Homework1

O(1) Push，Pop, FindMin

问题分析

Push Pop本身复杂度即O（1），故关键在于如何FindMin
我们可以设计一个包含两个栈的栈，DataStack存所有数据，minStack存最小数据，最小数据对应index放在栈顶

算法思路

- 入栈：
    - First Element: 第一个元素进入栈DataStack时候，其对应index 0 进入minStack
    - Other Elements X: 若X<DataStack[minStack[0]],则X的index入minStack栈
- 出栈
    - 当index of DataStack[0]==minStack[0],minStack[0]出栈，此时新栈顶为DataStack最小元素index
- FindMin
    - 取DataStack[minStack[0]]，通过调用peek函数获得

算法代码实现及优化

算法代码实现

java代码实现如下：

```
1
2  class MinStack {
3      private Stack<Integer> DataStack;
4      private Stack<Integer> minStack;
5      public MinStack() {
6          minStack = new Stack<>();
7          DataStack = new Stack<>(); //存储原始数据
8      }
9
10     public void push(int x) {
11         DataStack.push(x);
12         if (minStack.isEmpty() || x < minStack.peek()) {
13             minStack.push(x);
14             //栈空或元素更小时X入栈，若X==minStack.peek（），不更新
15         } else {
16             minStack.push(minStack.peek());
17         }
18     }
19
20     public void pop() {
```

```
        DataStack.pop();
        minStack.pop();
    }

    public int FindMin() {
        return minStack.peek();
        //返回minStack栈顶，即序列中最小元素
    }
}

```

优化方向

- 数据交换时可采用异或方式

最优情况O(N)BubbleSort

问题分析

最优情况即待排序列已经有序，若为O(N),即只需要一次遍历即可，故目的为优化BubbleSort,使其最优情况下只遍历一遍数组

算法思路

- 我们可以设计一个Flag，用以标记遍历过程是否发生交换。
- 初始Flag=false，遍历未排序部分，若存在需要交换的情况，更新Flag为true，并进行交换
- 若未发生交换，即序列已经有序，即可在遍历一次数组后跳出循环

算法代码实现及优化

算法代码实现

java代码实现如下,以int型数组为例：

```
    public static void BubbleSort(int[] Arr) {
    int n = Arr.length;
    for (int i = 0; i < n; i++) {
        boolean Flag = false;//初始标记为未发生交换
        for (int j = 0; j < n - i - 1; j++) {
            if (Arr[j] > Arr[j + 1]) {
                int temp = Arr[j];
                Arr[j] = Arr[j + 1];
                Arr[j + 1] = temp;
                Flag = true;
            }
        }
        if (!Flag) {
            break;// 未交换，说明已经有序，可跳出循环
        }
    }
}

```

优化方向

- 数据交换时可采用异或方式

Homework2

Calculate Greatest commom divisor

问题分析

求解最大公约数，既可以从质因数分解的角度出发，也可以从遍历数的角度出发，还可以从求模的角度出发等等

算法思路

本处罗列三种算法的处理思路

- 从因数的分解出发，即对两数分别进行因数分解，找出相同的因数，将其相乘即结果
- 从遍历数的角度出发，可以找到较小的数，并向下找，若均能整除，即gcd
- 从求模角度出发，即辗转相除法，直至余数为0，此时较小的数即gcd

算法设计与优化

算法设计

三种算法的代码如下：
因数分解：

```
    public static int Factorization(int a, int b) {
    int gcd = 1;
    int i = 2;
    while (i <= a && i <= b) {
        if (a % i == 0 && b % i == 0) {
            //进入if，说明i为a，b公共因子，故可乘入gcd
            gcd *= i;
            a /= i;
            b /= i;
        } else {
            i++;
        }
    }
    return gcd;
}
```

遍历数

```
   public static int Decreased(int a, int b) {
    int gcd = 1;
    int min = Math.min(a, b);
    for (int i = min; i > 0; i--) {
        if (a % i == 0 && b % i == 0) {
            gcd = i;
            break;
        }
    }
    return gcd;
}
```

辗转相除法

```
    public static int Euclidean(int a, int b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

算法优化

- 在数值较小时选用辗转相除法，避免大数除法的复杂运算处理

Find Max Online

问题分析

本题为动态查找最大值，故可采用最大堆的方式进行数据的存储，并在每一次插入删除操作时进行堆的调整

算法思路

- 最大值，一直为Data[1]处
- 管理最大堆，首先需要判断堆满或空，故设置了IsEmpty,IsFull函数
- 插入之前，需创造一个空堆，故设计了CreateMaxHeap函数
- 当在MaxHeap中插入一个key时，根据树的性质，进行下滤，找到对应位置，故对于下滤操作设置了PercDown函数
- 插入key时，设置了Insert函数，并在函数内部调用PercDown函数
- 将序列建造为最大堆，即进行多次插入，故设置了BuildMaxHeap函数，进行最大堆的创建
- 删除最大元素时，我们用堆中最后一个数据进行下滤，调整其位置，并处理堆的规模，故设置了DeleteMax函数

算法设计与优化

算法设计

算法设计如下所示

```
typedef struct HNode* Heap;
struct HNode{
    ElementType* Data;
    int Size;
    int Capacity;
};
typedef Heap MaxHeap;
#define MaxData 1001
#define MinData -1
#define Error -6

int IsEmpty(Heap H){
    return (H->Size == 0);
}
int IsFull(Heap H){
    return (H->Capacity == H->Size);
}

MaxHeap CreateMaxHeap(int MaxSize){
    MaxHeap H = (MaxHeap)malloc(sizeof(struct HNode));
    H->Data = (ElementType*)malloc((MaxSize + 1) * sizeof(ElementType));
    H->Size = 0;
    H->Data[0] = MaxData;
    H->Capacity = MaxSize;
    return H;
}

void InsertMaxHeap(MaxHeap H, ElementType X){
    //将元素X插入最大堆H，其中H->Data[0]已经定义为哨兵
    int i;
    if (!IsFull(H))
    {
        i = ++H->Size;
        for (; H->Data[i / 2] < X; i /= 2)
            H->Data[i] = H->Data[i / 2];
        H->Data[i] = X;
    }
}

void PercDownMaxHeap(MaxHeap H, int p){
    //下滤：将H中以H->Data[p]为根的子堆调整为最大堆
    int Parent, Child;
    ElementType X;
    X = H->Data[p];
    for (Parent = p; Parent * 2 <= H->Size; Parent = Child)
    {
        Child = Parent * 2;
```

```
        if ((Child != H->Size) && (H->Data[Child] < H->Data[Child + 1]))
            Child++;//Child指向左右子结点的较大者
        if (X >= H->Data[Child])
            break;
        else
            H->Data[Parent] = H->Data[Child];
    }
    H->Data[Parent] = X;
}

void BuildMaxHeap(MaxHeap H)//建造最大堆{
    调整H->Data[]中的元素，使满足最大堆的有序性
    假设所有H->Size个元素已经存在H->Data[]中
    int i;
    for (i = H->Size / 2; i > 0; i--)
        PercDownMaxHeap(H, i);
}

ElementType DeleteMax(MaxHeap H){
    //从最大堆H中取出键值为最大的元素，并删除一个结点
    int Parent, Child;
    ElementType MaxItem, X;
    if (!IsEmpty(H)){
        MaxItem = H->Data[1];//取出根结点存放的最大值
        X = H->Data[H->Size--];
        //用最大堆中最后一个元素从根结点开始向上过滤下层结点,当前堆的规模要减小
        for (Parent = 1; Parent * 2 <= H->Size; Parent = Child)
        {
            Child = Parent * 2;
            if ((Child != H->Size) && (H->Data[Child] < H->Data[Child + 1]))
                Child++;//Child指向左右子结点的较大者
            if (X >= H->Data[Child])
                break;//找到了合适位置
            else
                H->Data[Parent] = H->Data[Child];//下滤X
        }
        H->Data[Parent] = X;
        return MaxItem;
    }
    else
        return Error;
}

```

算法优化

- 可采用PriorityQueue，减少设计的手动维护函数

Homework3

Find All Subset

问题分析

从数学上，易知具有n个元素的集合，其子集个数共 2^n个，且不同子集之间存在包含关系，因而可使用两种方法

- 使用二进制码，1代表存在，0表示不存在，二进制数表示该元素在集合中
- 使用迭代法，从空集开始，把集合中的元素加入当前子集，再将子集加入之前生成的子集

由于任意算法其构造的子集均有2^n 个，故本问题的时间复杂度至少为2^n

算法分析与思路

- 二进制码法：
    - 采用二重循环的方式
    - 外层循环遍历所有二进制码，内层循环根据二进制码生成集合
    - 因需遍历所有二进制码，并根据其二进制码生成对应集合，故其时间复杂度为O(n*2^n)
- 迭代法：
    - 采用二重循环的方式
    - 外层循环为遍历原集合的所有元素，内层循环遍历已生成的所有子集
    - 因需遍历所有元素，及当前生成的所有子集，故时间复杂度为O(n*2^n)

算法实现与选择

算法实现

- 二进制法的代码实现如下：

```
public class SubsetBinary {
   public static List<List<Integer>> subsets(int[] nums) {
      List<List<Integer>> result = new ArrayList<>();
      int n = nums.length;
      for (int i = 0; i < (1 << n); i++) {
          //考虑第0到第2^n-1 个集合
          List<Integer> subset = new ArrayList<>();
          for (int j = 0; j < n; j++) {
              if ((i & (1 << j)) != 0) {
                  // 如果i的第j位为1，则把集合中第j个元素添加到当前子集
                  subset.add(nums[j]);
              }
          }
          // 把当前创建的子集加入结果中
          result.add(subset);
      }
```

```
        return result;
    }
}
```

- 迭代法的代码实现如下：

```
    public class SubsetIteration {
        public static List<List<Integer>> subsets(int[] nums) {
            List<List<Integer>> subsets = new ArrayList<>();
            subsets.add(new ArrayList<>()); // 添加空集
            for (int num : nums) {
                //对于原集合的每个元素
                int size = subsets.size(); // 获取当前子集的数量
                for (int i = 0; i < size; i++) {
                //对于目前已经创建的所有子集
                List<Integer> subset = new ArrayList<>(subsets.get(i));
                subset.add(num); // 将当前数添加到子集中
                subsets.add(subset); // 添加新的子集
            }
         }
         return subsets;
     }
}
```

算法选择

- 集合较大时，因迭代法的空间复杂度为O(2^n),二进制法的空间复杂度为O(1),故应该选择二进制法
- 当集合较小时，两种方法差异不明显

Application of similarity index

定义

数学意义上的相似度指标是一种用于衡量两个或多个对象之间相似程度的指标。在计算相似度时，通常将每个对象表示为数学空间中的一个向量，然后计算这些向量之间的距离或夹角等度量指标，以判断它们之间的相似度。

常见应用

1. 余弦相似度 (Cosine Similarity):

$$\text{Cosine Similarity}\,(x,y)=\frac{x\cdot y}{\|x\|\|y\|} \tag{1}$$

其中， x 和 y是两个向量，$x\cdot y$ 表示向量内积， $\|x\|$表示向量 x 的范数。

2. 欧几里得距离 (Euclidean Distance) :

$$\text{Euclidean Distance}\,(x,y)=\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2} \tag{2}$$

其中， x 和 y 是两个向量， n 是向量的维度。

3. 曼哈顿距离 (Manhattan Distance) :

$$\text{Manhattan Distance } (x, y) = \sum_{i=1}^{n} |x_i - y_i| \tag{3}$$

其中， x 和 y 是两个向量， n 是向量的维度。

4. 皮尔逊相关系数 (Pearson Correlation Coefficient) :

$$\text{Pearson Correlation Coefficient}(x, y) = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^{n} (y_i - \bar{y})^2}} \tag{4}$$

其中， x 和 y 是两个向量， $\bar{x}$ 和 $\bar{y}$ 分别表示向量 x 和 y 的均值。

5. Jaccard相似系数 (Jaccard Similarity Coefficient)

$$\text{Jaccard Similarity Coefficient } (A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{5}$$

其中， A 和 B 是两个集合。

6. 信息熵 (Information Entropy) :

$$\text{Entropy}(x) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{6}$$

其中， X 是一个随机变量， p_i 是 X 取值为 i 的概率。

Homework4

CompareTwoFindWay

有序序列

最坏情况下二分查找的时间复杂度为 $O(logn)$,而采用顺序查找的方法时间复杂度为O(n),故有序情况下二分查找效率更高

无序序列

查找一次

使用二分查找需先进行排序，排序操作额外的时间复杂度为$O(nlogn)$,故此时二分查找的时间复杂度为$O(nlogn)$，而采用顺序查找的时间复杂度为O(n)，故此时使用顺序查找效率更高

查找多次

当需要查找多次时，二分查找在经过一次排序后，查找的时间复杂度为$O(logn)$，而每次顺序查找的时间复杂度均为O（n）,故当搜索次数较大时，排序所耗费的代价可以接受，此时更应当选择二分查找

无序例子

单次查找

当无序数组的长度为1000000

- 顺序查找：最坏情况下，需要查找的元素在数组的最后一位，需要进行1000000次比较才能找到该元素，因此基本操作次数为1000000。
- 二分查找：首先需要对数组进行排序，假设采用快速排序算法，则时间复杂度为O(nlogn) ≈ 20000000。排序完成后，采用二分查找，最坏情况下，需要查找的元素不在数组中，此时二分查找会执行log2(1000000) + 1 ≈ 21次比较，因此基本操作次数为20000000 + 21 ≈ 20000021。可见此情况下顺序查找的基本操作次数远少于二分查找，故此时应选择顺序查找

多次查找

当无序数组的长度为1000000，查找次数为100000时

- 顺序查找：一次顺序查找的基本操作次数为1000000，共查找1000次，故基本操作次数为 $1000000 * 100000 = 10^{11}$
- 二分查找：首先需要对数组进行排序，假设采用快速排序算法，则时间复杂度为O(nlogn) ≈ 20000000。排序完成后，采用二分查找，最坏情况下，需要查找的元素不在数组中，此时二分查找会执行log2(1000000) + 1 ≈ 21次比较，因此基本操作次数为20000000 + 21*100000 ≈ 22100000= $2.21 * 10^7$
- 由于$2.21 * 10^7 << 10^{11}$,可见此情况下二分查找的基本操作次数远少于顺序查找，故此时应选用二分查找

FindItem

问题分析

由于序列本身是有序的，绕元素旋转后序列整体尽管无序，但在其内可划分成两个有序的子序列。故实施查找仍可采取二分的思路，并可基于序列特点采取改进的二分查找算法

算法思路分析

- 旋转后的序列可分为两个有序的子序列，称为LeftSub和RightSub,其中LeftSub的元素均大于RightSub的元素
- 然后递归判断中间元素和最后一个元素的关系
    - 如果中间元素大于最后一个元素，则说明中间元素在LeftSub中，RightSub必然满足LeftSub所有元素的大小关系，因此应该在LeftSub中继续查找。
    - 如果中间元素小于最后一个元素，则说明中间元素在RightSub中，LeftSub必然满足LeftSub所有元素的大小关系，因此应该在RightSub中继续查找。
    - 当找到时，返回对应索引

算法实现

具体递归实现代码如下：

```
public static int searchRotatedArray(int[] nums, int target) {
    return search(nums, 0, nums.length - 1, target);
}
private static int search(int[] nums, int left, int right, int target) {
    if (left > right) {
        return -1;
    }
    int mid = left + (right - left) / 2;
    if (nums[mid] == target) {
        return mid;
    }
    if (nums[mid] >= nums[left]) {
        if (target >= nums[left] && target < nums[mid]) {
            return search(nums, left, mid - 1, target);
        } else {
            return search(nums, mid + 1, right, target);
        }
    } else {
        if (target > nums[mid] && target <= nums[right]) {
            return search(nums, mid + 1, right, target);
        } else {
            return search(nums, left, mid - 1, target);
        }
    }
}
```

显然，因采用二分查找的策略，时间复杂度均为$O(logn)$，空间复杂度均为$O(1)$

算法优化

- 如果数组中有重复元素，可以考虑在二分查找中添加一些额外的判断，例如当中间值与左边界和右边界相等时，需要特殊处理
- 二分查找中可以使用位运算替代除法运算，提升代码效率
- 可以通过使用位运算和三元运算符等技巧来尽量减少 if 判断

基于上述优化思路，优化代码如下：

```
public class RotatedSortedArraySearch {
    public int search(int[] nums, int target) {
        return search(nums, target, 0, nums.length - 1);
    }
    private int search(int[] nums, int target, int left, int right) {
        if (left > right) {
            // 没有找到目标值，返回 -1
            return -1;
        }
        // 计算中间位置
        int mid = (left + right) >>> 1;
        if (nums[mid] == target) {
            // 如果中间位置的值就是目标值，直接返回中间位置的下标
            return mid;
        }
        // 判断目标值可能存在的区间，同时考虑到相等的情况
        boolean leftHalf = (nums[mid] >= nums[left]) == (target >=
nums[left]);
        if (leftHalf) {
            // 如果目标值可能存在于左半边，继续在左半边递归查找
            return search(nums, target, left, mid - 1);
        } else {
            // 否则，目标值可能存在于右半边，继续在右半边递归查找
            return search(nums, target, mid + 1, right);
        }
    }
}
```

Homework5

Find Top-k

问题分析

找出序列中前k个大的元素，常见的有直接排序，最小堆法，桶排序法

算法思路

- 直接排序：将整个数组进行排序，然后取前k个元素即可，时间复杂度为O(nlogn)
- 最小堆法：将整个序列插入到大小为k的最小堆里，最终全部pop，时间复杂度为O(nlogk)
- 桶排序：将数据分到n个桶里，然后按递减顺序遍历桶即可得

算法实现

- 直接排序代码如下：

```
public static int[] topK(int[] nums, int k) {
    Arrays.sort(nums);
    return Arrays.copyOfRange(nums, nums.length - k, nums.length);
}

```

- 最小堆法代码如下：

```
public static int[] topK(int[] nums, int k) {
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    for (int num : nums) {
        minHeap.push(num);
        if (minHeap.size() > k) {
            minHeap.pop();
        }
    }
    var result = new int[k];
    for (int i = 0; i < k; i++) {
        result[i] = minHeap.pop();
    }
    return result;
}



```

- 桶排序代码如下：

```

public static List<Integer> bucketSort(int[] nums, int k) {
```

```
    List<Integer>[] buckets = new List[nums.length + 1];
    for (int i = 0; i < buckets.length; i++) {
        buckets[i] = new ArrayList<>();
    }
    for (int num : nums) {
        buckets[num].add(num);
    }
    List<Integer> result = new ArrayList<>();
    for (int i = buckets.length - 1; i >= 0 && result.size() < k; i--) {
        result.addAll(buckets[i]);
    }
    return result;
}


```

对比分析

- 从空间复杂度来看，堆排序和直接排序的空间复杂度都是 O(1)，而桶排序的空间复杂度是 O(n+k)，需要开辟额外的桶空间，因而在空间较小且数据量较大时，应谨慎选择桶排序
- 从时间复杂度来看，最小堆法是O(nlogk)，直接排序法是 O(nlogn) ，而桶排序是 O(n+k) 的， 因而数据量庞大时候，更倾向于选择桶排序法

递归堆排序

算法实现

具体实现代码如下：

```
public static void HeapSort(int[] nums) {
    if (nums == null || nums.length <= 1) {
        return;
    }
    // 构建最大堆
    BuildMaxHeap(nums, nums.length);
    // 重新调整堆
    for (int i = nums.length - 1; i >= 1; i--) {
        Swap(nums, 0, i);
        MaxHeapify(nums, 0, i);
    }
}

private static void BuildMaxHeap(int[] nums, int heapSize) {
    // 从最后一个非叶节点开始向上调整,直至到顶，即构建好堆
    for (int i = heapSize / 2 - 1; i >= 0; i--) {
        MaxHeapify(nums, i, heapSize);
    }
}

private static void MaxHeapify(int[] nums, int i, int heapSize) {
    int left = 2 * i + 1;
    int right = 2 * i + 2;
```

```
    int largest = i;
    if (left < heapSize && nums[left] > nums[largest]) {
        largest = left;
    }
    if (right < heapSize && nums[right] > nums[largest]) {
        largest = right;
    }
    if (largest != i) {
        Swap(nums, i, largest);
        MaxHeapify(nums, largest, heapSize);
    }
}

private static void Swap(int[] nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}

```

算法分析

- 递归法的堆排序实际上就是在原始的堆排序算法的基础上加了递归过程，每次对左右子树进行递归调用。因此它的时间复杂度也是 O(nlogn)。
- 因为它需要在递归调用时创建一些额外的栈空间来保存递归调用的函数返回地址和局部变量，故其空间复杂度相比原始堆排序高。它的空间复杂度为 O(logn)，其中 logn 是递归调用的深度

Homework6

Voronoi 图

基本概念

Voronoi图（维诺图）又叫泰森多边形或者狄利克雷图。它由n个选定点和个区域构成。它将平面分成n个部分，每个部分满足如下的条件

- 每个区域仅一个选定点
- 区域内任意点，该点到该区域内选定点的欧氏距离恒小于该点到其他选定点
- 对于区域边界上的点，到该边界确定的几个相邻区域的选定点距离相等

构造方法

- 通过一些算法（如Delaunay三角剖分）计算出一组点的Voronoi图，其中每个点对应一个Voronoi cell（也称为面）和一些Voronoi边（也称为线段）。
- 对于每个Voronoi cell，需要计算其边界上的点，这些点是Voronoi图中的特殊点，也称为Voronoi顶点。这可以通过找到与该cell关联的Voronoi边，以及它们之间的交点来完成。
- 对于每个Voronoi边，需要确定其两个端点。这可以通过找到该边所关联的两个Voronoi cell，并找到它们之间的垂线交点来完成。
- 根据Voronoi图的定义，图中每个cell都由一组Voronoi边围成，因此需要为每个cell建立一个边集合，其中包含所有关联该cell的Voronoi边。
- 最后，将计算得到的Voronoi边和Voronoi cell保存到数据结构中，以便后续的应用

Delaunay三角网生成思路

- 构造一个超级三角形，包含所有散点，放入三角形链表
- 在三角形链表中找出其外接圆包含插入点的三角形，删除该三角形的公共边，将插入点同该三角形的全部顶点连接起来
- 根据优化准则对局部新形成的三角形进行优化。将形成的三角形放入Delaunay三角形链表。
- 循环执行，直到所有散点插完

算法应用

- 几何建模和计算机图形学：Voronoi 图广泛用于计算机图形学中，用于对地形、云和树木等自然现象进行建模，以及构建用于有限元分析的网格。
- 模式识别和图像处理：Voronoi 图可用于检测图像中的模式以及根据像素的强度或颜色将图像分割成区域。
- 优化与运筹学：Voronoi图可用于解决最优化问题，如最近邻问题、设施选址问题、旅行商问题。
- 空间分析和地理学：Voronoi 图用于分析空间数据并模拟对象或区域之间的空间关系，例如土地利用模式、城市增长和交通网络。
- 分子建模和化学：Voronoi 图用于对分子建模并计算它们的属性，例如表面积、体积和可及性。它们还用于药物设计和蛋白质结构预测

二维矩阵查找

问题思路

- 在二维矩阵中查找特定元素一维矩阵类似，同样可以采取暴力查找、二分查找等思路.
- 基于矩阵本身升序特点，故还可采用分治查找和Z字型查找的方法

算法设计

- 暴力查找：
    - 遍历整个矩阵，时间复杂度为O(nm)。这种方法的缺点是时间复杂度较高
- 二分查找：
    - 与一维矩阵类似，可以将每行看成一个有序数组，分别对每行进行二分查找
    - 时间复杂度为O(mlog n)，其中
- 分治查找
    - 在升序方阵中查找特定元素时，可以将方阵分成四个部分，分别递归查找目标元素所在的部分。时间复杂度为O(log^2n)
- Z字型查找
    - 从矩阵的右上角 (0,n−1)进行搜索。在每一步的搜索过程中，如果我们位于位置 (x,y)，那么我们希望在以 matrix的左下角为左下角、以 (x,y)为右上角的矩阵中进行搜索，即行的范围为 [x,m−1]，列的范围为 [0,y]
    - 在搜索的过程中，如果我们没有找到 target，那么我们要么将 y 减少 1，要么将 x 增加 1。由于 (x,y)的初始值分别为 (0,n−1)，因此 y 最多能被减少 n 次，x 最多能被增加 m 次，总搜索次数为 m+n,即时间复杂度为O(m+n)

分治时间复杂度证明

- 矩阵的大小为n * n，则在每一层递归中，矩阵的大小都会减半。因此，递归树的深度为logn。在每一层递归中，需要对一个长度为n的一维数组进行二分查找，时间复杂度为O(logn),因而合起来即O(log^2n)
    - 第0层：矩阵大小为n * n，需要对一个长度为n的一维数组进行二分查找，时间复杂度为O(logn)。
    - 第1层：矩阵大小为n/2 * n/2，需要对一个长度为n/2的一维数组进行二分查找，时间复杂度为O(log(n/2)) = O(logn)。
    - 第2层：矩阵大小为n/4 * n/4，需要对一个长度为n/4的一维数组进行二分查找，时间复杂度为O(log(n/4)) = O(logn)。

      ...
    - 第k层：矩阵大小为n/2^k * n/2^k，需要对一个长度为n/2k的一维数组进行二分查找，时间复杂度为O($log(n/2^k)$) = O(logn)
- 每一层递归的时间复杂度均为O(logn)，递归树的深度为logn，因此分治查找的时间复杂度为O(log^2n)

代码实现

```java
// 暴力搜索
public static boolean searchMatrix(int[][] matrix, int target) {
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[0].length; j++) {
            if (matrix[i][j] == target) {
                return true;
            }
        }
    }
    return false;
}

```

```java
//二分查找
public static boolean searchMatrix(int[][] matrix, int target) {
    for (int i = 0; i < matrix.length; i++) {
        int left = 0, right = matrix[i].length - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (matrix[i][mid] == target) {
                return true;
            } else if (matrix[i][mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }
    return false;
}

```

```java
//分治查找
public class DivideAndConquerSearch {
    public static boolean search(int[][] matrix, int target) {
        int rows = matrix.length;
        int cols = matrix[0].length;
        return search(matrix, target, 0, rows - 1, 0, cols - 1);
    }

    private static boolean search(int[][] matrix, int target, int rowStart, int rowEnd, int colStart, int colEnd) {
        if (rowStart > rowEnd || colStart > colEnd) {
            return false;
        }

        int midRow = (rowStart + rowEnd) / 2;
        int midCol = (colStart + colEnd) / 2;

        if (matrix[midRow][midCol] == target) {
            return true;
```

```
        } else if (matrix[midRow][midCol] > target) {
            return search(matrix, target, rowStart, midRow - 1, colStart, colEnd)
                    || search(matrix, target, midRow, rowEnd, colStart, midCol - 1);
        } else {
            return search(matrix, target, rowStart, midRow, midCol + 1, colEnd)
                    || search(matrix, target, midRow + 1, rowEnd, colStart, colEnd);
        }
    }
}

```

```
//Z字型查找
public boolean searchMatrix(int[][] matrix, int target) {
        int m = matrix.length, n = matrix[0].length;
        int x = 0, y = n - 1;
        while (x < m && y >= 0) {
            if (matrix[x][y] == target) {
                return true;
            }
            if (matrix[x][y] > target) {
                --y;
            } else {
                ++x;
            }
        }
        return false;
    }
```

Homework 8

Print Matrix

问题分析

因需要螺旋向内打印矩阵，故需要考虑矩阵的边界问题，即矩阵的行数和列数，以及矩阵的起始位置。

算法思路

- 将矩阵分为若干层，从外向内打印矩阵
- 对于每一层，分别沿着上、右、下、左四个方向打印矩阵
- 打印完该层后，将矩阵的行数和列数减去2，同时将矩阵的起始位置向内移动一行一列

代码实现

给出递归版实现和非递归版实现

```
//非递归版本
public class PrintMatrix {
    public static void printMatrix(int[][] matrix){
        if(matrix==null)
            return;
        int rows=matrix.length;
        int cols=matrix[0].length;
        int start=0;
```

```
        while(rows>start*2&&cols>start*2){
            printMatrixInCircle(matrix,rows,cols,start);
            start++;
        }
    }
    public static void printMatrixInCircle(int[][] matrix,int rows,int cols,int start){
        int endX=cols-1-start;
        int endY=rows-1-start;
        //从左到右打印一行
        for(int i=start;i<=endX;i++){
            System.out.print(matrix[start][i]+" ");
        }
        //从上到下打印一列
        if(start<endY){
            for(int i=start+1;i<=endY;i++){
                System.out.print(matrix[i][endX]+" ");
            }
        }
        //从右到左打印一行
        if(start<endX&&start<endY){
            for(int i=endX-1;i>=start;i--){
                System.out.print(matrix[endY][i]+" ");
            }
        }
        //从下到上打印一列
        if(start<endX&&start<endY-1){
            for(int i=endY-1;i>=start+1;i--){
                System.out.print(matrix[i][start]+" ");
            }
        }
    }
}
```

```
//递归版本
public class PrintMatrix {
    public static void printMatrix(int[][] matrix){
        if(matrix==null)
            return;
        int rows=matrix.length;
        int cols=matrix[0].length;
        int start=0;
        printMatrix(matrix,rows,cols,start);
    }
    public static void printMatrix(int[][] matrix,int rows,int cols,int start){
        if(rows<=start*2||cols<=start*2)
            return;
        int endX=cols-1-start;
        int endY=rows-1-start;
        //从左到右打印一行
        for(int i=start;i<=endX;i++){
```

```
            System.out.print(matrix[start][i]+" ");
        }
        //从上到下打印一列
        if(start<endY){
            for(int i=start+1;i<=endY;i++){
                System.out.print(matrix[i][endX]+" ");
            }
        }
        //从右到左打印一行
        if(start<endX&&start<endY){
            for(int i=endX-1;i>=start;i--){
                System.out.print(matrix[endY][i]+" ");
            }
        }
        //从下到上打印一列
        if(start<endX&&start<endY-1){
            for(int i=endY-1;i>=start+1;i--){
                System.out.print(matrix[i][start]+" ");
            }
        }
        printMatrix(matrix,rows,cols,start+1);
    }
}
```

- 以上两种算法，时间复杂度均为O(mn)
- 非递归版本的空间复杂度为O(1)，递归版本的空间复杂度为O(mn)，因为递归版本每次调用都会创建新的栈帧，栈帧的大小取决于函数的参数和局部变量的大小，因而递归版本的空间复杂度较高

矩阵旋转

问题分析

- 矩阵旋转，即将矩阵中的元素按照顺时针方向旋转90度，即将矩阵中的元素按照对角线进行翻转，然后将每一行的元素进行翻转。
- 对于矩阵中的第 i 行的第 j 个元素，在旋转后，它出现在倒数第 i 列的第 j 个位置，即它的新位置为 (j, n-1-i)

算法思路

- 若使用辅助数组，根据旋转规则，将矩阵中的元素复制到辅助数组中，然后将辅助数组中的元素复制到原数组中，时间复杂度为O(mn)，空间复杂度为O(mn)
- 原地旋转，即不使用辅助数组，直接在原数组上进行旋转，时间复杂度为O(mn)，空间复杂度为O(1)
- 翻转代替旋转，即先将矩阵中的元素按照对角线进行翻转，然后将每一行的元素进行翻转，时间复杂度为O(mn)，空间复杂度为O(1)

代码实现

```java
//辅助数组
public class RotateMatrix {
    public static void rotateMatrix(int[][] matrix){
        if(matrix==null)
            return;
        int rows=matrix.length;
        int cols=matrix[0].length;
        int[][] newMatrix=new int[cols][rows];
        for(int i=0;i<rows;i++){
            for(int j=0;j<cols;j++){
                newMatrix[j][rows-1-i]=matrix[i][j];
            }
        }
        for(int i=0;i<cols;i++){
            for(int j=0;j<rows;j++){
                matrix[i][j]=newMatrix[i][j];
            }
        }
    }
}
```

```java
//原地旋转
public class RotateMatrix {
    public static void rotateMatrix(int[][] matrix){
        if(matrix==null)
            return;
        int rows=matrix.length;
        int cols=matrix[0].length;
        //先将矩阵中的元素按照对角线进行翻转
        for(int i=0;i<rows;i++){
            for(int j=i+1;j<cols;j++){
                int temp=matrix[i][j];
                matrix[i][j]=matrix[j][i];
                matrix[j][i]=temp;
            }
        }
        //再将每一行的元素进行翻转
        for(int i=0;i<rows;i++){
            for(int j=0;j<cols/2;j++){
                int temp=matrix[i][j];
                matrix[i][j]=matrix[i][cols-1-j];
                matrix[i][cols-1-j]=temp;
            }
        }
    }
}
```

```java
//翻转代替旋转
public class RotateMatrix {
```

```
    public static void rotateMatrix(int[][] matrix){
        if(matrix==null)
            return;
        int rows=matrix.length;
        int cols=matrix[0].length;
        //先将矩阵中的元素按照对角线进行翻转
        for(int i=0;i<rows;i++){
            for(int j=i+1;j<cols;j++){
                int temp=matrix[i][j];
                matrix[i][j]=matrix[j][i];
                matrix[j][i]=temp;
            }
        }
        //再将每一行的元素进行翻转
        for(int i=0;i<rows;i++){
            for(int j=0;j<cols/2;j++){
                int temp=matrix[i][j];
                matrix[i][j]=matrix[i][cols-1-j];
                matrix[i][cols-1-j]=temp;
            }
        }
    }
}
```

- 若需降低空间复杂度，方法三可使用异或操作，可省去临时变量的空间

之字形打印矩阵

问题分析

- 之字形打印矩阵，即从左上角开始，按照之字形的顺序打印矩阵中的元素，即先从左上角开始，向右打印一行，然后向下打印一列，再向左打印一行，如此循环，直至打印完所有的元素。

算法思路

- 使用两个变量row和col来追踪当前位置，并使用一个布尔值变量up来表示方向。如果up为true，则从左下向右上打印，否则从右上向左下打印
- 对于每个方向，我们沿着对角线移动，直到到达边界。我们需要处理边界情况，例如当到达右上角或左下角时，需要调整row和col的值以确保打印正确。
- 该算法的时间复杂度是O(M*N)，其中M和N分别是矩阵的行数和列数。它的空间复杂度是O(1)

代码实现

```
public static void printMatrixZigZag(int[][] matrix) {
    int m = matrix.length;
    int n = matrix[0].length;
    int row = 0, col = 0;
    boolean up = true;

    while (row < m && col < n) {
        if (up) {
            // 从左下向右上打印
```

```
            while (row >= 0 && col < n) {
                System.out.print(matrix[row][col] + " ");
                row--;
                col++;
            }
            if (row < 0 && col < n) {
                row = 0;
            } else if (col == n) {
                row += 2;
                col--;
            }
        } else {
            // 从右上向左下打印
            while (row < m && col >= 0) {
                System.out.print(matrix[row][col] + " ");
                row++;
                col--;
            }
            if (col < 0 && row < m) {
                col = 0;
            } else if (row == m) {
                col += 2;
                row--;
            }
        }
        up = !up;
    }
}

```

最小路径

问题分析

- 给定一个二维数组，每个元素代表从左上角到该位置的最小路径和，求从左上角到右下角的最小路径和

算法思路

- 从暴力的角度，我们可以使用递归的方法，从左上角开始，向右或向下移动，直到到达右下角，然后返回路径和。但是这种方法的时间复杂度为O(2^(m+n))，空间复杂度为O(m+n)，时间复杂度高，不适用于大型矩阵
- 从贪心的角度，从左上角出发，每次选择向右或向下移动一格，使得移动后的格子的权值最小。该算法的时间复杂度是O(m+n)，空间复杂度为O(1),其中m和n分别是矩阵的行数和列数，因此比暴力枚举法快得多。然而，并不能保证总是能找到最优解，因此该算法并不适用于所有情况
- 从分治的角度，将矩阵划分成四个部分，分别计算从左上角到右下角的最小数字和。如果只有一个格子，则直接返回该格子的数字。如果只有一行或一列，则计算这一行或这一列的数字和。否则，分别计算右下部分、下方部分和右侧部分的最小数字和。最后，返回从左上角到右下角的最小数字和
- 从动态规划的角度，我们可以使用一个二维数组dp来存储从左上角到右下角的最小数字和。dp[i][j]表示从左上角到(i,j)位置的最小数字和。我们可以使用递推式dp[i][j]=min(dp[i-1][j],dp[i][j-

1])+matrix[i][j]来计算dp[i][j]的值。最后，返回dp[m-1][n-1]即可。该算法的时间复杂度是O(mn)，空间复杂度是O(mn)

代码实现

```java
//暴力枚举法
public static int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    return dfs(grid, 0, 0, m, n);
}

public static int dfs(int[][] grid, int i, int j, int m, int n) {
    if (i == m - 1 && j == n - 1) {
        return grid[i][j];
    }
    if (i == m - 1) {
        return grid[i][j] + dfs(grid, i, j + 1, m, n);
    }
    if (j == n - 1) {
        return grid[i][j] + dfs(grid, i + 1, j, m, n);
    }
    int rightSum = dfs(grid, i, j + 1, m, n);
    int downSum = dfs(grid, i + 1, j, m, n);
    return grid[i][j] + Math.min(rightSum, downSum);
}
```

```java
//贪心算法
public static int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    int row = 0, col = 0;
    int sum = 0;
    while (row < m && col < n) {
        if (row == m - 1) {
            sum += grid[row][col];
            col++;
        } else if (col == n - 1) {
            sum += grid[row][col];
            row++;
        } else {
            if (grid[row][col + 1] < grid[row + 1][col]) {
                sum += grid[row][col];
                col++;
            } else {
                sum += grid[row][col];
                row++;
            }
        }
    }
    return sum;
}
```

```java
//分治算法
public static int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    return divideAndConquer(grid, 0, 0, m - 1, n - 1);
}

public static int divideAndConquer(int[][] grid, int startX, int startY, int endX, int endY) {
    if (startX > endX || startY > endY) {
        return Integer.MAX_VALUE;
    }
    if (startX == endX && startY == endY) {
        return grid[startX][startY];
    }
    int midX = startX + (endX - startX) / 2;
    int midY = startY + (endY - startY) / 2;
    int rightSum = divideAndConquer(grid, startX, midY + 1, endX, endY);
    int downSum = divideAndConquer(grid, midX + 1, startY, endX, midY);
    int minSum = Math.min(rightSum, downSum);
    if (midX < endX && midY < endY) {
        int diagonalSum = divideAndConquer(grid, midX + 1, midY + 1, endX, endY);
        minSum = Math.min(minSum, diagonalSum);
    }
    return grid[startX][startY] + minSum;
}
```

```java
//动态规划
public static List<int[]> minSumPath(int[][] matrix) {
    int m = matrix.length;
    int n = matrix[0].length;

    int[][] dp = new int[m][n];
    dp[0][0] = matrix[0][0];

    // 第一列只能从上面一格过来
    for (int i = 1; i < m; i++) {
        dp[i][0] = dp[i - 1][0] + matrix[i][0];
    }

    // 第一行只能从左边一格过来
    for (int j = 1; j < n; j++) {
        dp[0][j] = dp[0][j - 1] + matrix[0][j];
    }

    // 填充剩下的格子，每个格子只能从左边或上面过来
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = matrix[i][j] + Math.min(dp[i - 1][j], dp[i][j - 1]);
        }
```

```
    }

    // 反向遍历dp数组，找出最小路径
    List<int[]> path = new ArrayList<>();
    int i = m - 1, j = n - 1;
    while (i > 0 || j > 0) {
        path.add(new int[]{i, j});
        if (i == 0) {
            j--;
        } else if (j == 0) {
            i--;
        } else if (dp[i - 1][j] < dp[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }
    path.add(new int[]{0, 0});
    Collections.reverse(path);

    return path;
}
```

- 四种算法中，空间复杂度最优的是贪心算法，因为它不需要额外的存储空间。动态规划和暴力枚举法的空间复杂度相同，都为 O(mn)。分治算法的空间复杂度也为 O(mn)，但由于递归调用，还需要额外的栈空间

Homework 9

Binary Search and Fibonacci search

定义

- 折半查找，也称二分查找，是一种非常高效的查找算法，适用于有序数组的查找。它的思想是每次将查找区间折半，缩小查找范围，直到找到目标元素或者区间为空。折半查找的时间复杂度为O(log n)，空间复杂度为O(1)，是一种非常高效的查找算法
- 斐波那契查找，是一种基于斐波那契数列的查找算法。它的思想是将数组分成若干个斐波那契数列的长度，并将查找点分别作为起点和终点，然后通过对比目标元素和数组中间点的大小，不断缩小查找范围，直到找到目标元素或者区间为空。斐波那契查找的时间复杂度为O(log n)，空间复杂度为O(1)。

实现思路

- 折半查找：
    - 定义数组范围：确定在哪个范围内进行查找。
    - 求出中间位置：将左右范围之和除以二即可。
    - 判断目标值与中间位置的大小关系：如果目标值比中间位置的值大，则在右半部分继续查找；如果目标值比中间位置的值小，则在左半部分继续查找；如果相等，则直接返回中间位置的下标。
    - 更新查找范围：根据判断结果，更新查找范围，继续执行步骤2-3，直到找到目标值或范围缩小为0
- 斐波那契查找：
    - 定义斐波那契数列：将数组的长度与斐波那契数列进行比较，找到大于等于数组长度的第一个斐波那契数。

- 将数组长度扩展到斐波那契数列对应的长度，将扩展的部分用最后一个元素进行填充。
- 定义三个指针：left指向数组的第一个元素，right指向数组的最后一个元素，mid指向斐波那契数列中第k个元素。
- 将要查找的目标值与mid位置的元素进行比较：如果目标值比mid位置的元素大，则在右半部分继续查找；如果目标值比mid位置的元素小，则在左半部分继续查找；如果相等，则直接返回mid位置的下标。
- 更新left、right、mid的值：根据查找结果，更新left、right、mid的值，继续执行步骤4，直到找到目标值或范围缩小为0

实现代码

```java
// 折半查找
public class BinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int left = 0;
        int right = arr.length - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (arr[mid] == target) {
                return mid;
            } else if (arr[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1;
    }
}
```

```java
// 斐波那契查找
public class FibonacciSearch {
    public static int fibonacciSearch(int[] arr, int target) {
        int left = 0;
        int right = arr.length - 1;
        int k = 0;
        int mid = 0;
        int[] f = fibonacci();
        while (right > f[k] - 1) {
            k++;
        }
        int[] temp = Arrays.copyOf(arr, f[k]);
        for (int i = right + 1; i < temp.length; i++) {
            temp[i] = arr[right];
        }
        while (left <= right) {
            mid = left + f[k - 1] - 1;
            if (target < temp[mid]) {
                right = mid - 1;
                k--;
            } else if (target > temp[mid]) {
```

```
                left = mid + 1;
                k -= 2;
            } else {
                if (mid <= right) {
                    return mid;
                } else {
                    return right;
                }
            }
        }
        return -1;
    }

    public static int[] fibonacci() {
        int[] f = new int[20];
        f[0] = 1;
        f[1] = 1;
        for (int i = 2; i < f.length; i++) {
            f[i] = f[i - 1] + f[i - 2];
        }
        return f;
    }
}
```

性能比较

- 折半查找的时间复杂度为O(log n)，空间复杂度为O(1)。斐波那契查找的时间复杂度为O(log n)，空间复杂度为O(1)。
- 二分查找的优点是实现简单、代码容易理解，但只适用于有序数组，如果数组未排序，则需要先排序，时间复杂度为O(n log n)。斐波那契查找的优点是不需要先排序，但是需要额外的空间来存储斐波那契数列，空间复杂度为O(n)。
- 空间较小时，折半查找的性能优于斐波那契查找。空间较大时，斐波那契查找的性能优于折半查找。
- 当数组长度较大时，斐波那契查找的性能优于折半查找。

实际应用

- 二分查找：
    - 排序算法：在排序算法中，二分查找可以用于查找某个元素在已排序的数组中的位置，从而实现快速排序等算法。
    - 数据库查询：在大型的数据库中，二分查找可以用于高效的查找操作，可以帮助我们快速查询到需要的信息。
    - 网络寻址：在网络中，二分查找可以用于查找IP地址等信息，帮助我们快速定位到目标位置。
    - 代码查找：在编写代码时，二分查找可以用于快速查找代码库中某个特定的函数或者变量。
    - 数学计算：在数学计算中，二分查找可以用于计算一些复杂函数的零点或者极值点，从而实现数值计算等操作。
- Fibonacci查找

    - 数据库查询：在大型的内存数据库中，Fibonacci查找可以用于高效的查找操作，可以帮助我们快速查询到需要的信息。
    - 图像处理：在图像处理中，Fibonacci查找可以用于查找目标图像中的某个元素，比如某个像素的位置等。
    - 金融领域：在金融领域中，Fibonacci查找可以用于快速查找某个客户的交易记录，以了解该客户的交易情况。
    - 网络寻址：在网络中，Fibonacci查找可以用于查找IP地址等信息，帮助我们快速定位到目标位置。
    - 数学计算：在数学计算中，Fibonacci查找可以用于计算斐波那契数列的第n项

Bloom Filter

概述

- Bloom Filter是一种基于概率的，空间效率很高的随机数据结构。它可以快速地判断一个元素是否存在于一个集合中。它的核心思想是使用多个哈希函数对元素进行哈希，将其映射到一个位数组中，并将每个位数组的值设为1。
- 当判断一个元素是否存在时，同样使用多个哈希函数对该元素进行哈希，并检查哈希结果所对应的位数组是否都为1，如果存在任意一个位数组的值为0，则可以确定该元素不存在于集合中，否则认为该元素可能存在于集合中。

实现原理

- 初始化位数组：Bloom过滤器需要一个位数组，用来表示集合中的元素是否存在。一开始将所有位都设为0，表示集合为空。
- 添加元素：将要添加的元素使用多个哈希函数进行哈希，得到多个哈希值，将每个哈希值对应的位数组的值设为1。
- 判断元素是否存在：同样使用多个哈希函数对要查找的元素进行哈希，得到多个哈希值，并检查每个哈希值对应的位数组的值是否都为1。如果存在任意一个哈希值对应的位数组的值为0，则可以确定该元素不存在于集合中，否则认为该元素可能存在于集合中。

算法思路

- 初始化位数组：Bloom过滤器需要一个位数组，用来表示集合中的元素是否存在。一开始将所有位都设为0，表示集合为空。
- 将每个要插入的元素经过k个不同的哈希函数进行哈希得到k个哈希值，这些哈希值分别对应比特数组中的k个位置，并将这k个位置的比特位都设置为1。
- 当要查询一个元素是否存在于Bloom过滤器中时，同样地将这个元素进行k次哈希，检查每个哈希值对应的比特位是否都为1，若其中有任意一个比特位为0，则说明该元素一定不存在于Bloom过滤器中；若所有比特位都为1，则说明该元素可能存在于Bloom过滤器中，但不一定存在，需要进一步查询。

代码实现

```java
import java.util.BitSet;

public class BloomFilter {
    private int m; // 比特数组的长度
    private int k; // 哈希函数的个数
```

```
    private BitSet bitSet;

    // 构造函数
    public BloomFilter(int m, int k) {
        this.m = m;
        this.k = k;
        bitSet = new BitSet(m);
    }

    // 插入一个元素
    public void add(String element) {
        for (int i = 0; i < k; i++) {
            int hash = MurMurHash3.hash(element.getBytes(), i);
            int index = Math.abs(hash) % m;
            bitSet.set(index);
        }
    }

    // 查询一个元素是否可能存在于Bloom过滤器中
    public boolean mightContain(String element) {
        for (int i = 0; i < k; i++) {
            int hash = MurMurHash3.hash(element.getBytes(), i);
            int index = Math.abs(hash) % m;
            if (!bitSet.get(index)) {
                return false;
            }
        }
        return true;
    }
}
```

复杂度分析

- 时间复杂度：Bloom过滤器的时间复杂度与哈希函数的个数k有关，添加元素的时间复杂度为O(k)，查询元素的时间复杂度为O(k)。
- 空间复杂度：Bloom过滤器的空间复杂度是O(m),但其空间复杂度主要取决于以下三个参数：期望存储的元素数量 n, 允许的误判率 fpp , 哈希函数的数量 k 。通常情况下，n和fpp是由应用场景决定的，k的选择一般为常数值。
    - 假设每个哈希函数得到的哈希值都是独立的，那么对于一个给定的元素，它被置为1的概率是p，即：$p = (1 - e^{(} - kn/m))^k$ 。其中，m表示Bloom过滤器使用的位数组的长度，也就是存储的总bit数。
    - 对于一个给定的误判率fpp，我们可以通过调整m和k来满足需求。假设我们固定了fpp和n的值，那么最小的位数组长度m和哈希函数数量k分别为：
      $m = -nln(fpp)/(ln2)^2$ ， $k = (ln2 * m)/n$
    - 通过这两个公式，我们可以得到Bloom过滤器的空间复杂度是O(m)，也就是与预期存储元素数量n和允许的误判率fpp有关

优缺点分析

- 优点
    - 空间效率高，Bloom过滤器存储数据只需要占用很少的内存空间。
    - 查询速度快，查询一个元素是否存在，只需要进行k次哈希计算即可，速度很快。
    - 可以进行分布式存储，因为Bloom过滤器只需要判断元素是否存在，而不需要存储具体的元素信息，所以可以将Bloom过滤器存储在不同的机器上。
- 缺点：
    - 有一定的误判率，可能会将不存在的元素误判为存在的元素。这个误判率与哈希函数的个数、哈希函数的质量、存储空间的大小等因素有关。
    - 不支持元素的删除操作，因为删除一个元素可能会影响到其他元素的判断结果，而Bloom过滤器的判断结果只能是可能存在或一定不存在，不能将一个可能存在的元素改为一定不存在。
    - 存在哈希碰撞的问题，当哈希函数不够好或者存储空间不够大时，会出现哈希碰撞的情况，导致误判率进一步增加

优化思路

- 增加哈希函数的数量：通过增加哈希函数的数量，可以降低误判率，但是也会增加计算哈希的时间和占用的空间。
- 优化哈希函数：使用高质量的哈希函数可以提高Bloom过滤器的准确性，同时降低误判率。
- 动态调整过滤器大小：在实际使用中，可以根据需要动态调整过滤器的大小，以便更好地平衡存储空间和准确性。
- 结合其他数据结构：可以将Bloom过滤器与其他数据结构结合使用，例如基于树或链表的数据结构，以提高准确性和效率

实际应用

- URL去重：
  在爬虫领域，我们需要从互联网上爬取海量的网页数据，而URL是其中一个重要的标识符。在去重时，可以使用Bloom过滤器来存储已经抓取过的URL，从而避免重复抓取相同的URL，提高爬取效率。
- 缓存穿透：
  在分布式缓存系统中，如果有一个热点key被频繁地查询，但是这个key对应的值并不存在于缓存中，那么这个查询就会直接落到数据库上，从而导致数据库的压力急剧增加，甚至可能引起宕机。这种情况被称为缓存穿透。为了解决这个问题，可以使用Bloom过滤器来快速判断一个key是否存在于缓存中，从而避免无效的查询落到数据库上。
- 恶意URL过滤：
  在网络安全领域，恶意URL是指恶意软件或者病毒通过URL传播，从而感染用户电脑的一种手段。为了防止用户误点击这些恶意URL，可以使用Bloom过滤器来快速过滤掉这些URL，从而保障用户的网络安全。
- 数据库查询优化：
  在查询数据时，如果数据量非常大，那么查询效率会非常低。为了提高查询效率，可以使用Bloom过滤器来预先过滤掉不存在的记录，从而减少查询的数量。这种方法在一些大型的分布式数据库系统中得到了广泛的应用。

Homework 10

LU Decomposition

问题分析

- LU分解是将一个矩阵分解成一个下三角矩阵和一个上三角矩阵的乘积，即A = LU，其中L为下三角矩阵，U为上三角矩阵。
- 常用的方法有Gauss消元法和Doolittle算法。

算法思路

- Gauss消元法
    - 从第一行开始，将第一行的第一个元素作为主元，将该行的其他元素除以主元，得到一个系数矩阵。
    - 将系数矩阵与原矩阵相乘，得到一个新的矩阵。
    - 重复上述步骤，直到得到一个上三角矩阵。
    - 在Gauss消元法中，每一步消元需要操作 n 行和列，所以总操作数为 n^3
- Doolittle算法
    - 将矩阵分解成L和U，其中L的对角线元素均为1。
    - 将A的第一行元素赋值给U的第一行，将L的第一列元素赋值为A的第一列元素除以U的第一个元素
    - 根据U的第i行，将A的第i+1行到最后一行的相应元素消为零
    - 重复上述过程，直到分解完毕，得到L和U
    - Doolittle算法中，每一步操作需要消去 n 个元素，所以总操作数同样为 n^3

代码实现

```java
//高斯消元法
public static void LUDecomposition(double[][] A) {
    int n = A.length;
    double[][] L = new double[n][n];
    double[][] U = new double[n][n];

    for (int i = 0; i < n; i++) {
        // 计算上三角矩阵U
        for (int j = i; j < n; j++) {
            double sum = 0;
            for (int k = 0; k < i; k++) {
                sum += L[i][k] * U[k][j];
            }
            U[i][j] = A[i][j] - sum;
        }

        // 计算下三角矩阵L
        for (int j = i; j < n; j++) {
            if (i == j) {
```

```
                L[i][i] = 1;
            } else {
                double sum = 0;
                for (int k = 0; k < i; k++) {
                    sum += L[j][k] * U[k][i];
                }
                L[j][i] = (A[j][i] - sum) / U[i][i];
            }
        }
    }

    System.out.println("L = ");
    printMatrix(L);
    System.out.println("U = ");
    printMatrix(U);
}
```

```
//Doolittle算法
public class LU_Doolittle {
    public static double[][] decompose(double[][] A) {
        int n = A.length;
        double[][] L = new double[n][n];
        double[][] U = new double[n][n];

        for (int i = 0; i < n; i++) {
            // 计算U的第一行
            U[0][i] = A[0][i];
            // 计算L的第一列
            L[i][0] = A[i][0] / U[0][0];
        }

        for (int i = 1; i < n; i++) {
            for (int j = i; j < n; j++) {
                double sum = 0.0;
                for (int k = 0; k < i; k++) {
                    sum += L[i][k] * U[k][j];
                }
                U[i][j] = A[i][j] - sum;
            }

            for (int j = i + 1; j < n; j++) {
                double sum = 0.0;
                for (int k = 0; k < i; k++) {
                    sum += L[j][k] * U[k][i];
                }
                L[j][i] = (A[j][i] - sum) / U[i][i];
            }
        }

        for (int i = 0; i < n; i++) {
            L[i][i] = 1.0;
        }
```

```
36
37          double[][] LU = new double[n][2 * n];
38          for (int i = 0; i < n; i++) {
39              for (int j = 0; j < n; j++) {
40                  LU[i][j] = L[i][j];
41              }
42              for (int j = n; j < 2 * n; j++) {
43                  LU[i][j] = U[i][j - n];
44              }
45          }
46
47          return LU;
48      }
49  }
```

算法分析

- Gauss消元法
    - 该算法的时间复杂度为 $O(n^3)$
    - 该算法的空间复杂度为 $O(n^2)$
    - 需要对系数矩阵进行消元，计算量较大，但该算法对于系数矩阵具有任意性，可以处理一些特殊的系数矩阵
- Doolittle算法
    - 该算法的时间复杂度为 $O(n^3)$
    - 该算法的空间复杂度为 $O(n^2)$
    - 计算量较小，但该算法只适用于主元素都不为0的系数矩阵

求解逆矩阵

问题分析

求解逆矩阵，常用的有Gauss-Jordan消元法和伴随矩阵法。

算法思路

- Gauss-Jordan消元法
    - 将矩阵A扩展为一个2n×n的矩阵M，其左半部分为矩阵A，右半部分为n×n的单位矩阵I
    - 对矩阵M进行Gauss-Jordan消元，将左半部分变为一个上三角矩阵。在进行消元的过程中，对右半部分也进行相同的消元操作
    - 经过Gauss-Jordan消元后，矩阵M的右半部分就是矩阵A的逆矩阵
- 伴随矩阵法
    - 先求解矩阵A的行列式，若行列式为0，则矩阵A不可逆
    - 再根据代数余子式求解矩阵A的伴随矩阵
    - 最后将伴随矩阵的每个元素除以行列式的值，得到矩阵A的逆矩阵

代码实现

```java
//Gauss-Jordan消元法
public static double[][] inverseMatrix(double[][] A) {
    int n = A.length;
    double[][] B = new double[n][2 * n];

    // 将矩阵 A 和单位矩阵 I 合并
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            B[i][j] = A[i][j];
            B[i][j + n] = (i == j) ? 1 : 0;
        }
    }

    // 高斯-约旦消元求解
    for (int i = 0; i < n; i++) {
        // 求出列 i 的主元素
        double max = Math.abs(B[i][i]);
        int maxRow = i;
        for (int j = i + 1; j < 2 * n; j++) {
            if (Math.abs(B[j][i]) > max) {
                max = Math.abs(B[j][i]);
                maxRow = j;
            }
        }

        // 将主元素所在行与当前行交换
        for (int j = i; j < 2 * n; j++) {
            double temp = B[i][j];
            B[i][j] = B[maxRow][j];
            B[maxRow][j] = temp;
        }

        // 将主元素缩放为 1
        double scale = B[i][i];
        for (int j = i; j < 2 * n; j++) {
            B[i][j] /= scale;
        }

        // 用主元素所在行消元
        for (int j = 0; j < n; j++) {
            if (j != i) {
                double factor = B[j][i];
                for (int k = i; k < 2 * n; k++) {
                    B[j][k] -= factor * B[i][k];
                }
            }
        }
    }

    // 返回逆矩阵
```

```
    double[][] inv = new double[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            inv[i][j] = B[i][j + n];
        }
    }
    return inv;
}
```

```
//伴随矩阵法
public static double[][] inverse(double[][] matrix) {
    int n = matrix.length;
    double[][] adjugate = adjugate(matrix);
    double determinant = determinant(matrix);
    double inversedDeterminant = 1 / determinant;
    double[][] inverse = new double[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            inverse[i][j] = adjugate[i][j] * inversedDeterminant;
        }
    }
    return inverse;
}

public static double[][] adjugate(double[][] matrix) {
    int n = matrix.length;
    double[][] adjugate = new double[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            adjugate[i][j] = Math.pow(-1, i + j) * determinant(minor(matrix,
i, j));
        }
    }
    return transpose(adjugate);
}

public static double[][] minor(double[][] matrix, int row, int col) {
    int n = matrix.length;
    double[][] minor = new double[n - 1][n - 1];
    int rowOffset = 0;
    for (int i = 0; i < n; i++) {
        if (i == row) {
            rowOffset = 1;
            continue;
        }
        int colOffset = 0;
        for (int j = 0; j < n; j++) {
            if (j == col) {
                colOffset = 1;
                continue;
            }
            minor[i - rowOffset][j - colOffset] = matrix[i][j];
```

```
        }
    }
    return minor;
}

public static double[][] transpose(double[][] matrix) {
    int n = matrix.length;
    int m = matrix[0].length;
    double[][] transposed = new double[m][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            transposed[j][i] = matrix[i][j];
        }
    }
    return transposed;
}

public static double determinant(double[][] matrix) {
    int n = matrix.length;
    if (n == 1) {
        return matrix[0][0];
    }
    double determinant = 0;
    int sign = 1;
    for (int i = 0; i < n; i++) {
        determinant += sign * matrix[0][i] * determinant(minor(matrix, 0, i));
        sign = -sign;
    }
    return determinant;
}
```

算法分析

- Gauss-Jordan消元法
    - 上述方法需要求解n个线性方程组，因此时间复杂度为O(n^3)。
    - 由于Gauss-Jordan消元涉及到除法操作，因此在计算过程中可能会出现数值精度损失的问题
- 伴随矩阵法
    - 因需计算行列式，因此时间复杂度为O(n!)

Homework 11

顶点v_i到顶点v_j之间长度为k的路径数量

对含有m个节点的有向无权非多重图G,用邻接矩阵A对其进行表示，矩阵A的第i列第j行元素表示以节点i为弧尾,节点j为弧头的边，记为a_{ij}。
由于该图是无权图，因此，如果a_{ij}的值为0，说明a_{ij}，所代表的边不存在；如果a_{ij}的值为 1，说明a_{ij}所代表的边存在。

k=1时

- 由于图 G 为非多重图，即不存在平行边，因此以节点i为弧尾，节点j为弧的边最多只能存在1条，即节点i到节点j距离为1的路径最多也只能存在1条。
- 综上，当a_{ij}的值为1时，图 G 中唯一存在以节点i为弧尾节点、j为弧头的边，自然也唯一存在节点i到节点j距离为1的路径。

令k=n时成立，则k=n+1时

假设矩阵A^n的元素 a_{ij}^n表示节点i到节点j距离为n的路径数，则根据矩阵乘法的运算定义，可得矩阵A^n的元素a_{ij}^{n+1}的计算式如下：
$a_{ij}^{n+1} = \sum_{k=1}^{m} a_{ik}a_{kj}^n$

- 由于a_{ij}^n表示节点i到节点j距离为n的路径数。而a_{ij}表示以节点i 为弧尾，节点j为孤头的边是否唯一存在，由此可知，$a_{ij}^n * a_{kj}$的含义为从节点i到节点j,且倒数第二个节点为节点k的、距离为n+1的路径的总数。
- 从而,a_{ij}^{n+1}表示了从节点i到节点j且倒数第二个节点为任意节点的、距离为n+1的路径的总数.
- 综上，A^{n+1}的元素a_{ij}^{n+1}表示节点 i到节点j距离为 n+1的路径数。

结论

最后，根据数学归纳法，结论成立。

Homework12

Find maximum indpendent set

问题分析

最大独立集问题是一个NP完全问题，因此我们可以采用近似算法来解决。本次作业中，我们采用的是贪心算法，即每次选择当前最优的解，直到所有的点都被覆盖。

算法设计

算法流程

1. 从图中选择一个度数最大的点，将其加入到独立集中
2. 将该点的所有邻居从图中删除
3. 重复1、2步骤，直到所有的点都被删除
4. 输出独立集

算法分析

- 时间复杂度：O(V^2)，空间复杂度：O(V)

算法实现

```
maxDegreeVertex = -1
maxDegree = 0
for v in G.V():
    if G.degree(v) > maxDegree:
        maxDegree = G.degree(v)
        maxDegreeVertex = v
maxIndependentSet = []
while maxDegreeVertex != -1:
    maxIndependentSet.append(maxDegreeVertex)
    for w in G.adj(maxDegreeVertex):
        G.removeVertex(w)
    maxDegreeVertex = -1
    maxDegree = 0
    for v in G.V():
        if G.degree(v) > maxDegree:
            maxDegree = G.degree(v)
            maxDegreeVertex = v
return maxIndependentSet
```

```

public class MaxIndependentSet {
    private Graph G;
    private int[] colors;
    private int maxDegree;
    private int maxDegreeVertex;
```

```
    private int[] maxIndependentSet;
    private int maxIndependentSetSize;

    public MaxIndependentSet(Graph G) {
        this.G = G;
        colors = new int[G.V()];
        maxDegree = 0;
        maxDegreeVertex = -1;
        maxIndependentSet = new int[G.V()];
        maxIndependentSetSize = 0;
        for (int v = 0; v < G.V(); v++) {
            colors[v] = -1;
            if (G.degree(v) > maxDegree) {
                maxDegree = G.degree(v);
                maxDegreeVertex = v;
            }
        }
        greedy();
    }

    private void greedy() {
        while (maxDegreeVertex != -1) {
            maxIndependentSet[maxIndependentSetSize++] = maxDegreeVertex;
            colors[maxDegreeVertex] = maxIndependentSetSize;
            for (int w : G.adj(maxDegreeVertex)) {
                colors[w] = maxIndependentSetSize;
            }
            maxDegree = 0;
            maxDegreeVertex = -1;
            for (int v = 0; v < G.V(); v++) {
                if (colors[v] == -1 && G.degree(v) > maxDegree) {
                    maxDegree = G.degree(v);
                    maxDegreeVertex = v;
                }
            }
        }
    }

    public int[] getMaxIndependentSet() {
        return maxIndependentSet;
    }

    public int getMaxIndependentSetSize() {
        return maxIndependentSetSize;
    }
}
```

优化思路

- 在算法实现中，我们每次都需要遍历所有的点，找到度数最大的点，这样的时间复杂度为O(V^2)，我们可以通过维护一个最大堆来优化这一步骤，这样的时间复杂度为O(VlogV)。

Minimize Weighted Completion Time

问题分析

在本次作业中，我们需要实现两种算法来解决最小化加权完成时间问题，分别是贪心算法和动态规划算法。

算法设计

贪心算法

采用的是自底向上的策略，即先安排最短的任务，再安排次短的任务，直到所有的任务都被安排完毕。

动态规划算法

采用的是自顶向下的策略，即先安排最长的任务，再安排次长的任务，直到所有的任务都被安排完毕。

算法实现

贪心算法

```

sort jobs by processing time
for job in jobs:
    schedule job
return schedule
```

```

public class Greedy {
    private Job[] jobs;
    private int[] schedule;
    private int[] completionTime;
    private int[] startTime;
    private int[] finishTime;
    private int[] waitingTime;
    private int[] processingTime;
    private int[] weight;
    private int totalWeight;
    private int n;

    public Greedy(Job[] jobs) {
        this.jobs = jobs;
        n = jobs.length;
        schedule = new int[n];
```

```
        completionTime = new int[n];
        startTime = new int[n];
        finishTime = new int[n];
        waitingTime = new int[n];
        processingTime = new int[n];
        weight = new int[n];
        totalWeight = 0;
        for (int i = 0; i < n; i++) {
            processingTime[i] = jobs[i].getProcessingTime();
            weight[i] = jobs[i].getWeight();
            totalWeight += weight[i];
        }
        greedy();
    }

    private void greedy() {
        int[] index = new int[n];
        for (int i = 0; i < n; i++) {
            index[i] = i;
        }
        Arrays.sort(index, (o1, o2) -> processingTime[o1] -
processingTime[o2]);
        int time = 0;
        for (int i = 0; i < n; i++) {
            int j = index[i];
            schedule[i] = j;
            startTime[j] = time;
            finishTime[j] = time + processingTime[j];
            completionTime[j] = finishTime[j];
            waitingTime[j] = startTime[j];
            time += processingTime[j];
        }
    }
}
```

动态规划算法

```

sort jobs by processing time
for job in jobs:
    schedule job
return schedule

```

```



public class DynamicProgramming {
    private Job[] jobs;
    private int[] schedule;
    private int[] completionTime;
    private int[] startTime;
    private int[] finishTime;
```

```
    private int[] waitingTime;
    private int[] processingTime;
    private int[] weight;
    private int totalWeight;
    private int n;

    public DynamicProgramming(Job[] jobs) {
        this.jobs = jobs;
        n = jobs.length;
        schedule = new int[n];
        completionTime = new int[n];
        startTime = new int[n];
        finishTime = new int[n];
        waitingTime = new int[n];
        processingTime = new int[n];
        weight = new int[n];
        totalWeight = 0;
        for (int i = 0; i < n; i++) {
            processingTime[i] = jobs[i].getProcessingTime();
            weight[i] = jobs[i].getWeight();
            totalWeight += weight[i];
        }
        dynamicProgramming();
    }

    private void dynamicProgramming() {
        int[] index = new int[n];
        for (int i = 0; i < n; i++) {
            index[i] = i;
        }
        Arrays.sort(index, (o1, o2) -> processingTime[o1] -
processingTime[o2]);
        int[][] dp = new int[n + 1][totalWeight + 1];
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= totalWeight; j++) {
                dp[i][j] = Integer.MAX_VALUE;
            }
        }
        dp[0][0] = 0;
        for (int i = 1; i <= n; i++) {
            int j = index[i - 1];
            for (int k = 0; k <= totalWeight; k++) {
                if (k >= weight[j]) {
                    dp[i][k] = Math.min(dp[i - 1][k], dp[i - 1][k - weight[j]]
+ processingTime[j]);
                } else {
                    dp[i][k] = dp[i - 1][k];
                }
            }
        }
        int k = totalWeight;
        for (int i = n; i >= 1; i--) {
            int j = index[i - 1];
```

```
            if (dp[i][k] == dp
                    [i - 1][k - weight[j]] + processingTime[j]) {
                schedule[i - 1] = j;
                startTime[j] = dp[i - 1][k - weight[j]];
                finishTime[j] = dp[i][k];
                completionTime[j] = finishTime[j];
                waitingTime[j] = startTime[j];
                k -= weight[j];
            }

        }
    }
}
```

算法分析

- 贪心算法的时间复杂度为$O(nlogn)$，空间复杂度为$O(n)$。
- 动态规划算法的时间复杂度为$O(nW)$，空间复杂度为$O(nW)$。

Homework 13

最省时路线

问题抽象

鉴于由题图知，如果将两地行驶时间抽象为该边的权重，则该问题实际上为求解矩阵中从一个顶点到与之对应的另一个顶点的最短距离问题。

问题分析

- 根据上述抽象，本题可以先将原有矩阵进行上下翻转，再整理成一个二维数组，每个元素代表从左上角到该位置的最小路径和，则原问题即变成求从左上角到右下角的最小路径和

算法思路

- 从暴力的角度，我们可以使用递归的方法，从左上角开始，向右或向下移动，直到到达右下角，然后返回路径和。但是这种方法的时间复杂度为O(2^(m+n))，空间复杂度为O(m+n)，时间复杂度高，不适用于大型矩阵
- 从贪心的角度，从左上角出发，每次选择向右或向下移动一格，使得移动后的格子的权值最小。该算法的时间复杂度是O(m+n)，空间复杂度为O(1),其中m和n分别是矩阵的行数和列数，因此比暴力枚举法快得多。然而，并不能保证总是能找到最优解，因此该算法并不适用于所有情况
- 从分治的角度，将矩阵划分成四个部分，分别计算从左上角到右下角的最小数字和。如果只有一个格子，则直接返回该格子的数字。如果只有一行或一列，则计算这一行或这一列的数字和。否则，分别计算右下部分、下方部分和右侧部分的最小数字和。最后，返回从左上角到右下角的最小数字和
- 从动态规划的角度，我们可以使用一个二维数组dp来存储从左上角到右下角的最小数字和。dp[i][j]表示从左上角到(i,j)位置的最小数字和。我们可以使用递推式dp[i][j]=min(dp[i-1][j],dp[i][j-1])+matrix[i][j]来计算dp[i][j]的值。最后，返回dp[m-1][n-1]即可。该算法的时间复杂度是O(mn)，空间复杂度是O(mn)

代码实现

```java
//暴力枚举法
public static int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    return dfs(grid, 0, 0, m, n);
}

public static int dfs(int[][] grid, int i, int j, int m, int n) {
    if (i == m - 1 && j == n - 1) {
        return grid[i][j];
    }
    if (i == m - 1) {
        return grid[i][j] + dfs(grid, i, j + 1, m, n);
    }
    if (j == n - 1) {
        return grid[i][j] + dfs(grid, i + 1, j, m, n);
    }
    int rightSum = dfs(grid, i, j + 1, m, n);
    int downSum = dfs(grid, i + 1, j, m, n);
    return grid[i][j] + Math.min(rightSum, downSum);
}
```

```java
//贪心算法
public static int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    int row = 0, col = 0;
    int sum = 0;
    while (row < m && col < n) {
        if (row == m - 1) {
            sum += grid[row][col];
            col++;
        } else if (col == n - 1) {
            sum += grid[row][col];
            row++;
        } else {
            if (grid[row][col + 1] < grid[row + 1][col]) {
                sum += grid[row][col];
                col++;
            } else {
                sum += grid[row][col];
                row++;
            }
        }
    }
    return sum;
}
```

```java
//分治算法
```

```java
public static int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    return divideAndConquer(grid, 0, 0, m - 1, n - 1);
}

public static int divideAndConquer(int[][] grid, int startX, int startY, int endX, int endY) {
    if (startX > endX || startY > endY) {
        return Integer.MAX_VALUE;
    }
    if (startX == endX && startY == endY) {
        return grid[startX][startY];
    }
    int midX = startX + (endX - startX) / 2;
    int midY = startY + (endY - startY) / 2;
    int rightSum = divideAndConquer(grid, startX, midY + 1, endX, endY);
    int downSum = divideAndConquer(grid, midX + 1, startY, endX, midY);
    int minSum = Math.min(rightSum, downSum);
    if (midX < endX && midY < endY) {
        int diagonalSum = divideAndConquer(grid, midX + 1, midY + 1, endX, endY);
        minSum = Math.min(minSum, diagonalSum);
    }
    return grid[startX][startY] + minSum;
}
```

```java
//动态规划
public static List<int[]> minSumPath(int[][] matrix) {
    int m = matrix.length;
    int n = matrix[0].length;

    int[][] dp = new int[m][n];
    dp[0][0] = matrix[0][0];

    // 第一列只能从上面一格过来
    for (int i = 1; i < m; i++) {
        dp[i][0] = dp[i - 1][0] + matrix[i][0];
    }

    // 第一行只能从左边一格过来
    for (int j = 1; j < n; j++) {
        dp[0][j] = dp[0][j - 1] + matrix[0][j];
    }

    // 填充剩下的格子，每个格子只能从左边或上面过来
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = matrix[i][j] + Math.min(dp[i - 1][j], dp[i][j - 1]);
        }
    }

```

```
    // 反向遍历dp数组，找出最小路径
    List<int[]> path = new ArrayList<>();
    int i = m - 1, j = n - 1;
    while (i > 0 || j > 0) {
        path.add(new int[]{i, j});
        if (i == 0) {
            j--;
        } else if (j == 0) {
            i--;
        } else if (dp[i - 1][j] < dp[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }
    path.add(new int[]{0, 0});
    Collections.reverse(path);

    return path;
}
```

- 四种算法中，空间复杂度最优的是贪心算法，因为它不需要额外的存储空间。动态规划和暴力枚举法的空间复杂度相同，都为 O(mn)。分治算法的空间复杂度也为 O(mn)，但由于递归调用，还需要额外的栈空间

芯片最大连接

问题分析

因需连接的仅为输入和输出端口，且只能由输入端口连接到输出端口，因此本题可转换成一个二部图进行求解。即将输入端口和输出端口分别看成二部图的两个顶点集合，将芯片之间的连接看成二部图的边集合，求解二部图的最大匹配即可。

算法设计

- 根据题目需求，本题可采用匈牙利算法暴力搜索法，或者映射增广路径算法求解二部图的最大匹配。
- 同时，为确定是否存在连接冲突的情况，既可以采取映射用以判断是否冲突，也可以采取暴力搜索法，判断是否存在交叉连接。

代码实现

```
//匈牙利算法
class ChipDesign {
    public static int calculateMaxConnections(List<Integer> inputPorts,
List<Integer> outputPorts) {
        Set<Integer> usedInputPorts = new HashSet<>();
        Set<Integer> usedOutputPorts = new HashSet<>();
        int maxConnections = 0;

        for (int inputPort : inputPorts) {
```

```
            if (usedInputPorts.contains(inputPort)) {
                continue; // 如果已经使用过该输入端口，跳过
            }

            for (int outputPort : outputPorts) {
                if (usedOutputPorts.contains(outputPort)) {
                    continue; // 如果已经使用过该输出端口，跳过
                }

                if (inputPort == outputPort) {
                    // 判断连接是否相交
                    boolean isIntersect = false;
                    for (int i = 0; i < inputPorts.size(); i++) {
                        if (inputPorts.get(i) == outputPorts.get(i) &&
usedInputPorts.contains(inputPorts.get(i))) {
                            isIntersect = true;
                            break;
                        }
                    }

                    if (!isIntersect) {
                        usedInputPorts.add(inputPort);
                        usedOutputPorts.add(outputPort);
                        maxConnections++;
                        break;
                    }
                }
            }
        }

        return maxConnections;
    }
}
```

```
// 暴力搜索法
class ChipDesign {
    public static int calculateMaxConnections(List<Integer> inputPorts,
List<Integer> outputPorts) {
        Set<Integer> usedInputPorts = new HashSet<>();
        Set<Integer> usedOutputPorts = new HashSet<>();
        int maxConnections = 0;

        Map<Integer, Integer> outputToInputMapping = new HashMap<>();
        for (int i = 0; i < inputPorts.size(); i++) {
            int inputPort = inputPorts.get(i);
            int outputPort = outputPorts.get(i);
            if (inputPort == outputPort && !usedInputPorts.contains(inputPort)
&& !usedOutputPorts.contains(outputPort)) {
                outputToInputMapping.put(outputPort, inputPort);
                usedInputPorts.add(inputPort);
                usedOutputPorts.add(outputPort);
                maxConnections++;
```

```
            }
        }

        for (int i = 0; i < inputPorts.size(); i++) {
            int inputPort = inputPorts.get(i);
            int outputPort = outputPorts.get(i);
            if (inputPort != outputPort && !usedInputPorts.contains(inputPort)
&& !usedOutputPorts.contains(outputPort)) {
                Integer mappedInputPort =
outputToInputMapping.get(outputPort);
                if (mappedInputPort == null) {
                    outputToInputMapping.put(outputPort, inputPort);
                    usedInputPorts.add(inputPort);
                    usedOutputPorts.add(outputPort);
                    maxConnections++;
                } else if (mappedInputPort == inputPort) {
                    // 当出现重复映射时，选择连接数更多的映射
                    int mappedOutputPort = outputPorts.indexOf(outputPort);
                    int mappedConnections = countConnections(inputPorts,
outputPorts, usedInputPorts, usedOutputPorts);
                    int currentConnections = countConnections(inputPorts,
outputPorts, usedInputPorts, usedOutputPorts, inputPort, outputPort);
                    if (currentConnections > mappedConnections) {
                        outputToInputMapping.put(outputPort, inputPort);
                        usedInputPorts.remove(mappedInputPort);
                        usedOutputPorts.remove(mappedOutputPort);
                        usedInputPorts.add(inputPort);
                        usedOutputPorts.add(outputPort);
                        maxConnections += (currentConnections -
mappedConnections);
                    }
                }
            }
        }

        return maxConnections;
    }

    private static int countConnections(List<Integer> inputPorts,
List<Integer> outputPorts, Set<Integer> usedInputPorts, Set<Integer>
usedOutputPorts) {
        int count = 0;
        for (int i = 0; i < inputPorts.size(); i++) {
            int inputPort = inputPorts.get(i);
            int outputPort = outputPorts.get(i);
            if (inputPort == outputPort && usedInputPorts.contains(inputPort)
&& usedOutputPorts.contains(outputPort)) {
                count++;
            }
        }
        return count;
    }

```

```
    private static int countConnections(List<Integer> inputPorts, List<Integer> outputPorts, Set<Integer> usedInputPorts, Set<Integer> usedOutputPorts, int excludeInputPort, int excludeOutputPort) {
        int count = 0;
        for (int i = 0; i < inputPorts.size(); i++) {
            int inputPort = inputPorts.get(i);
            int outputPort = outputPorts.get(i);
            if (inputPort == outputPort && usedInputPorts.contains(inputPort) && usedOutputPorts.contains(outputPort)
                    && (inputPort != excludeInputPort || outputPort != excludeOutputPort)) {
                count++;
            }
        }
        return count;
    }
}
```

```
// 映射计数
class ChipDesign {
    public static int calculateMaxConnections(List<Integer> inputPorts, List<Integer> outputPorts) {
        Map<Integer, Integer> inputCount = new HashMap<>();
        Map<Integer, Integer> outputCount = new HashMap<>();
        int maxConnections = 0;

        for (int i = 0; i < inputPorts.size(); i++) {
            int inputPort = inputPorts.get(i);
            int outputPort = outputPorts.get(i);
            inputCount.put(inputPort, inputCount.getOrDefault(inputPort, 0) + 1);
            outputCount.put(outputPort, outputCount.getOrDefault(outputPort, 0) + 1);
        }

        for (int i = 0; i < inputPorts.size(); i++) {
            int inputPort = inputPorts.get(i);
            int outputPort = outputPorts.get(i);
            if (inputPort == outputPort && inputCount.get(inputPort) > 0 && outputCount.get(outputPort) > 0) {
                maxConnections++;
                inputCount.put(inputPort, inputCount.get(inputPort) - 1);
                outputCount.put(outputPort, outputCount.get(outputPort) - 1);
            }
        }

        return maxConnections;
    }
}
```

算法分析

- 匈牙利算法时间复杂度为O(n^3)，空间复杂度为O(n^2)。
- 暴力搜索法的时间复杂度为 O(n^2)，映射计数法的时间复杂度为 O(n)；二者空间复杂度为 O(n)。