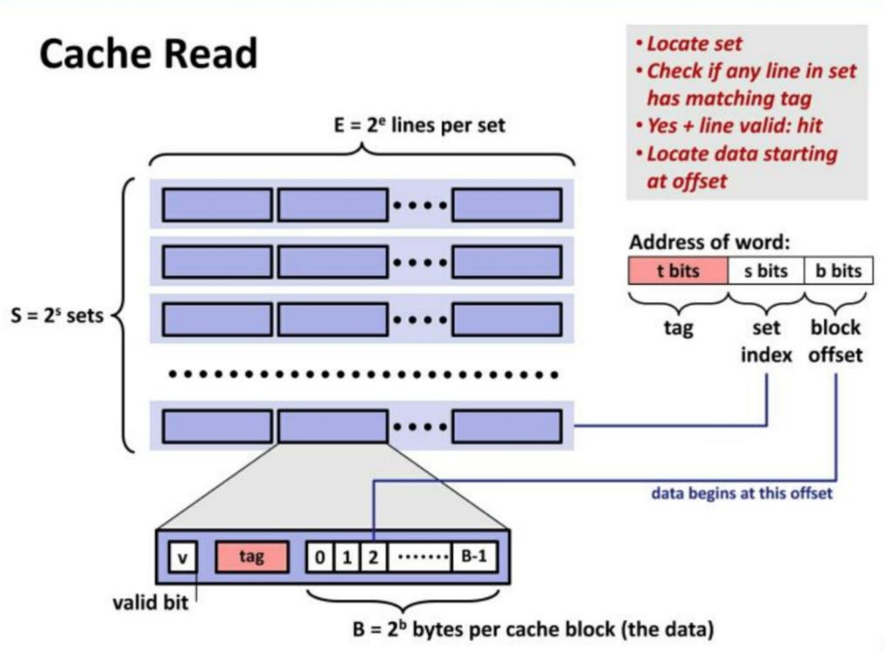
Part A

本部分的实验要求我们实现一个缓存模拟器；它应该能实现简单的组相联映射功能

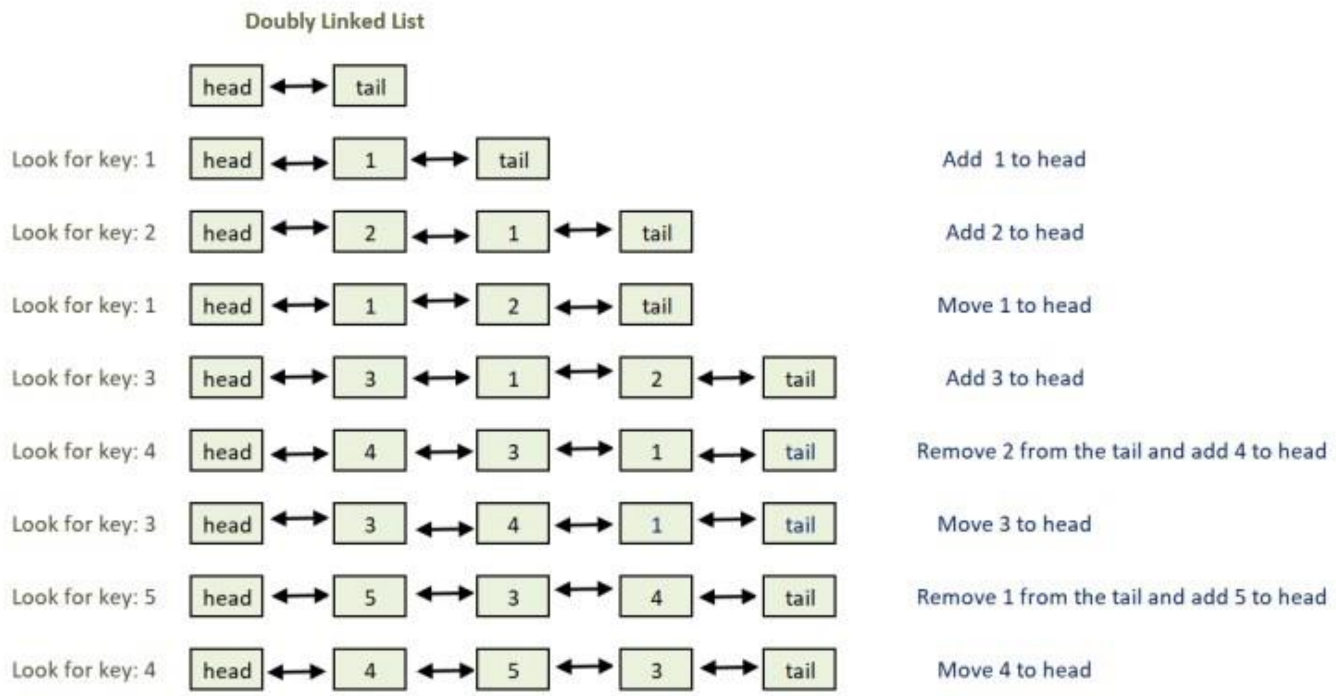


内存以块为单位被存储到缓存内，存储的下标是 set index(图中所示的中间几位)，即 set index=i 的地址请求会被放置到第 i 组缓存。在组相联中，每组 cache 都有2^s个块，每个块都有2^b位来存储数据，所以缓存的总大小是$E \cdot 2^{s+b}$位。

读取地址的 tag 和 set index 的操作由位运算完成

```
ADDR getIndex() {
    // get cache index of variable 'addr'
    ADDR mask = 0;
    mask = ~mask;
    mask = ~(mask << n_setbits);
    return mask & (addr >> n_blockbits);
}

ADDR getTag() {
    // get cache tag of variable 'addr'
    return addr >> n_setbits >> n_blockbits;
}
```

因为每组的容量有限，所以我们不能无限地向某个组内放置数据；当数据达到限额时，就必须把一些块从组中移出。本实验中希望我们使用 LRU 策略。



LRU 使用一个有序的数据结构描述所有块，块按照最近使用的时间为键排序。因为考虑到这样的有序结构常常需要随机的删除和在首部插入，使用连续内存会导致大量的元素移动，所以使用双向链表实现是比较明智的。有时 LRU 的容量比较大，线性查找链表的效率比较低，这时可以用哈希表快速索引链表结点。

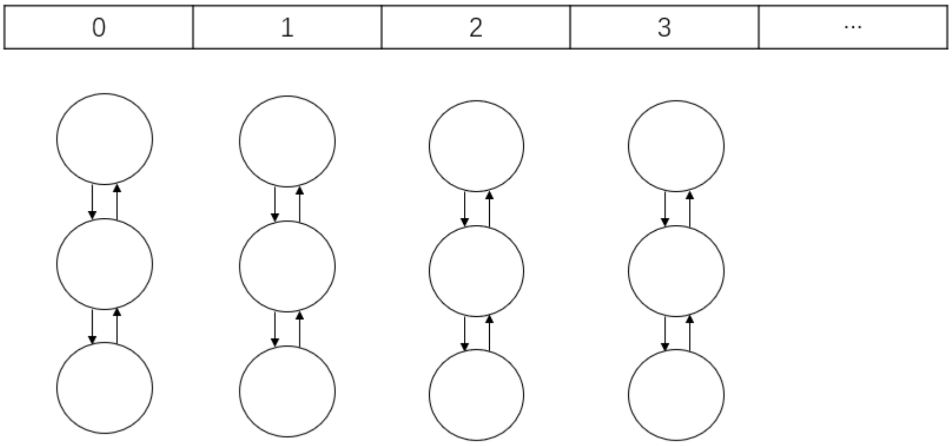
每个链表结点包含它的键，前驱后继指针(因为是模拟，我们无需真正存储2^b位的整块数据)。而每个组需要保存该组元素的链表首位，以及链表长度。

```
struct ListNode {
    ADDR tag;
    ListNode* prev;
    ListNode* next;
};

struct LRUCache {
    ListNode* head;
    ListNode* tail;
    int size;
};
```

我们模拟用到的主数据结构是一个组组成的数组，也就是

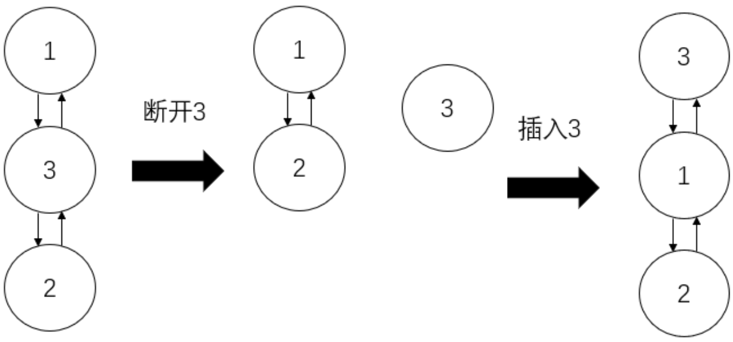
```
LRUCache* cacheList;


void initCache() {
    // initialize cache in memory
    int numCache = 1 << n_setbits;
    cacheList = (LRUCache *) malloc(sizeof(LRUCache)*numCache);
    for (int i = 0;i < numCache;i++) {
        cacheList[i].head = NULL;
        cacheList[i].tail = NULL;
        cacheList[i].size = 0;
    }
```

```
}
```

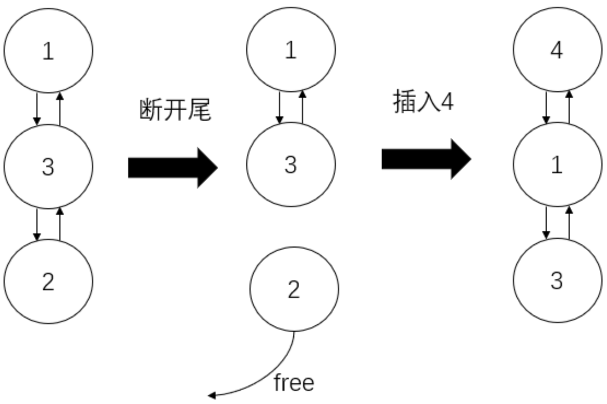


为了实现 LOAD，STORE 和 MODIFY，我们需要一个统一的接口 query，从而 LOAD，STORE 等价为一次 query，MODIFY 是两次 query。query 又分为命中和不命中两种情况，我们来看看分别应该怎样处理。

请求3，命中



请求4，不命中



可以看出，我们需要的辅助函数有两个；一个是实现链表任意位置的随机删除，另一个是实现在链表头插入结点。
头部插入的函数要考虑链表为空的边界条件

```
void pushFront(ListNode* node, ADDR index) {
    if (cacheList[index].head) {
        node->prev = NULL;
```

```
        node->next = cacheList[index].head;
        cacheList[index].head->prev = node;
        cacheList[index].head = node;
    }
    else {
        cacheList[index].head = node;
        cacheList[index].tail = node;
        node->next = NULL;
        node->prev = NULL;
    }
    cacheList[index].size++;
}
```

随机删除的函数要考虑目标结点在首尾的边界条件

```
void remove(ListNode* node, ADDR index) {
    ListNode* prev = node->prev;
    ListNode* next = node->next;
    if (prev) prev->next = next;
    if (next) next->prev = prev;
    if (node == (cacheList[index].head)) cacheList[index].head = next;
    if (node == (cacheList[index].tail)) cacheList[index].tail = prev;
    cacheList[index].size--;
}
```

仅用这两者，以及 malloc，就能写出 query 函数

```
void query(ADDR index, ADDR tag) {
    // load or store
    // hit: find node 'tag' in list cacheList[index], hit++
    // miss: cant find, create a new node, put it at head.
    // evict: if size>n_lines, delete tail
    if (cacheList[index].size==0) {
        ListNode* newnode = (ListNode*)malloc(sizeof(ListNode));
        newnode->tag = tag;
        pushFront(newnode, index);
        miss_count++;
        if (verbose)
            printf(" miss");
    }
    else {
        ListNode* node = cacheList[index].head;
        while (node) {
            if (node->tag == tag) break;
            node = node->next;
        }
        if (node) {
            remove(node, index);
            pushFront(node, index);
            hit_count++;
            if (verbose)
                printf(" hit");
        }
```

```
        else {
            node = (ListNode*)malloc(sizeof(ListNode));
            node->tag = tag;
            pushFront(node, index);
            miss_count++;
            if (verbose)
                printf(" miss");
            if (cacheList[index].size > n_lines) {
                remove(cacheList[index].tail, index);
                eviction_count++;
                if (verbose)
                    printf(" eviction");
            }
        }
    }
}
```

基于 sscanf 设计一个用于解析文件的行的函数

```
void parseLine(char line[255]) {
    if (line[0] != ' ') {
        oper = 'I';
        return;
    }
    sscanf(line, " %c %llx,%d", &oper, &addr, &opSize);
}
```

运行模拟，得到和正确值相同的结果

```
int main()
{
    // show verbosely
    verbose = 0;
    // set file name
    sprintf(tracefile,"%s","./traces/long.trace");
    // set -s -E -b
    initCacheParams(5, 1, 5);
    // allocate memory for cache
    initCache();
    // open trace file
    fp = fopen(tracefile, "r");
    if (!fp) {
        printf("File %s isn\'t existing!",tracefile);
        return 1;
    }
    // read lines
    while (fgets(buf, 200, fp)) {
        parseLine(buf);
        if (oper == 'I') continue;
        ADDR index = getIndex();
        ADDR tag = getTag();
        // printf("Index: %llx, tag: %llx\n", index, tag);
        if (verbose)
```

```
            printf("%c %llx,%d", oper, addr, opSize);
        if (oper == 'L' || oper == 'S')
            query(index, tag);
        else if (oper == 'M')
            modifyQuery(index, tag);
        if (verbose)
            printf("\n");
    }
    printSummary(hit_count, miss_count, eviction_count);
    // close trace file
    fclose(fp);
}
```

Part B

本部分的实验要求我们实现能充分利用缓存的矩阵转置算法;

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N]);
```

实验的要求是特殊化的，缓存的参数为 s = 5 ,E = 1, b = 5，也就是一个每组 32 字节(8 个整形)，共 32 组的直接映射缓存。我们要特别处理三个测试例，分别是 32x32，64x64 和 61x67 三种 size 的矩阵。实验允许我们为这三个矩阵设计专用的算法，以实现最大化的缓存利用率。

1) 32x32

第一个矩阵是 32x32 的矩阵，它的转置也是 32x32，两个矩阵在内存上连续排布。我们可以考察$A[i][j]$和$B[j][i]$所被映射到的缓存。设$A[0][0]$所映射到的缓存为第 0 号，$A[0][8]$映射到的缓存为第 1 号，按这个顺序递推，考察矩阵中各元素所属的缓存。
显然，j 每增 8 则 cache index 增 1(越界则取模 32)，i 每增 8 则 cache index 增 4

$$cache\_index[i][j] = [4i + (j/8)]\%32$$

画出缓存序号矩阵:

```
for i in range(32):
    for j in range(32):
        I[i][j] = (4*i+(j/8))%32
```

```
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 4,  4,  4,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  7,  7,  7,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15, 15,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
20, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 23, 23, 23,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
28, 28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29, 29, 29, 29, 29, 30, 30, 30, 30, 30, 30, 30, 30, 31, 31, 31, 31, 31, 31, 31, 31,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 4,  4,  4,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  7,  7,  7,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15, 15,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
20, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 23, 23, 23,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
28, 28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29, 29, 29, 29, 29, 30, 30, 30, 30, 30, 30, 30, 30, 31, 31, 31, 31, 31, 31, 31, 31,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 4,  4,  4,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  7,  7,  7,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15, 15,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
20, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 23, 23, 23,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
28, 28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29, 29, 29, 29, 29, 30, 30, 30, 30, 30, 30, 30, 30, 31, 31, 31, 31, 31, 31, 31, 31,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 4,  4,  4,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  7,  7,  7,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15, 15,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
20, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 23, 23, 23,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
28, 28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29, 29, 29, 29, 29, 30, 30, 30, 30, 30, 30, 30, 30, 31, 31, 31, 31, 31, 31, 31, 31,
```

可以看到整个矩阵在行顺序上$cache_index[i][j] = cache_index[i+8][j]$，在列顺序上每 8 个元素变化一次。假设我们现在从$A[0][0]$开始操作，LOAD $A[0][0]$将把$A[0][0], A[0][1], \ldots, A[0][7]$都取入缓存，如果不希望重复取这块内存，则最好一次性把这 8 个数都 STORE 到 B 矩阵的$B[j][i]$处，然后再也不访问这块内存。这样对 A 来说，缓存利用率最高。这 8 个数的目标地址为$B[0][0], B[1][0], \ldots, B[7][0]$，对应的缓存块为0,4,8,12,16,20,24,28。也就是说，如果希望存下$A[0][0], A[0][1], \ldots, A[0][7]$，则需要用到0,4,8,12,16,20,24,28这些缓存，也就是同时取出$B[0][0] \sim B[0][7], B[1][0] \sim B[1][7], \ldots, B[7][0] \sim B[7][7]$进入缓存等待存储。

问题来了，在存储完$A[0][0], A[0][1], \ldots, A[0][7]$之后的下一步，该存储哪些$A[i][j]$呢？这时，现在留在缓存中内部的$B[0][0] \sim B[0][7], B[1][0] \sim B[1][7], \ldots, B[7][0] \sim B[7][7]$仅仅被存储了 1/8，把它们都处理完毕后，再把它们换出缓存显然对 B 来说利用率更高。这也就告诉我们，以 8x8 的滑动窗口来处理矩阵转置是明智的。

现在的结论是：以 8x8 的滑动窗口来处理矩阵转置充分利用了缓存；但现在仍然有一个问题，当滑动窗口在经过对角线的 8x8 矩阵上时，将有$B[i][i] \coloneqq A[i][i]$型的赋值操作。因为$B[i][i]$和$A[i][i]$使用相同的缓存，如果按顺序赋值，如顺序执行

$$B[0][0] \coloneqq A[0][0], B[1][0] \coloneqq A[0][1], B[2][0] \coloneqq A[0][2], B[3][0] \coloneqq A[0][3] \ldots$$

则在$B[0][0] \coloneqq A[0][0]$时，会面临缓存抖动；即$B[0][0]$入缓存，$A[0][0]$出缓存，然后$A[0][1]$又被用到，$A[0][0]$入缓存，这就造成了比较大的性能损失。为了避免这种损失，可以在$A[0][0{:}8]$不再会被用到时，再执行$B[0][0] \coloneqq A[0][0]$，即调整顺序，把对角线的赋值放到循环的最后。

$$B[1][0] \coloneqq A[0][1], B[2][0] \coloneqq A[0][2], \ldots, B[7][0] \coloneqq A[0][7], B[0][0] \coloneqq A[0][0]$$

这样，在完成上述赋值后，$A[0][0{:}8]$也无需再入缓存；类似的，有

$$B[0][1] \coloneqq A[1][0], B[2][1] \coloneqq A[1][2], \ldots, B[7][1] \coloneqq A[1][7], B[1][1] \coloneqq A[1][1]$$

至于不经过对角线的 8x8 矩阵，则没有这种担忧。在其他位置，LOAD A 使用的缓存始终和 STORE B 使用的缓存不相交：

```python
for i in range(32):
    for j in range(32):
        if I[i][j]==I[j][i]:
            print("(%d,%d)"%(i,j),end=',')
```

executed in 8ms, finished 21:23:53 2021-02-19

(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),(11,11),(12,12),(13,13),(14,14),(15,15),(16,16),(17,17),(18,18),(19,19),(20,20),(21,21),(22,22),(23,23),(24,24),(25,25),(26,26),(27,27),(28,28),(29,29),(30,30),(31,31),

综上，得到能高效处理 32x32 型矩阵的算法。

```c
if (M == 32 && N == 32) { // unique trans function designed for 32x32 matrix.
```

```
        int i, j, x, y, tmp;

        for (x = 0; x < 32; x += 8) {
            for (y = 0; y < 32; y += 8) {
                // step of silding window = 8
                for (i = 0; i < 8; ++i) {
                    for (j = 0; j < 8; ++j) {
                        // treat blocks at diagonal specially
                        if (x==y && i==j)
                            continue;
                        tmp = A[x + i][y + j];
                        B[y + j][x + i] = tmp;
                    }
                    // now transfer diagonal
                    if (x==y) {
                        tmp = A[x + i][y + i];
                        B[y + i][x + i] = tmp;
                    }
                }
            }
        }
    }
```

2) 64x64

第二个问题相比第一个难度提升了好几倍。类似的，我们能推理出在 64x64 的矩阵中缓存序号的公式

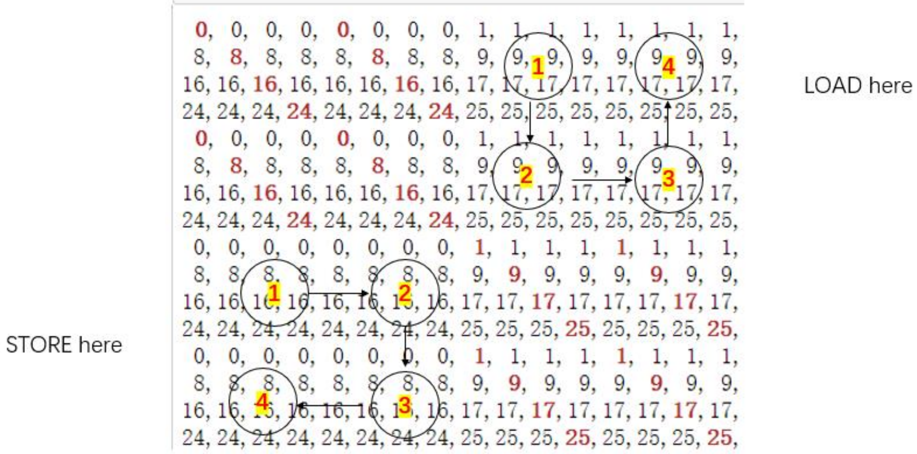
$$cache\_index[i][j] = [8i + (j/8)]\%32$$

观察缓存序号矩阵，有

```
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
```

这里只截取了 32x32 的一部分，但是仍然可以看出这个矩阵有着 4x8 的周期性。但如果像上一题那样，使用 4x8 的滑动窗口，又会有一些新的麻烦。因为 A 的 4x8 的滑动窗口写入 B 的 8x4 的滑动窗口，同列的 8 行中存在重复的 cache index，所以会有额外开销。

为此，可以用顺时针方法遍历 4 个 4x4 矩阵，以尽可能减少抖动开销。



求解出哪些下标存在读写缓存的冲突。

```python
for i in range(64):
    for j in range(64):
        if I[i][j]==I[j][i]:
            print("(%d,%d)"%(i,j),end=',')
```

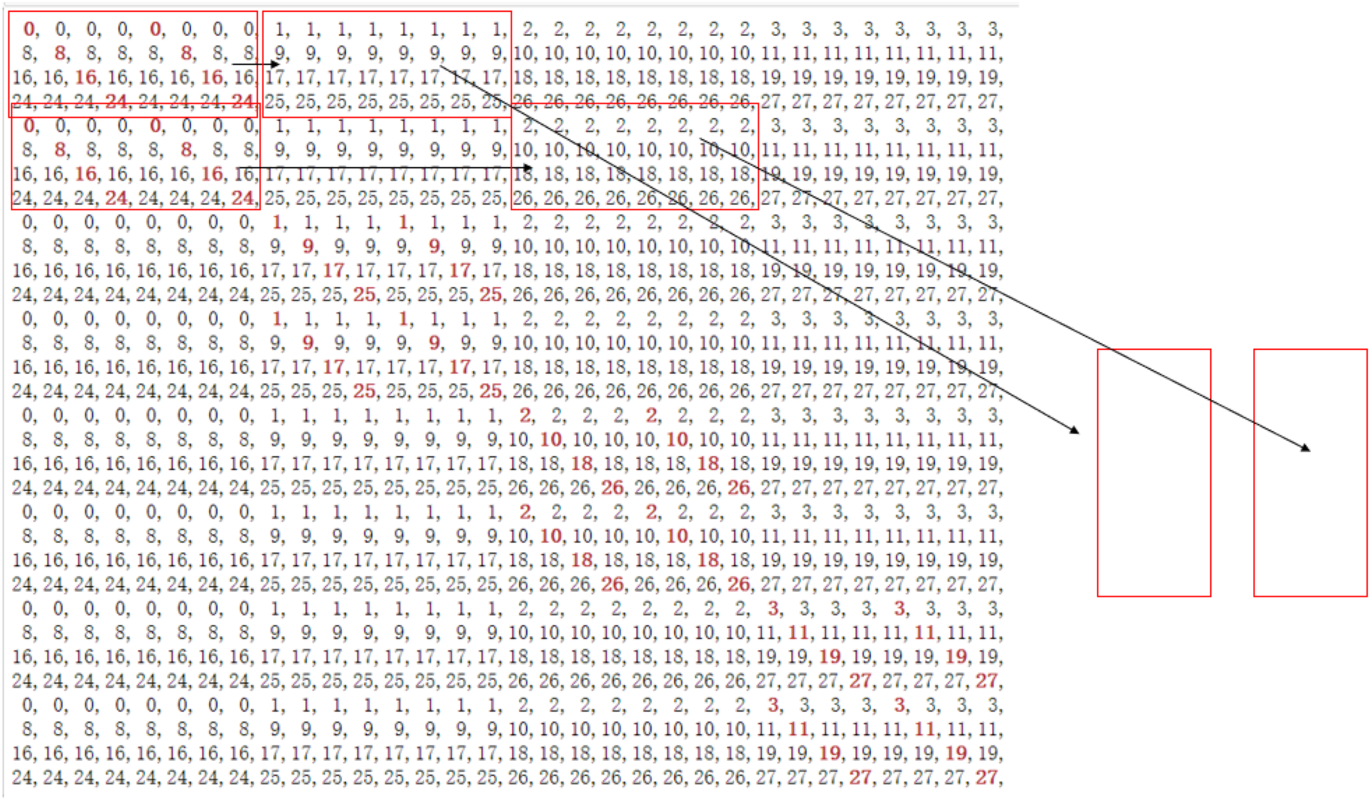
executed in 12ms, finished 21:38:59 2021-02-19

```
(0,0),(0,4),(1,1),(1,5),(2,2),(2,6),(3,3),(3,7),(4,0),(4,4),(5,1),(5,5),(6,2),(6,6),(7,3),(7,7),(8,8),(8,12),(9,9),(9,13),(10,10),(10,14),(11,11),(11,15),(12,8),(12,12),(13,9),(13,13),(14,10),(14,14),(15,11),(15,15),(16,16),(16,20),(17,17),(17,21),(18,18),(18,22),(19,19),(19,23),(20,16),(20,20),(21,17),(21,21),(22,18),(22,22),(23,19),(23,23),(24,24),(24,28),(25,25),(25,29),(26,26),(26,30),(27,27),(27,31),(28,24),(28,28),(29,25),(29,29),(30,26),(30,30),(31,27),(31,31),(32,32),(32,36),(33,33),(33,37),(34,34),(34,38),(35,35),(35,39),(36,32),(36,36),(37,33),(37,37),(38,34),(38,38),(39,35),(39,39),(40,40),(40,44),(41,41),(41,45),(42,42),(42,46),(43,43),(43,47),(44,40),(44,44),(45,41),(45,45),(46,42),(46,46),(47,43),(47,47),(48,48),(48,52),(49,49),(49,53),(50,50),(50,54),(51,51),(51,55),(52,48),(52,52),(53,49),(53,53),(54,50),(54,54),(55,51),(55,55),(56,56),(56,60),(57,57),(57,61),(58,58),(58,62),(59,59),(59,63),(60,56),(60,60),(61,57),(61,61),(62,58),(62,62),(63,59),(63,63),
```

欧吼，我们发现部分满足$|i-j|=4$的$A[i][j]$在写入时也会面临和对角线相同的缓存抖动。如图所示

```
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11,
16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27,
```

这些麻烦的红色数字出现在经过对角线的 8x8 矩阵内，而且不容易使用 1)的方法消除掉。我们可以特别地处理这些矩阵，这里使用的方法是霸占未使用的缓存的方法。



借助未使用的缓存，临时存储经过 A 对角线的 8x8 矩阵，然后再复制到 B 的对应区域，这样就只是多了从 A 存储到临时缓存的开销，而解决了对角线上的缓存抖动问题。

这样就得到了 64x64 型矩阵的转置算法

```
if (M == 64 && N == 64) { // unique trans function designed for 64x64 matrix.
        int i, j, x, y, tmp;

        for (x = 0;x < 64;x += 8) {
            // deal with diagnal
            // store 8x8 diagnal matrix into unused caches
            for (i = 0;i < 4;i++) {
                for (j = 0;j < 8;j++) {
                    tmp = A[x + i][x + j];
                    B[x + i][(x + 8) % 64 + j] = tmp;
                }
            }
            for (i = 4;i < 8;i++) {
                for (j = 0;j < 8;j++) {
                    tmp = A[x + i][x + j];
                    B[x + i][(x + 16) % 64 + j] = tmp;
                }
            }
            for (i = 0;i < 4;i++) {
                for (j = 0;j < 4;j++) {
                    tmp = B[x + j][(x + 8) % 64 + i];
                    B[x + i][x + j] = tmp;
                }
            }
            for (i = 0;i < 4;i++) {
                for (j = 4;j < 8;j++) {
```

```
                tmp = B[x + j][(x + 16) % 64 + i];
                B[x + i][x + j] = tmp;
            }
        }
        for (i = 4;i < 8;i++) {
            for (j = 0;j < 4;j++) {
                tmp = B[x + j][(x + 8) % 64 + i];
                B[x + i][x + j] = tmp;
            }
        }
        for (i = 4;i < 8;i++) {
            for (j = 4;j < 8;j++) {
                tmp = B[x + j][(x + 16) % 64 + i];
                B[x + i][x + j] = tmp;
            }
        }
        // other 8x8 matrix
        for (y = 0;y < 64;y += 8) {
            if (x == y) continue;
            for (j = 0;j < 8;j++) {
                for (i = 0;i < 4;i++) {
                    tmp = A[y + j][x + i];
                    B[x + i][y + j] = tmp;
                }
            }
            for (j = 7;j >= 0;j--) {
                for (i = 4;i < 8;i++) {
                    tmp = A[y + j][x + i];
                    B[x + i][y + j] = tmp;
                }
            }
        }
    }
}
```

3) 61x67

有

$$cache\_index[i][j] = [((61i + j)/8)]\%32$$

```
 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,
 8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12,
16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19, 20, 20, 20, 20, 20, 20,
25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27, 28, 28, 28, 28, 28, 28, 28, 28, 29,
 1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,  4,  4,  4,  4,  5,  5,  5,  5,
 9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13,
18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19, 20, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 21, 21, 21, 21, 22, 22,
26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27, 28, 28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29, 29, 29, 29, 29, 30, 30, 30, 30, 30,
 3,  3,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  6,  6,  6,
11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 15, 15, 15,
19, 19, 20, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 23,
28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29, 29, 29, 29, 29, 30, 30, 30, 30, 30, 30, 30, 30, 31, 31, 31, 31, 31, 31, 31, 31,  0,
 4,  4,  4,  4,  5,  5,  5,  5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  7,  7,  7,  8,  8,  8,  8,
12, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15, 15, 16, 16, 16, 16, 16, 16, 16,
21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 23, 23, 23, 24, 24, 24, 24, 24, 24, 24, 24, 25, 25,
29, 29, 29, 30, 30, 30, 30, 30, 30, 30, 30, 31, 31, 31, 31, 31, 31, 31, 31,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,
 6,  6,  6,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  7,  7,  7,  8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9,
14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15, 15, 16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18,
22, 22, 23, 23, 23, 23, 23, 23, 23, 23, 24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26,
31, 31, 31, 31, 31, 31, 31,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,
 7,  7,  7,  7,  8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11,
15, 16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19,
24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27, 28, 28,
 0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,  4,
 9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12,
17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19, 20, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21,
25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27, 28, 28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29, 29, 29,
 2,  2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  5,  5,  5,  6,
10, 10, 10, 10, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14,
18, 19, 19, 19, 19, 19, 19, 19, 19, 20, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22,
27, 27, 27, 27, 27, 27, 28, 28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29, 29, 29, 29, 29, 30, 30, 30, 30, 30, 30, 30, 30, 31, 31,
 3,  3,  3,  4,  4,  4,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,
```

同样先检测哪些下标存在冲突

```python
for i in range(61):
    for j in range(61):
        if I[i][j]==I[j][i]:
            print("(%d,%d)"%(i,j),end=',')
```

executed in 10ms, finished 10:11:25 2021-02-20

```
(0,0),(0,31),(1,1),(1,32),(2,2),(2,33),(3,3),(3,34),(4,4),(4,35),(5,5),(5,36),(6,6),(6,37),(7,7),(7,38),(8,8),(8,39),(9,9),(9,40),(10,10),(10,41),(11,11),(11,42),(12,12),(12,43),(13,13),(13,44),(14,14),(14,45),(15,15),(15,46),(16,16),(16,47),(17,17),(17,48),(18,18),(18,49),(19,19),(19,50),(20,20),(20,51),(21,21),(21,52),(22,22),(22,53),(23,23),(23,54),(24,24),(24,55),(25,25),(25,56),(26,26),(26,57),(27,27),(27,58),(28,28),(28,59),(29,29),(29,60),(30,30),(31,0),(31,31),(32,1),(32,32),(33,2),(33,33),(34,3),(34,34),(35,4),(35,35),(36,5),(36,36),(37,6),(37,37),(38,7),(38,38),(39,8),(39,39),(40,9),(40,40),(41,10),(41,41),(42,11),(42,42),(43,12),(43,43),(44,13),(44,44),(45,14),(45,45),(46,15),(46,46),(47,16),(47,47),(48,17),(48,48),(49,18),(49,49),(50,19),(50,50),(51,20),(51,51),(52,21),(52,52),(53,22),(53,53),(54,23),(54,54),(55,24),(55,55),(56,25),(56,56),(57,26),(57,57),(58,27),(58,58),(59,28),(59,59),(60,29),(60,60),
```

即出现在主对角线，以及左下的 31x31 的子矩阵的对角线，和右上的 31x31 的子矩阵的对角线。也就是

$$(0,0)\sim(60,60)$$

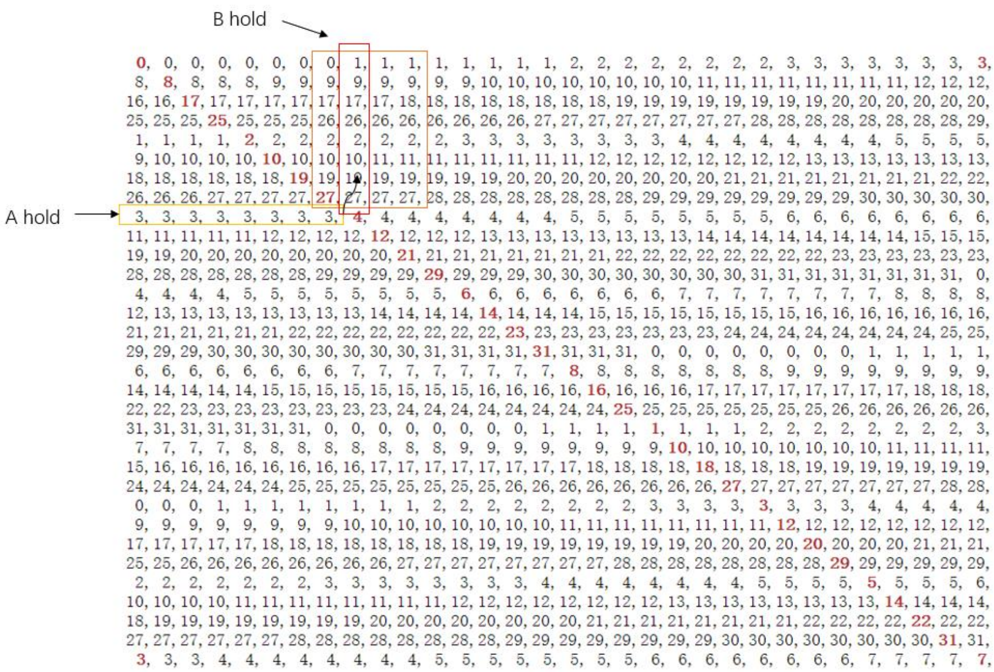
$$(0,31)\sim(29,60)$$

$$(31,0)\sim(60,29)$$

因为 61x67 的 size 不能被 8 整除，所以本矩阵的缓存下标不像 1 和 2 一样呈周期性分布，但是这也让同一列上的下标更不容易重复。结合对角线上的下标冲突，这里，对 56x64 的子矩阵进行 8x8 分块，在对块的每一行的扫描的最后处理对角线元素是一种较好的策略。在块间，使用行优先而非列优先遍历，可以减小块间滑动的抖动开销，这个结论如下图所示。

在完成对一个 8x8 子矩阵的扫描后， A 和 B 的缓存状态如图

此时 A 占用的缓存只有 2 块，B 则占用了 8 块。如果块间使用列优先，当前 B 则占用的 8 块缓存的剩余部分失效。为了充分利用 B 中尚未未写入的缓存，应该尽可能把这 8 块缓存写完再换出。为此，A hold 应该下移。



这样，我们的缓存损失是 A 的较少几块，而非 B 的 8 块，这就大大减小了缓存的未利用率。

对 56x64 的子矩阵以外的区域，考虑下半区的 5x64 矩阵，可以分解为 8 个 5x8 的矩阵块来处理；对右半区的 61x3 的矩阵，可以直接用简单的列优先的二重循环解决。

```
if (M == 61 && N == 67) { // unique trans function designed for 61x67 matrix.
        int i, j, x, y, tmp;

        // 56x64 submatrix
        for (y = 0;y < 64;y += 8) {
            for (x = 0;x < 56;x += 8) {
```

```c
                for (i = 0;i < 8;i++) {
                    for (j = 0;j < 8;j++) {
                        if (i == j) continue;
                        tmp = A[x+i][y+j];
                        B[y+j][x+i] = tmp;
                    }
                    tmp = A[x+i][y+i];
                    B[y+i][x+i] = tmp;
                }
            }
        }
        // foot 5x64 submatrix
        for (y = 0;y < 64;y+=8) {
            for (i = 56;i < 61;i++) {
                for (j=0;j<8;j++) {
                    tmp = A[i][y+j];
                    B[y+j][i] = tmp;
                }
            }
        }
        // right side 61x3 submatrix
        for (i = 0;i < 61;i++) {
            for (j = 64;j < 67;j++) {
                tmp = A[i][j];
                B[j][i] = tmp;
            }
        }
    }
```

经测试满足要求。到此，cache lab 完全结束。