令应是连续的一个整体，为了达到代码指令分布的要求，因而使用数组来记录虚拟地址与其对应的物理地址关系。

附·源代码：

1. **public** **static** **void** init() {//生成地址流
2. Random random=**new** Random();
3. **int** m,m1,m2,i=0;
4. **while**(i<=319) {
5. m = random.nextInt(320);//[0,319]
6. instruction[i++] = m;
7. instruction[i++] = m + 1;
8. m1 = random.nextInt(m + 1);
9. instruction[i++] = m1;
10. m2 = random.nextInt(320 - m1 - 1) + m1 + 1;
11. instruction[i++] = m2;
12. }
13. }

### ⑶ OPT()函数（最佳页面置换法）

核心思想：若内存足够，判断内存中是否有进程所需页面，是，命中次数+1；否，未命中次数+1，将该进程对应的页面装入内存。若内存不够，判断内存中是否有进程所需页面，是，命中次数+1；否，未命中次数+1，找出已存在的页面中未来最后被使用的，将其用下一指令所对应的页面覆盖。本程序使用max记录后续最晚出现的页面，最后删除即可。但是，只要发现内存中有一个后续不再出现的页面，就可以停止后续搜索，立即置换页面。由于页面后续不会出现，不管内存中现有多少个不会出现的页面，都不会影响命中结果。

附·源代码：

1. **public** **static** **void** OPT() {//最优置换法
2. Msize = 2;
3. **while** (Msize <= 32) {
4. num = 1;
5. Aimed = 0;  //每次开始前，命中次数赋为0
6. Error = 0;  //每次开始前，未命中次数赋为0
7. List<Integer> list = **new** ArrayList<>();
8. **for** (**int** i = 0; i <= 319; i++) {
9. **if** (list.size() < Msize) {
10. **if** (list.contains(instruction[i] / 10)) {
11. Aimed++;
12. **if** (Msize == 2) {
13. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，成功命中。");
14. printStatus(list);
15. }
16. } **else** {
17. list.add(instruction[i] / 10);
18. Error++;
19. **if** (Msize == 2) {
20. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，未能命中，将页面" + (instruction[i] / 10) + "装入页表。");
21. printStatus(list);
22. }
23. }
24. } **else** {//满了
25. **if** (list.contains(instruction[i] / 10)) {
26. Aimed++;
27. **if** (Msize == 2) {
28. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，成功命中。");
29. printStatus(list);
30. }
31. } **else** {
32. **int** state=1;
33. **int** Disappear=i+1;//记录出现的次数
34. **if**(Disappear==320) Disappear=318;
35. here:
36. **for**(**int** j=0;j<Msize;j++){
37. **if**(i<319)
38. **for**(**int** k=i+1;k<=319;k++){
39. **if**(list.get(j)==instruction[k]/10){//该页面后续出现
40. **if**(k>Disappear||Disappear==i+1) {
41. Disappear=k;
42. }
43. **break**;
44. }
45. **if**(list.get(j)!=(instruction[k]/10)&&k==319) {//该页面后续不出现，直接剔除
46. list.remove(j);
47. state=0;
48. **break** here;
49. }
50. }
51. }
52. **if**(state==1) {
53. list.remove((Object) (instruction[Disappear] / 10));
54. }
55. list.add(instruction[i] / 10);
56. Error++;
57. **if** (Msize == 2) {
58. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，未能命中，将页面" + (instruction[i] / 10) + "装入页表。");
59. printStatus(list);
60. }
61. }
62. }
63. }
64. printStatus(1);
65. Msize++;
66. }
67. }

### ⑷ FIFO()函数（先进先出法）

核心思想：使用Arraylist作为基本数据结构，用队列的特征限制其操作，即只能从队首出，从队尾进，用它来模拟内存空间。若队列未满，则页面直接入队；若队列已满，则先出队，再入队。

附·源代码：

1. **public** **static** **void** FIFO(){
2. Msize=2;
3. **while**(Msize<=32){
4. num=1;
5. Aimed=0;  //每次开始前，命中次数赋为0
6. Error=0;  //每次开始前，未命中次数赋为0
7. List<Integer> list =**new** ArrayList<Integer>();
8. **for**(**int** i=0;i<=319;i++){
9. **if**(list.size()<Msize){
10. **if**(list.contains(instruction[i]/10)) {
11. Aimed++;
12. **if** (Msize == 2) {
13. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，成功命中。");
14. printStatus(list);
15. }
16. }
17. **else**{
18. list.add(instruction[i]/10);
19. Error++;
20. **if** (Msize == 2) {
21. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，未能命中，将页面" + (instruction[i] / 10) + "装入页表。");
22. printStatus(list);
23. }
24. }
25. }
26. **else**{
27. **if**(list.contains(instruction[i]/10)) {
28. Aimed++;
29. **if** (Msize == 2) {
30. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，成功命中。");
31. printStatus(list);
32. }
33. }
34. **else** {
35. list.remove(0);
36. list.add(instruction[i] / 10);
37. Error++;
38. **if** (Msize == 2) {
39. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，未能命中，将页面" + (instruction[i] / 10) + "装入页表。");
40. printStatus(list);
41. }
42. }
43. }
44. }
45. printStatus(2);
46. Msize++;
47. }
48. }

### ⑸ LRU()函数（最近最少使用算法）

核心思想：使用LinkedList作为基本数据结构，每次页面置换都对页表进行重新排序，最近使用的在链尾，一旦内存空间满了，就移除链首元素（最近最少使用的）。

附·源代码：

**public static void** LRU(){*//最近最少使用算法  
 Msize*=2;  
 **while**(*Msize*<=32){  
 *num*=1;  
 *Aimed*=0; *//每次开始前，命中次数赋为0  
 Error*=0; *//每次开始前，未命中次数赋为0* List<Integer> list =**new** LinkedList<>();*//最新使用的在最前面* **for**(**int** i=0;i<=319;i++){  
 **if**(list.size()<*Msize*){  
 **if**(list.contains(*instruction*[i]/10)) {  
 *Aimed*++;  
 list.remove((Integer)(*instruction*[i]/10));  
 list.add(0,*instruction*[i]/10);  
 **if** (*Msize* == 2) {  
 System.***out***.print(**"第"** + (*num*++) + **"次置换，需要页面"** + (*instruction*[i] / 10) + **"，成功命中。"**);  
 *printStatus*(list);  
 }  
 }  
 **else** {  
 list.add(0,*instruction*[i]/10);  
 *Error*++;  
 **if** (*Msize* == 2) {  
 System.***out***.print(**"第"** + (*num*++) + **"次置换，需要页面"** + (*instruction*[i] / 10) + **"，未能命中，将页面"** + (*instruction*[i] / 10) + **"装入页表。"**);  
 *printStatus*(list);  
 }  
 }  
 }  
 **else**{  
 **if**(list.contains(*instruction*[i]/10)) {  
 *Aimed*++;  
 list.remove((Integer)(*instruction*[i]/10));  
 list.add(0,*instruction*[i]/10);  
 **if** (*Msize* == 2) {  
 System.***out***.print(**"第"** + (*num*++) + **"次置换，需要页面"** + (*instruction*[i] / 10) + **"，成功命中。"**);  
 *printStatus*(list);  
 }  
 }  
 **else**{  
 list.remove(list.size()-1);  
 list.add(*instruction*[i]/10);  
 *Error*++;  
 **if** (*Msize* == 2) {  
 System.***out***.print(**"第"** + (*num*++) + **"次置换，需要页面"** + (*instruction*[i] / 10) + **"，未能命中，将页面"** + (*instruction*[i] / 10) + **"装入页表。"**);  
 *printStatus*(list);  
 }  
 }  
 }  
 }  
 *printStatus*(3);  
 *Msize*++;  
 }  
}

### ⑹ LFU()函数（最少访问页面算法）

核心思想： 使用一个与页数相同的数组记录对应页被访问的次数，需要进行页面置换时，找出内存中现有页被访问次数最少的那个，置换即可。

附·源代码：

1. **public** **static** **void** LFU(){//最少访问页面算法
2. Msize=2;
3. **while**(Msize<=32){
4. num=1;
5. Aimed=0;  //每次开始前，命中次数赋为0
6. Error=0;  //每次开始前，未命中次数赋为0
7. List<Integer> list =**new** LinkedList<Integer>();
8. **int**[] record=**new** **int**[32]; //共32页
9. **for**(**int** i=0;i<=319;i++){
10. **if**(list.size()<Msize) {
11. **if** (list.contains(instruction[i] / 10)) {
12. Aimed++;
13. record[instruction[i]/10]++;
14. **if** (Msize == 2) {
15. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，成功命中。");
16. printStatus(list);
17. }
18. }
19. **else**{
20. Error++;
21. list.add(instruction[i]/10);
22. record[instruction[i]/10]++;
23. **if** (Msize == 2) {
24. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，未能命中，将页面" + (instruction[i] / 10) + "装入页表。");
25. printStatus(list);
26. }
27. }
28. }
29. **else**{
30. **if**(list.contains(instruction[i]/10)){
31. Aimed++;
32. record[instruction[i]/10]++;
33. **if** (Msize == 2) {
34. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，成功命中。");
35. printStatus(list);
36. }
37. }
38. **else**{
39. Error++;
40. **int** num\_min=0;  //最小的那个下标
41. **int** MIN=0; //最小值
42. **for**(**int** index=0;index<Msize;index++){//找出使用次数最少的那个
43. **if**(record[list.get(index)]<MIN){
44. MIN=record[list.get(index)];
45. num\_min=index;
46. }
47. }
48. list.remove(num\_min);
49. list.add(instruction[i]/10);
50. record[instruction[i]/10]++;
51. **if** (Msize == 2) {
52. System.out.print("第" + (num++) + "次置换，需要页面" + (instruction[i] / 10) + "，未能命中，将页面" + (instruction[i] / 10) + "装入页表。");
53. printStatus(list);
54. }
55. }
56. }
57. }
58. printStatus(4);
59. Msize++;
60. }
61. }

### ⑺ printStatus()函数

说明：进行每次页面置换后，需要打印内存中现有页面信息；每轮指令执行完成后，也需要打印该算法的命中次数信息。

附·源代码：

1. **public** **static** **void** printStatus(List<Integer> list){
2. System.out.print("内存现装有：");
3. **for**(**int** i=0;i<list.size();i++){
4. System.out.print("页面"+list.get(i)+"\t");
5. }
6. System.out.println();
7. }
8. **public** **static** **void** printStatus(**int** which){
9. Effective=1-(**double**)(Error/(**double**)(320));
10. **switch** ((which)){
11. **case** 1:{System.out.print("OPT :Msize:");**break**;}
12. **case** 2:{System.out.print("FIFO :Msize:");**break**;}
13. **case** 3:{System.out.print("LRU :Msize:");**break**;}
14. **case** 4:{System.out.print("LFU :Msize:");**break**;}
15. **default**:{}
16. }
17. System.out.println(Msize+"\t缺失个数："+Error+"\t命中个数"+Aimed+"\t命中率："+Effective);
18. }

### ⑻ 主函数

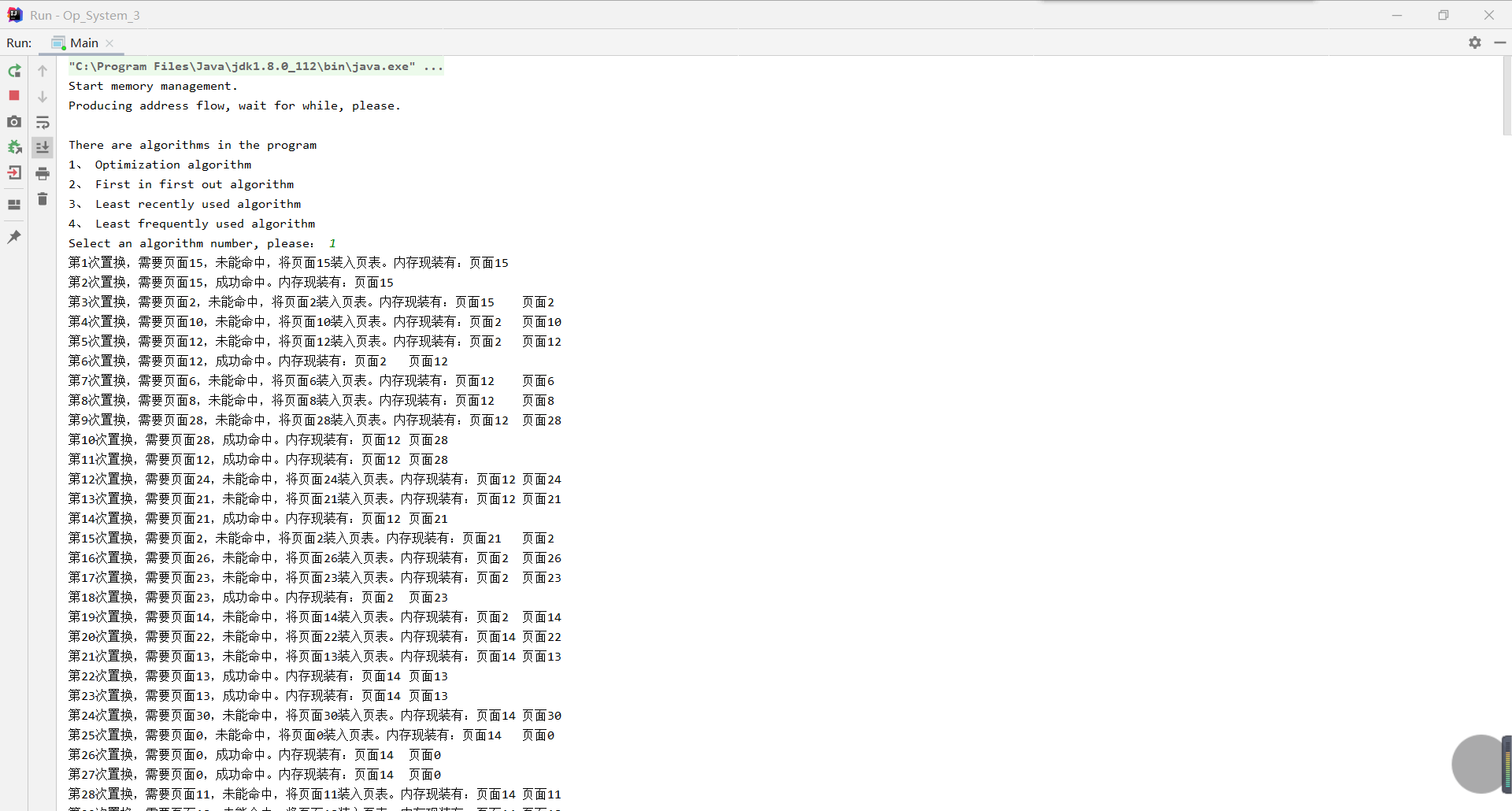
说明：主要起控制程序流程的作用。

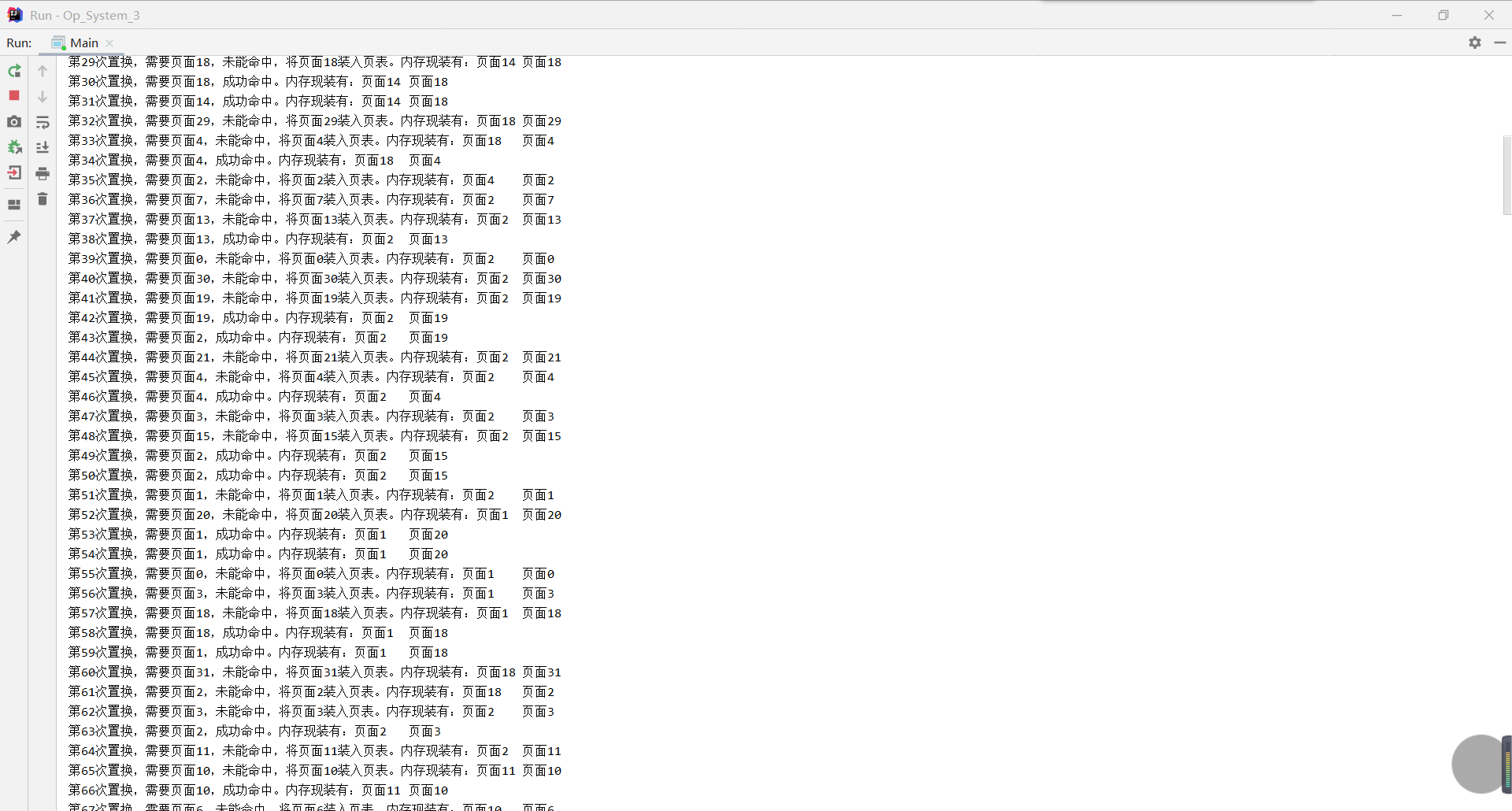
附·源代码：

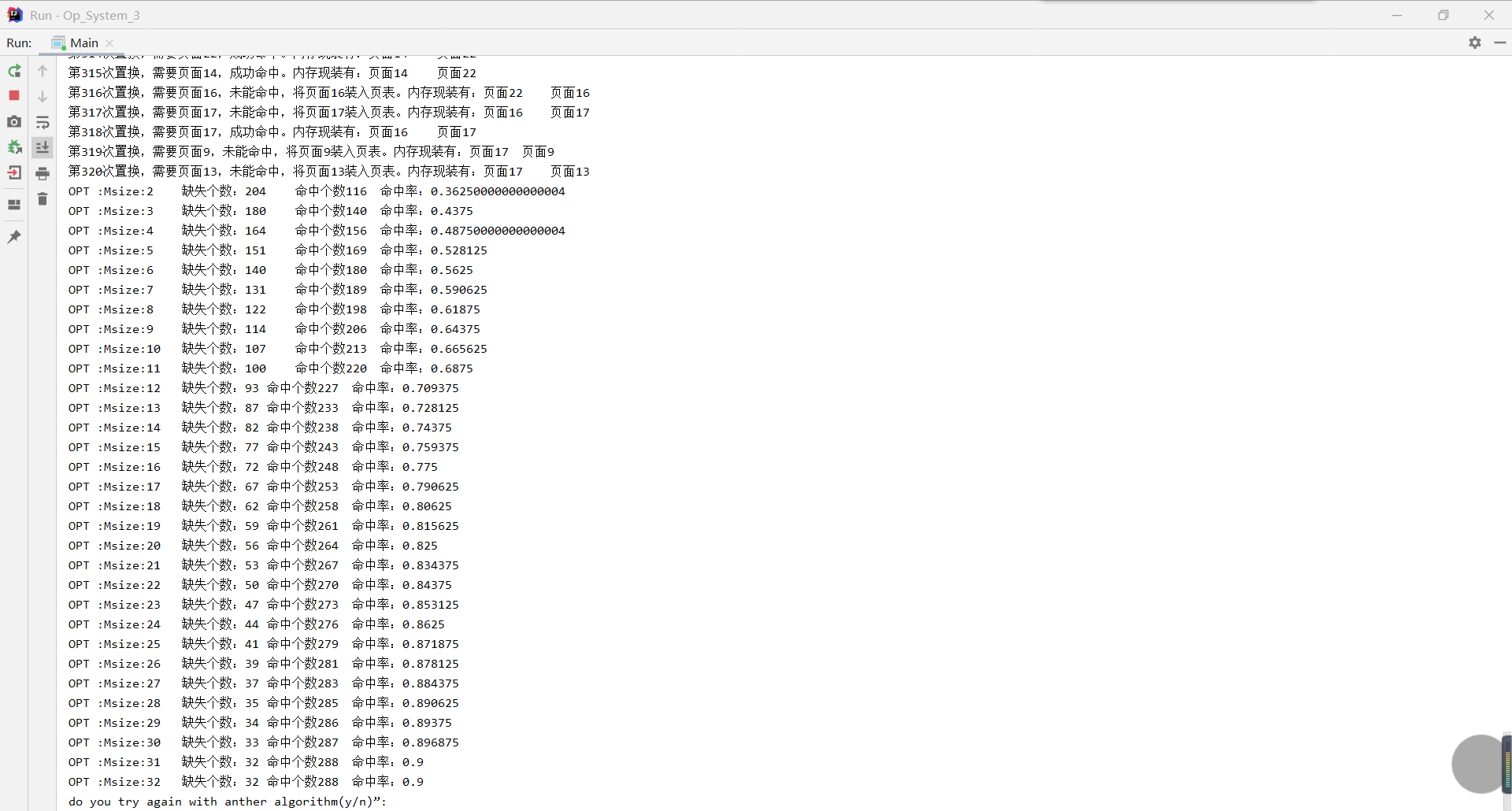
1. **public** **static** **void** main(String[] args) {
2. System.out.println("Start memory management.");
3. System.out.println("Producing address flow, wait for while, please.");
4. init();
5. **do** {
6. //选择算法
7. System.out.println("\nThere are algorithms in the program\n1、 Optimization algorithm\n2、 Least recently used algorithm\n3、 First in first out algorithm\n4、 Least frequently used algorithm ");
8. System.out.print("Select an algorithm number, please： ");
9. Scanner scanner=**new** Scanner(System.in);
10. **int** choi=scanner.nextInt();
11. **while**(!(choi<=4&&choi>=1)){
12. System.out.print("不存在您选择的算法序号，请重新选择管理算法：");
13. choi=scanner.nextInt();
14. };
15. **switch**(choi){
16. **case** 1:{ OPT();**break**;}
17. **case** 2:{ FIFO();**break**;}
18. **case** 3:{ LRU();**break**;}
19. **case** 4:{ LFU();**break**;}
20. }
21. System.out.print("do you try again with anther algorithm(y/n)”:");
22. **try** {
23. **char** choo=(**char**)System.in.read();
24. **while**(choo!='Y'&&choo!='y'&&choo!='N'&&choo!='n'){
25. System.out.print("您输入的格式有误，请重新输入(y/n)：");
26. choo=(**char**)System.in.read();
27. }
28. **if**(choo=='Y'||choo=='y') ;
29. **else** **break**;
30. } **catch** (IOException e) {
31. e.printStackTrace();
32. }
33. }**while**(**true**);
34. }

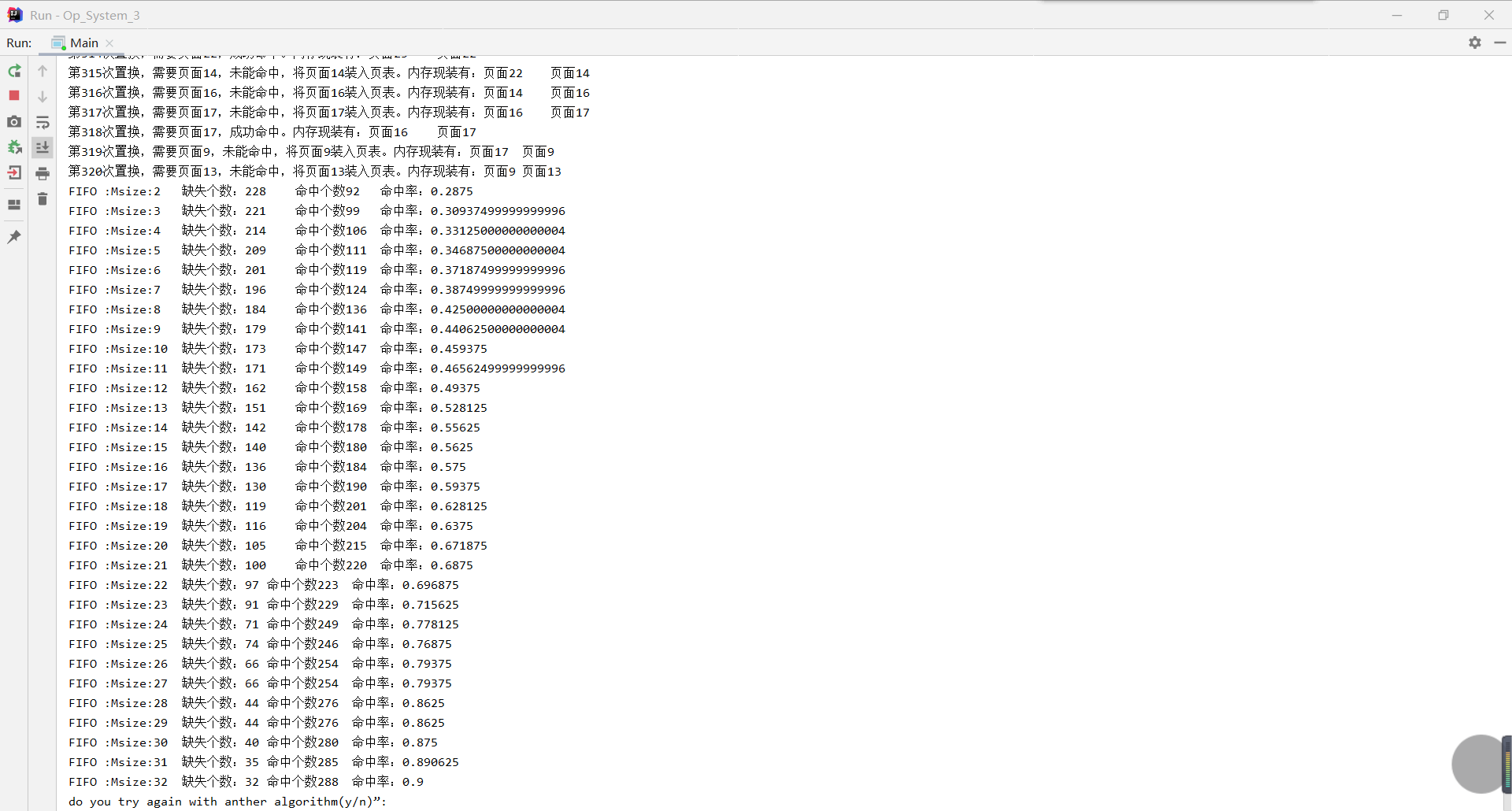
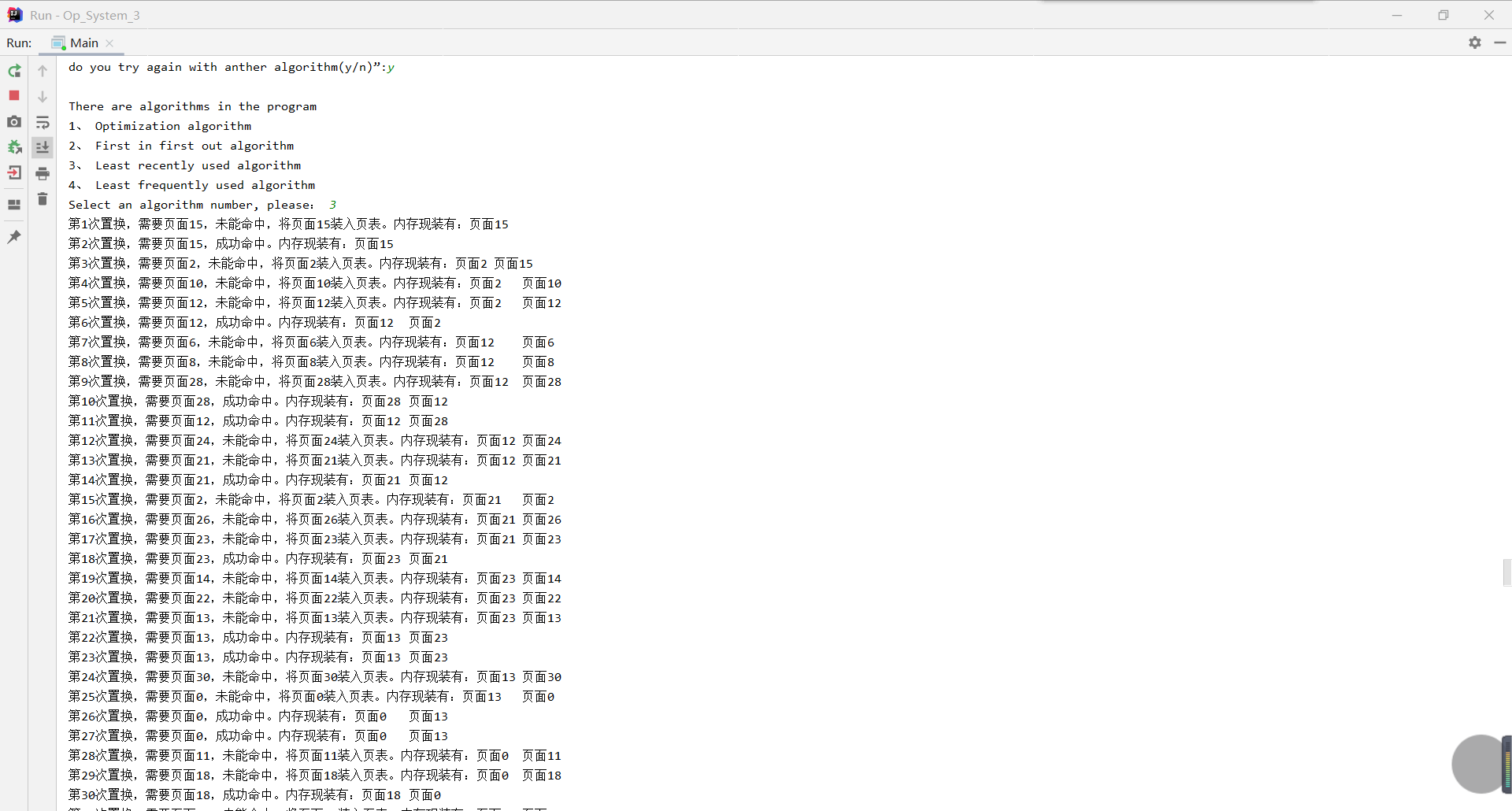
# 运行结果

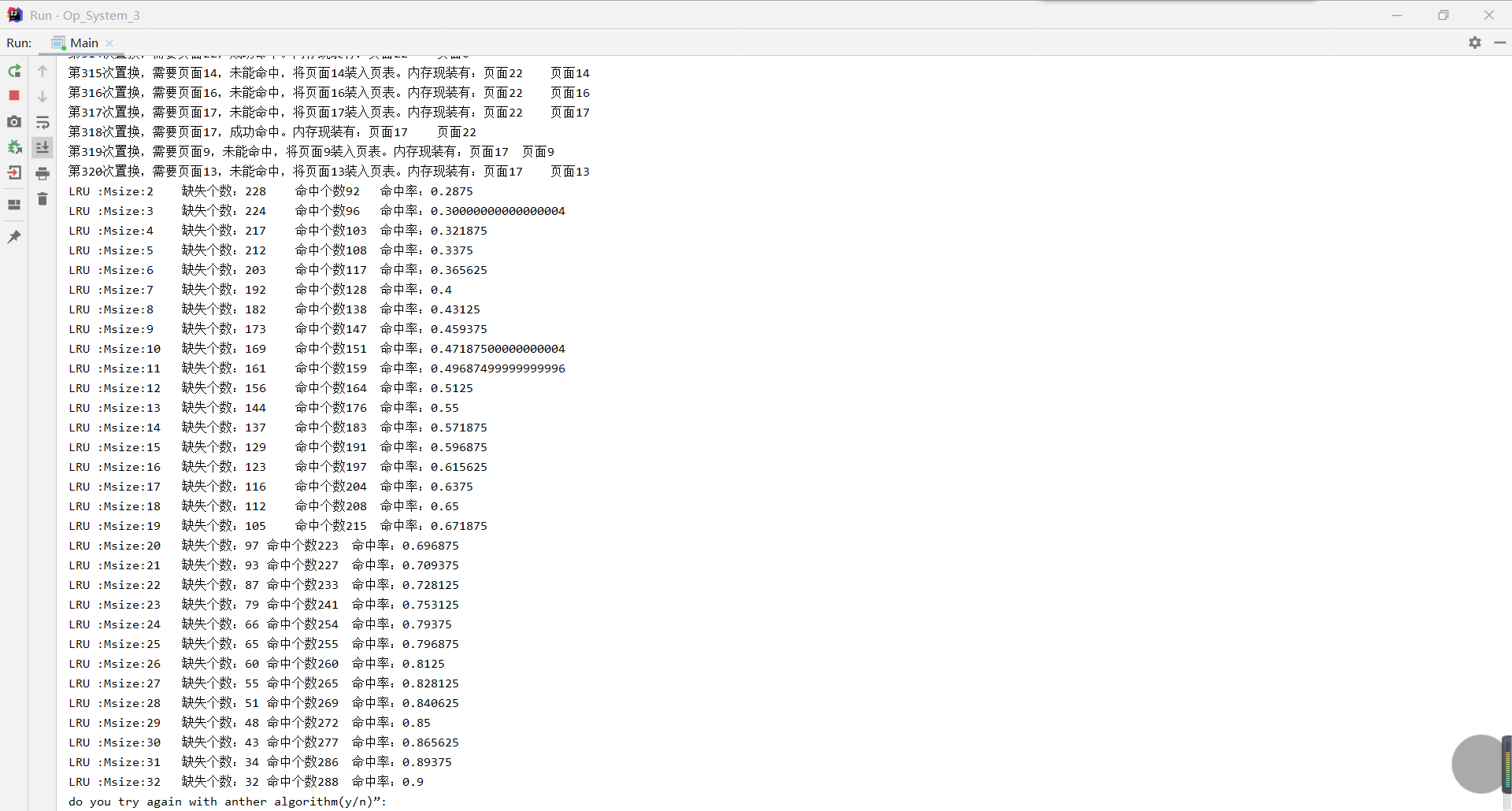
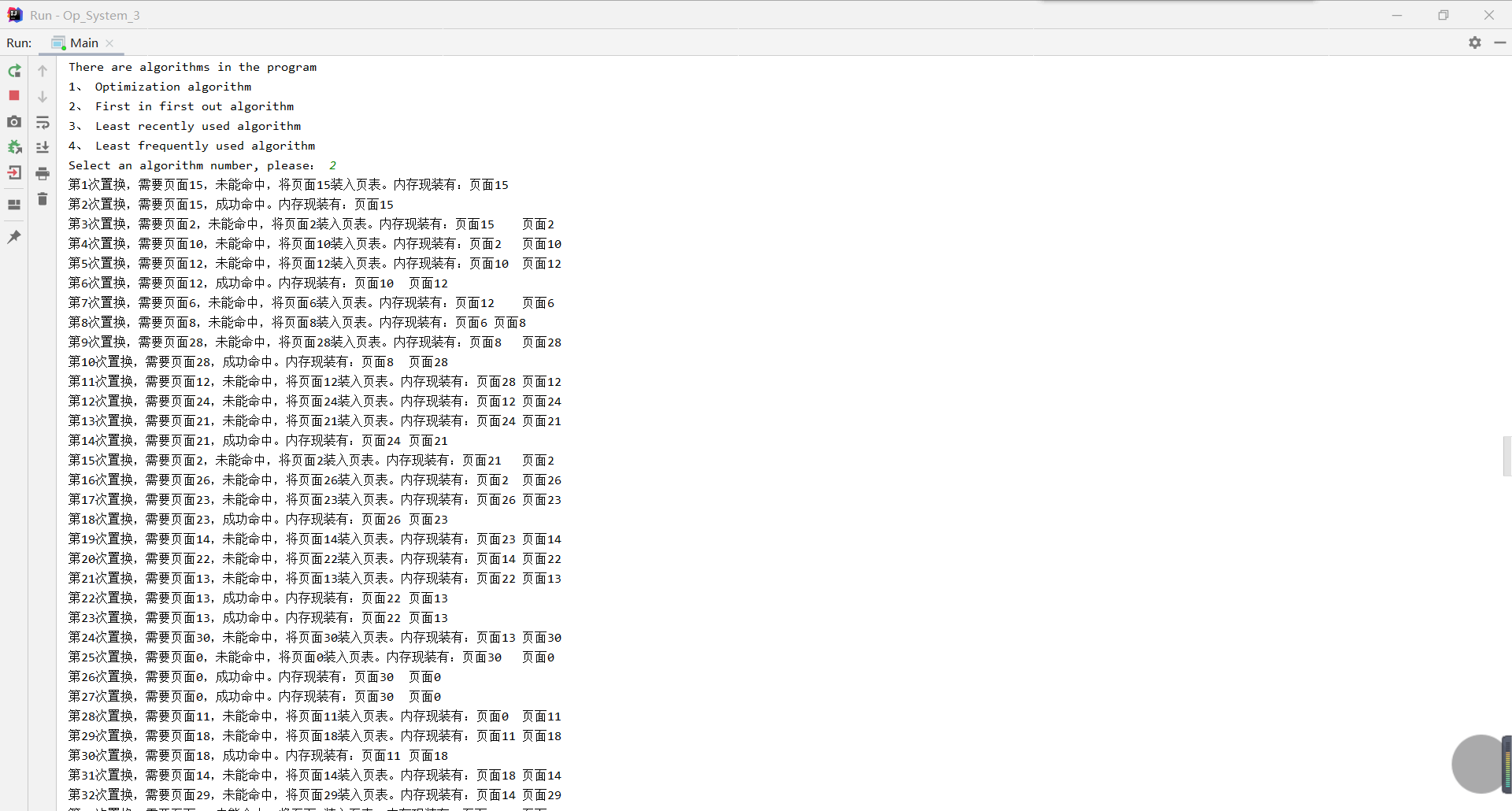
◎OPT算法（最佳页面置换法）

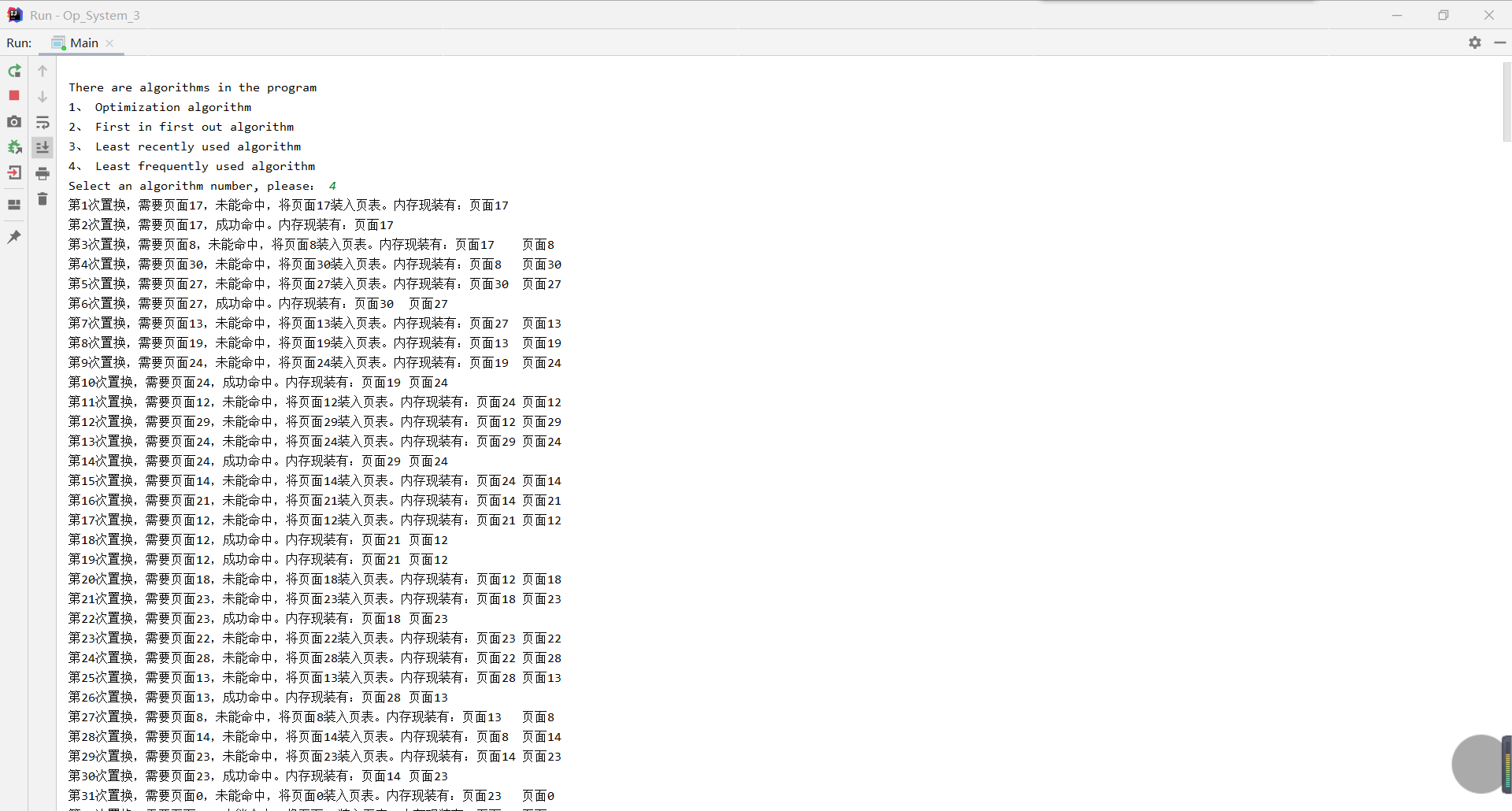


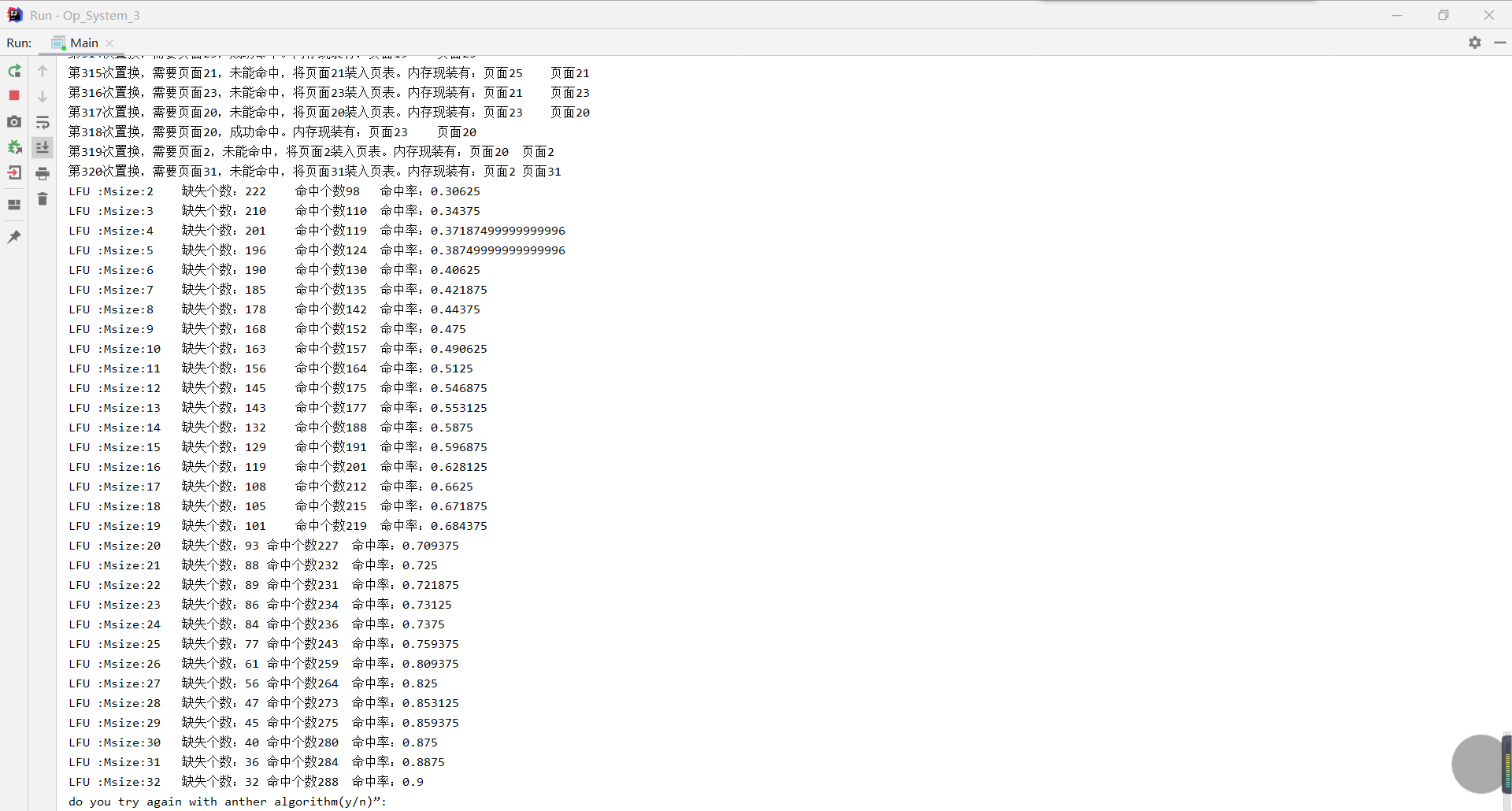




◎FIFO算法（先进先出置换法）

◎LRU算法（最近最少使用置换法）：

◎LFU算法（最少访问页面算法）



# 实验分析与结论

## 1.实验结果展示

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 命中率  Msize | OPT算法 | FIFO算法 | LRU算法 | LFU算法 |
| 2 | 0.36 | 0.29 | 0.29 | 0.29 |
| 3 | 0.44 | 0.31 | 0.30 | 0.31 |
| 4 | 0.49 | 0.33 | 0.32 | 0.33 |
| 5 | 0.53 | 0.35 | 0.34 | 0.35 |
| 6 | 0.57 | 0.37 | 0.37 | 0.37 |
| 7 | 0.59 | 0.39 | 0.40 | 0.39 |
| 8 | 0.62 | 0.43 | 0.43 | 0.43 |
| 9 | 0.64 | 0.44 | 0.46 | 0.44 |
| 10 | 0.67 | 0.46 | 0.47 | 0.46 |
| 11 | 0.69 | 0.47 | 0.50 | 0.47 |
| 12 | 0.71 | 0.49 | 0.51 | 0.50 |
| 13 | 0.73 | 0.53 | 0.55 | 0.53 |
| 14 | 0.74 | 0.56 | 0.57 | 0.56 |
| 15 | 0.76 | 0.56 | 0.60 | 0.56 |
| 16 | 0.78 | 0.58 | 0.62 | 0.58 |
| 17 | 0.79 | 0.59 | 0.64 | 0.59 |
| 18 | 0.81 | 0.63 | 0.65 | 0.63 |
| 19 | 0.82 | 0.64 | 0.67 | 0.64 |
| 20 | 0.83 | 0.67 | 0.70 | 0.67 |
| 21 | 0.83 | 0.69 | 0.71 | 0.69 |
| 22 | 0.84 | 0.70 | 0.73 | 0.70 |
| 23 | 0.85 | 0.72 | 0.75 | 0.72 |
| 24 | 0.86 | 0.78 | 0.79 | 0.78 |
| 25 | 0.87 | 0.77 | 0.80 | 0.77 |
| 26 | 0.88 | 0.79 | 0.81 | 0.79 |
| 27 | 0.88 | 0.79 | 0.83 | 0.79 |
| 28 | 0.89 | 0.86 | 0.84 | 0.86 |
| 29 | 0.89 | 0.86 | 0.85 | 0.86 |
| 30 | 0.90 | 0.88 | 0.87 | 0.88 |
| 31 | 0.90 | 0.89 | 0.89 | 0.89 |
| 32 | 0.90 | 0.90 | 0.90 | 0.90 |

## 2.结论

①无论是哪一种算法，只要内存够大，命中率都很高。因为内存足够大的话，每个页面只需要置入内存一次，无论后续需不需要，内存始终都有该页面，无需置换或者置换次数很少。

②无论是哪一种算法，在内存空间较小的时候命中率都不是很高。这是由于内存较小，无法置入太多的页面，内存中页面能置入的页面太少，导致命中率大大降低。

③ 大多数情况下，操作系统的内存空间不会大于执行所有指令所需要的空间，也就是多道程序运行过程中需要发生页面置换。同等情况下，最佳置换法OPT的命中次数最多，因为它考虑到了现有页面中后续出现的次数，换句话说，它尽可能地增加了命中次数。但是，操作系统往往是无法预知进程的调度情况的，因而这只是理想的置换法。

④当页面分布较为分散，即前面出现过的页面后续出现的频率不高时，FIFO算法与LRU算法、LFU算法极为接近。

⑤正常情况下（即除最佳置换法），LFU算法的命中率最高，FIFO算法的命中率最低。LRU算法的命中率与地址流分布、内存容量等有很大的关系，具有一定的不确定性，在小作业中较不稳定。