目录

- 补充问题
- 链表
  - 反转单链表和双向链表
  - 判断一个链表是否为回文结构
  - 链表与荷兰国旗问题
  - 复制含有随机指针结点的链表
    - 借助哈希表，额外空间`O(N)`
    - 进阶操作：额外空间`O(1)`
  - 若两个可能有环的单链表相交，请返回相交的第一个结点
- 栈和队列
  - 用数组结构实现大小固定的栈和队列
  - 取栈中最小元素
  - 仅用队列结构实现栈结构
  - 仅用栈结构实现队列结构
- 二叉树
  - 实现二叉树的先序、中序、后续遍历，包括递归方式和非递归方式
    - 递归方式
    - 非递归方式
      - 先序遍历
      - 中序遍历
      - 后序遍历
  - 在二叉树中找一个结点的后继结点，结点除lleft,right指针外还包含一个parent指针
  - 介绍二叉树的序列化和反序列化
    - 序列化
    - 重建
  - 判断一个树是否是平衡二叉树
  - 判断一棵树是否是搜索二叉树
  - 判断一棵树是否是完全二叉树
  - 已知一棵完全二叉树，求其结点个数，要求时间复杂度0(N)
- 并查集
  - 并查集结构的实现
  - 并查集的应用

贪心策略

- 拼接最小字典序
- 金条和铜板
- IPO
- 会议室项目宣讲

递归和动态规划

- 暴力递归
  - n!问题
  - 汉诺塔问题
  - 打印一个字符串的所有子序列
  - 打印一个字符串的所有全排列结果
  - 母牛生牛问题
- 暴力递归改为动态规划
  - 最小路径和
    - 递归尝试版本
    - 根据尝试版本改动态规划
  - 一个数是否是数组中任意个数的和
    - 暴力递归版本
    - 暴力递归改动态规划（高度套路）
  - 哪些暴力递归能改为动态规划
    - 有后效性和无后效性

时间复杂度

时间复杂度是衡量算法好坏的重要指标之一。时间复杂度反映的是不确定性样本量的增长对于算法操作所需时间的影响程度，与算法操作是否涉及到样本量以及涉及了几次直接相关，如遍历数组时时间复杂度为数组长度n（对应时间复杂度为`O(n)`），而对数据的元操作（如加减乘除与或非等）、逻辑操作（如if判断）等都属于**常数**时间内的操作（对应时间复杂度`O(1)`）。

在化简某算法时间复杂度表达式时需遵循以下规则：

- 对于同一样本量，可省去低阶次数项，仅保留高阶次数项，如`O(n^2)+O(n)`可化简为`O(n^2)`，`O(n)+O(1)`可化简为`O(n)`
- 可省去样本量前的常量系数，如`O(2n)`可化简为`O(n)`，`O(8)`可化简为`O(1)`
- 对于不同的不确定性样本量，不能按照上述两个规则进行化简，要根据实际样本量的大小分析表达式增量。如`O(logm)+O(n^2)`不能化简为`O(n^2)`或`O(logm)`。而要视m、n两者之间的差距来化简，比如m>>n时可以化简为`O(logm)`，因为表达式增量是由样本量决定的。

额外空间复杂度

算法额外空间复杂度指的是对于输入样本，经过算法操作需要的额外空间。比如使用冒泡排序对一个数组排序，期间只需要一个临时变量`temp`，那么该算法的额外空间复杂度为`O(1)`。又如归并排序，在排序过程中需要创建一个与样本数组相同大小的辅助数组，尽管在排序过后该数组被销毁，但该算法的额外空间复杂度为`O(n)`。

经典例题——举一反三

找出B中不属于A的数

> 找出数组B中不属于A的数，数组A有序而数组B无序。假设数组A有n个数，数组B有m个数，写出算法并分析时间复杂度。

方法一：遍历

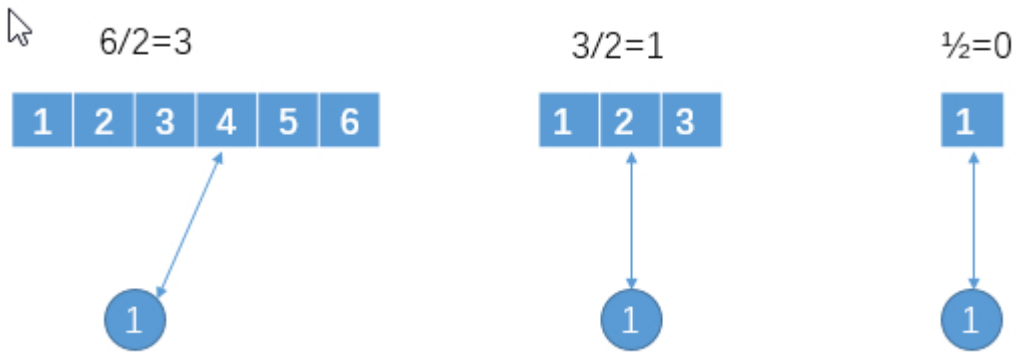
首先遍历B，将B中的每个数拿到到A中找，若找到则打印。对应算法如下：

```
int A[] = {1, 2, 3, 4, 5};
int B[] = {1, 4, 2, 6, 5, 7};

for (int i = 0; i < 6; ++i) {
  int temp = B[i];
  bool flag = false;
  for (int j = 0; j < 5; ++j) {
    if (A[j] == temp) {
      flag = true;    //找到了
      break;
    }
  }
  if (!flag) {    //没找到
    printf("%d", temp);
  }
}
```

不难看出上述算法的时间复杂度为`O(m*n)`，因为将两个数组都遍历了一遍

方法二：二分查找

由于数组A是有序的，**在一个有序序列中查找一个元素可以使用二分法（也称折半法）**。原理就是将查找的元素与序列的中位数进行比较，如果小于则去掉中位数及其之后的序列，如果大于则去掉中位数及其之前的序列，如果等于则找到了。如果不等于那么再将其与剩下的序列继续比较直到找到或剩下的序列为空为止。



利用二分法对应题解的代码如下：

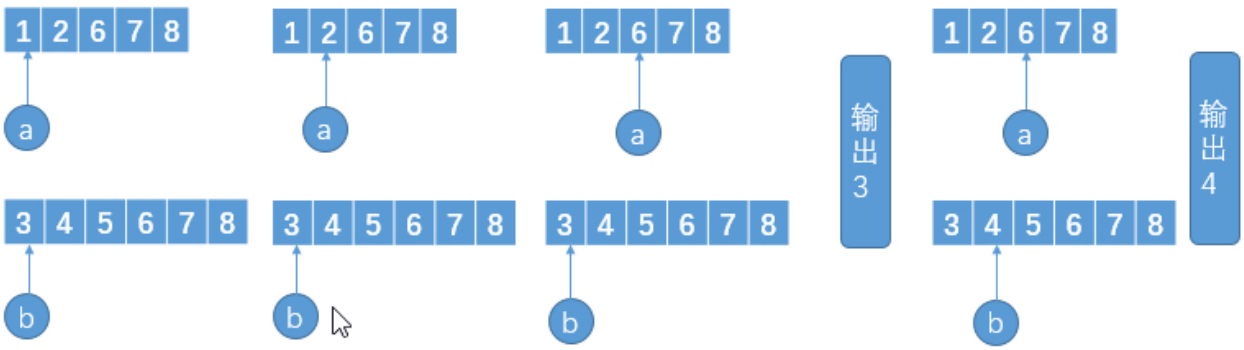
```
for (int i = 0; i < 6; ++i) {       //B的长度为6
  int temp = B[i];
```

```
  //二分法查找
  int left = 0,right = 5-1;          //A的长度为5
  int mid = (left + right) / 2;
  while (left < right && A[mid] != temp) {
    if (A[mid] > temp) {
      right = mid - 1;
    } else {
      left = mid + 1;
    }
    mid = (left + right) / 2;
  }
  if (A[mid] != temp) {
    printf("%d", temp);
  }
}
```

`for` 循环 `m` 次，`while` 循环 `logn` 次（如果没有特别说明，log均以2为底），此算法的时间复杂度为 `O(mlogn)`

方法三：排序+外排

第三种方法就是将数组B也排序，然后使用**逐次比对**的方式来查找A数组中是否含有B数组中的某元素。引入a、b两个指针分别指向数组A、B的首元素，比较指针指向的元素值，当 `a<b` 时，向后移动a指针查找该元素；当 `a=b` 时，说明A中存在该元素，跳过该元素查找，向后移动b；当 `a>b` 时说明A中不存在该元素，打印该元素并跳过该元素的查找，向后移动b。直到a或b有一个到达数组末尾为止（若a先到达末尾，那么b和b之后的数都不属于A）



对应题解的代码如下：

```
void fun3(int A[],int a_length,int B[],int b_length){
    quickSort(B, 0, b_length - 1);  //使用快速排序法对数组B排序->O(mlogm)
    int* a = A,*b=B;
    while (a <= A + a_length - 1 || b <= B + b_length - 1) {
        if (*a == *b) {
            b++;
            continue;
        }
        if (*a > *b) {
            printf("%d", *b);
            b++;
        } else {
            a++;
        }
```

```
    }

    if (a == A + a_length) {    //a先到头
        while (b < B + b_length) {
            printf("%d", *b);
            b++;
        }
    }
}
```

快速排序的代码如下：

```
#include <stdlib.h>
#include <time.h>

//交换两个int变量的值
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

//产生一个low~high之间的随机数
int randomInRange(int low, int high){
    srand((int) time(0));
    return (rand() % (high - low))+low;
}

//快速排序的核心算法，随机选择一个数，将比该数小的移至数组左边，比该数大的移至
//数组右边，最后返回该数的下标（移动完之后该数的下标可能与移动之前不一样）
int partition(int arr[],int start,int end){
    if (arr == NULL || start < 0 || end <= 0 || start > end) {
        return -1;
    }

    int index = randomInRange(start, end);//随机选择一个数
    swap(arr[index], arr[end]);//将该数暂时放至末尾

    int small = start - 1;
    //遍历前n-1个数与该数比较并以该数为界限将前n-1个数
    //分为两组，small指向小于该数的那一组的最后一个元素
    for (index = start; index < end; index++) {
        if (arr[index] < arr[end]) {
            small++;
            if (small != index) {
                swap(arr[small], arr[index]);
            }
        }
    }

    //最后将该数放至数值较小的那一个组的中间
    ++small;
    swap(arr[small], arr[end]);
```

```
    return small;
}

void quickSort(int arr[],int start,int end) {
    if (start == end) {
        return;
    }
    int index = partition(arr, start, end);
    if (index > start) {
        quickSort(arr,start, index - 1);
    }
    if (index < end) {
        quickSort(arr, index + 1, end);
    }
}
```

此种方法的时间复杂度为：`O(mlogm)`（先对B排序）+`O(m+n)`（最坏的情况是指针a和b都到头）。

三种方法的比较

1. `O(m*n)`
2. `O(mlogn)`（以2为底）
3. `O(mlogm)+O(m+n)`（以2为底）

易知算法2比1更优，因为增长率`logn<n`。而2和3的比较取决于样本量m和n之间的差距，若`m>>n`那么2更优，不难理解：数组B元素较多，那么对B的排序肯定要花费较长时间，而这一步并不是题解所必需的，不如采用二分法；相反地，若`m<<n`，那么3更优。

荷兰国旗问题

给定一个数组arr，和一个数num，请把小于num的数放在数组的左边，等于num的数放在数组的中间，大于num的数放在数组的右边。

> 要求额外空间复杂度O(1)，时间复杂度O(N)

思路：利用两个指针`L`、`R`，将`L`指向首元素之前，将`R`指向尾元素之后。从头遍历序列，将当前遍历元素与`num`比较，若`num`，则将其与`L`的右一个元素交换位置并遍历下一个元素、右移`L`；若`=num`则直接遍历下一个元素；若`>num`则将其和`R`的左一个元素交换位置，并重新判断当前位置元素与`num`的关系。直到遍历的元素下标到为`R-1`为止。

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}
void partition(int arr[],int startIndex,int endIndex,int num){
    int L = startIndex - 1, R = endIndex + 1, i = startIndex;
    while (i <= R - 1) {
        if (arr[i] < num) {
            swap(arr[i++], arr[++L]);
        } else if (arr[i] > num) {
            swap(arr[i], arr[--R]);
```

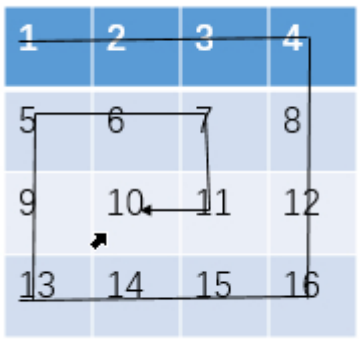
```
        } else {
            i++;
        }
    }
}

int main(){
    int arr[] = {1,2, 1, 5, 4, 7, 2, 3, 9,1};
    travles(arr, 8);
    partition(arr, 0, 7, 2);
    travles(arr, 8);
    return 0;
}
```

`L` 代表小于 `num` 的数的右界，`R` 代表大于 `num` 的左界，`partition` 的过程就是遍历元素、不断壮大 `L、R` 范围的过程。这里比较难理解的地方可能是为什么 `arr[i]<num` 时要右移 `L` 而 `arr[i]>num` 时却不左移 `R`，这是因为对于当前元素 `arr[i]`，如果 `arr[i]<num` 进行 `swap(arr[i],arr[L+1])` 之后对于当前下标的数据状况是知晓的（一定有 `arr[i]=arr[L+1]`），因为是从头遍历到 `i` 的，而 `L+1<=i`。但是如果 `arr[i]>num` 进行 `swap(arr[i],arr[R-1])` 之后对于当前元素的数据状况是不清楚的，因为 `R-1>=i`，`arr[R-1]` 还没遍历到。

矩阵打印问题

转圈打印方块矩阵

给定一个4阶矩阵如下：

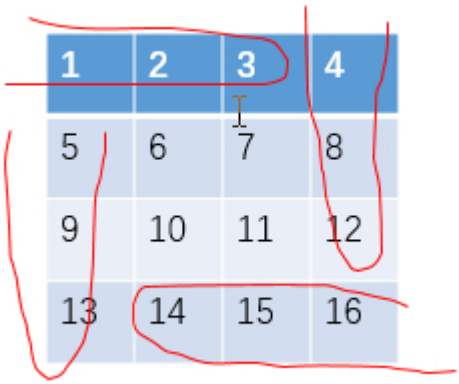


打印结果如下（要求额外空间复杂度为 `O(1)`）：

```
1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
```

> 思路：这类问题需要将思维打开，从宏观的层面去找出问题存在的共性从而求解。如果你的思维局限在1是如何变到2的、4是怎么变到8的、11之后为什么时10、它们之间有什么关联，那么你就陷入死胡同了。
>
> 从宏观的层面找共性，其实转圈打印的过程就是不断顺时针打印外围元素的过程，只要给你一个左上角的点（如 `(0,0)`）和右下角的点（如 `(3,3)`），你就能够打印出 `1 2 3 4 8 12 16 15 14 13 9 5`；同样，给你 `(1,1)` 和 `(2,2)`，你就能打印出 `6 7 11 10`。这个根据两点打印正方形上元素的过程可以抽取出来，整个问题也就迎刃而解了。

打印一个矩阵某个正方形上的点的逻辑如下：



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

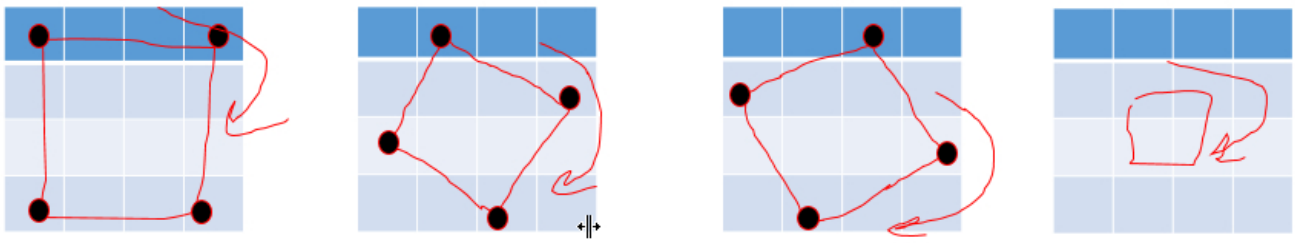
```c
//
// Created by zaw on 2018/10/21.
//
#include <stdio.h>
#define FACTORIAL 4

void printSquare(int leftUp[], int rigthDown[],int matrix[][FACTORIAL]){
    int i = leftUp[0], j = leftUp[1];
    while (j < rigthDown[1]) {
        printf("%d ", matrix[i][j++]);
    }
    while (i < rigthDown[0]) {
        printf("%d ", matrix[i++][j]);
    }
    while (j > leftUp[1]) {
        printf("%d ", matrix[i][j--]);
    }
    while (i > leftUp[0]) {
        printf("%d ", matrix[i--][j]);
    }
}

void printMatrixCircled(int matrix[][FACTORIAL]){
    int leftUp[] = {0, 0}, rightDown[] = {FACTORIAL-1,FACTORIAL-1};
    while (leftUp[0] < rightDown[0] && leftUp[1] < rightDown[1]) {
        printSquare(leftUp, rightDown, matrix);
        ++leftUp[0];
        ++leftUp[1];
        --rightDown[0];
        --rightDown[1];
    }
}

int main(){
    int matrix[4][4] = {
            {1,  2,  3,  4},
            {5,  6,  7,  8},
            {9,  10, 11, 12},
            {13, 14, 15, 16}
```

```
    };
    printMatrixCircled(matrix);//1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
}
```

旋转方块矩阵

给定一个方块矩阵，请把该矩阵调整成顺时针旋转90°之后的样子，要求额外空间复杂度为`O(1)`。

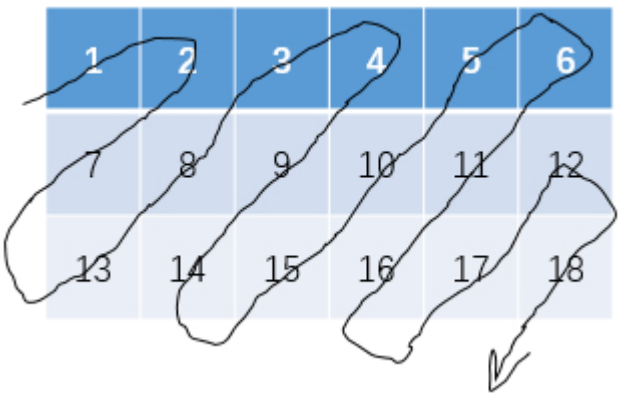
> 思路：拿上图举例，首先选取矩阵四个角上的点`1,3,9,7`，按顺时针的方向`1`到`3`的位置（`1->3`）、`3->9`、`9->7`、`7->1`，这样对于旋转后的矩阵而言，这四个点已经调整好了。接下来只需调整`2,6,8,4`的位置，调整方法是一样的。只需对矩阵第一行的前n-1个点采用同样的方法进行调整、对矩阵第二行的前前n-3个点……，那么调整n阶矩阵就容易了。
>
> 这也是在宏观上观察数据变动的一般规律，找到以不变应万变的通解（给定一个点，确定矩阵上以该点为角的正方形，将该正方形旋转90°），整个问题就不攻自破了。



```
//
// Created by zaw on 2018/10/21.
//
#include <stdio.h>
#define FACTORIAL 4

void circleSquare(int leftUp[],int rightDown[],int matrix[][FACTORIAL]){
    int p1[] = {leftUp[0], leftUp[1]};
    int p2[] = {leftUp[0], rightDown[1]};
    int p3[] = {rightDown[0], rightDown[1]};
    int p4[] = {rightDown[0],leftUp[1]};
    while (p1[1] < rightDown[1]) {
        //swap
        int tmp = matrix[p4[0]][p4[1]];
        matrix[p4[0]][p4[1]] = matrix[p3[0]][p3[1]];
        matrix[p3[0]][p3[1]] = matrix[p2[0]][p2[1]];
        matrix[p2[0]][p2[1]] = matrix[p1[0]][p1[1]];
        matrix[p1[0]][p1[1]] = tmp;

        p1[1]++;
```

```
        p2[0]++;
        p3[1]--;
        p4[0]--;
    }
}

void circleMatrix(int matrix[][FACTORIAL]){
    int leftUp[] = {0, 0}, rightDown[] = {FACTORIAL - 1, FACTORIAL - 1};
    while (leftUp[0] < rightDown[0] && leftUp[1] < rightDown[1]) {
        circleSquare(leftUp, rightDown, matrix);
        leftUp[0]++;
        leftUp[1]++;
        --rightDown[0];
        --rightDown[1];
    }
}

void printMatrix(int matrix[][FACTORIAL]){
    for (int i = 0; i < FACTORIAL; ++i) {
        for (int j = 0; j < FACTORIAL; ++j) {
            printf("%2d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main(){
    int matrix[FACTORIAL][FACTORIAL] = {
            {1,  2,  3,  4},
            {5,  6,  7,  8},
            {9,  10, 11, 12},
            {13, 14, 15, 16}
    };
    printMatrix(matrix);
    circleMatrix(matrix);
    printMatrix(matrix);
}
```

之字形打印矩阵



对如上矩阵的打印结果如下（要求额外空间复杂度为`O(1)`）：

```
1 2 7 13 8 3 4 9 14 15 10 5 6 11 16 17 12 18
```

> 此题也是需要从宏观上找出一个共性：给你两个，你能否将该两点连成的45°斜线上的点按给定的打印方向打印出来。拿上图举例，给出`(2,0)`、`(0,2)`和`turnUp=true`，应该打印出`13,8,3`。那么整个问题就变成了两点的走向问题了，开始时两点均为`(0,0)`，然后一个点往下走，另一个点往右走（如`1->7`，`1->2`）；当往下走的点是边界点时就往右走（如`13->14`），当往右走的点到边界时就往下走（如`6->12`）。每次两点走一步，并打印两点连线上的点。

```
//
// Created by zaw on 2018/10/22.
//
#include <stdio.h>

const int rows = 3;
const int cols = 6;

void printLine(int leftDown[],int rightUp[], bool turnUp,int matrix[rows][cols]){
    int i,j;
    if (turnUp) {
        i = leftDown[0], j = leftDown[1];
        while (j <= rightUp[1]) {
            printf("%d ", matrix[i--][j++]);
        }
    } else {
        i = rightUp[0], j = rightUp[1];
        while (i <= leftDown[0]) {
            printf("%d ", matrix[i++][j--]);
        }
    }
}

void zigZagPrintMatrix(int matrix[rows][cols]){
    if (matrix==NULL)
        return;
    int leftDown[] = {0, 0}, rightUp[] = {0, 0};
    bool turnUp = true;
    while (leftDown[1] <= cols - 1) {
        printLine(leftDown, rightUp, turnUp, matrix);
        turnUp = !turnUp;
        if (leftDown[0] < rows - 1) {
            leftDown[0]++;
        } else {
            leftDown[1]++;
        }
        if (rightUp[1] < cols - 1) {
            ++rightUp[1];
        } else {
            ++rightUp[0];
        }
    }
```

```
}

int main(){
    int matrix[rows][cols] = {
            {1,  2,  3,  4,  5,  6},
            {7,  8,  9,  10, 11, 12},
            {13, 14, 15, 16, 17, 18}
    };
    zigZagPrintMatrix(matrix);//1 2 7 13 8 3 4 9 14 15 10 5 6 11 16 17 12 18
    return 0;
}
```

在行和列都排好序的矩阵上找数

如图:

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

任何一列或一行上的数是有序的，实现一个函数，判断某个数是否存在于矩阵中。要求时间复杂度为`O(M+N)`，额外空间复杂度为`O(1)`。

> 从矩阵右上角的点开始取点与该数比较，如果大于该数，那么说明这个点所在的列都不存在该数，将这个点左移；如果这个点上的数小于该数，那么说明这个点所在的行不存在该数，将这个点下移。直到找到与该数相等的点为止。最坏的情况是，该数只有一个且在矩阵左下角上，那么时间复杂度为`O(M-1+N-1)=O(M+N)`

```
//
// Created by zaw on 2018/10/22.
//
#include <stdio.h>
const int rows = 4;
const int cols = 4;

bool findNumInSortedMatrix(int num,int matrix[rows][cols]){
    int i = 0, j = cols - 1;
    while (i <= rows - 1 && j <= cols - 1) {
        if (matrix[i][j] > num) {
            --j;
        } else if (matrix[i][j] < num) {
            ++i;
        } else {
            return true;
        }
    }
```

```
    return false;
}

int main(){
    int matrix[rows][cols] = {
            {1, 2, 3, 4},
            {2, 4, 5, 8},
            {3, 6, 7, 9},
            {4, 8, 9, 10}
    };
    if (findNumInSortedMatrix(7, matrix)) {
        printf("find!");
    } else {
        printf("not exist!");
    }
    return 0;
}
```

岛问题

一个矩阵中只有0和1两种值，每个位置都可以和自己的上、下、左、右四个位置相连，如果有一片1连在一起，这个部分叫做一个岛，求一个矩阵中有多少个岛?

比如矩阵:

1	**0**	**1**
0	1	0
1	1	1

就有3个岛。

> 分析：我们可以遍历矩阵中的每个位置，如果遇到1就将与其相连的一片1都感染成2，并自增岛数量。

```
public class IslandNum {

    public static int getIslandNums(int matrix[][]){
        int res = 0 ;
        for(int i = 0 ; i < matrix.length ; i++){
            for(int j = 0 ; j < matrix[i].length ; j++){
                if(matrix[i][j] == 1){
                    res++;
                    infect(matrix , i , j);
                }
            }
        }
        return res;
    }

    public static void infect(int matrix[][], int i ,int j){
```

```
        if(i < 0 || i >= matrix.length || j < 0 || j >= matrix[i].length ||
matrix[i][j] != 1){
            return;
        }
        matrix[i][j] = 2;
        infect(matrix , i-1 , j);
        infect(matrix , i+1 , j);
        infect(matrix , i , j-1);
        infect(matrix , i , j+1);
    }

    public static void main(String[] args){
        int matrix[][] = {
            {1,0,0,1,0,1},
            {0,1,1,0,0,0},
            {1,0,0,0,1,1},
            {1,1,1,1,1,1}
        };
        System.out.println(getIslandNums(matrix));
    }
}
```

经典结构和算法

字符串

KMP算法

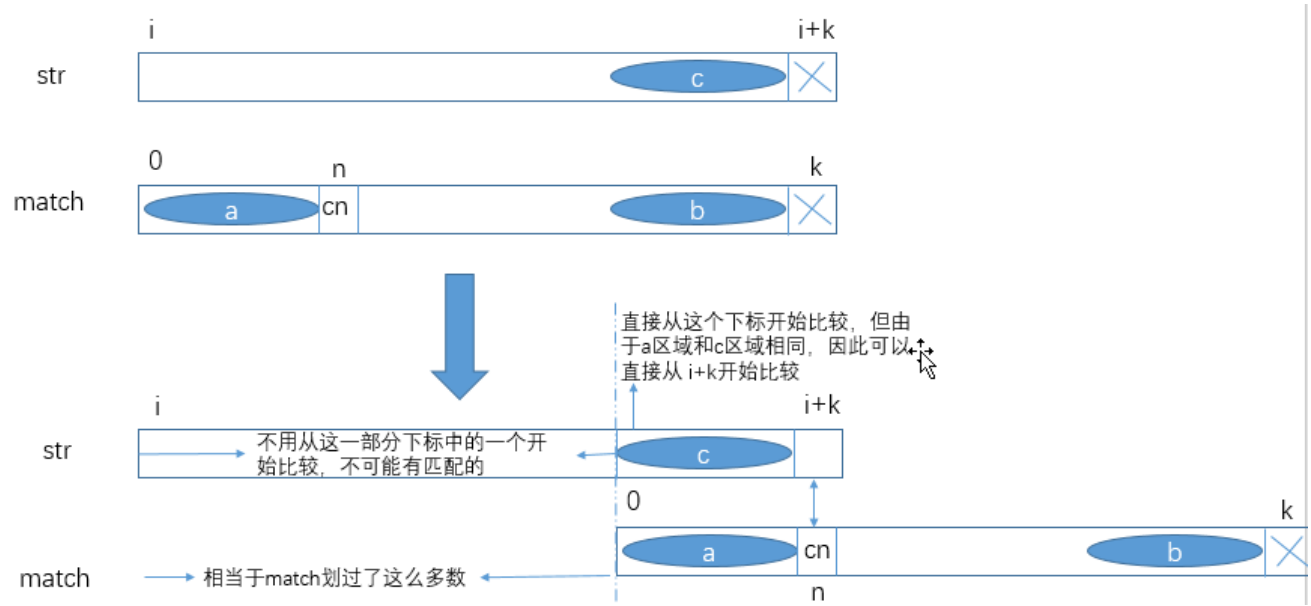
KMP算法是由一个问题而引发的：对于一个字符串`str`（长度为N）和另一个字符串`match`（长度为M），如果`match`是`str`的子串，请返回其在`str`第一次出现时的首字母下标，若`match`不是`str`的子串则返回`-1`。

最简单的方法是将`str`从头开始遍历并与`match`逐次比较，若碰到了不匹配字母则终止此次遍历转而从`str`的第二个字符开始遍历并与`match`逐次比较，直到某一次的遍历每个字符都与`match`匹配否则返回`-1`。易知此种做法的时间复杂度为`O(N*M)`。

> KMP算法则给出求解该问题时间复杂度控制在`O(N)`的解法。

首先该算法需要对应`match`创建一个与`match`长度相同的辅助数组`help[match.length]`，该数组元素表示`match`某个下标之前的子串的**前后缀子串最大匹配长度。前缀子串**表示一个串中以串首字符开头的不包含串尾字符的任意个连续字符，**后缀子串**则表示一个串中以串尾字符结尾的不包括串首字符的任意个连续字符。比如`abcd`的前缀子串可以是`a`、`ab`、`abc`，但不能是`abcd`，而`abcd`的后缀字串可以是`d`、`cd`、`bcd`，但不能是`abcd`。再来说一下`help`数组，对于`char match[]="abc1abc2"`来说，有`help[7]=3`，因为`match[7]='2'`，因此`match`下标在`7`之前的子串`abc1abc`的前缀子串和后缀子串相同的情况下，前缀子串的最大长度为3（即前缀字串和后缀字串都取`abc`）；又如`match="aaaab"`，有`help[4]=3`（前缀子串和后缀子串最大匹配长度当两者为`aaa`时取得），相应的有`help[3]=2`、`help[2]=1`。

假设当要寻找的子串 `match` 的 `help` 数组找到之后（对于一个串的 `help` 数组的求法在介绍完 `KMP` 算法之后再详细说明）。就可以进行 `KMP` 算法求解此问题了。 `KMP` 算法的逻辑（结论）是，对于 `str` 的 `i~(i+k)` 部分（ `i` 、 `i+k` 均为 `str` 的合法下标）和 `match` 的 `0~k` 部分（ `k` 为 `match` 的合法下标），如果有 `str[i]=match[0]` 、 `str[i+1]=match[1]` ..... `str[i+k-1]=match[k-1]` ，但 `str[i+k]!=[k]` ，那么 `str` 的下标不用从 `i+k` 变为 `i+1` 重新比较，只需将子串 `str[0]~str[i+k-1]` 的最大匹配前缀子串的后一个字符 `cn` 重新与 `str[i+k]` 向后依次比较，后面如果又遇到了不匹配的字符重复此操作即可：



当遇到不匹配字符时，常规的做法是将 `str` 的遍历下标 `sIndex` 移到 `i+1` 的位置并将 `match` 的遍历下标 `mIndex` 移到 `0` 再依次比较，这种做法并没有利用上一轮的比较信息（对下一轮的比较没有任何优化）。而 `KMP` 算法则不是这样，当遇到不匹配的字符 `str[i+k]` 和 `match[k]` 时， `str` 的遍历指针 `sIndex=i+k` 不用动，将 `match` 右滑并将其遍历指针 `mIndex` 打到子串 `match[0]~match[k-1]` 的最大匹配前缀子串的后一个下标 `n` 的位置。然后 `sIndex` 从 `i+k` 开始， `mIndex` 从 `n` 开始，依次向后比较，若再遇到不匹配的数则重复此过程。

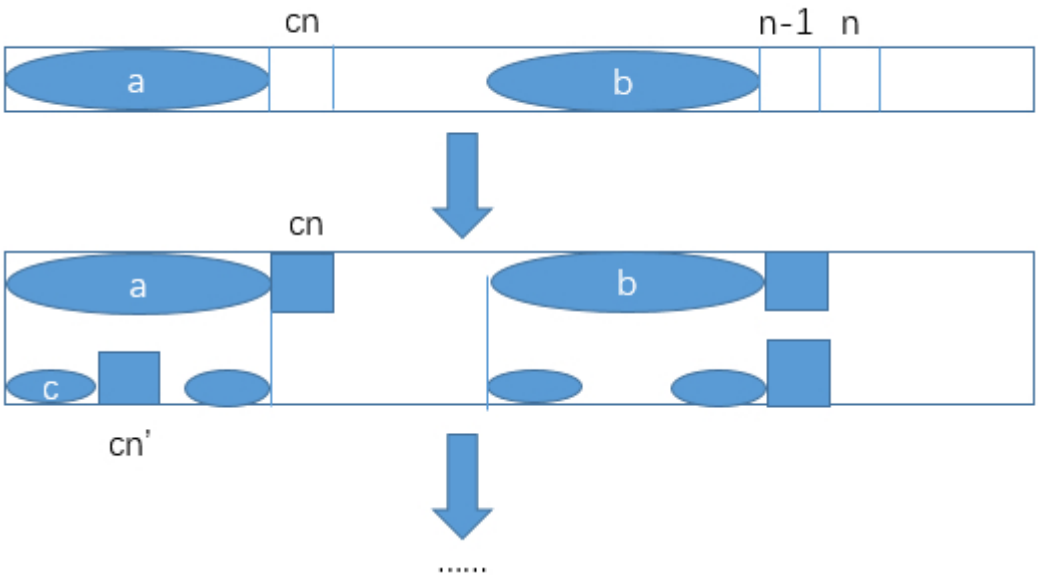
对应代码如下：

```
void length(char* str){
  if(str==NULL)
    return -1;
  int len=0;
  while(*(str++)!='\0'){
    len++;
  }
  return len;
}

int getIndexOf(char* str,char* m){
    int slen = length(str) , mlen = length(m);
    if(mlen > slen)
        return -1;
    int help[mlen];
    getHelpArr(str,help);
    int i=0,j=0;    //sIndex,mIndex
    while(i < slen && j < mlen){
        if(str[i] == m[j]){
            i++;
```

```
            j++;
        }else if(help[j] != -1){
            j = help[j];    //mIndex -> cn's index
        }else{  //the first char is not match,move the sIndex
            i++;
        }
    }
    return j == mlen ? i - mlen : -1;
}
```

可以发现 KMP 算法中 str 的遍历指针并没有回溯这个动作（只向后移动），当完成匹配时 sIndex 的移动次数小于 N，否则 sIndex 移动到串尾也会终止循环，所以 while 对应的匹配过程的时间复杂度为 O(N)（ if(help[j] != -1){ j = help[j] } 的执行次数只会是常数次，因此可以忽略）。

下面只要解决如何求解一个串的 help 数组，此问题就解决了。 help 数组要从前到后求解，直接求 help[n] 是很难有所头绪的。当串 match 长度 mlen=1 时，规定 help[0]=-1 。当 mlen=2 时，去掉 match[1] 之后只剩下 match[0] ，最大匹配子串长度为0（因为前缀子串不能包含串尾字符，后缀子串不能包含串首字符），即 help[1]=0 。当 mlen>2 时， help[n] （n>=2）都可以推算出来：



如上图所示，如果我们知道了 help[n-1] ，那么 help[n] 的求解有两种情况：如果 match[cn]=match[n-1] ，那么由a区域与b区域（a、b为子串 match[0~n-2] 的最大匹配前缀子串和后缀字串）相同可知 help[n]=help[n-1]+1 ；如果 match[cn]!=match[n-1] ，那么求a区域中下一个能和b区域后缀子串中匹配的较大的一个，即a区域的最大匹配前缀字串 c区域 ，将 match[n-1] 和c区域的后一个位置（ cn' ）上的字符比较，如果相等则 help[n] 等于c区域的长度+1，而c区域的长度就是 help[cn] （ help 数组的定义如此）；如果不等则将 cn 打到 cn' 的位置继续和 match[n-1] 比较，直到 cn 被打到 0 为止（即 help[cn]=-1 为止），那么此时 help[n]=0 。

对应代码如下：

```
int* getHelpArr(char* s,int help[]){
    if(s==NULL)
        return NULL;
    int slen = length(s);
    help[0]=-1;
```

```
    help[1]=0;
    int index = 2;//help数组从第三个元素开始的元素值需要依次推算
    int cn = 0;      //推算help[2]时，help[1]=0，即s[1]之前的字符组成的串中不存在最大匹配前后子串，那么cn作为最大匹配前缀子串的后一个下标自然就是0了
    while(index < slen){
        if(s[index-1] == s[cn]){    //if match[n-1] == match[cn]
            help[index] = help[index-1] + 1;
            index++;
            cn++;
        }else if(help[cn] == -1){   //cn reach 0
            help[index]=0;
            index++;
            cn++;
        }else{
            cn = help[cn];  //set cn to cn' and continue calculate help[index]
        }
    }
    return help;
}
```

那么这个求解 `help` 数组的过程的时间复杂度如何计算呢？仔细观察克制 `while` 循环中仅涉及到 `index` 和 `cn` 这两个变量的变化：

	第一个if分支	第二个if分支	第三个if分支
index	增大	增大	不变
index-cn	不变	不变	增大

可以发现 `while` 循环执行一次不是 `index` 增大就是 `index-cn` 增大，而 `index < slen`、`index - cn < slen`，即 `index` 最多自增 `M`（`match` 串的长度）次，`index-cn` 最多增加 `M` 次，如此 `while` 最多执行 `M+M` 次，即时间复杂为 `O(2M)=O(M)`。

综上所述，使用 `KMP` 求解此问题的时间复杂度为 `O(M)`（求解 `match` 的 `help` 数组的时间复杂度）+ `O(N)`（匹配的时间复杂度）= `O(N)`（因为 `N > M`）。

KMP算法的应用

1. 判断一个二叉树是否是另一棵二叉树的子树（即某棵树的结构和数据状态和另一棵二叉树的子树样）。

   思路：如果这棵树的序列化串是另一棵树的序列化串的子串，那么前者必定是后者的子树。

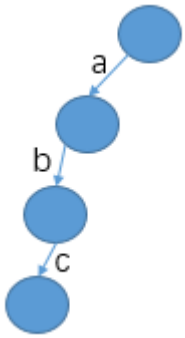
前缀树（字典树）

前缀树的介绍

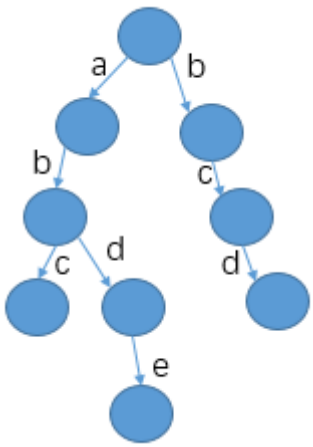
前缀树是一种存储字符串的高效容器，基于此结构的操作有：

- `insert` 插入一个字符串到容器中
- `search` 容器中是否存在某字符串，返回该字符串进入到容器的次数，没有则返回0
- `delete` 将某个字符串进入到容器的次数减1
- `prefixNumber` 返回所有插入操作中，以某个串为前缀的字符串出现的次数

设计思路：该结构的重点实现在于存储。前缀树以字符为存储单位，将其存储在结点之间的树枝上而非结点上，如插入字符串`abc`之后前缀树如下：



每次插入串都要从头结点开始，遍历串中的字符依次向下“铺路”，如上图中的`abc`3条路。对于每个结点而言，它可以向下铺`a~z`26条不同的路，假如来到某个结点后，它要向下铺的路（取决于遍历到哪个字符来了）被之前插入串的过程铺过了那么就可以直接走这条路去往下一个结点，否则就要先铺路再去往下一个结点。如再插入串`abde`和`bcd`的前缀树将如下所示：



根据前缀树的`search`和`prefixNumber`两个操作，我们还需要在每次铺路后记录以下每个结点经过的次数（`across`），以及每次插入操作每个结点作为终点结点的次数（`end`）。

前缀树的实现

前缀树的实现示例：

```
public class TrieTree {

  public static class TrieNode {
    public int across;
    public int end;
    public TrieNode[] paths;

    public TrieNode() {
      super();
      across = 0;
      end = 0;
      paths = new TrieNode[26];
    }
  }
```

```
15
16  private TrieNode root;
17
18  public TrieTree() {
19    super();
20    root = new TrieNode();
21  }
22
23  //向树中插入一个字符串
24  public void insert(String str) {
25    if (str == null || str.length() == 0) {
26      return;
27    }
28    char chs[] = str.toCharArray();
29    TrieNode cur = root;
30    for (char ch : chs) {
31      int index = ch - 'a';
32      if (cur.paths[index] == null) {
33        cur.paths[index] = new TrieNode();
34      }
35      cur = cur.paths[index];
36      cur.across++;
37    }
38    cur.end++;
39  }
40
41  //查询某个字符串插入的次数
42  public int search(String str) {
43    if (str == null || str.length() == 0) {
44      return 0;
45    }
46    char chs[] = str.toCharArray();
47    TrieNode cur = root;
48    for (char ch : chs) {
49      int index = ch - 'a';
50      if (cur.paths[index] == null) {
51        return 0;
52      }else{
53        cur = cur.paths[index];
54      }
55    }
56    return cur.end;
57  }
58
59  //删除一次插入过的某个字符串
60  public void delete(String str) {
61    if (search(str) > 0) {
62      char chs[] = str.toCharArray();
63      TrieNode cur = root;
64      for (char ch : chs) {
65        int index = ch - 'a';
66        if (--cur.paths[index].across == 0) {
67          cur.paths[index] = null;
```

```
68            return;
69          }
70          cur = cur.paths[index];
71        }
72        cur.end--;
73      }
74    }
75
76      //查询所有插入的字符串中，以prefix为前缀的有多少个
77    public int prefixNumber(String prefix) {
78      if (prefix == null || prefix.length() == 0) {
79        return 0;
80      }
81      char chs[] = prefix.toCharArray();
82      TrieNode cur = root;
83      for (char ch : chs) {
84        int index = ch - 'a';
85        if (cur.paths[index] == null) {
86          return 0;
87        }else{
88          cur = cur.paths[index];
89        }
90      }
91      return cur.across;
92    }
93
94    public static void main(String[] args) {
95      TrieTree tree = new TrieTree();
96      tree.insert("abc");
97      tree.insert("abde");
98      tree.insert("bcd");
99      System.out.println(tree.search("abc")); //1
100     System.out.println(tree.prefixNumber("ab"));    //2
101   }
102 }
```

前缀树的相关问题

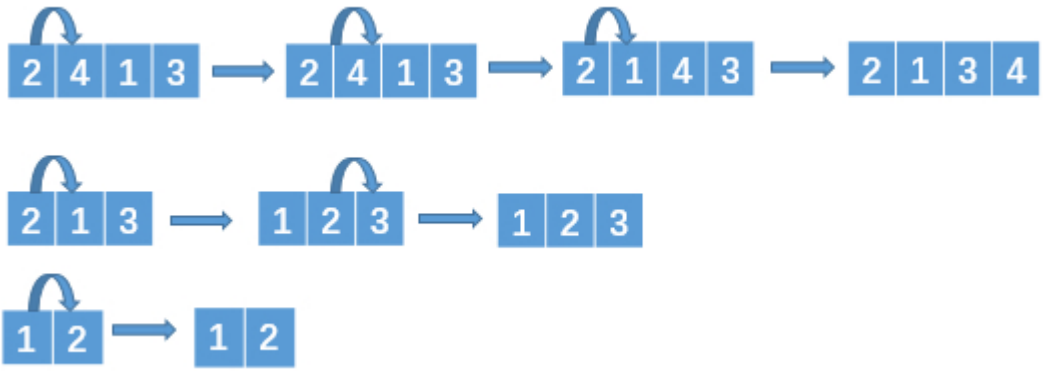
一个字符串类型的数组arr1，另一个字符串类型的数组arr2：

- arr2中有哪些字符，是arr1中出现的？请打印
- arr2中有哪些字符，是作为arr1中某个字符串前缀出现的？请打印
- arr2中有哪些字符，是作为arr1中某个字符串前缀出现的？请打印arr2中出现次数最大的前缀。

数组

冒泡排序

冒泡排序的核心是从头遍历序列。以升序排列为例：将第一个元素和第二个元素比较，若前者大于后者，则交换两者的位置，再将第二个元素与第三个元素比较，若前者大于后者则交换两者位置，以此类推直到倒数第二个元素与最后一个元素比较，若前者大于后者，则交换两者位置。这样一轮比较下来将会把序列中最大的元素移至序列末尾，这样就安排好了最大数的位置，接下来只需对剩下的（n-1）个元素，重复上述操作即可。

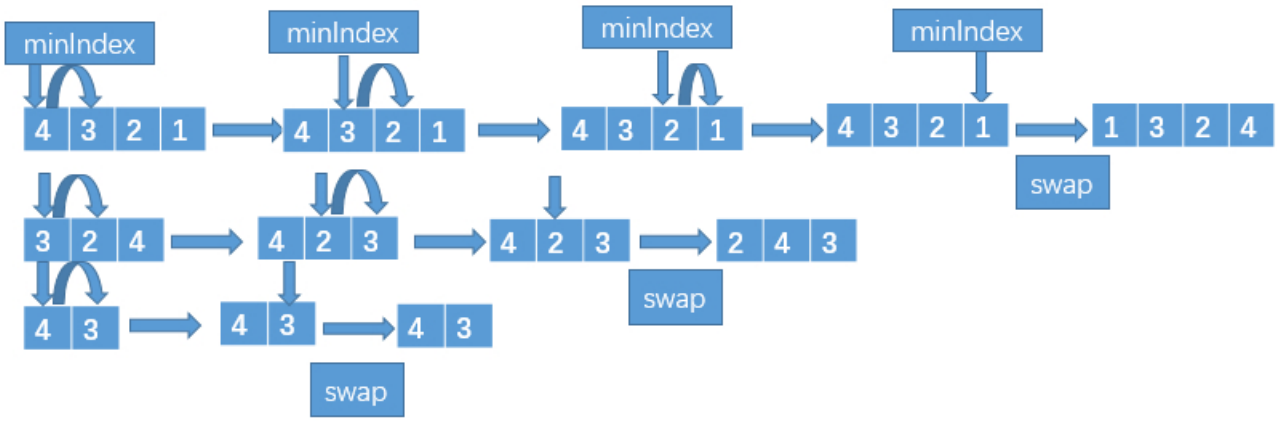



```c
void swap(int *a, int *b){
  int temp = *a;
  *a = *b;
  *b = temp;
}

void bubbleSort(int arr[], int length) {
  if(arr==NULL || length<=1){
    return;
  }
  for (int i = length-1; i > 0; i--) {  //只需比较(length-1)轮
    for (int j = 0; j < i; ++j) {
      if (arr[j] > arr[j + 1]) {
        swap(&arr[j], &arr[j + 1]);
      }
    }
  }
}
```

该算法的时间复杂度为`n+(n-1)+...+1`，很明显是一个等差数列，由（首项+末项）*项数/2求其和为`(n+1)n/2`，可知时间复杂度为`O(n^2)`

选择排序

以升序排序为例：找到最小数的下标`minIndex`，将其与第一个数交换，接着对子序列（1-n）重复该操作，直到子序列只含一个元素为止。(即选出最小的数放到第一个位置，该数安排好了，再对剩下的数选出最小的放到第二个位置，以此类推)

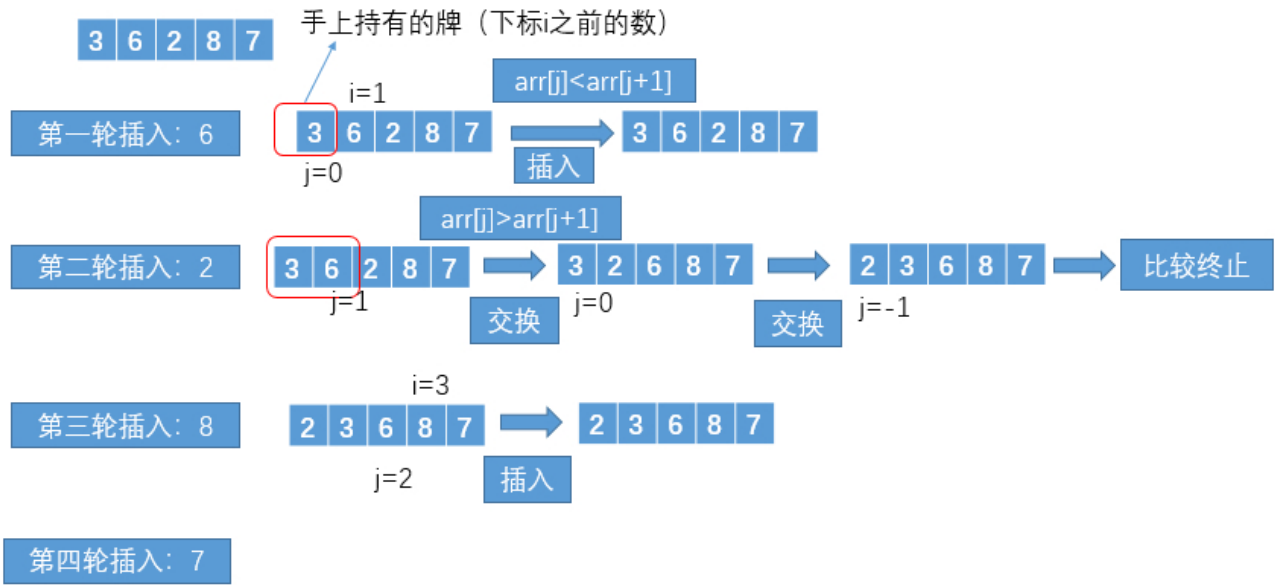


```
void selectionSort(int arr[], int length) {
  for (int i = 0; i < length-1; ++i) {    //要进行n-1次选择，选出n-1个数分别放在前n-1个位置上
    if(arr==NULL || length<=1){
      return;
    }
    int minIndex = i;   //记录较小数的下标
    for (int j = i+1; j < length; ++j) {
      if (arr[minIndex] > arr[j]) {
        minIndex = j;
      }
    }
    if (minIndex != i) {
      swap(&arr[minIndex],&arr[i]);
    }
  }
}
```

同样，不难得出该算法的时间复杂度（big o）为`o(n^2)`（n-1+n-2+n-3+...+1）

插入排序

插入排序的过程可以联想到打扑克时揭一张牌然后将其到手中有序纸牌的合适位置上。比如我现在手上的牌是7、8、9、J、Q、K，这时揭了一张10，我需要将其依次与K、Q、J、9、8、7比较，当比到9时发现大于9，于是将其插入到9之后。对于一个无序序列，可以将其当做一摞待揭的牌，首先将首元素揭起来，因为揭之前手上无牌，因此此次揭牌无需比较，此后每揭一次牌都需要进行上述的插牌过程，当揭完之后，手上的握牌顺序就对应着该序列的有序形式。




```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
void insertionSort(int arr[], int length){
```

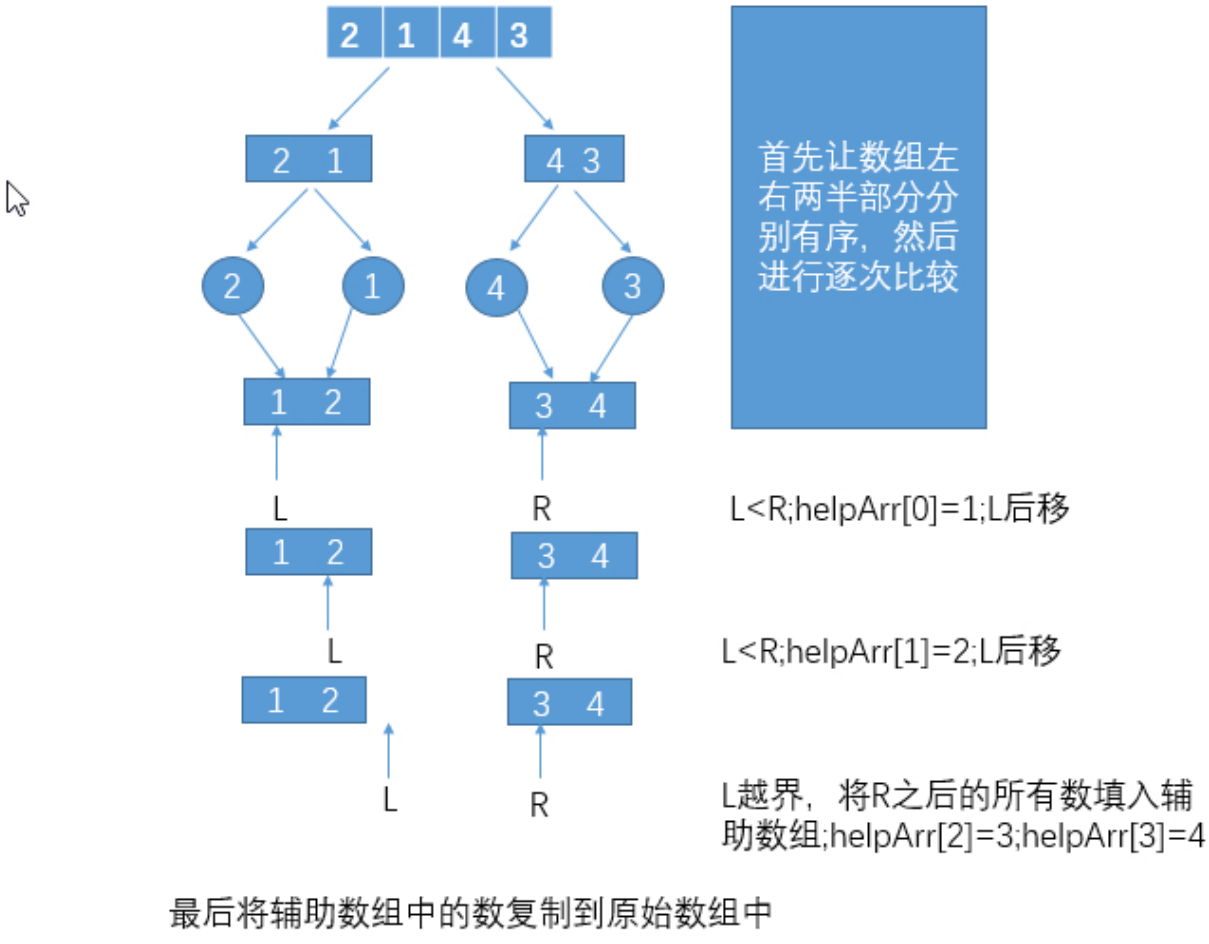
```
    if(arr==NULL || length<=1){
        return;
    }
    for (int i = 1; i < length; ++i) {      //第一张牌无需插入，直接入手，后续揭牌需比较然后插入，因此从第二个元素开始遍历（插牌）
        //将新揭的牌与手上的逐次比较，若小于则交换，否则停止，比较完了还没遇到更小的也停止
        for (int j = i - 1; j >= 0 || arr[j] <= arr[j + 1]; j--) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}
```

插入排序的big o该如何计算？可以发现如果序列有序，那么该算法的big o为`O(n)`，因为只是遍历了一次序列（这时最好情况）；如果序列降序排列，那么该算法的big o为`O(n^2)`（每次插入前的比较交换加起来要：1+2+...+n-1）（最坏情况）。**一般应用场景中都是按算法的最坏情况来考量算法的效率的，因为你做出来的应用要能够承受最坏情况。**即该算法的big o为`O(n^2)`

归并排序

归并排序的核心思想是先让序列的左半部分有序、再让序列的右半部分有序，最后从两个子序列（左右两半）从头开始逐次比较，往辅助序列中填较小的数。

以序列`{2,1,4,3}`为例，归并排序的过程大致如下：



算法代码示例：

```
void merge(int arr[],int helpArr[], int startIndex, int midIndex,int endIndex) {
    int L = startIndex, R = midIndex + 1, i = startIndex;
    while (L <= midIndex && R <= endIndex) { //只要没有指针没越界就逐次比较
        helpArr[i++] = arr[L] < arr[R] ? arr[L++] : arr[R++];
    }
    while (L != midIndex + 1) {
        helpArr[i++] = arr[L++];
    }
    while (R != endIndex + 1) {
        helpArr[i++] = arr[R++];
    }
    for (i = startIndex; i <= endIndex; i++) {
        arr[i] = helpArr[i];
    }
}

void mergeSort(int arr[],int helpArr[], int startIndex, int endIndex) {
    int midIndex;
    if (startIndex < endIndex) {  //当子序列只含一个元素时，不再进行此子过程
        //(endIndex+startIndex)/2可能会导致int溢出，下面求中位数的做法更安全
        midIndex = startIndex + ((endIndex - startIndex) >> 1);
        mergeSort(arr, helpArr, startIndex, midIndex);        //对左半部分排序
        mergeSort(arr, helpArr, midIndex + 1, endIndex);      //对右半部分排序
        merge(arr, helpArr, startIndex, midIndex, endIndex);  //使整体有序
    }
}

int main(){
    int arr[] = {9, 1, 3, 4, 7, 6, 5};
    travels(arr, 7);//遍历打印
    int helpArr[7];
    mergeSort(arr, helpArr, 0, 7);
    travels(arr, 7);

    return 0;
}
```

此算法的核心就是第 `24、25、26` 这三行。第 `26` 行应该不难理解，就是使用两个指针 `L、R` 外加一个辅助数组，将两个序列有序地**并入**辅助数组。但为什么 `24、25` 行执行过后数组左右两半部分就分别有序了呢？这就又牵扯到了归并排序的核心思想：先让一个序列左右两半部分有序，然后再并入使整体有序。因此 `24、25` 是对左右两半部分分别递归执行归并排序，直到某次递归时左右两半部分均为一个元素时递归终止。当一个序列只含两个元素时，调用 `mergeSort` 会发现 `24、25` 行是无效操作，直接执行 `merge`。就像上图所示，两行递归完毕后，左右两半部分都会变得有序。

> 当一个递归过程比较复杂时（不像递归求阶乘那样一幕了然），我们可以列举简短样本进行分析。
>
> 对于这样复杂的递归行为，千万不要想着追溯整个递归过程，只需分析第一步要做的事（比如此例中第一步要做的是就是 `mergeSort` 函数所呈现出来的那样：对左半部分排序、对右半部分排序、最后并入，你先不管是怎么排序的，不要被24、25行的 `mergeSort` 给带进去了）和递归终止的条件（比如此例中是 `` `startIndex>=endIndex ``，即要排序的序列只有一个元素时）。

归并排序的时间复杂度是`O(nlogn)`，额外空间复杂度是`O(n)`。

根据**Master公式**（本文 **小技巧**一节中有讲到）可得`T(n)=2T(n/2)+O(n)`，第一个2的含义是子过程（对子序列进行归并排序）要执行两次，第二个2的含义是子过程样本量占一半（因为分成了左右两半部分），最后`O(n)`表示左右有序之后进行的并入操作为`O(n+n)=O(n)`(L、R指针移动次数总和为n，将辅助数组覆盖源数组为n)，符合`T(n)=aT(n/b)+O(n^d)`，经计算该算法的时间复杂度为`O(nlogn)`

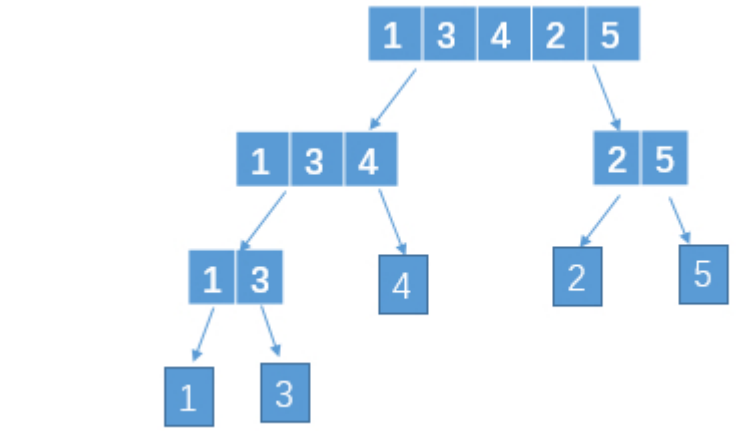
小和问题

在一个数组中，每一个数左边比当前数小的数累加起来，叫做这个数组的小和。求一个数组的小和。例如：

```
对于数组[1,3,4,2,5]
1左边比1小的数，没有；
3左边比3小的数，1；
4左边比4小的数，1、3；
2左边比2小的数，1；
5左边比5小的数，1、3、4、2；
所以小和为1+1+3+1+1+3+4+2=16
```

简单的做法就是遍历一遍数组，将当前遍历的数与该数之前数比较并记录小于该数的数。易知其时间复杂度为`O(n^2)`（0+1+2+.....+n-1）。

更优化的做法是利用归并排序的**并入逻辑**：



并入　1并入3　1<3，sum+=1

13并入4 1<4，sum+=1，3<4，sum+=3

2并入5　2<5，sum+=2

134并入25 1<2和5，sum+=1*2，

3>2，2入helpArr

3<5，sum+=3

4<5，sum+=4

sum=1+1+3+2+1*2+3+4=16

对应代码：

```c
int merge(int arr[],int helpArr[], int startIndex, int midIndex,int endIndex) {
    int L = startIndex, R = midIndex + 1, i = startIndex;
    int res=0;
    while (L <= midIndex && R <= endIndex ) { //只要没有指针没越界就逐次比较
        res += arr[L] < arr[R] ? arr[L] * (endIndex - R + 1) : 0;
        helpArr[i++] = arr[L] < arr[R] ? arr[L++] : arr[R++];
    }
    while (L != midIndex + 1) {
        helpArr[i++] = arr[L++];
    }
    while (R != endIndex + 1) {
        helpArr[i++] = arr[R++];
    }
    for (i = startIndex; i <= endIndex; i++) {
        arr[i] = helpArr[i];
    }
    return res;
}

int mergeSort(int arr[],int helpArr[], int startIndex, int endIndex) {
    int midIndex;
    if (startIndex < endIndex) {  //当子序列只含一个元素时，不再进行此子过程
        midIndex = startIndex + ((endIndex - startIndex) >> 1);
        return mergeSort(arr, helpArr, startIndex, midIndex) +         //对左半部分排序
               mergeSort(arr, helpArr, midIndex + 1, endIndex) +     //对右半部分排序
               merge(arr, helpArr, startIndex, midIndex, endIndex);  //使整体有序
    }
    return 0;   //一个元素时不存在小和
}

int main(){
    int arr[] = {1,3,4,2,5};
    int helpArr[5];
    printf("small_sum:%d\n",mergeSort(arr, helpArr, 0, 4)) ;
    return 0;
}
```

该算法在归并排序的基础上做了略微改动，即`merge`中添加了变量`res`记录每次**并入**操作应该累加的小和、`mergeSort`则将每次并入应该累加的小和汇总。此种做法的复杂度与归并排序的相同，优于遍历的做法。可以理解，依次求每个数的小和过程中有很多比较是重复的，而利用归并排序求小和时利用了并入的两个序列分别有序的特性省去了不必要的比较，如`134并入25`时，`2>1`直接推出`2`后面的数都`>1`，因此直接`1*(endIndex-indexOf(2)+1)`即可。这在样本量不大的情况下看不出来优化的效果，试想一下如果样本量为`2^32`，那么依照前者求小和`O(n^2)`可知时间复杂度为`O(21亿的平方)`，而归并排序求小和则只需`O(21亿*32)`，足以见得`O(n^2)`和`O(nlogn)`的优劣。

逆序对问题

在一个数组中，左边的数如果比右边的数大，则这两个数构成一个逆序对，请打印所有逆序对。

> 这题的思路也可以利用归并排序来解决，在并入操作时记录`arr[L]>arr[R]`的情况即可。

快速排序

经典快排

经典快排就是将序列中比尾元素小的移动到序列左边，比尾元素大的移动到序列右边，对以该元素为界的左右两个子序列（均不包括该元素）重复此操作。

首先我们要考虑的是对给定的一个数，如何将序列中比该数小的移动到左边，比该数大的移动到右边。

> 思路：利用一个辅助指针`small`，代表较小数的右边界（初始指向首元素前一个位置），遍历序列每次遇到比该数小的数就将其与`arr[small+1]`交换并右移`small`，最后将该数与`arr[small+1]`交换即达到目的。对应算法如下：

```
void partition(int arr[], int startIndex, int endIndex){
    int small = startIndex - 1;
    for (int i = startIndex; i < endIndex; ++i) {
        if(arr[i] < arr[endIndex]) {
            if (small + 1 != i) {
                swap(arr[++small], arr[i]);
            } else {
                //如果small、i相邻则不用交换
                small++;
            }
        }
    }
    swap(arr[++small], arr[endIndex]);
}
int main(){
    int arr[] = {1, 2, 3, 4, 6, 7, 8, 5};
    travles(arr, 8);//1 2 3 4 6 7 8 5
    partition(arr, 0, 7);
    travles(arr, 8);//1 2 3 4 5 7 8 6
    return 0;
}
```

接着就是快排的递归逻辑：对`1 2 3 4 6 7 8 5`序列`partition`之后，去除之前的比较参数`5`，对剩下的子序列`1234`和`786`继续`partition`，直到子序列为一个元素为止：

```
int partition(int arr[], int startIndex, int endIndex){
    int small = startIndex - 1;
    for (int i = startIndex; i < endIndex; ++i) {
        if(arr[i] < arr[endIndex]) {
            if (small + 1 != i) {
                swap(arr[++small], arr[i]);
            } else {
                //如果small、i相邻则不用交换
                small++;
            }
        }
    }
    swap(arr[++small], arr[endIndex]);
    return small;
}

```

```
void quickSort(int arr[], int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return;
    }
    int index = partition(arr, startIndex, endIndex);
    quickSort(arr, startIndex, index - 1);
    quickSort(arr, index + 1, endIndex);
}
int main(){
    int arr[] = {1, 5, 6, 2, 7, 3, 8, 0};
    travles(arr, 8);    //1 5 6 2 7 3 8 0
    quickSort(arr, 0,7);
    travles(arr, 8);    //0 1 2 3 5 6 7 8
    return 0;
}
```

经典排序的时间复杂度与数据状况有关，如果**每一次** `partition` 时，尾元素都是序列中最大或最小的，那么去除该元素序列并未如我们划分为样本量相同的左右两个子序列，而是只安排好了一个元素（就是去掉的那个元素），这样的话时间复杂度就是 `O(n-1+n-2+……+1)=O(n^2)` ；但如果每一次 `partition` 时，都将序列分成了两个样本量相差无几的左右两个子序列，那么时间复杂度就是 `O(nlogn)` （使用Master公式求解）。

由荷兰国旗问题引发对经典快排的改进

可以发现这里 `partition` 的过程与荷兰国旗问题中的 `partition` 十分相似，能否以后者的 `partition` 实现经典快排呢？我们来试一下：

```
int* partition(int arr[], int startIndex, int endIndex){ ;
    int small = startIndex - 1, great = endIndex + 1, i = startIndex;
    while (i <= great - 1) {
        if (arr[i] < arr[endIndex]) {
            swap(arr[++small], arr[i++]);
        } else if (arr[i] > arr[endIndex]){
            swap(arr[--great], arr[i]);
        } else {
            i++;
        }
    }
    int range[] = {small, great};
    return range;
}

void quickSort(int arr[], int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return;
    }
    int* range = partition(arr, startIndex, endIndex);
    quickSort(arr, startIndex, range[0]);
    quickSort(arr, range[1], endIndex);
}

int main(){
    int arr[] = {1, 5, 6, 2, 7, 3, 8, 0};
```

```
    travles(arr, 8);    //1 5 6 2 7 3 8 0
    quickSort(arr, 0,7);
    travles(arr, 8);    //0 1 2 3 5 6 7 8
    return 0;
}
```

比较一下经典排序和使用荷兰国旗问题改进后的经典排序，不难发现，后者一次`partition`能去除一个以上的元素（等于`arr[endIndex]`的区域），而前者每次`partition`只能去除一个元素，这里的去除相当于安排（排序）好了对应元素的位置。因此后者比经典排序更优，但是优化不大，只是常数时间内的优化，实质上的效率还是要看数据状况（最后的情况为`O(nlogn)`，最坏的情况为`O(n^2)`）。

随机快排——O(nlogn)

上面谈到了快排的短板是依赖数据状况，那么我们有没有办法消除这个依赖，让他成为真正的`O(nlogn)`呢?

> 事实上，为了让算法中的操作不依托于数据状况（如快排中每一次`partition`取尾元素作为比较，这就没有规避样本的数据状况，如果尾元素是最大或最小值就成了最坏情况）常常有两种做法：
>
> 1、使用随机取数
>
> 2、将样本数据哈希打乱

随机快排就是采用上了上述第一种解决方案，在每一轮的`partition`中随机选择序列中的一个数作为要比较的数：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

//产生[startIndex,endIndex]之间的随机整数
int randomInRange(int startIndex,int endIndex){
    return rand() % (endIndex - startIndex + 1) + startIndex;
}

int* partition(int arr[], int startIndex, int endIndex){ ;
    int small = startIndex - 1, great = endIndex + 1, i = startIndex;
    int randomNum = arr[randomInRange(startIndex, endIndex)];
    while (i <= great - 1) {
        if (arr[i] < randomNum) {
            swap(arr[++small], arr[i++]);
        } else if (arr[i] > randomNum){
            swap(arr[--great], arr[i]);
        } else {
            i++;
        }
    }
    int range[] = {small, great};
```

```
    return range;
}

void quickSort(int arr[], int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return;
    }
    int* range = partition(arr, startIndex, endIndex);
    quickSort(arr, startIndex, range[0]);
    quickSort(arr, range[1], endIndex);
}

void travles(int dataArr[], int length){
    for (int i = 0; i < length; ++i) {
        printf("%d ", dataArr[i]);
    }
    printf("\n");
}

int main(){
    srand(time(NULL));//此后调用rand()时将以调用时的时间为随机数种子
    int arr[] = {9,7,1,3,2,6,8,4,5};
    travles(arr, 9);
    quickSort(arr, 0,8);
    travles(arr, 9);
    return 0;
}
```

观察比较代码可以发现随机快排只不过是在`partition`时随机选出一个下标上的数作为比较对象，从而避免了每一轮选择尾元素会受数据状况的影响的问题。

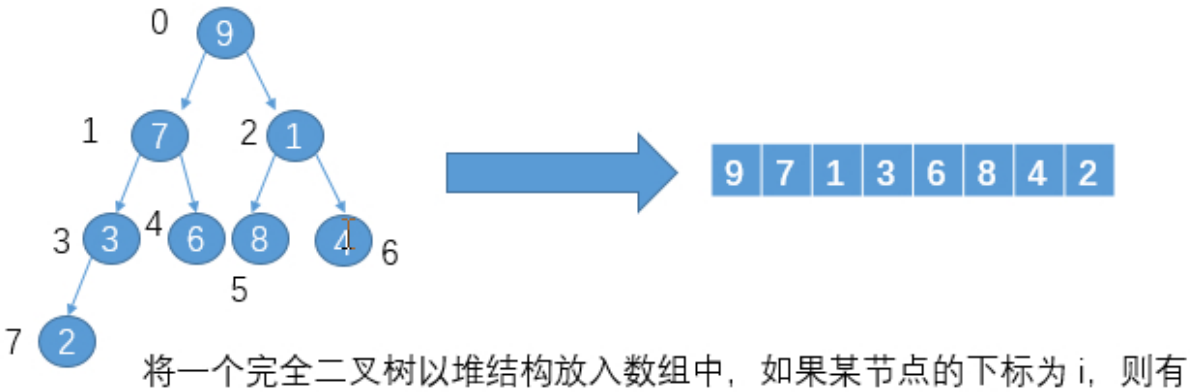
> 那么随机快排的时间复杂度又为多少呢?

经数学论证，由于每一轮`partition`选出的作为比较对象的数是随机的，即序列中的每个数都有`1/n`的概率被选上，那么该算法时间复杂度为概率事件，经数学论证该算法的**数学期望**为`O(nlogn)`。虽然说是数学期望，但在实际工程中，常常就把随机快排的时间复杂度当做`O(nlog)`。

堆排序

什么是堆

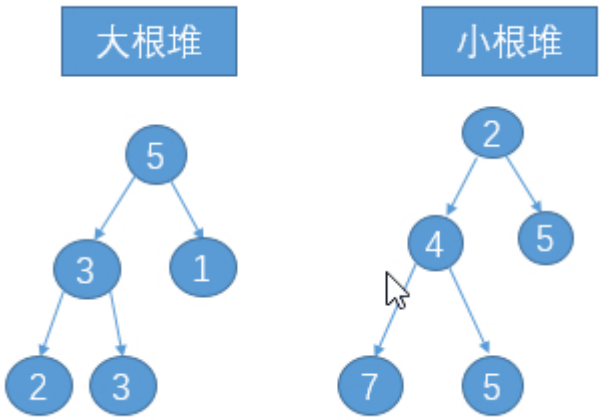
堆结构就是将一颗**完全二叉树**映射到数组中的一种存储方式:



将一个完全二叉树以堆结构放入数组中，如果某节点的下标为 i，则有
该节点的父节点=(i-1)/2
该节点的左孩子节点=2*i+1
该节点的右孩子节点=2*i+2

大根堆和小根堆

当堆的每一颗子树（包括树本身）的最大值就是其结点时称为大根堆；相反，当堆的每一颗子树的最小值就是其根结点时称为小根堆。其中大根堆的应用较为广泛，是一种很重要的数据结构。



heapInsert和heapify

大根堆最重要的两个操作就是`heapInsert`和`heapify`，前者是当一个元素加入到大根堆时应该自底向上与其父结点比较，若大于父结点则交换；后者是当堆中某个结点的数值发生变化时，应不断向下与其孩子结点中的最大值比较，若小于则交换。下面是对应的代码：

```
//index之前的序列符合大根堆排序，将index位置的元素加入堆结构，但不能破坏大根堆的特性
void heapInsert(int arr[],int index){
    while (arr[index] > arr[(index - 1) / 2]) { //当该结点大于父结点时
        swap(arr[index], arr[(index - 1) / 2]);
        index = (index - 1) / 2;    //继续向上比较
    }
}

//数组中下标从0到heapSize符合大根堆排序
//index位置的值发生了变化，重新调整堆结构为大根堆
//heapSize指的是数组中符合大根堆排序的范围而不是数组长度,最大为数组长度，最小为0
void heapify(int arr[], int heapSize, int index){
```

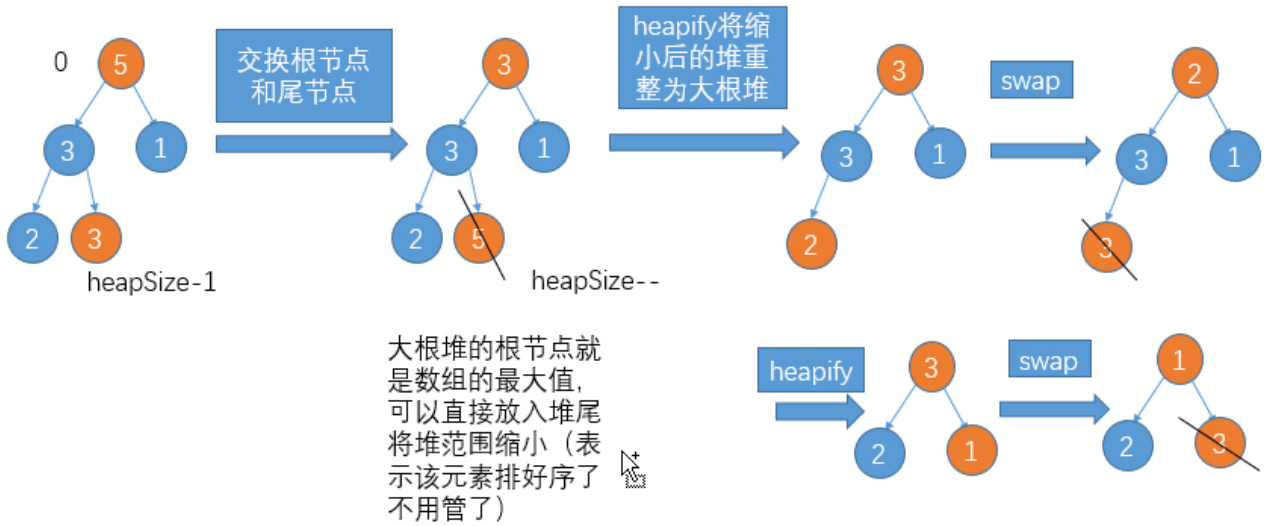
```
    int leftChild = index * 2 + 1;
    while (leftChild < heapSize) {  //当该结点有左孩子时
        int greatOne = leftChild + 1 < heapSize && arr[leftChild + 1] > arr[leftChild] ?
                leftChild + 1 : leftChild;  //只有当右孩子存在且大于左孩子时，最大值是右孩子，否则是左孩子
        greatOne = arr[greatOne] > arr[index] ? greatOne : index;//将父结点与最大孩子结点比较，确定最大值
        if (greatOne == index) {
            //如果最大值是本身，则不用继续向下比较
            break;
        }
        swap(arr[index], arr[greatOne]);

        //next turn下一轮
        index = greatOne;
        leftChild = index * 2 + 1;
    }
}
```

建立大根堆

```
void buildBigRootHeap(int arr[],int length){
    if (arr == NULL || length <= 1) {
        return;
    }
    for (int i = 0; i < length; ++i) {
        heapInsert(arr, i);
    }
}
```

利用heapify排序

前面做了那么多铺垫都是为了建立大根堆，那么如何利用它来排序呢?



对应代码实现如下:

```
void heapSort(int arr[],int length){
    if (arr == NULL || length <= 1) {
        return;
    }
    //先建立大根堆
    for (int i = 0; i < length; ++i) {
        heapInsert(arr, i);
    }
      //循环弹出堆顶元素并heapify
    int heapSize = length;
    swap(arr[0], arr[--heapSize]);//相当于弹出堆顶元素
    while (heapSize > 0) {
        heapify(arr, heapSize, 0);
        swap(arr[0], arr[--heapSize]);
    }
}

int main(){
    int arr[] = {9,7,1,3,6,8,4,2,5};
    heapSort(arr, 9);
    travles(arr, 9);
    return 0;
}
```

堆排序的优势在于无论是入堆一个元素`heapInsert`还是出堆一个元素之后的`heapify`都不是将整个样本遍历一遍（`O(n)`级别的操作），而是树层次上的遍历（`O(logn)`级别的操作）。

这样的话堆排序过程中，建立堆的时间复杂度为`O(nlogn)`，循环弹出堆顶元素并`heapify`的时间复杂度为`O(nlogn)`，整个堆排序的时间复杂度为`O(nlogn)`，额外空间复杂度为`O(1)`

> 优先级队列结构（比如Java中的`PriorityQueue`）就是堆结构。

排序算法的稳定性

排序算法的稳定性指的是排序前后是否维持值相同的元素在序列中的相对次序。如序列`271532`，在排序过程中如果能维持第一次出现的`2`在第二次出现的`2`的前面，那么该排序算法能够保证稳定性。首先我们来分析一下前面所讲排序算法的稳定性，再来谈谈稳定性的意义。

- **冒泡排序**。可以保证稳定性，只需在比较相邻两个数时只在后一个数比前一个数大的情况下才交换位置即可。
- **选择排序**。无法保证稳定性，比如序列`926532`，在第一轮`maxIndex`的选择出来之后（`maxIndex=0`），第二次出现的`2`（尾元素）将与`9`交换位置，那么两个`2`的相对次序就发生了变化，而这个交换是否会影响稳定性在我们`coding`的时候是不可预测的。
- **插入排序**。可以保证稳定性，每次插入一个数到有序序列中时，遇到比它大的就替换，否则不替换。这样的话，值相同的元素，后面插入的就总在前面插入的后面了。
- **归并排序**。可以保证稳定性，在左右两半子序列排好序后的`merge`过程中，比较大小时如果相等，那么优先插入左子序列中的数。
- **快排**。不能保证稳定性，因为`partition`的过程会将比`num`小的与`small`区域的右一个数交换位置，将比`num`大的与`great`区域的左一个数交换位置，而`small`、`great`分居序列两侧，很容易打乱值相同元素的相对次序。

- **堆排序**。不能保证稳定性。二叉树如果交换位置的结点是相邻层次的可以保证稳定性，但堆排序中弹出堆顶元素后的 `heapify` 交换的是第一层的结点和最后一层的结点。

维持稳定性一般是为了满足业务需求。假设下面是一张不同厂商下同一款产品的价格和销售情况表：

品牌	价格	销量
三星	1603	92
小米	1603	74
vivo	1604	92

要求先按价格排序，再按销量排序。如果保证稳定性，那么排序后应该是这样的：

品牌	价格	销量
三星	1603	92
vivo	1604	92
小米	1603	74

即按销量排序后，销量相同的两条记录会保持之前的按价格排序的状态，这样先前的价格排序这个工作就没白做。

比较器的使用

之前所讲的一些算法大都是对基本类型的排序，但实际工程中要排序的对象可能是无法预测的，那么如何实现一个通用的排序算法以应对呢？事实上，之前的排序都可以归类为**基于比较的排序**。也就是说我们只需要对要比较的对象实现一个比较器，然后排序算法基于比较器来排序，这样算法和具体要排序的对象之间就解耦了。以后在排序之前，基于要排序的对象实现一个比较器（定义了如何比较对象大小的逻辑），然后将比较器丢给排序算法即可，这样就实现了复用。

在 `Java`（本人学的是 `Java` 方向）中，这个比较器就是 `Comparator` 接口，我们需要实现其中的 `compare` 方法，对于要排序的对象集合定义一个比较大小的逻辑，然后在构造用来添加这类对象的有序容器时传入这个构造器即可。封装好的容器会在容器元素发生改变时使用我们的比较器来重新组织这些元素。

```
import lombok.AllArgsConstructor;
import lombok.Data;
import java.util.PriorityQueue;
import java.util.Comparator;

public class ComparatorTest {

    @Data
    @AllArgsConstructor
    static class Student {
        private long id;
        private String name;
        private double score;
    }

```

```
    static class IdAscendingComparator implements Comparator<Student> {
        /**
         * 底层排序算法对两个元素比较时会调用这个方法
         * @param o1
         * @param o2
         * @return  若返回正数则认为o1<o2，返回0则认为o1=o2，否则认为o1>o2
         */
        @Override
        public int compare(Student o1, Student o2) {
            return o1.getId() < o2.getId() ? -1 : 1;
        }
    }

    public static void main(String[] args) {
        //大根堆
        PriorityQueue heap = new PriorityQueue(new IdAscendingComparator());
        Student zhangsan = new Student(1000, "zhangsan", 50);
        Student lisi = new Student(999, "lisi", 60);
        Student wangwu = new Student(1001, "wangwu", 50);
        heap.add(zhangsan);
        heap.add(lisi);
        heap.add(wangwu);
        while (!heap.isEmpty()) {
            System.out.println(heap.poll());//弹出并返回堆顶元素
        }
    }
}
```

还有`TreeSet`等，都是在构造是传入比较器，否则将直接根据元素的值（`Java`中引用类型变量的值为地址，比较将毫无意义）来比较，这里就不一一列举了。

有关排序问题的补充

1. **归并排序**可以做到额外空间复杂度为`O(1)`，但是比较难，感兴趣的可以搜 **归并排序 内部缓存法**
2. **快速排序**可以做到保证稳定性，但是很难，可以搜`01 stable sort`（论文）
3. 有一道题是：是奇数放到数组左边，是偶数放到数组右边，还要求奇数和奇数之间、偶数和偶数之间的原始相对次序不变。这道题和归并排序如出一辙，只不过归并排序是将`arr[length-1]`或`arr[randomIndex]`作为比较的标准，而这道题是将是否能整除2作为比较的标准，这类问题都同称为`o1 sort`，要使这类问题做到稳定性，要看`01 stable sort`这篇论文。

工程中的综合排序算法

实际工程中的排序算法一般会将 **归并排序**、**插入排序**、**快速排序**综合起来，集大家之所长来应对不同的场景要求：

- 当要排序的元素为基本数据类型且元素个数较少时，直接使用 **插入排序。**因为在样本规模较小时（比如60），`O(NlogN)`的优势并不明显甚至不及`O(N^2)`，而在`O(N^2)`的算法中，插入排序的常数时间操作最少。
- 当要排序的元素为对象数据类型（包含若干字段），为保证稳定性将采用 **归并排序。**
- 当要排序的元素为基本数据类型且样本规模较大时，将采用 **快速排序。**

桶排序

上一节中所讲的都是基于比较的排序，也即通过比较确定每个元素所处的位置。那么能不能不比较而实现排序呢？这就涉及到了 **桶排序** 这个方法论：准备一些桶，将序列中的元素按某些规则放入翻入对应的桶中，最后根据既定的规则依次倒出桶中的元素。

> 非基于比较的排序，与被排序的样本的实际数据状况有很大关系，所以在实际中并不常用。

计数排序

计数排序是 **桶排序** 方法论的一种实现，即准备一个与序列中元素的数据范围大小相同的数组，然后遍历序列，将遇到的元素作为数组的下标并将该位置上的数加1。例如某序列元素值在0~100之间，请设计一个算法对其排序，要求时间复杂度为`O(N)`。

```c
#include <stdio.h>
void countSort(int arr[],int length){
    int bucketArr[101];
    int i;
    for(i = 0 ; i <= 100 ; i++){
        bucketArr[i]=0; //init buckets
    }
    for(i = 0 ; i < length ; i++){
        bucketArr[arr[i]]++;    //put into buckets
    }
    int count, j=0;
    for(i = 0 ; i <= 100 ; i++) {
        if (bucketArr[i] != 0) { //pour out
            count = bucketArr[i];
            while (count-- > 0) {
                arr[j++] = i;
            }
        }
    }
}

void travels(int arr[], int length){
    for (int i = 0; i < length; ++i) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main(){
    int arr[] = {9, 2, 1, 4, 5, 2, 1, 6, 3, 8, 1, 2};
    travels(arr, 12);//9 2 1 4 5 2 1 6 3 8 1 2
    countSort(arr, 12);
    travels(arr, 12);//1 1 1 2 2 2 3 4 5 6 8 9
    return 0;
}
```

> 如果下次面试官问你有没有事件复杂度比`O(N)`更优的排序算法时，不要忘了计数排序哦！！！

补充问题

1. 给定一个数组，求如果排序后，相邻两数的最大值，要求时间复杂度为`O(N)`，且要求不能用非基于比较的排序。

   这道题的思路比较巧妙：首先为这N个数准备N+1个桶，然后以其中的最小值和最大值为边界将数值范围均分成N等分，然后遍历数组将对应范围类的数放入对应的桶中，下图以数组长度为9举例

   1、为9个数准备10个桶，分别编号0~9
   2、找出最小值（比如0）放入0号桶；找出最大值（比如100）放入9号桶
   3、将最小值~最大值的数值范围（0~100）均分为N等份分别作为每个桶的收集的数的范围
   4、遍历数组将元素放入对应的桶中，并更新该桶的入数记录（所有进入该桶的数的最大值、最小值、是否有数进入过该桶）
   5、数组排序后的相邻两数的最大差值一定当两数分属不同的桶时产生，理由如下：
   　　a、将N个数放入N+1个桶，必然有一个桶是空桶
   　　b、相邻的两个非空桶之间，前一个桶的最大值和后一个桶的最小值一定对应着数组排序后其中相邻的两个数

   这里比较难理解的是：

   - 题目问的是求**如果排序后，相邻两数的最大差值。**该算法巧妙的借助一个空桶（N个数进N+1个桶，必然有一个是空桶），将问题转向了求**两个相邻非空桶**（其中可能隔着若干个空桶）之间前桶的最大值和后桶最小值的差值，而无需在意每个桶中进了哪些数（**只需记录每个桶入数的最大值和最小值以及是否有数**）

   对应代码如下：

```c
#include <stdio.h>

//根据要入桶的数和最大最小值得到对应桶编号
int getBucketId(int num,int bucketsNum,int min,int max){
    return (num - min) * bucketsNum / (max - min);
}

int max(int a, int b){
    return a > b ? a : b;
}

int min(int a, int b){
    return a < b ? a : b;
}

int getMaxGap(int arr[], int length) {
    if (arr == NULL || length < 2) {
        return -1;
    }
    int maxValue = -999999, minValue = 999999;
    int i;
    //找出最大最小值
    for (i = 0; i < length; ++i) {
        maxValue = max(maxValue, arr[i]);
```

```
        minValue = min(minValue, arr[i]);
    }
    //记录每个桶的最大最小值以及是否有数，初始时每个桶都没数
    int maxs[length + 1], mins[length + 1];
    bool hasNum[length + 1];
    for (i = 0; i < length + 1; i++) {
        hasNum[i] = false;
    }
    //put maxValue into the last bucket
    mins[length] = maxs[length] = maxValue;
    hasNum[length] = true;

    //iterate the arr
    int bid; //bucket id
    for (i = 0; i < length; i++) {
        if (arr[i] != maxValue) {
            bid = getBucketId(arr[i], length + 1, minValue, maxValue);
            //如果桶里没数，则该数入桶后，最大最小值都是它，否则更新最大最小值
            mins[bid] = !hasNum[bid] ? arr[i] : arr[i] < mins[bid] ? arr[i] : mins[bid];
            maxs[bid] = !hasNum[bid] ? arr[i] : arr[i] > maxs[bid] ? arr[i] : maxs[bid];
            hasNum[bid] = true;
        }
    }

    //find the max gap between two nonEmpty buckets
    int res = 0, j = 0;
    for (i = 0; i < length; ++i) {
        j = i + 1;//the next nonEmtpy bucket id
        while (!hasNum[j]) {//the last bucket must has number
            j++;
        }
        res = max(res, (mins[j] - maxs[i]));
    }

    return res;
}

int main(){
    int arr[] = {13, 41, 67, 26, 55, 99, 2, 82, 39, 100};
    printf("%d", getMaxGap(arr, 9));    //17
    return 0;
}
```

链表

反转单链表和双向链表

实现反转单向链表和反转双向链表的函数，要求时间复杂度为`O(N)`，额外空间复杂度为`O(1)`

> 此题的难点就是反转一个结点的next指针后，就无法在该结点通过next指针找到后续的结点了。因此每次反转之前需要将该结点的后继结点记录下来。

```
#include<stdio.h>
#include<malloc.h>
#define MAX_SIZE 100

struct LinkNode{
    int data;
    LinkNode* next;
};

void init(LinkNode* &head){
    head = (LinkNode*)malloc(sizeof(LinkNode));
    head->next=NULL;
}

void add(int i,LinkNode* head){
    LinkNode* p = (LinkNode*)malloc(sizeof(LinkNode));
    p->data = i;
    p->next =  head->next;
    head->next = p;
}

void printList(LinkNode* head){
    if(head==NULL)
        return;
    LinkNode* p = head->next;
    while(p != NULL){
        printf("%d ",p->data);
        p = p->next;
    }
    printf("\n");
}
```

```
#include<stdio.h>
#include "LinkList.cpp"

void reverseList(LinkNode *head){
    if(head == NULL)
        return;
    LinkNode* cur = head->next;
    LinkNode* pre = NULL;
    LinkNode* next = NULL;
    while(cur != NULL){
        next = cur->next;
        cur->next = pre;
        pre = cur;
        cur = next;
    }
    //pre -> end node
    head->next = pre;
```

```
    return;
}

int main(){
    LinkNode* head;
    init(head);
    add(1,head);
    add(2,head);
    add(3,head);
    add(4,head);

    printList(head);
    reverseList(head);
    printList(head);
}
```

判断一个链表是否为回文结构

请实现一个函数判断某个单链表是否是回文结构，如`1->3->1`返回`true`、`1->2->2->1`返回`true`、`2->3->1`返回`false`。

> 我们可以利用回文链表前后两半部分逆序的特点、结合栈先进后出来求解此问题。将链表中间结点之前的结点依次压栈，然后从中间结点的后继结点开始遍历链表的后半部分，将遍历的结点与栈弹出的结点比较。

代码示例如下：

```
#include<stdio.h>
#include "LinkList.cpp"
#include "SqStack.cpp"

/*
    判断某链表是否是回文结构
    1、首先找到链表的中间结点（若是偶数个结点则是中间位置的左边一个结点）
    2、使用一个栈将中间结点之前的结点压栈，然后从中间结点的后一个结点开始从栈中拿出结点比较
*/

bool isPalindromeList(LinkNode* head){
    if(head == NULL)
        return false;

    LinkNode *slow = head , *fast = head;
    SqStack* stack;
    init(stack);

    //fast指针每走两步，slow指针才走一步
    while(fast->next != NULL && fast->next->next != NULL){
        fast = fast->next->next;
        slow = slow->next;
        push(slow,stack);
    }

    //链表没有结点或只有一个结点，不是回文结构
    if(isEmpty(stack))
```

```
        return false;

    //判断偶数个结点还是奇数个结点
    if(fast->next != NULL){ //奇数个结点,slow需要再走一步
        slow = slow->next;
    }

    //从slow的后继结点开始遍历链表，将每个结点与栈顶结点比较
    LinkNode* node;
    slow = slow->next;
    while(slow != NULL){
        pop(stack,node);
        //一旦发现有一个结点不同就不是回文结构
        if(slow->data != node->data)
            return false;
        slow = slow->next;
    }
    return true;
}

int main(){

    LinkNode* head;
    init(head);
    add(2,head);
    add(3,head);
    add(3,head);
    add(2,head);
    printList(head);

    if(isPalindromeList(head)){
        printf("是回文链表");
    }else{
        printf("不是回文链表");
    }
    return 0;
}
```

`LinkList.cpp`:

```
#include<stdio.h>
#include<malloc.h>
#define MAX_SIZE 100

struct LinkNode{
    int data;
    LinkNode* next;
};

void init(LinkNode* &head){
    head = (LinkNode*)malloc(sizeof(LinkNode));
    head->next=NULL;
}
```

```
14
15 void add(int i,LinkNode* head){
16     LinkNode* p = (LinkNode*)malloc(sizeof(LinkNode));
17     p->data = i;
18     p->next =  head->next;
19     head->next = p;
20 }
21
22 void printList(LinkNode* head){
23     if(head==NULL)
24         return;
25     LinkNode* p = head->next;
26     while(p != NULL){
27         printf("%d ",p->data);
28         p = p->next;
29     }
30     printf("\n");
31 }
```

`SqStack`:

```
1  #include<stdio.h>
2  #include<malloc.h>
3
4  struct SqStack{
5      LinkNode* data[MAX_SIZE];
6      int length;
7  };
8
9  void init(SqStack* &stack){
10     stack = (SqStack*)malloc(sizeof(SqStack));
11     stack->length=0;
12 }
13
14 bool isEmpty(SqStack* stack){
15     if(stack->length > 0)
16         return false;
17     return true;
18 }
19
20 bool isFull(SqStack* stack){
21     if(stack->length == MAX_SIZE)
22         return true;
23     return false;
24 }
25
26 void push(LinkNode* i,SqStack* stack){
27     if(stack==NULL)
28         return;
29     if(!isFull(stack)){
30         stack->data[stack->length++] = i;
31     }
32 }
```

```
33
34  bool pop(SqStack* stack,LinkNode* &i){
35      if(stack==NULL)
36          return false;
37      if(!isEmpty(stack))
38          i = stack->data[--stack->length];
39          return true;
40  }
```

进阶：要求使用时间复杂度为`O(N)`，额外空间复杂度为`O(1)`求解此问题。

思路：我们可以先将链表的后半部分结点的`next`指针反向，然后从链表的两头向中间推进，逐次比较。（当然了，为了不破坏原始数据结构，我们在得出结论之后还需要将链表指针恢复原样）

```
1   #include<stdio.h>
2   #include "LinkList.cpp"
3   #include "SqStack.cpp"
4
5   bool isPalindromeList(LinkNode* head){
6       /*第一步、与方法一一样，找到中间结点*/
7       if(head == NULL)
8           return false;
9
10      LinkNode *n1 = head , *n2 = head;
11      while(n2->next != NULL && n2->next->next != NULL){
12          n2 = n2->next->next;
13          n1 = n1->next;
14      }
15      //如果没有结点或者只有一个首结点
16      if(n2 == head){
17          return false;
18      }
19      //如果是奇数个结点
20      if(n2->next != NULL){
21          n1 = n1->next;  //n1 -> middle node
22      }
23
24      /*第二步、不使用额外空间，在链表自身上做文章：反转链表后半部分结点的next指针*/
25      n2 = n1->next;  // n2 -> right part first node
26      n1->next = NULL;//middle node->next = NULL
27      LinkNode *n3 = NULL;
28      while (n2 != NULL) {
29          n3 = n2->next;  //记录下一个要反转指针的结点
30          n2->next = n1;  //反转指针
31          n1 = n2;
32          n2 = n3;
33      }
34      //n1 -> end node
35      n3 = n1;  //record end node
36      n2 = head->next;
37      while (n2 != NULL) {
38          if (n2->data != n1->data) {
39              return false;
```

```
        }
        n2 = n2->next;  //move n2 forward right
        n1 = n1->next;  //move n1 forward left
    }
    //recover the right part nodes
    n2 = n3; //n2 -> end node
    n1 = NULL;
    while (n2 != NULL) {
        n3 = n2->next;
        n2->next = n1;
        n1=n2;
        n2 = n3;
    }

    return true;
}


/*bool isPalindromeList(LinkNode* head){
    if(head == NULL)
        return false;

    LinkNode *slow = head , *fast = head;
    SqStack* stack;
    init(stack);

    //fast指针每走两步，slow指针才走一步
    while(fast->next != NULL && fast->next->next != NULL){
        fast = fast->next->next;
        slow = slow->next;
        push(slow,stack);
    }

    //链表没有结点或只有一个结点，不是回文结构
    if(isEmpty(stack))
        return false;

    //判断偶数个结点还是奇数个结点
    if(fast->next != NULL){ //奇数个结点,slow需要再走一步
        slow = slow->next;
    }

    //从slow的后继结点开始遍历链表，将每个结点与栈顶结点比较
    LinkNode* node;
    slow = slow->next;
    while(slow != NULL){
        pop(stack,node);
        //一旦发现有一个结点不同就不是回文结构
        if(slow->data != node->data)
            return false;
        slow = slow->next;
    }
    return true;
```

```
}*/

int main(){

    LinkNode* head;
    init(head);
    add(2,head);
    add(3,head);
    add(3,head);
    add(1,head);
    printList(head);

    if(isPalindromeList(head)){
        printf("yes");
    }else{
        printf("no");
    }
    return 0;
}
```

链表与荷兰国旗问题

将单向链表按某值划分成左边小、中间相等、右边大的形式

```
#include<stdio.h>
#include "LinkList.cpp"

/*
    partition一个链表有两种做法。
    1，将链表中的所有结点放入一个数组中，那么就转换成了荷兰国旗问题，但这种做法会使用O(N)的额外空间;
    2，分出逻辑上的small,equal,big三个区域，遍历链表结点将其添加到对应的区域中，最后再将这三个区域连起来。
    这里只示范第二种做法:
*/
void partitionList(LinkNode *head,int val){
    if(head == NULL)
        return;
    LinkNode *smH = NULL;   //small area head node
    LinkNode *smT = NULL;   //small area tail node
    LinkNode *midH = NULL;  //equal area head node
    LinkNode *midT = NULL;  //equal area tail node
    LinkNode *bigH = NULL;  //big area head node
    LinkNode *bigT = NULL;  //big area tail node
    LinkNode *cur = head->next;
    LinkNode *next = NULL;//next node need to be distributed to the three areas
    while(cur != NULL){
        next = cur->next;
        cur->next = NULL;
        if(cur->data > val){
            if(bigH == NULL){
                bigH = bigT = cur;
            }else{
```

```
                bigT->next = cur;
                bigT = cur;
            }
        }else if(cur->data == val){
            if(midH == NULL){
                midH = midT = cur;
            }else{
                midT->next = cur;
                midT = cur;
            }
        }else{
            if(smH == NULL){
                smH = smT = cur;
            }else{
                smT->next = cur;
                smT = cur;
            }
        }
        cur = next;
    }
    //reconnect small and equal
    if(smT != NULL){
        smT->next = midH;
        midT = midT == NULL ? midT : smT;
    }
    //reconnect equal and big
    if(bigT != NULL){
        midT->next = bigH;
    }

    head = smH != NULL ? smH : midH != NULL ? midH : bigH;

    return;
}

int main(){
    LinkNode* head;
    init(head);
    add(5,head);
    add(2,head);
    add(7,head);
    add(9,head);
    add(1,head);
    add(3,head);
    add(5,head);
    printList(head);
    partitionList(head,5);
    printList(head);
}
```

复制含有随机指针结点的链表

借助哈希表，额外空间 O(N)

将链表的所有结点复制一份，以 `key,value` 为 `源结点，副本结点` 的方式存储到哈希表中，再建立副本结点之间的关系（`next、rand` 指针域）

```java
import java.util.HashMap;
import java.util.Map;

public class CopyLinkListWithRandom {

    public static class Node {
        public Node(int data) {
            this.data = data;
        }

        public Node() {
        }

        int data;
        Node next;
        Node rand;
    }

    public static Node copyLinkListWithRandom(Node head) {
        if (head == null) {
            return null;
        }
        Node cur = head;
        Map<Node, Node> copyMap = new HashMap<>();
        while (cur != null) {
            copyMap.put(cur, new Node(cur.data));
            cur = cur.next;
        }
        cur = head;
        while (cur != null) {
            copyMap.get(cur).next = copyMap.get(cur.next);
            copyMap.get(cur).rand = copyMap.get(cur.rand);
            cur = cur.next;
        }
        return copyMap.get(head);
    }

    public static void printListWithRandom(Node head) {
        if (head != null) {
            while (head.next != null) {
                head = head.next;
                System.out.print("node data:" + head.data);
                if (head.rand != null) {
                    System.out.println(",rand data:" + head.rand.data);
                } else {
                    System.out.println(",rand is null");
                }
            }
        }
    }
```

```
51
52      public static void main(String[] args) {
53          Node head = new Node();
54          head.next = new Node(1);
55          head.next.next = new Node(2);
56          head.next.next.next = new Node(3);
57          head.next.next.next.next = new Node(4);
58          head.next.rand = head.next.next.next.next;
59          head.next.next.rand = head.next.next.next;
60          printListWithRandom(head);
61
62          System.out.println("==========");
63
64          Node copy = copyLinkListWithRandom(head);
65          printListWithRandom(copy);
66      }
67  }
```

进阶操作：额外空间 O(1)

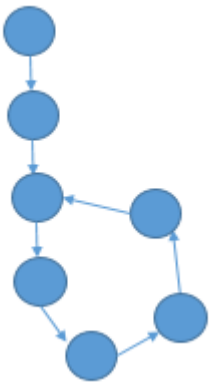
将副本结点追加到对应源结点之后，建立副本结点之间的指针域，最后将副本结点从该链表中分离出来。

```
1  //extra area O(1)
2  public static Node copyLinkListWithRandom2(Node head){
3    if (head == null) {
4      return null;
5    }
6    Node cur = head;
7    //copy every node and append
8    while (cur != null) {
9      Node copy = new Node(cur.data);
10     copy.next = cur.next;
11     cur.next = copy;
12     cur = cur.next.next;
13   }
14   //set the rand pointer of every copy node
15   Node copyHead = head.next;
16   cur = head;
17   Node curCopy = copyHead;
18   while (curCopy != null) {
19     curCopy.rand = cur.rand == null ? null : cur.rand.next;
20     cur = curCopy.next;
21     curCopy = cur == null ? null : cur.next;
22   }
23   //split
24   cur = head;
25   Node next = null;
26   while (cur != null) {
27     curCopy = cur.next;
28     next = cur.next.next;
29     curCopy.next = next == null ? null : next.next;
30     cur.next = next;
31     cur = next;
```

```
32      }
33      return copyHead;
34  }
```

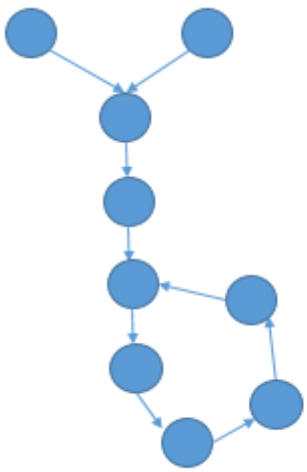
若两个可能有环的单链表相交，请返回相交的第一个结点

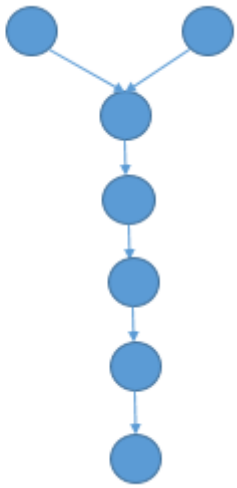
根据单链表的定义，每个结点有且只有一个`next`指针，那么如果单链表有环，它的结构将是如下所示：



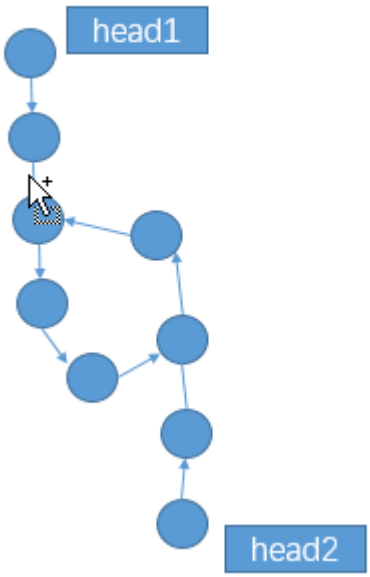
相交会导致两个结点指向同一个后继结点，但不可能出现一个结点有两个后继结点的情况。

1、当相交的结点不在环上时，有如下两种情况：





2、当相交的结点在环上时，只有一种情况：



综上，两单链表若相交，要么都无环，要么都有环。

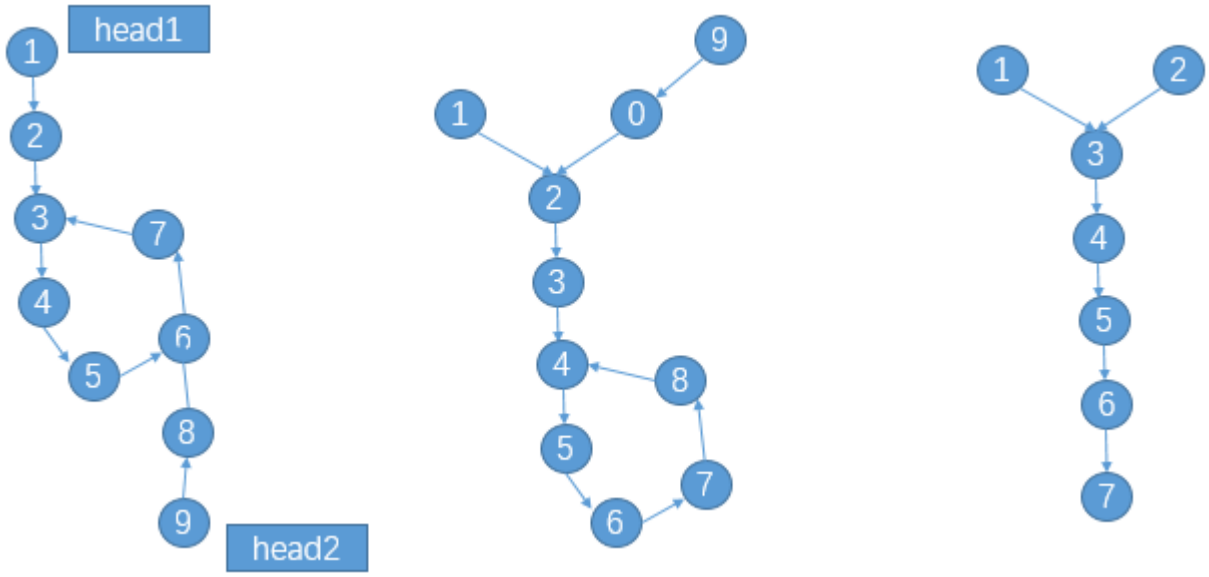
> 此题还需要注意的一点是如果链表有环，那么如何获取入环呢（因为不能通过`next`是否为空来判断是否是尾结点了）。这里就涉及到了一个规律：如果快指针`fast`和慢指针`slow`同时从头结点出发，`fast`走两步而`slow`走一步，当两者相遇时，将`fast`指针指向头结点，使两者都一次只走一步，两者会在入环结点相遇。

```java
public class FirstIntersectNode {
    public static class Node{
        int data;
        Node next;
        public Node(int data) {
            this.data = data;
        }
    }

    public static Node getLoopNode(Node head) {
        if (head == null) {
            return null;
        }
        Node fast = head;
        Node slow = head;
        do {
            slow = slow.next;
            if (fast.next == null || fast.next.next == null) {
                return null;
            } else {
                fast = fast.next.next;
            }
        } while (fast != slow);
        //fast == slow
        fast = head;
        while (fast != slow) {
            slow = slow.next;
```

```
            fast = fast.next;
        }
        return slow;
    }

    public static Node getFirstIntersectNode(Node head1, Node head2) {
        if (head1 == null || head2 == null) {
            return null;
        }
        Node loop1 = getLoopNode(head1);    //两链表的入环结点loop1和loop2
        Node loop2 = getLoopNode(head2);
        //no loop
        if (loop1 == null && loop2 == null) {
            return noLoop(head1, head2);
        }
        //both loop
        if (loop1 != null && loop2 != null) {
            return bothLoop(head1, head2, loop1, loop2);
        }
        //don't intersect
        return null;
    }

    private static Node bothLoop(Node head1, Node head2, Node loop1, Node loop2) {
        Node cur1 = head1;
        Node cur2 = head2;
        //入环结点相同，相交点不在环上
        if (loop1 == loop2) {
            int n = 0;
            while (cur1.next != loop1) {
                n++;
                cur1 = cur1.next;
            }
            while (cur2.next != loop1) {
                n--;
                cur2 = cur2.next;
            }
            cur1 = n > 0 ? head1 : head2;           //将cur1指向结点数较多的链表
            cur2 = cur1 == head1 ? head2 : head1;   //将cur2指向另一个链表
            n = Math.abs(n);
            while (n != 0) {                        //将cur1先走两链表结点数差值个结点
                cur1 = cur1.next;
                n--;
            }
            while (cur1 != cur2) {                  //cur1和cur2会在入环结点相遇
                cur1 = cur1.next;
                cur2 = cur2.next;
            }
            return cur1;
        }
        //入环结点不同，相交点在环上
        cur1 = loop1.next;
        while (cur1 != loop1) {
```

```
            if (cur1 == loop2) {    //链表2的入环结点在链表1的环上，说明相交
                return loop1;   //返回loop1或loop2均可，因为整个环就是两链表的相交部分
            }
            cur1 = cur1.next;
        }
        //在链表1的环上转了一圈也没有找到链表2的入环结点，说明不想交
        return null;
    }

    private static Node noLoop(Node head1, Node head2) {
        Node cur1 = head1;
        Node cur2 = head2;
        int n = 0;
        while (cur1.next != null) {
            n++;
            cur1 = cur1.next;
        }
        while (cur2.next != null) {
            n--;
            cur2 = cur2.next;
        }
        if (cur1 != cur2) {     //两链表的尾结点不同，不可能相交
            return null;
        }
        cur1 = n > 0 ? head1 : head2;           //将cur1指向结点数较多的链表
        cur2 = cur1 == head1 ? head2 : head1;   //将cur2指向另一个链表
        n = Math.abs(n);
        while (n != 0) {                        //将cur1先走两链表结点数差值个结点
            cur1 = cur1.next;
            n--;
        }
        while (cur1 != cur2) {                  //cur1和cur2会在入环结点相遇
            cur1 = cur1.next;
            cur2 = cur2.next;
        }
        return cur1;
    }

    public static void printList(Node head) {
        for (int i = 0; i < 50; i++) {
            System.out.print(head.data+" ");
            head = head.next;
        }
        System.out.println();
    }

}
```

对应三种情况测试如下：




```
public static void main(String[] args) {

        //==================== both loop ======================
        //1->2->[3]->4->5->6->7->[3]...
        Node head1 = new Node(1);
        head1.next = new Node(2);
        head1.next.next = new Node(3);
        head1.next.next.next = new Node(4);
        head1.next.next.next.next = new Node(5);
        head1.next.next.next.next.next = new Node(6);
        head1.next.next.next.next.next.next = new Node(7);
        head1.next.next.next.next.next.next.next = head1.next.next;

        //9->8->[6]->7->3->4->5->[6]...
        Node head2 = new Node(9);
        head2.next = new Node(8);
        head2.next.next = head1.next.next.next.next.next;
        head2.next.next.next = head1.next.next.next.next.next.next;
        head2.next.next.next.next = head1.next.next;
        head2.next.next.next.next.next = head1.next.next.next;
        head2.next.next.next.next.next.next = head1.next.next.next.next.next;
        head2.next.next.next.next.next.next.next = head1.next.next.next.next.next.next;

        printList(head1);
        printList(head2);
        System.out.println(getFirstIntersectNode(head1, head2).data);
        System.out.println("==================");

        //1->[2]->3->4->5->6->7->8->4...
        Node head3 = new Node(1);
        head3.next = new Node(2);
        head3.next.next = new Node(3);
        head3.next.next.next = new Node(4);
        head3.next.next.next.next = new Node(5);
        head3.next.next.next.next.next = new Node(6);
```

```
        head3.next.next.next.next.next.next = new Node(7);
        head3.next.next.next.next.next.next.next = new Node(8);
        head3.next.next.next.next.next.next.next.next = head1.next.next.next;

        //9->0->[2]->3->4->5->6->7->8->4...
        Node head4 = new Node(9);
        head4.next = new Node(0);
        head4.next.next = head3.next;
        head4.next.next.next = head3.next.next;
        head4.next.next.next.next = head3.next.next.next;
        head4.next.next.next.next.next = head3.next.next.next.next;
        head4.next.next.next.next.next.next = head3.next.next.next.next.next;
        head4.next.next.next.next.next.next.next = head3.next.next.next.next.next.next;
        head4.next.next.next.next.next.next.next.next = head3.next.next.next.next.next.next.next;
        head4.next.next.next.next.next.next.next.next.next = head3.next.next.next;

        printList(head3);
        printList(head4);
        System.out.println(getFirstIntersectNode(head3,head4).data);
        System.out.println("==================");

        //============== no loop ==============
        //1->[2]->3->4->5
        Node head5 = new Node(1);
        head5.next = new Node(2);
        head5.next.next = new Node(3);
        head5.next.next.next = new Node(4);
        head5.next.next.next.next = new Node(5);
        //6->[2]->3->4->5
        Node head6 = new Node(6);
        head6.next = head5.next;
        head6.next.next = head5.next.next;
        head6.next.next.next = head5.next.next.next;
        head6.next.next.next.next = head5.next.next.next.next;

        System.out.println(getFirstIntersectNode(head5,head6).data);
    }
```

栈和队列

用数组结构实现大小固定的栈和队列

```
//
// Created by zaw on 2018/10/21.
//
#include <stdio.h>
#include <malloc.h>
#define MAX_SIZE 1000

struct ArrayStack{
```

```
    int data[MAX_SIZE];
    int top;
};

void init(ArrayStack *&stack) {
    stack = (ArrayStack *) malloc(sizeof(ArrayStack));
    stack->top = -1;
}

bool isEmpty(ArrayStack* stack){
    return stack->top == -1 ?;
}

bool isFull(ArrayStack *stack){
    return stack->top == MAX_SIZE - 1 ?;
}

void push(int i, ArrayStack *stack){
    if (!isFull(stack)) {
        stack->data[++stack->top] = i;
    }
}

int pop(ArrayStack* stack){
    if (!isEmpty(stack)) {
        return stack->data[stack->top--];
    }
}

int getTopElement(ArrayStack *stack){
    if (!isEmpty(stack)) {
        return stack->data[stack->top];
    }
}

int main(){

    ArrayStack* stack;
    init(stack);
    push(1, stack);
    push(2, stack);
    push(3, stack);

    printf("%d ", pop(stack));
    printf("%d ", getTopElement(stack));
    printf("%d ", pop(stack));
    printf("%d ", pop(stack));
        //3 2 2 1
    return 0;
}
```

```
//
// Created by zaw on 2018/10/21.
```

```c
//
#include <stdio.h>
#include <malloc.h>
#define MAX_SIZE 1000

//数组结构实现的环形队列
struct ArrayCircleQueue{
    int data[MAX_SIZE];
    int front,rear;
};

void init(ArrayCircleQueue *&queue){
    queue = (ArrayCircleQueue *) malloc(sizeof(ArrayCircleQueue));
    queue->front = queue->rear = 0;
}

bool isEmpty(ArrayCircleQueue *queue){
    return queue->front == queue->rear;
}

bool isFull(ArrayCircleQueue *queue){
    return (queue->rear+1)%MAX_SIZE==queue->front;
}

void enQueue(int i, ArrayCircleQueue *queue){
    if (!isFull(queue)) {
        //move the rear and fill it
        queue->data[++queue->rear] = i;
    }
}

int deQueue(ArrayCircleQueue *queue){
    if (!isEmpty(queue)) {
        return queue->data[++queue->front];
    }
}

int main(){
    ArrayCircleQueue* queue;
    init(queue);
    enQueue(1, queue);
    enQueue(2, queue);
    enQueue(3, queue);
    while (!isEmpty(queue)) {
        printf("%d ", deQueue(queue));
    }
}
```

取栈中最小元素

实现一个特殊的栈，在实现栈的基本功能的基础上，再实现返回栈中最小元素的操作`getMin`。要求如下：

- `pop`、`push`、`getMin`操作的时间复杂度都是`O(1)`。

- 设计的栈类型可以使用现成的栈结构。

> 思路：由于每次`push`之后都会可能导致栈中已有元素的最小值发生变化，因此需要一个容器与该栈联动（记录每次`push`产生的栈中最小值）。我们可以借助一个辅助栈，数据栈`push`第一个元素时，将其也`push`到辅助栈，此后每次向数据栈`push`元素的同时将其和辅助栈的栈顶元素比较，如果小，则将其也`push`到辅助栈，否则取辅助栈的栈顶元素`push`到辅助栈。（数据栈正常`push`、`pop`数据，而辅助栈`push`每次数据栈`push`后产生的栈中最小值；但数据栈`pop`时，辅助栈也只需简单的`pop`即可，保持同步）

```
//
// Created by zaw on 2018/10/21.
//
#include <stdio.h>
#include <malloc.h>
#include "ArrayStack.cpp"

int min(int a, int b){
    return a < b ? a : b;
}

struct GetMinStack{
    ArrayStack* dataStack;
    ArrayStack* helpStack;
};

void initGetMinStack(GetMinStack* &stack){
    stack = (GetMinStack *) malloc(sizeof(GetMinStack));
    init(stack->dataStack);
    init(stack->helpStack);
}

void push(int i, GetMinStack *stack) {
    if (!isFull(stack->dataStack)) {
        push(i, stack->dataStack);  //ArrayStack.cpp
        if (!isEmpty(stack->helpStack)) {
            i = min(i, getTopElement(stack->helpStack));
        }
        push(i, stack->helpStack);
    }
}

int pop(GetMinStack* stack){
    if (!isEmpty(stack->dataStack)) {
        pop(stack->helpStack);
        return pop(stack->dataStack);
    }
}

int getMin(GetMinStack *stack){
    if (!isEmpty(stack->dataStack)) {
        return getTopElement(stack->helpStack);
    }
```

```
}

int main(){
    GetMinStack *stack;
    initGetMinStack(stack);
    push(6, stack);
    printf("%d ", getMin(stack));//6
    push(3, stack);
    printf("%d ", getMin(stack));//3
    push(1, stack);
    printf("%d ", getMin(stack));//1
    pop(stack);
    printf("%d ", getMin(stack));//3

    return 0;
}
```

仅用队列结构实现栈结构

> 思路：只要将关注点放在 **后进先出** 这个特性就不难实现了。使用一个数据队列和辅助队列，当放入数据时使用队列的操作正常向数据队列中放，但出队元素时，需将数据队列的前n-1个数入队辅助队列，而将数据队列的队尾元素弹出来，最后数据队列和辅助队列交换角色。

```
//
// Created by zaw on 2018/10/21.
//
#include <stdio.h>
#include <malloc.h>
#include "../queue/ArrayCircleQueue.cpp"

struct DoubleQueueStack{
    ArrayCircleQueue* dataQ;
    ArrayCircleQueue* helpQ;
};

void init(DoubleQueueStack* &stack){
    stack = (DoubleQueueStack *) malloc(sizeof(DoubleQueueStack));
    init(stack->dataQ);
    init(stack->helpQ);
}

void swap(ArrayCircleQueue *&dataQ, ArrayCircleQueue *&helpQ){
    ArrayCircleQueue* temp = dataQ;
    dataQ = helpQ;
    helpQ = temp;
}

void push(int i,DoubleQueueStack* stack){
    if (!isFull(stack->dataQ)) {
        return enQueue(i, stack->dataQ);
    }
}
```

```
30
31  int pop(DoubleQueueStack* stack){
32      if (!isEmpty(stack->dataQ)) {
33          int i = deQueue(stack->dataQ);
34          while (!isEmpty(stack->dataQ)) {
35              enQueue(i, stack->helpQ);
36              i = deQueue(stack->dataQ);
37          }
38          swap(stack->dataQ, stack->helpQ);
39          return i;
40      }
41  }
42
43  bool isEmpty(DoubleQueueStack* stack){
44      return isEmpty(stack->dataQ);
45  }
46
47  int getTopElement(DoubleQueueStack* stack){
48      if (!isEmpty(stack->dataQ)) {
49          int i = deQueue(stack->dataQ);
50          while (!isEmpty(stack->dataQ)) {
51              enQueue(i, stack->helpQ);
52              i = deQueue(stack->dataQ);
53          }
54          enQueue(i, stack->helpQ);
55          swap(stack->dataQ, stack->helpQ);
56          return i;
57      }
58  }
59
60  int main(){
61
62      DoubleQueueStack *stack;
63      init(stack);
64      push(1, stack);
65      push(2, stack);
66      push(3, stack);
67      while (!isEmpty(stack)) {
68          printf("%d ", pop(stack));
69      }
70      push(4, stack);
71      printf("%d ", getTopElement(stack));
72
73      return 0;
74  }
```

仅用栈结构实现队列结构

> 思路：使用两个栈，一个栈`PutStack`用来放数据，一个栈`GetStack`用来取数据。取数据时，如果`PulllStack`为空则需要将`PutStack`中的**所有元素**一次性依次`pop`并放入`GetStack`。
>
> 特别要注意的是这个 **倒数据**的时机：

- 只有当`GetStack`为空时才能往里倒
- **倒数据**时必须一次性将`PutStack`中的数据倒完

```
//
// Created by zaw on 2018/10/21.
//
#include <stdio.h>
#include <malloc.h>
#include "../stack/ArrayStack.cpp"

struct DoubleStackQueue{
    ArrayStack* putStack;
    ArrayStack* getStack;
};

void init(DoubleStackQueue *&queue){
    queue = (DoubleStackQueue *) malloc(sizeof(DoubleStackQueue));
    init(queue->putStack);
    init(queue->getStack);
}

bool isEmpty(DoubleStackQueue *queue){
    return isEmpty(queue->getStack) && isEmpty(queue->putStack);
}

void pour(ArrayStack *stack1, ArrayStack *stack2){
    while (!isEmpty(stack1)) {
        push(pop(stack1), stack2);
    }
}

void enQueue(int i, DoubleStackQueue *queue){
    if (!isFull(queue->putStack)) {
        push(i, queue->putStack);
    } else {
        if (isEmpty(queue->getStack)) {
            pour(queue->putStack, queue->getStack);
            push(i, queue->putStack);
        }
    }
}

int deQueue(DoubleStackQueue* queue){
    if (!isEmpty(queue->getStack)) {
        return pop(queue->getStack);
    } else {
        if (!isEmpty(queue->putStack)) {
            pour(queue->putStack, queue->getStack);
            return pop(queue->getStack);
        }
    }
}

```

```
51
52  int main(){
53      DoubleStackQueue *queue;
54      init(queue);
55      enQueue(1, queue);
56      printf("%d\n", deQueue(queue));
57      enQueue(2, queue);
58      enQueue(3, queue);
59      while (!isEmpty(queue)) {
60          printf("%d ", deQueue(queue));
61      }
62      return 0;
63  }
```

二叉树

实现二叉树的先序、中序、后续遍历，包括递归方式和非递归方式

递归方式

```
1   public static class Node{
2     int data;
3     Node left;
4     Node right;
5     public Node(int data) {
6       this.data = data;
7     }
8   }
9
10  public static void preOrderRecursive(Node root) {
11    if (root != null) {
12      System.out.print(root.data+" ");
13      preOrderRecursive(root.left);
14      preOrderRecursive(root.right);
15    }
16  }
17
18  public static void medOrderRecursive(Node root) {
19    if (root != null) {
20      medOrderRecursive(root.left);
21      System.out.print(root.data+" ");
22      medOrderRecursive(root.right);
23    }
24  }
25
26  public static void postOrderRecursive(Node root) {
27    if (root != null) {
28      postOrderRecursive(root.left);
29      postOrderRecursive(root.right);
30      System.out.print(root.data+" ");
31    }
32  }
```

```
33
34  public static void main(String[] args) {
35    Node root = new Node(1);
36    root.left = new Node(2);
37    root.right = new Node(3);
38    root.left.left = new Node(4);
39    root.left.right = new Node(5);
40    root.right.left = new Node(6);
41    root.right.right = new Node(7);
42    preOrderRecursive(root);  //1 2 4 5 3 6 7
43    System.out.println();
44    medOrderRecursive(root);  //4 2 5 1 6 3 7
45    System.out.println();
46    postOrderRecursive(root); //4 5 2 6 7 3 1
47    System.out.println();
48  }
```

以先根遍历二叉树为例，可以发现递归方式首先尝试打印当前结点的值，随后尝试打印左子树，打印完左子树后尝试打印右子树，递归过程的`base case`是当某个结点为空时停止子过程的展开。这种递归尝试是由二叉树本身的结构所决定的，因为二叉树上的任意结点都可看做一棵二叉树的根结点（即使是叶子结点，也可以看做是一棵左右子树为空的二叉树根结点）。

观察先序、中序、后序三个递归方法你会发现，不同点在于打印当前结点的值这一操作的时机。**你会发现每个结点会被访问三次**：进入方法时算一次、递归处理左子树完成之后返回时算一次、递归处理右子树完成之后返回时算一次。因此在`preOrderRecursive`中将打印语句放到方法开始时就产生了先序遍历；在`midOrderRecursive`中，将打印语句放到递归chu处理左子树完成之后就产生了中序遍历。

非递归方式

先序遍历

拿到一棵树的根结点后，首先打印该结点的值，然后将其非空右孩子、非空左孩子依次压栈。栈非空循环：从栈顶弹出结点（一棵子树的根节点）并打印其值，再将其非空右孩子、非空左孩子依次压栈。

```
1   public static void preOrderUnRecur(Node root) {
2     if (root == null) {
3       return;
4     }
5     Stack<Node> stack = new Stack<>();
6     stack.push(root);
7     Node cur;
8     while (!stack.empty()) {
9       cur = stack.pop();
10      System.out.print(cur.data+" ");
11      if (cur.right != null) {
12        stack.push(cur.right);
13      }
14      if (cur.left != null) {
15        stack.push(cur.left);
16      }
17    }
18    System.out.println();
19  }
```

你会发现压栈的顺序和打印的顺序是相反的，压栈是先根结点，然后有右孩子就压右孩子、有左孩子就压左孩子，这是利用栈的后进先出。每次获取到一棵子树的根节点之后就可以获取其左右孩子，因此无需保留其信息，直接弹出并打印，然后保留其左右孩子到栈中即可。

中序遍历

对于一棵树，将该树的左边界全部压栈，`root`的走向是只要左孩子不为空就走向左孩子。当左孩子为空时弹出栈顶结点（此时该结点是一棵左子树为空的树的根结点，根据中序遍历可以直接打印该结点，然后中序遍历该结点的右子树）打印，如果该结点的右孩子非空（说明有右子树），那么将其右孩子压栈，这个右孩子又可能是一棵子树的根节点，因此将这棵子树的左边界压栈，这时回到了开头，以此类推。

```
public static void medOrderUnRecur(Node root) {
  if (root == null) {
    return;
  }
  Stack<Node> stack = new Stack<>();
  while (!stack.empty() || root != null) {
    if (root != null) {
      stack.push(root);
      root = root.left;
    } else {
      root = stack.pop();
      System.out.print(root.data+" ");
      root = root.right;
    }
  }
  System.out.println();
}
```

后序遍历

> 思路一：准备两个栈，一个栈用来保存遍历时的结点信息，另一个栈用来排列后根顺序（根节点先进栈，右孩子再进，左孩子最后进）。

```
public static void postOrderUnRecur1(Node root) {
  if (root == null) {
    return;
  }
  Stack<Node> stack1 = new Stack<>();
  Stack<Node> stack2 = new Stack<>();
  stack1.push(root);
  while (!stack1.empty()) {
    root = stack1.pop();
    if (root.left != null) {
      stack1.push(root.left);
    }
    if (root.right != null) {
      stack1.push(root.right);
    }
    stack2.push(root);
  }
```

```
  while (!stack2.empty()) {
    System.out.print(stack2.pop().data + " ");
  }
  System.out.println();
}
```

> 思路二：只用一个栈。借助两个变量`h`和`c`，`h`代表最近一次打印过的结点，`c`代表栈顶结点。首先将根结点压栈，此后栈非空循环，令`c`等于栈顶元素（`c=stack.peek()`）执行以下三个分支：
>
> 1. `c`的左右孩子是否与`h`相等，如果都不相等，说明`c`的左右孩子都不是最近打印过的结点，由于左右孩子是左右子树的根节点，根据后根遍历的特点，左右子树肯定都没打印过，那么将左孩子压栈（打印左子树）。
> 2. 分支1没有执行说明`c`的左孩子要么不存在；要么左子树刚打印过了；要么右子树刚打印过了。这时如果是前两种情况中的一种，那就轮到打印右子树了，因此如果`c`的右孩子非空就压栈。
> 3. 如果前两个分支都没执行，说明`c`的左右子树都打印完了，因此弹出并打印`c`结点，更新一下`h`。

```
public static void postOrderUnRecur2(Node root) {
  if (root == null) {
    return;
  }
  Node h = null;  //最近一次打印的结点
  Node c = null;  //代表栈顶结点
  Stack<Node> stack = new Stack<>();
  stack.push(root);
  while (!stack.empty()) {
    c = stack.peek();
    if (c.left != null && c.left != h && c.right != h) {
      stack.push(c.left);
    } else if (c.right != null && c.right != h) {
      stack.push(c.right);
    } else {
      System.out.print(stack.pop().data + " ");
      h = c;
    }
  }
  System.out.println();
}
```

在二叉树中找一个结点的后继结点，结点除lleft,right指针外还包含一个parent指针

> 这里的后继结点不同于链表的后继结点。在二叉树中，前驱结点和后继结点是按照二叉树中两个结点被中序遍历的先后顺序来划分的。比如某二叉树的中序遍历是`2 1 3`，那么`1`的后继结点是`3`，前驱结点是`2`

你当然可以将二叉树中序遍历一下，在遍历到该结点的时候标记一下，那么下一个要打印的结点就是该结点的后继结点。

我们可以推测一下，当我们来到二叉树中的某个结点时，如果它的右子树非空，那么它的后继结点一定是它的右子树中最靠左的那个结点；如果它的右孩子为空，那么它的后继结点一定是它的祖先结点中，把它当做左子孙（它存在于祖先结点的左子树中）的那一个，否则它没有后继结点。

这里如果它的右孩子为空的情况比较难分析，我们可以借助一个指针`parent`，当前来到的结点`node`和其父结点`parent`的`parent.left`比较，如果相同则直接返回`parent`，否则`node`来到`parent`的位置，`parent`则继续向上追溯，直到`parent`到达根节点为止若`node`还是不等于`parent`的左孩子，则返回`null`表明给出的结点没有后继结点。

```
public class FindSuccessorNode {

    public static class Node{
        int data;
        Node left;
        Node right;
        Node parent;

        public Node(int data) {
            this.data = data;
        }
    }

    public static Node findSuccessorNode(Node node){
        if (node == null) {
            return null;
        }
        if (node.right != null) {
            node = node.right;
            while (node.left != null) {
                node = node.left;
            }
            return node;
        } else {
            Node parent = node.parent;
            while (parent != null && parent.left != node) {
                node = parent;
                parent = parent.parent;
            }
            return parent == null ? null : parent;
        }
    }

    public static void main(String[] args) {
        Node root = new Node(1);
        root.left = new Node(2);
        root.left.parent = root;
        root.left.left = new Node(4);
        root.left.left.parent = root.left;
        root.left.right = new Node(5);
        root.left.right.parent = root.left;
        root.right = new Node(3);
        root.right.parent = root;
        root.right.right = new Node(6);
        root.right.right.parent = root.right;

       if (findSuccessorNode(root.left.right) != null) {
```

```
            System.out.println("node5's successor node
is:"+findSuccessorNode(root.left.right).data);
        } else {
            System.out.println("node5's successor node doesn't exist");
        }

        if (findSuccessorNode(root.right.right) != null) {
            System.out.println("node6's successor node
is:"+findSuccessorNode(root.right.right).data);
        } else {
            System.out.println("node6's successor node doesn't exist");
        }
    }
}
```

介绍二叉树的序列化和反序列化

序列化

> 二叉树的序列化要注意的两个点如下：
>
> 1. 每序列化一个结点数值之后都应该加上一个结束符表示一个结点序列化的终止，如`!`。
> 2. 不能忽视空结点的存在，可以使用一个占位符如`#`表示空结点的序列化。

```
/**
     * 先根遍历的方式进行序列化
     * @param node  序列化来到了哪个结点
     * @return
     */
public static String serializeByPre(Node node) {
  if (node == null) {
    return "#!";
  }
  //收集以当前结点为根节点的树的序列化信息
  String res = node.data + "!";
  //假设能够获取左子树的序列化结果
  res += serializeByPre(node.left);
  //假设能够获取右子树的序列化结果
  res += serializeByPre(node.right);
  //返回以当前结点为根节点的树的序列化结果
  return res;
}

public static void main(String[] args) {
  Node root = new Node(1);
  root.left = new Node(2);
  root.left.left = new Node(4);
  root.left.right = new Node(5);
  root.right = new Node(3);
  root.right.right = new Node(6);

  System.out.println(serializeByPre(root));
}
```

重建

> 怎么序列化的，就怎么反序列化

```java
public static Node reconstrut(String serializeStr) {
  if (serializeStr != null) {
    String[] datas = serializeStr.split("!");
    if (datas.length > 0) {
      //借助队列保存结点数值
      Queue<String> queue = new LinkedList<>();
      for (String data : datas) {
        queue.offer(data);
      }
      return recon(queue);
    }
  }
  return null;
}

private static Node recon(Queue<String> queue) {
  //依次出队元素重建结点
  String data = queue.poll();
  //重建空结点，也是base case，当要重建的某棵子树为空时直接返回
  if (data.equals("#")) {
    return null;
  }
  //重建头结点
  Node root = new Node(Integer.parseInt(data));
  //重建左右子树
  root.left = recon(queue);
  root.right = recon(queue);
  return root;
}

public static void main(String[] args) {
  Node root = new Node(1);
  root.left = new Node(2);
  root.left.left = new Node(4);
  root.left.right = new Node(5);
  root.right = new Node(3);
  root.right.right = new Node(6);

  String str = serializeByPre(root);
  Node root2 = reconstrut(str);
  System.out.println(serializeByPre(root2));
}
```

判断一个树是否是平衡二叉树

> 平衡二叉树的定义：当二叉树的任意一棵子树的左子树的高度和右子树的高度相差不超过1时，该二叉树为平衡二叉树。

根据定义可知，要确认一个二叉树是否是平衡二叉树势必要遍历所有结点。而遍历到每个结点时，要想知道以该结点为根结点的子树是否是平衡二叉树，我们要收集两个信息：

1. 该结点的左子树、右子树是否是平衡二叉树
2. 左右子树的高度分别是多少，相差是否超过1

那么我们来到某个结点时（子过程），我们需要向上层（父过程）返回的信息就是该结点为根结点的树是否是平衡二叉树以及该结点的高度，这样的话，父过程就能继续向上层返回应该收集的信息。

```
package top.zhenganwen.algorithmdemo.recursive;

/**
 * 判断是否为平衡二叉树
 */
public class IsBalanceBTree {
    public static class Node{
        int data;
        Node left;
        Node right;
        public Node(int data) {
            this.data = data;
        }
    }
    /**
     * 遍历时，来到某个结点需要收集的信息
     * 1、以该结点为根节点的树是否是平衡二叉树
     * 2、该结点的高度
     */
    public static class ReturnData {
        public boolean isBalanced;
        public int height;
        public ReturnData(boolean isBalanced, int height) {
            this.isBalanced = isBalanced;
            this.height = height;
        }
    }

    public static ReturnData isBalancedBinaryTree(Node node){
        if (node == null) {
            return new ReturnData(true, 0);
        }
        ReturnData leftData = isBalancedBinaryTree(node.left);
        if (leftData.isBalanced == false) {
            //只要有一棵子树不是平衡二叉树，则会一路返回false，该树的高度自然不必收集了
            return new ReturnData(false, 0);
        }
        ReturnData rightDta = isBalancedBinaryTree(node.right);
        if (rightDta.isBalanced == false) {
            return new ReturnData(false, 0);
        }
        //返回该层收集的结果
        if (Math.abs(leftData.height - rightDta.height) > 1) {
            return new ReturnData(false, 0);
```

```
        }
        //若是平衡二叉树，树高等于左右子树较高的那个加1
        return new ReturnData(true, Math.max(leftData.height, rightDta.height) + 1);
    }

    public static void main(String[] args) {
        Node root = new Node(1);
        root.left = new Node(2);
        root.left.left = new Node(4);
        root.right = new Node(3);
        root.right.right = new Node(5);
        root.right.right.right = new Node(6);
        System.out.println(isBalancedBinaryTree(root).isBalanced);  //false
    }
}
```

> 递归很好用，该题中的递归用法也是一种经典用法，可以高度套路：
>
> 1. 分析问题的解决需要哪些步骤（这里是遍历每个结点，确认每个结点为根节点的子树是否为平衡二叉树）
> 2. 确定递归：父问题是否和子问题相同
> 3. 子过程要收集哪些信息
> 4. 本次递归如何利用子过程返回的信息得到本过程要返回的信息
> 5. `base case`

判断一棵树是否是搜索二叉树

> 搜索二叉树的定义：对于二叉树的任意一棵子树，其左子树上的所有结点的值小于该子树的根节点的值，而其右子树上的所有结点的值大于该子树的根结点的值，并且整棵树上任意两个结点的值不同。

根据定义，搜索二叉树的中序遍历打印将是一个升序序列。因此我们可以利用二叉树的中序遍历的非递归方式，比较中序遍历时相邻两个结点的大小，只要有一个结点的值小于其后继结点的那就不是搜索二叉树。

```
import java.util.Stack;

/**
 * 判断是否是搜索二叉树
 */
public class IsBST {
    public static class Node {
        int data;
        Node left;
        Node right;

        public Node(int data) {
            this.data = data;
        }
    }

    public static boolean isBST(Node root) {
        if (root == null) {
            return true;
        }
```

```
        int preData = Integer.MIN_VALUE;
        Stack<Node> stack = new Stack<>();
        while (root != null || !stack.empty()) {
            if (root != null) {
                stack.push(root);
                root = root.left;
            } else {
                Node node = stack.pop();
                if (node.data < preData) {
                    return false;
                } else {
                    preData = node.data;
                }
                root = node.right;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        Node root = new Node(6);
        root.left = new Node(3);
        root.left.left = new Node(1);
        root.left.right = new Node(4);
        root.right = new Node(8);
        root.right.left = new Node(9);
        root.right.right = new Node(10);

        System.out.println(isBST(root));    //false
    }
}
```

判断一棵树是否是完全二叉树

根据完全二叉树的定义，如果二叉树上某个结点有右孩子无左孩子则一定不是完全二叉树；否则如果二叉树上某个结点有左孩子而没有右孩子，那么该结点所在的那一层上，该结点右侧的所有结点应该是叶子结点，否则不是完全二叉树。

```
import java.util.LinkedList;
import java.util.Queue;

/**
 * 判断是否为完全二叉树
 */
public class IsCompleteBTree {
    public static class Node {
        int data;
        Node left;
        Node right;
        public Node(int data) {
            this.data = data;
        }
```

```
    }

    public static boolean isCompleteBTree(Node root) {
        if (root == null) {
            return true;
        }
        Queue<Node> queue = new LinkedList<>();
        queue.offer(root);
        boolean leaf = false;
        while (!queue.isEmpty()) {
            Node node = queue.poll();
            //左空右不空
            if (node.left == null && node.right != null) {
                return false;
            }
            //如果开启了叶子结点阶段，结点不能有左右孩子
            if (leaf &&
                    (node.left != null || node.right != null)) {
                return false;
            }
            //将下一层要遍历的加入到队列中
            if (node.left != null) {
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            } else {
                //左右均为空，或左不空右空。该结点同层的右侧结点均为叶子结点，开启叶子结点阶段
                leaf = true;
            }

        }
        return true;
    }

    public static void main(String[] args) {
        Node root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);
        root.left.right = new Node(4);

        System.out.println(isCompleteBTree(root));//false
    }
}
```

已知一棵完全二叉树，求其结点个数，要求时间复杂度0(N)

如果我们遍历二叉树的每个结点来计算结点个数，那么时间复杂度将是`O(N^2)`，我们可以利用满二叉树的结点个数为`2^h-1`（h为树的层数）来加速这个过程。

首先完全二叉树，如果其左子树的最左结点在树的最后一层，那么其右子树肯定是满二叉树，且高度为 `h-1`；否则其左子树肯定是满二叉树，且高度为 `h-2`。也就是说，对于一个完全二叉树结点个数的求解，我们可以分解求解过程：1个根结点+ 一棵满二叉树（高度为h-1或者h-2）+ 一棵完全二叉树（高度为h-1）。前两者的结点数是可求的（1+2^level -1=2^level），后者就又成了求一棵完全二叉树结点数的问题了，可以使用递归。

```java
/**
 * 求一棵完全二叉树的节点个数
 */
public class CBTNodesNum {
  public static class Node {
    int data;
    Node left;
    Node right;
    public Node(int data) {
      super();
      this.data = data;
    }
  }

  // 获取完全二叉树的高度
  public static int getLevelOfCBT(Node root) {
    if (root == null)
      return 0;
    int level = 0;
    while (root != null) {
      level++;
      root = root.left;
    }
    return level;
  }

  public static int getNodesNum(Node node) {
    //base case
    if (node == null)
      return 0;
    int level = getLevelOfCBT(node);
    if (getLevelOfCBT(node.right) == level - 1) {
      // 左子树满，且高度为 level-1；收集左子树节点数2^(level-1)-1和头节点，对右子树重复此过程
      int leftNodesAndRoot = 1 << (level - 1);
      return getNodesNum(node.right) + leftNodesAndRoot;
    } else {
      // 右子树满，且高度为 level-2；收集右子树节点数2^(level-2)-1和头节点1,对左子树重复此过程
      int rightNodesAndRoot = 1 << (level - 2);
      return getNodesNum(node.left) + rightNodesAndRoot;

    }
  }

  public static void main(String[] args) {
    Node root = new Node(1);
    root.left = new Node(2);
    root.right = new Node(3);
    root.left.left = new Node(4);
```

```
49      root.left.right = new Node(5);
50      root.right.left = new Node(6);
51      root.right.right = new Node(7);
52
53      System.out.println(getNodesNum(root));
54    }
55  }
```

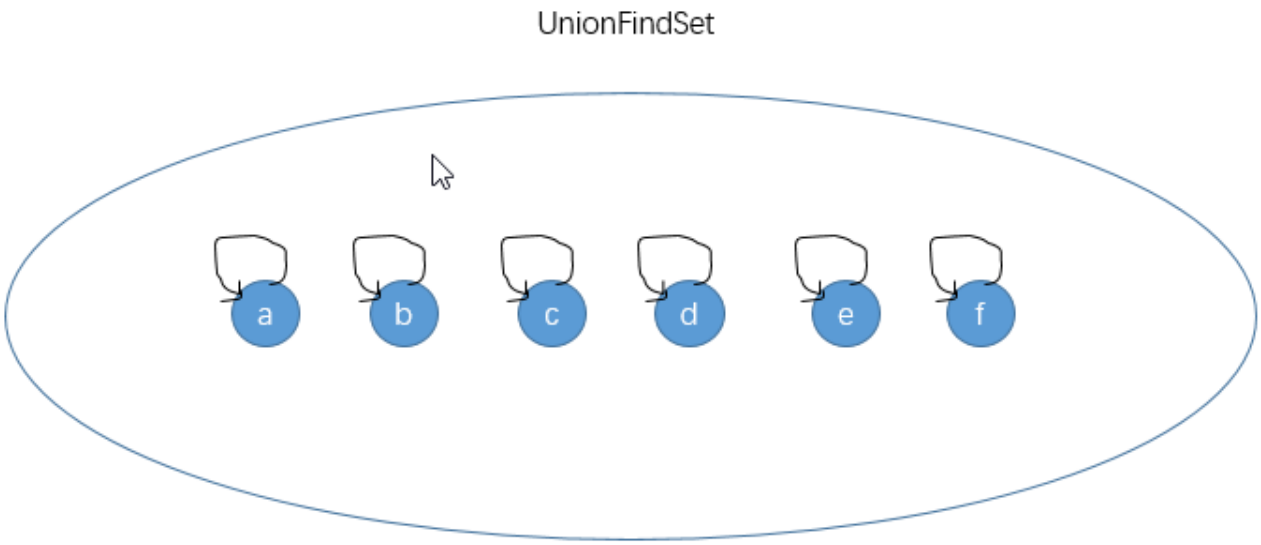
并查集

并查集是一种树型的数据结构，用于处理一些不交集（Disjoint Sets）的合并及查询问题。有一个**联合-查找算法**（**union-find algorithm**）定义了两个用于此数据结构的操作：

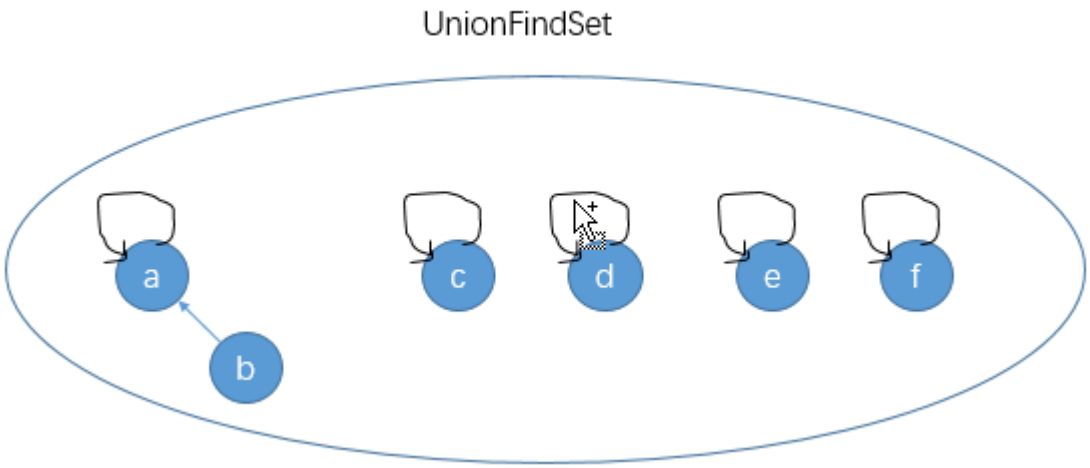
- Find：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。
- Union：将两个子集合并成同一个集合。

并查集结构的实现

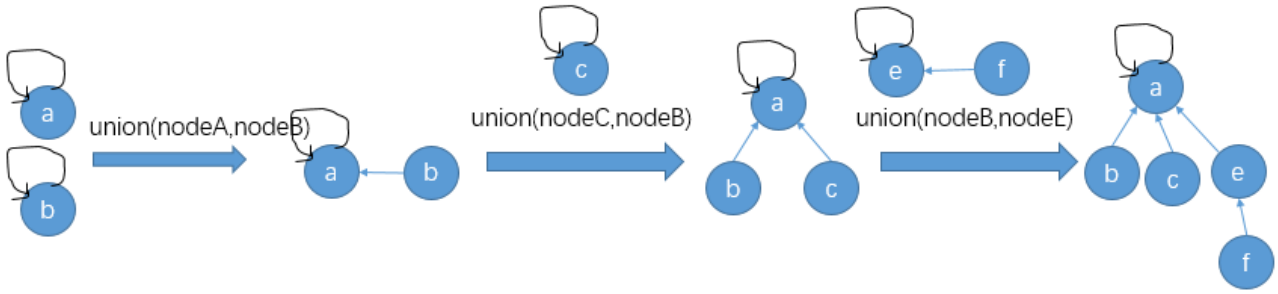
首先并查集本身是一个结构，我们在构造它的时候需要将所有要操作的数据扔进去，初始时每个数据自成一个结点，且每个结点都有一个父指针（初始时指向自己）。



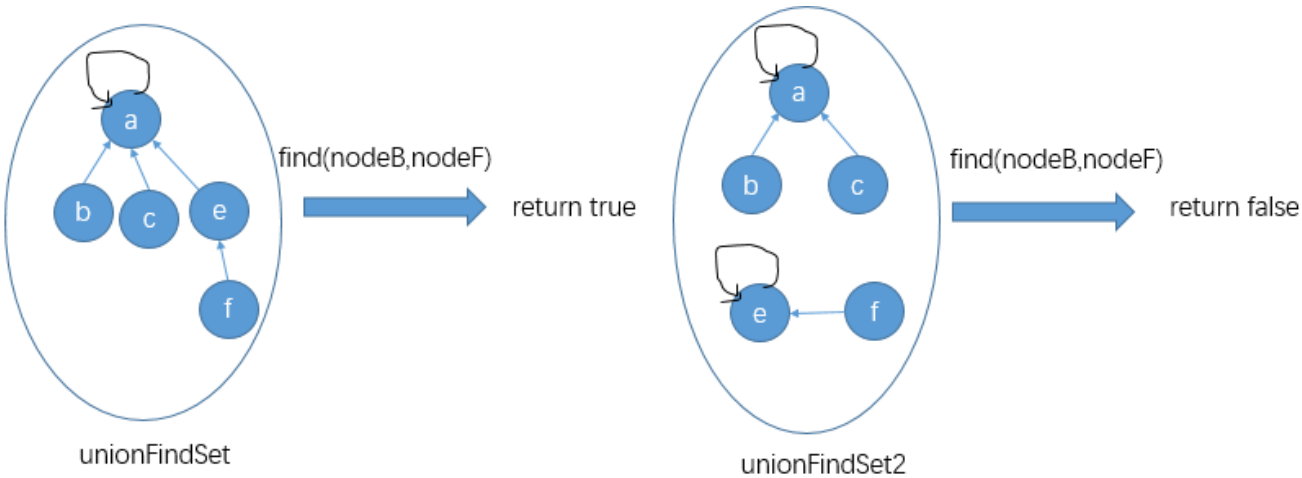
初始时并查集中的每个结点都算是一个子集，我们可以对任意两个元素进行合并操作。值得注意的是，`union(nodeA,nodeB)` 并不是将结点 `nodeA` 和 `nodeB` 合并成一个集合，而是将 `nodeA` 所在的集合和 `nodeB` 所在的集合合并成一个新的子集：



那么合并两个集合的逻辑是什么呢？首先要介绍一下**代表结点**这个概念：找一结点所在集合的代表结点就是找这个集合中父指针指向自己的结点（并查集初始化时，每个结点都是各自集合的代表结点）。那么合并两个集合就是将结点个数较少的那个集合的代表结点的父指针指向另一个集合的代表结点：



还有一个`find`操作：查找两个结点是否所属同一个集合。我们只需判断两个结点所在集合的代表结点是否是同一个就可以了：



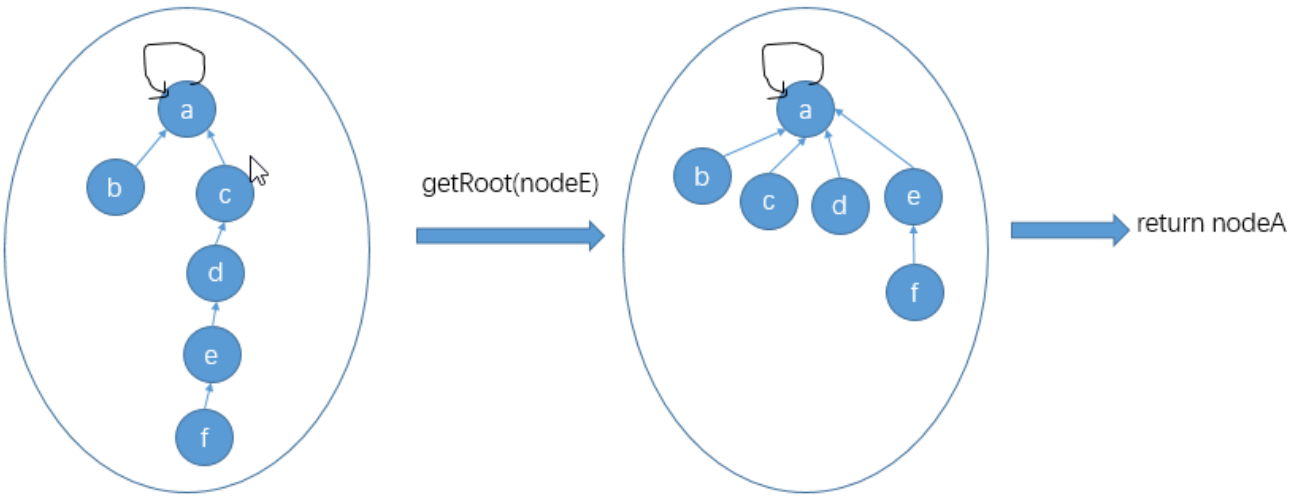
代码示例：

```
import java.util.*;

public class UnionFindSet{
    public static class Node{
        //whatever you like to store   int , char , String ..etc
    }
```

```
    private Map<Node,Node> fatherMap;
    private Map<Node,Integer> nodesNumMap;

    //give me the all nodes need to save into the UnionFindSet
    public UnionFindSet(List<Node> nodes){
        fatherMap = new HashMap();
        nodesNumMap = new HashMap();
        for(Node node : nodes){
            fatherMap.put(node,node);
            nodesNumMap.put(node,1);
        }
    }

    public void union(Node a,Node b){
        if(a == null || b == null){
            return;
        }
        Node rootOfA = getRoot(a);
        Node rootOfB = getRoot(b);
        if(rootOfA != rootOfB){
            int numOfA = nodesNumMap.get(rootOfA);
            int numOfB = nodesNumMap.get(rootOfB);
            if(numOfA >= numOfB){
                fatherMap.put(rootOfB , rootOfA);
                nodesNumMap.put(rootOfA, numOfA + numOfB);
            }else{
                fatherMap.put(rootOfA , rootOfB);
                nodesNumMap.put(rootOfB, numOfA + numOfB);
            }
        }
    }

    public boolean find(Node a,Node b){
        if(a == null || b == null){
            return false;
        }
        Node rootOfA = getRoot(a);
        Node rootOfB = getRoot(b);
        return rootOfA == rootOfB ? true : false;
    }

    public Node getRoot(Node node){
        if(node == null){
            return null;
        }
        Node father = fatherMap.get(node);
        if(father != node){
            father = fatherMap.get(father);
        }
        fatherMap.put(node, father);
        return father;
    }

```

```
60      public static void main(String[] args){
61          Node a = new Node();
62          Node b = new Node();
63          Node c = new Node();
64          Node d = new Node();
65          Node e = new Node();
66          Node f = new Node();
67          Node[] nodes = {a,b,c,d,e,f};
68
69          UnionFindSet set = new UnionFindSet(Arrays.asList(nodes));
70          set.union(a, b);
71          set.union(c, d);
72          set.union(b, e);
73          set.union(a, c);
74          System.out.println(set.find(d,e));
75      }
76  }
```

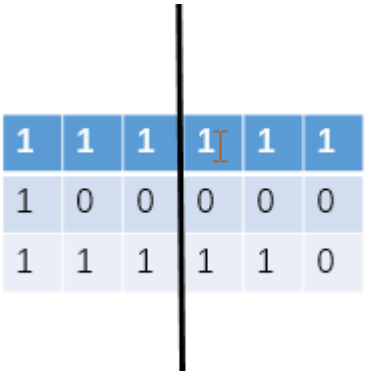
你会发现`union`和`find`的过程中都会有找一个结点所在集合的代表结点这个过程，所以我把它单独抽出来成一个`getRoot`，而且利用递归做了一个优化：找一个结点所在集合的代表结点时，会不停地向上找父指针指向自己的结点，最后在递归回退时将沿途路过的结点的父指针改为直接指向代表结点：



诚然，这样做是为了提高下一次查找的效率。

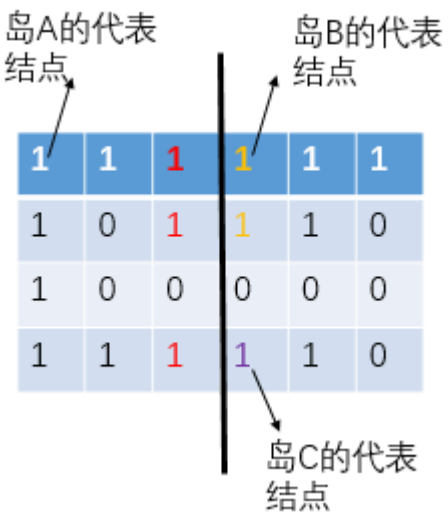
并查集的应用

并查集结构本身其实很简单，但是其应用却很难。这里以**岛问题**做引子，当矩阵相当大的时候，用单核CPU去跑这个遍历和感染效率是很低的，可能会使用并行计算框架来完成岛数量的统计。也就是说矩阵可能被分割成几个部分，逐个统计，最后在汇总。那么问题来了：



1	1	1	1	1	1
1	0	0	0	0	0
1	1	1	1	1	0

上面这个矩阵的岛数量是1；但如果从中间竖着切开，那么左边的岛数量是1，右边的岛数量是2，总数是3。如何处理切割后，相邻子矩阵之间的边界处的1相邻导致的重复统计呢？其实利用并查集的特性就很容易解决这个问题：



1	1	1	1	1	1
1	0	1	1	1	0
1	0	0	0	0	0
1	1	1	1	1	0

首先将切割边界处的数据封装成结点加入到并查集中并合并同一个岛上的结点，在分析边界时，查边界两边的1是否在同一个集合，如果不在那就`union`这两个结点，并将总的岛数量减1；否则就跳过此行继续分析下一行边界上的两个点。

贪心策略

拼接最小字典序

给定一个字符串类型的数组strs，找到一种拼接方式，使得把所有字符串拼起来之后形成的字符串具有最低的字典序。

> 此题很多人的想法是把数组按照字典序排序，然后从头到尾连接，形成的字符串就是所有拼接结果中字典序最小的那个。但这很容易证明是错的，比如`[ba,b]`的排序结果是`[b,ba]`，拼接结果是`bba`，但`bab`的字典序更小。
>
> 正确的策略是，将有序字符串数组从头到尾两两拼接时，应取两两拼接的拼接结果中字典序较小的那个。证明如下

如果令`.`代表拼接符号，那么这里的命题是如果`str1.str2 < str2.str2`且`str2.str3 < str3.str2`，那么一定有`str1.str3 < str3.str1`。这可以使用数学归纳法来证明。如果将`a~z`对应到`0~25`，比较两个字符串的字典序的过程，其实就比较两个26进制数大小的过程。`str1.str2`拼接的过程可以看做两个26进制数拼接的过程，若将两字符串解析成数字`int1`和`int2`，那么拼接就对应`int1 * 26^(str2的长度) + int2`，那么证明过程就变成了两个整数不等式递推另一个不等式了。

金条和铜板

一块金条切成两半，是需要花费和长度数值一样的铜板的。比如长度为20的 金条，不管切成长度多大的两半，都要花费20个铜板。一群人想整分整块金 条，怎么分最省铜板?

例如,给定数组{10,20,30}，代表一共三个人，整块金条长度为10+20+30=60. 金条要分成10,20,30三个部分。 如果， 先把长度60的金条分成10和50，花费60 再把长度50的金条分成20和30，花费50 一共花费110铜板。但是如果， 先把长度60的金条分成30和30，花费60 再把长度30金条分成10和20，花费30 一共花费90铜板。

输入一个数组，返回分割的最小代价。

> 贪心策略，将给定的数组中的元素扔进小根堆，每次从小根堆中先后弹出两个元素（如10和20），这两个元素的和（如30）就是某次分割得到这两个元素的花费，再将这个和扔进小根堆。直到小根堆中只有一个元素为止。（比如扔进30之后，弹出30、30，此次花费为30+30=60，再扔进60，堆中只有一个60了，结束，总花费30+60-=90）

```java
public stzuoatic int lessMoney(int arr[]){
  if (arr == null || arr.length == 0) {
    return 0;
  }
  //PriorityQueue是Java语言对堆结构的一个实现，默认将按自然顺序的最小元素放在堆顶
  PriorityQueue<Integer> minHeap = new PriorityQueue();
  for (int i : arr) {
    minHeap.add(i);
  }
  int res = 0;
  int curCost = 0;
  while (minHeap.size() > 1) {
    curCost = minHeap.poll() + minHeap.poll();
    res += curCost;
    minHeap.add(curCost);
  }
  return res;
}

public static void main(String[] args) {
  int arr[] = {10, 20, 30};
  System.out.println(lessMoney(arr));
}
```

IPO

输入： 参数1：正数数组costs；参数2：正数数组profits；参数3：正数k；参数4：正数m。costs[i]表示i号项目的花费（成本），profits[i]表示i号项目做完后在扣除花费之后还能挣到的钱(利润)，k表示你不能并行，只能串行的最多做k个项目 m表示你初始的资金。

说明：你每做完一个项目，马上获得的收益，可以支持你去做下一个项目。

输出： 你最后获得的最大钱数。

> 贪心策略：借助两个堆，一个是存放各个项目花费的小根堆、另一个是存放各个项目利润的大根堆。首先将所有项目放入小根堆而大根堆为空，对于手头上现有的资金（本金），将能做的项目（成本低于现有资金）从小根堆依次弹出并放入到大根堆，再弹出大根堆堆顶项目来完成，完成后根据利润更新本金。本金更新后，再将小根堆中能做的项目弹出加入到大根堆中，再弹出大根堆中的堆顶项目来做，重复此操作，直到某次本金更新和两个堆更新后大根堆无项目可做或者完成的项目个数已达k个为止。

```java
import java.util.Comparator;
import java.util.PriorityQueue;

public class IPO {

  public class Project{
    int cost;
    int profit;
    public Project(int cost, int profit) {
      this.cost = cost;
      this.profit = profit;
    }
  }

  public class MinCostHeap implements Comparator<Project> {
    @Override
    public int compare(Project p1, Project p2) {
      return p1.cost-p2.cost; //升序，由此构造的堆将把花费最小项目的放到堆顶
    }
  }

  public class MaxProfitHeap implements Comparator<Project> {
    @Override
    public int compare(Project p1, Project p2) {
      return p2.profit-p1.profit;
    }
  }

  public int findMaximizedCapital(int costs[], int profits[], int k, int m) {
    int res = 0;
    PriorityQueue<Project> minCostHeap = new PriorityQueue<>(new MinCostHeap());
    PriorityQueue<Project> maxProfitHeap = new PriorityQueue<>(new MaxProfitHeap());
    for (int i = 0; i < costs.length; i++) {
      Project project = new Project(costs[i], profits[i]);
      minCostHeap.add(project);
    }
    for (int i = 0; i < k; i++) {
      //unlock project
      while (minCostHeap.peek().cost < m) {
        maxProfitHeap.add(minCostHeap.poll());
      }
      if (maxProfitHeap.isEmpty()) {
        return m;
```

```
      }
      m +=  maxProfitHeap.poll().profit;
    }

    return m;
  }

}
```

会议室项目宣讲

一些项目要占用一个会议室宣讲，会议室不能同时容纳两个项目的宣讲。 给你每一个项目开始的时间和结束的时间(给你一个数组，里面 是一个个具体的项目)，你来安排宣讲的日程，要求会议室进行 的宣讲的场次最多。返回这个最多的宣讲场次。

> 贪心策略：
>
> 1、开始时间最早的项目先安排。反例：开始时间最早，但持续时间占了一整天，其他项目无法安排。
>
> 2、持续时间最短的项目先安排。反例：这样安排会导致结束时间在此期间和开始时间在此期间的所有项目不能安排。
>
> 3、最优策略：最先结束的项目先安排。

```
import java.util.Arrays;
import java.util.Comparator;

public class Schedule {

  public class Project {
    int start;
    int end;
  }

  public class MostEarlyEndComparator implements Comparator<Project> {
    @Override
    public int compare(Project p1, Project p2) {
      return p1.end-p2.end;
    }
  }

  public int solution(Project projects[],int currentTime) {
    //sort by the end time
    Arrays.sort(projects, new MostEarlyEndComparator());
    int res = 0;
    for (int i = 0; i < projects.length; i++) {
      if (currentTime <= projects[i].start) {
        res++;
        currentTime = projects[i].end;
      }
    }
    return res;
  }
```

```
30 }
```

> 经验：贪心策略相关的问题，累积经验就好，不必花费大量精力去证明。解题的时候要么找相似点，要么脑补策略然后用对数器、测试用例去证。

递归和动态规划

暴力递归：

1. 把问题转化为规模缩小了的同类问题的子问题
2. 有明确的不需要继续进行递归的条件(base case)
3. 有当得到了子问题的结果之后的决策过程
4. 不记录每一个子问题的解

动态规划：

1. 从暴力递归中来
2. 将每一个子问题的解记录下来，**避免重复计算**
3. 把暴力递归的过程，抽象成了状态表达
4. 并且存在化简状态表达，使其更加简洁的可能

P和NP

P指的是我明确地知道怎么算，计算的流程很清楚；而NP问题指的是我不知道怎么算，但我知道怎么尝试（暴力递归）。

暴力递归

n!问题

我们知道`n!`的定义，可以根据定义直接求解：

```
int getFactorial_1(int n){
  int res=1;
  for(int i = 1 ; i <= n ; n++){
    res*=i;
  }
  return res;
}
```

但我们可以这样想，如果知道`(n-1)!`，那通过`(n-1)! * n`不就得出`n!`了吗？于是我们就有了如下的尝试：

```
int getFactorial_2(int n){
  if(n=1)
    return 1;
  return getFactorial_2(n-1) * n;
}
```

`n!`的状态依赖`(n-1)!`，`(n-1)!`依赖`(n-2)!`，就这样依赖下去，直到`n=1`这个突破口，然后回溯，你会发现整个过程就回到了`1 * 2 * 3 * …… * (n-1) * n`的计算过程。

汉诺塔问题

该问题最基础的一个模型就是，一个竹竿上放了2个圆盘，需要先将最上面的那个移到辅助竹竿上，然后将最底下的圆盘移到目标竹竿，最后把辅助竹竿上的圆盘移回目标竹竿。

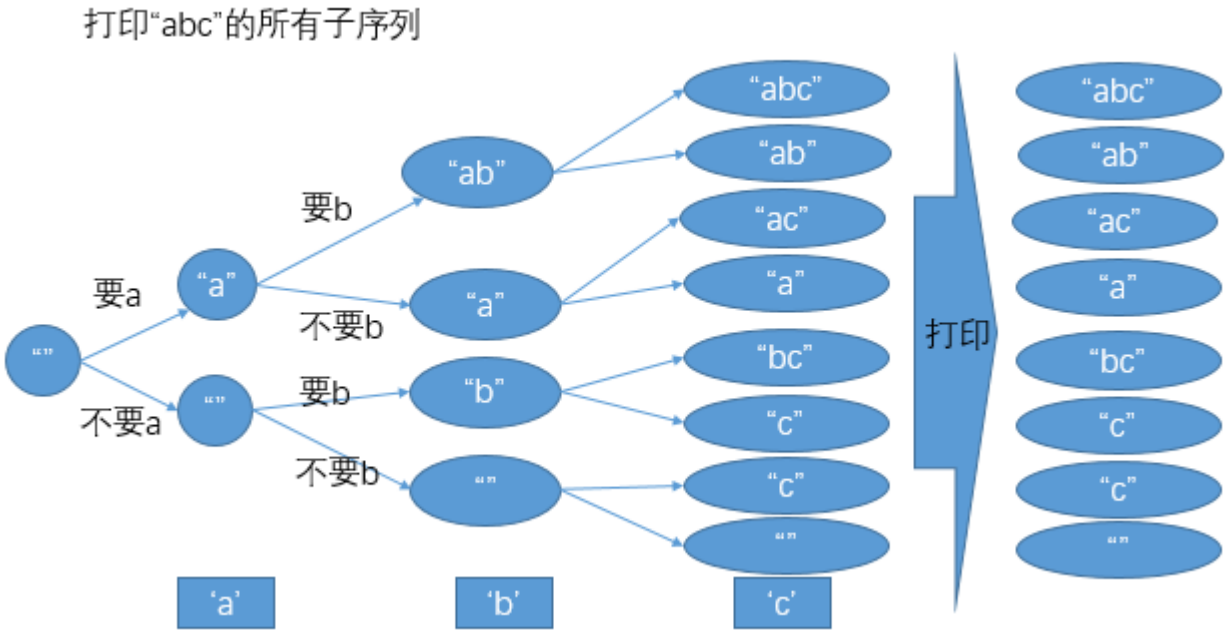
```java
public class Hanoi {

    public static void process(String source,String target,String auxiliary,int n){
        if (n == 1) {
            System.out.println("move 1 disk from " + source + " to " + target);
            return;
        }
        //尝试把前n-1个圆盘暂时放到辅助竹竿->子问题
        process(source, auxiliary, target, n - 1);
        //将底下最大的圆盘移到目标竹竿
        System.out.println("move 1 disk from "+source+" to "+target);
        //再尝试将辅助竹竿上的圆盘移回到目标竹竿->子问题
        process(auxiliary,target,source,n-1);
    }

    public static void main(String[] args) {
        process("Left", "Right", "Help", 3);
    }
}
```

根据Master公式计算得`T(N) = T(N-1)+1+T(N-1)`，时间复杂度为`O(2^N)`

打印一个字符串的所有子序列

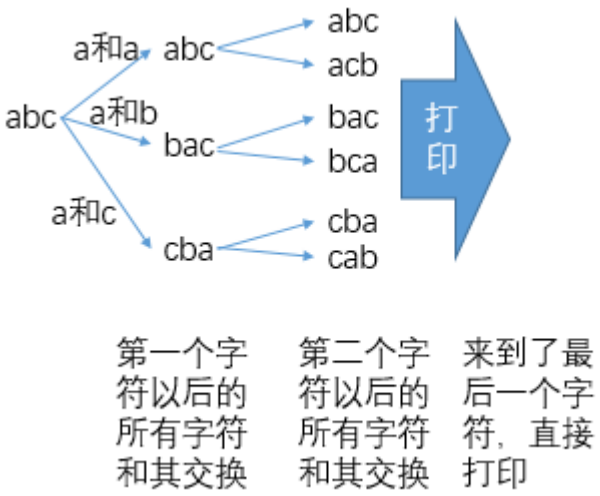
> 字符串的子序列和子串有着不同的定义。子串指串中相邻的任意个字符组成的串，而子序列可以是串中任意个不同字符组成的串。

尝试：开始时，令子序列为空串，扔给递归方法。首先来到字符串的第一个字符上，这时会有两个决策：将这个字符加到子序列和不加到子序列。这两个决策会产生两个不同的子序列，将这两个子序列作为这一级收集的信息扔给子过程，子过程来到字符串的第二个字符上，对上级传来的子序列又有两个决策，……这样最终能将所有子序列组合穷举出来：




```java
/**
     * 打印字符串的所有子序列-递归方式
     * @param str   目标字符串
     * @param index 当前子过程来到了哪个字符的决策上（要还是不要）
     * @param res   上级扔给本级的子序列
     */
public static void printAllSubSequences(String str,int index,String res) {
  //base case :  当本级子过程来到的位置到达串末尾，则直接打印
  if(index == str.length()) {
    System.out.println(res);
    return;
  }
  //决策是否要index位置上的字符
  printAllSubSequences(str, index+1, res+str.charAt(index));
  printAllSubSequences(str, index+1, res);
}

public static void main(String[] args) {
  printAllSubSequences("abc", 0, "");
}
```

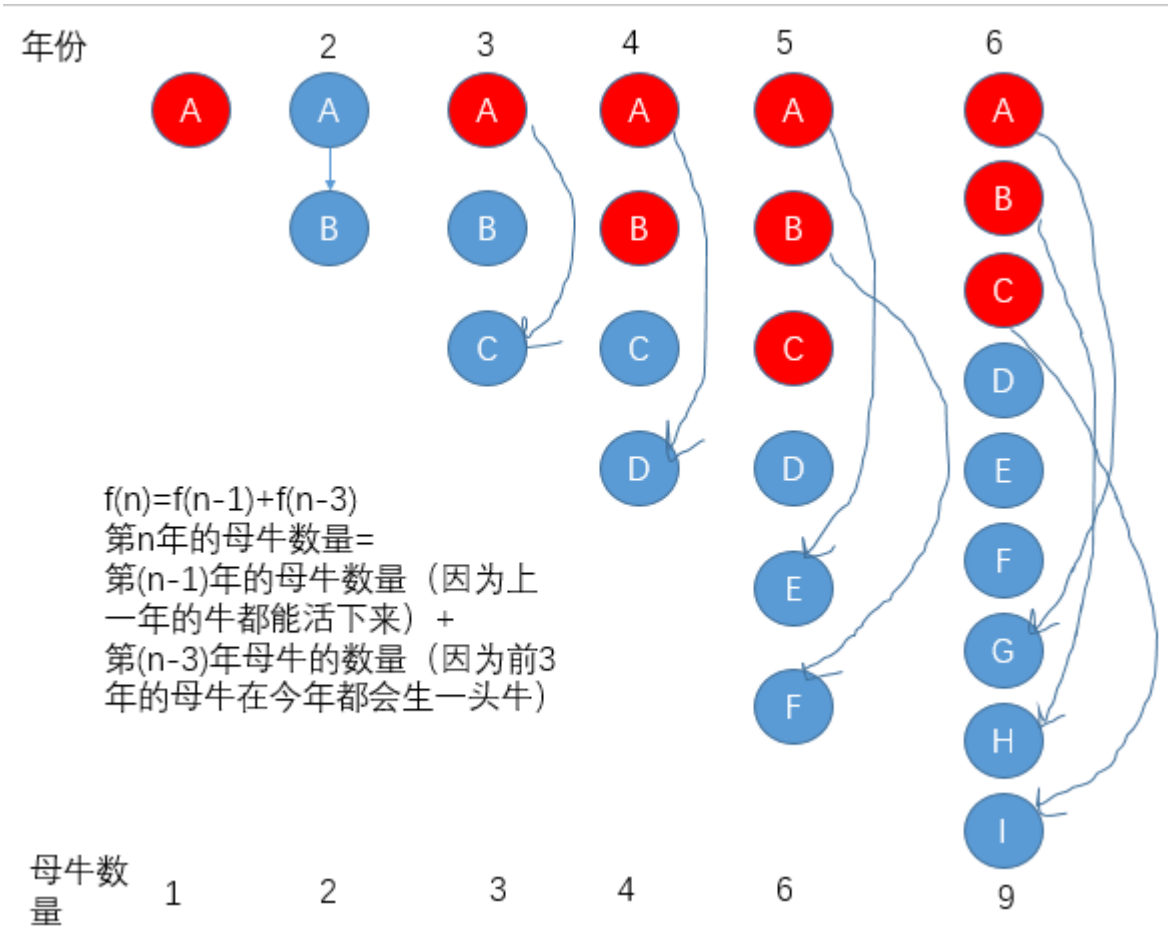
打印一个字符串的所有全排列结果




```java
/**
     * 本级任务：将index之后（包括index）位置上的字符和index上的字符交换，将产生的所有结果扔给下一级
     * @param str
     * @param index
     */
public static void printAllPermutations(char[] chs,int index) {
  //base case
  if(index == chs.length-1) {
    System.out.println(chs);
    return;
  }
  for (int j = index; j < chs.length; j++) {
    swap(chs,index,j);
    printAllPermutations(chs, index+1);
  }
}

public static void swap(char[] chs,int i,int j) {
  char temp = chs[i];
  chs[i] = chs[j];
  chs[j] = temp;
}

public static void main(String[] args) {
  printAllPermutations("abc".toCharArray(), 0);
}
```

母牛生牛问题

母牛每年生一只母牛，新出生的母牛成长三年后也能每年生一只母牛，假设不会死。求N年后，母牛的数量。



那么求第n年母牛的数量，按照此公式顺序计算即可，但这是`O(N)`的时间复杂度，存在`O(logN)`的算法（放到进阶篇中讨论）。

暴力递归改为动态规划

为什么要改动态规划？有什么意义？

动态规划由暴力递归而来，是对暴力递归中的重复计算的一个优化，策略是空间换时间。

最小路径和

给你一个二维数组，二维数组中的每个数都是正数，要求从左上角走到右下角，每一步只能向右或者向下。沿途经过的数字要累加起来。返回最小的路径和。

递归尝试版本

```
/**
     * 从矩阵matrix的(i,j)位置走到右下角元素，返回最小沿途元素和。每个位置只能向右或向下
     *
     * @param matrix
     * @param i
     * @param j
     * @return 最小路径和
     */
public static int minPathSum(int matrix[][], int i, int j) {
  // 如果(i,j)就是右下角的元素
```

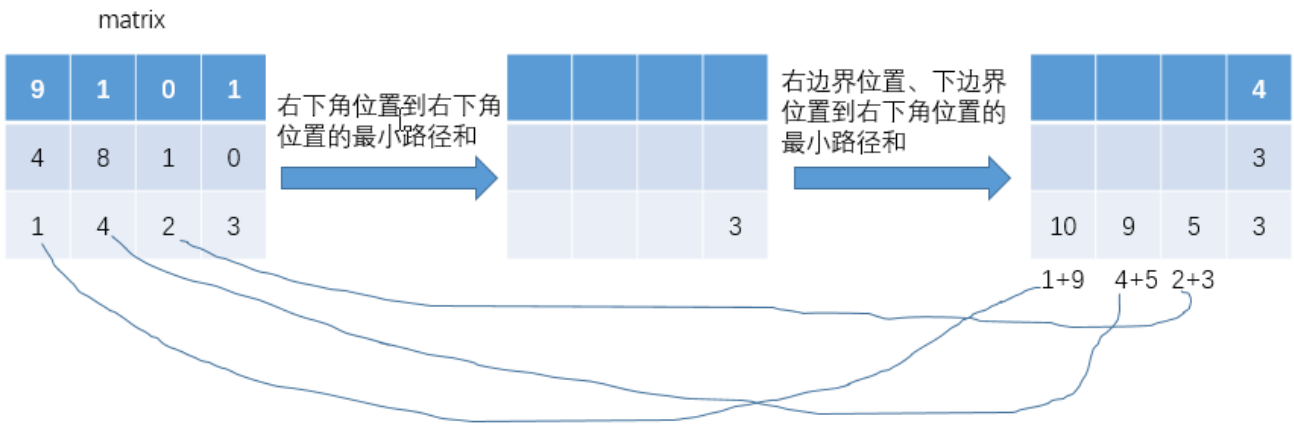
```
  if (i == matrix.length - 1 && j == matrix[0].length - 1) {
    return matrix[i][j];
  }
  // 如果(i,j)在右边界上，只能向下走
  if (j == matrix[0].length - 1) {
    return matrix[i][j] + minPathSum(matrix, i + 1, j);
  }
  // 如果(i,j)在下边界上，只能向右走
  if (i == matrix.length - 1) {
    return matrix[i][j] + minPathSum(matrix, i, j + 1);
  }
  // 不是上述三种情况，那么(i,j)就有向下和向右两种决策，取决策结果最小的那个
  int left = minPathSum(matrix, i, j + 1);
  int down = minPathSum(matrix, i + 1, j);
  return matrix[i][j] + Math.min(left,down );
}

public static void main(String[] args) {
  int matrix[][] = {
    { 9, 1, 0, 1 },
    { 4, 8, 1, 0 },
    { 1, 4, 2, 3 }
  };
  System.out.println(minPathSum(matrix, 0, 0)); //14
}
```

根据尝试版本改动态规划

上述暴力递归的缺陷在于有些子过程是重复的。比如 `minPathSum(matrix,0,1)` 和 `minPathSum(matrix,1,0)` 都会依赖子过程 `minPathSum(matrix,1,1)` 的状态（执行结果），那么在计算 `minPathSum(matrix,0,0)` 时势必会导致 `minPathSum(matrix,1,1)` 的重复计算。那我们能否通过对子过程计算结果进行缓存，在再次需要时直接使用，从而实现对整个过程的一个优化呢。

由暴力递归改动态规划的核心就是将每个子过程的计算结果进行一个记录，从而达到空间换时间的目的。那么 `minPath(int matrix[][],int i,int j)` 中变量 `i` 和 `j` 的不同取值将导致 `i*j` 种结果，我们将这些结果保存在一个 `i*j` 的表中，不就达到动态规划的目的了吗?

观察上述代码可知，右下角、右边界、下边界这些位置上的元素是不需要尝试的（只有一种走法，不存在决策问题），因此我们可以直接将这些位置上的结果先算出来：



而其它位置上的元素的走法则依赖右方相邻位置（i，j+1）走到右下角的最小路径和和下方相邻位置（i+1，j）走到右下角的最小路径和的大小比较，基于此来做一个向右走还是向左走的决策。但由于右边界、下边界位置上的结果我们已经计算出来了，因此对于其它位置上的结果也就不难确定了：

9+5=14	1+4=5	0+4=4	4
14	8+4=12	1+3=4	3
10	9	5	3

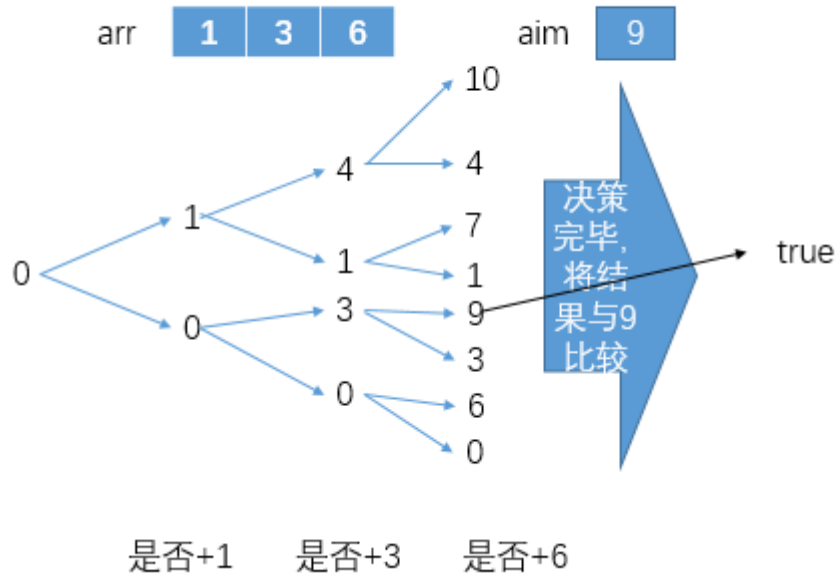
我们从`base case`开始，倒着推出了所有子过程的计算结果，并且没有重复计算。最后`minPathSum(matrix,0,0)`也迎刃而解了。

> 这就是动态规划，它不是凭空想出来的。首先我们尝试着解决这个问题，写出了暴力递归。再由暴力递归中的变量的变化范围建立一张对应的结果记录表，以`base case`作为突破口确定能够直接确定的结果，最后解决普遍情况对应的结果。

一个数是否是数组中任意个数的和

给你一个数组arr，和一个整数aim。如果可以任意选择arr中的数字，能不能累加得到aim，返回true或者false。

> 此题的思路跟求解一个字符串的所有子序列的思路一致，穷举出数组中所有任意个数相加的不同结果。



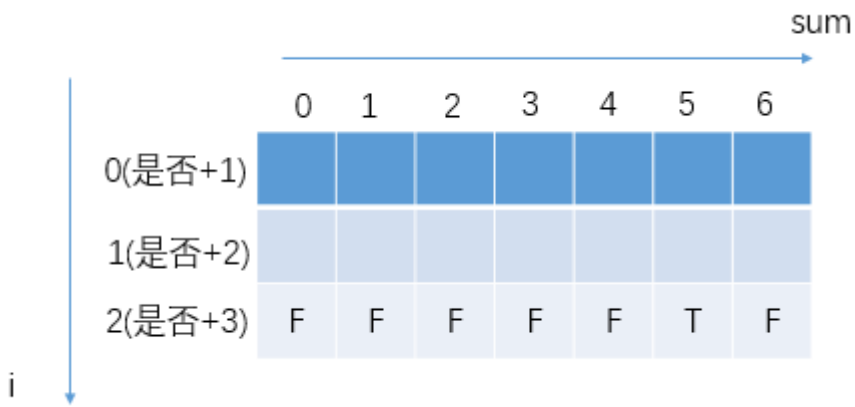
暴力递归版本

```
/**
     * 选择任意个arr中的元素相加是否能得到aim
     *
     * @param arr
     * @param aim
     * @param sum 上级扔给我的结果
```

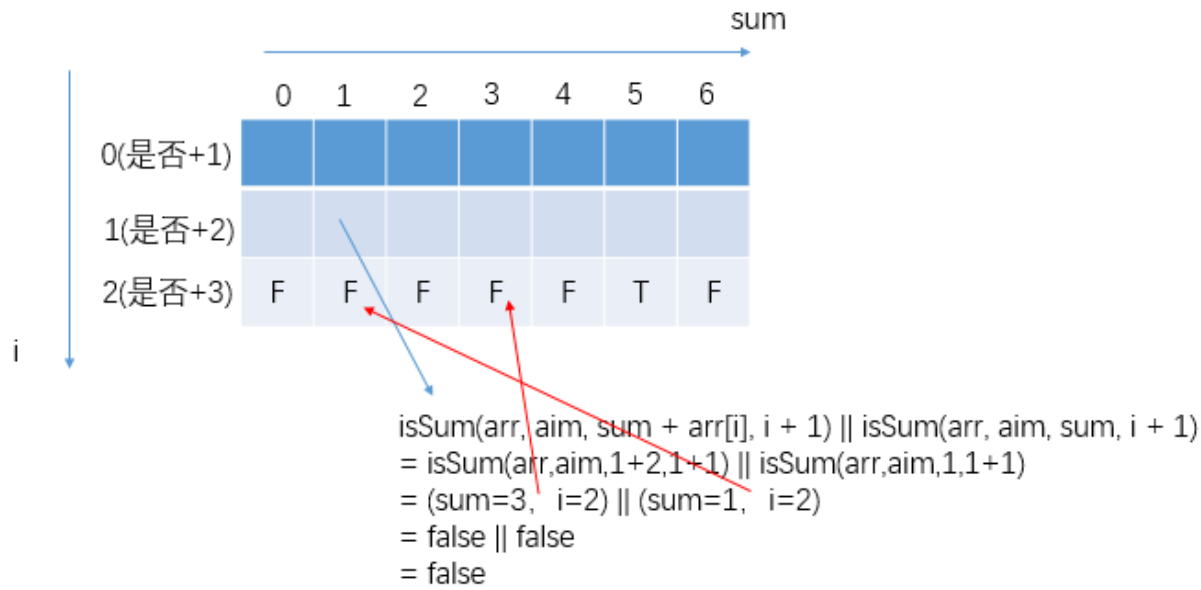
```
     * @param i    决策来到了下标为i的元素上
     * @return
     */
public static boolean isSum(int arr[], int aim, int sum,int i) {
  //决策完毕
  if (i == arr.length) {
    return sum == aim;
  }
  //决策来到了arr[i]：加上arr[i]或不加上。将结果扔给下一级
  return isSum(arr, aim, sum + arr[i], i + 1) || isSum(arr, aim, sum, i + 1);
}

public static void main(String[] args) {
  int arr[] = {1, 2, 3};
  System.out.println(isSum(arr, 5, 0, 0));
  System.out.println(isSum(arr, 6, 0, 0));
  System.out.println(isSum(arr, 7, 0, 0));
}
```

暴力递归改动态规划（高度套路）

1. 首先看递归函数的参数，找出变量。这里 `arr` 和 `aim` 是固定不变的，可变的只有 `sum` 和 `i` 。
2. 对应变量的变化范围建立一张表保存不同子过程的结果，这里 `i` 的变化范围是 `0~arr.length-1` 即 `0~2` ，而 `sum` 的变化范围是 `0~数组元素总和` ，即 `0~6` 。因此需要建一张 `3*7` 的表。
3. 从 `base case` 入手，计算可直接计算的子过程，以 `isSum(5,0,0)` 的计算为例，其子过程中“是否+3”的决策之后的结果是可以确定的：



4. 按照递归函数中 `base case` 下的尝试过程，推出其它子过程的计算结果，这里以 `i=1,sum=1` 的推导为例：



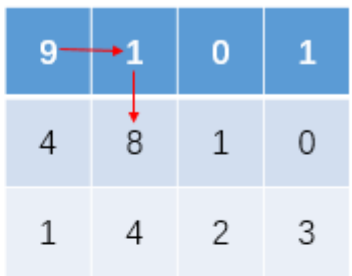
哪些暴力递归能改为动态规划

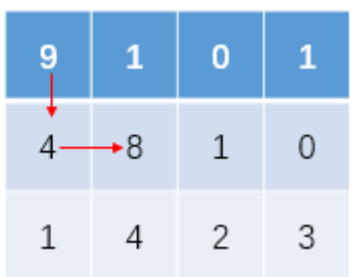
看过上述例题之后你会发现只要你能够写出尝试版本，那么改动态规划是高度套路的。但是不是所有的暴力递归都能够改动态规划呢？不是的，比如汉诺塔问题和N皇后问题，他们的每一步递归都是必须的，没有多余。这就涉及到了递归的有后效性和无后效性。

有后效性和无后效性

无后效性是指对于递归中的某个子过程，其上级的决策对该级的后续决策没有任何影响。比如最小路径和问题中以下面的矩阵为例：

9	1	0	1
4	8	1	0
1	4	2	3



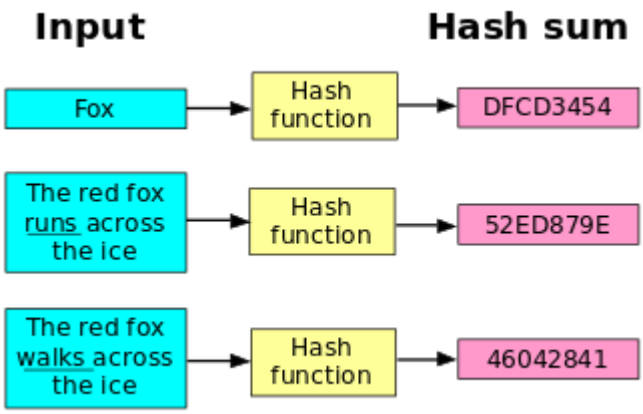


对于（1，1）位置上的8，无论是通过`9->1->8`还是`9->4->8`来到这个`8`上的，这个`8`到右下角的最小路径和的计算过程不会改变。这就是无后效性。

> 只有无后效性的暴力递归才能改动态规划。

哈希

哈希函数



> 百科：**散列函数**（英语：Hash function）又称**散列算法**、**哈希函数**，是一种从任何一种数据中创建小的**数字**"指纹"的方法。散列函数把消息或数据压缩成摘要，使得数据量变小，将数据的格式固定下来。该函数将输入域中的数据打乱混合，重新创建一个叫做**散列值**（hash values，hash codes，hash sums，或hashes）的指纹。

哈希函数的性质

哈希函数的输入域可以是非常大的范围，比如，任意一个字符串，但是输出域是固定的范围（一定位数的bit），假设为S，并具有如下性质：

1. 典型的哈希函数都有无限的输入值域。
2. 当给哈希函数传入相同的输入值时，返回值一样。
3. 当给哈希函数传入不同的输入值时，返回值可能一样，也可能不一样，这时当然的，因为输出域统一是S，所以会有不同的输入值对应在S中的一个元素上（这种情况称为 **哈希冲突**）。
4. 最重要的性质是很多不同的输入值所得到的返回值会均匀分布在S上。

前3点性质是哈希函数的基础，第4点是评价一个哈希函数优劣的关键，不同输入值所得到的所有返回值越均匀地分布在S上，哈希函数越优秀，并且这种均匀分布与输入值出现的规律无关。比如，"aaa1"、"aaa2"、"aaa3"三个输入值比较类似，但经过优秀的哈希函数计算后得到的结果应该相差非常大。

哈希函数的经典实现

> 参考文献：哈希函数的介绍

比如使用MD5对"test"和"test1"两个字符串哈希的结果如下（哈希结果为128个bit，数据范围为`0~(2^128)-1`，通常转换为32个16进制数显示）：

```
test    098f6bcd4621d373cade4e832627b4f6
test1 5a105e8b9d40e1329780d62ea2265d8a
```

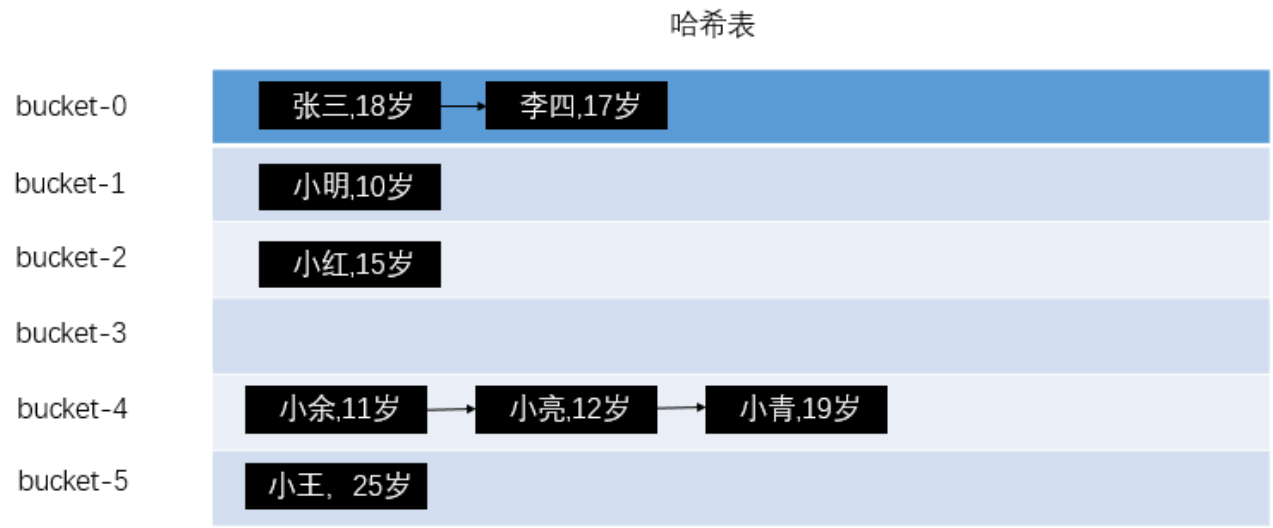
哈希表

> 百科：**散列表**（**Hash table**，也叫**哈希表**），是根据键（Key）而直接访问在内存存储位置的数据结构。也就是说，它通过计算一个关于键值的函数，将所需查询的数据映射到表中一个位置来访问记录，这加快了查找速度。这个映射函数称做散列函数，存放记录的数组称做**散列表**。

哈希表的经典实现

哈希表初始会有一个大小，比如16，表中每个元素都可以通过数组下标（0~15）访问。每个元素可以看做一个桶，当要往表里放数据时，将要存放的数据的键值通过哈希函数计算出的哈希值模上16，结果正好对应0~15，将这条数据放入对应下标的桶中。

那么当数据量超过16时，势必会存在哈希冲突（两条数据经哈希计算后放入同一个桶中），这时的解决方案就是将后一条入桶的数据作为后继结点链入到桶中已有的数据之后，如此，每个桶中存放的就是一个链表。那么这就是哈希表的经典结构：



当数据量较少时，哈希表的增删改查操作的时间复杂度都是`O(N)`的。因为根据一个键值就能定位一个桶，即使存在哈希冲突（桶里不只一条数据），但只要哈希函数优秀，数据量几乎均分在每个桶上（这样很少有哈希冲突，即使有，一个桶里也只会有很少的几条数据），那就在遍历一下桶里的链表比较键值进一步定位数据即可（反正链表很短）。

哈希表扩容

如果哈希表大小为16，对于样本规模N（要存储的数据数量）来说，如果N较小，那么根据哈希函数的散列特性，每个桶会均分这N条数据，这样落到每个桶的数据量也较小，不会影响哈希表的存取效率（这是由桶的链表长度决定的，因为存数据要往链表尾追加首先就要遍历得到尾结点，取数据要遍历链表比较键值）；但如果N较大，那么每个桶里都有`N/16`条数据，存取效率就变成`O(N)`了。因此哈希表哈需要一个扩容机制，当表中某个桶的数据量超过一个阀值时（`O(1)`到`O(N)`的转变，这需要一个算法来权衡），需要将哈希表扩容（一般是成倍的）。

扩容步骤是，创建一个新的较大的哈希表（假如大小为m），将原哈希表中的数据取出，将键值的哈希值模上m，放入新表对应的桶中，这个过程也叫`rehash`。

如此的话，那么原来的`O(N)`就变成了`O(log(m/16,N))`，比如扩容成5倍那就是`O(log(5,N))`（以5为底，N的对数）。当这个底数较大的时候就会将N的对数压得非常低而和`O(1)`非常接近了，并且实际工程中基本是当成`O(1)`来用的。

你也许会说`rehash`很费时，会导致哈希表性能降低，这一点是可以侧面避免的。比如扩容时将倍数提高一些，那么`rehash`的次数就会很少，平衡到整个哈希表的使用来看，影响就甚微了。或者可以进行**离线扩容**，当需要扩容时，原哈希表还是供用户使用，在另外的内存中执行`rehash`，完成之后再将新表替换原表，这样的话对于用户来说，他是感觉不到`rehash`带来的麻烦的。

哈希表的JVM实现

在`Java`中，哈希表的实现是每个桶中放的是一棵**红黑树**而非链表，因为红黑树的查找效率很高，也是对哈希冲突带来的性能问题的一个优化。

布隆过滤器

不安全网页的黑名单包含100亿个黑名单网页，每个网页的URL最多占用64B。现在想要实现一种网页过滤系统，可以根据网页的URL判断该网页是否在黑名单上，请设计该系统。

要求如下：

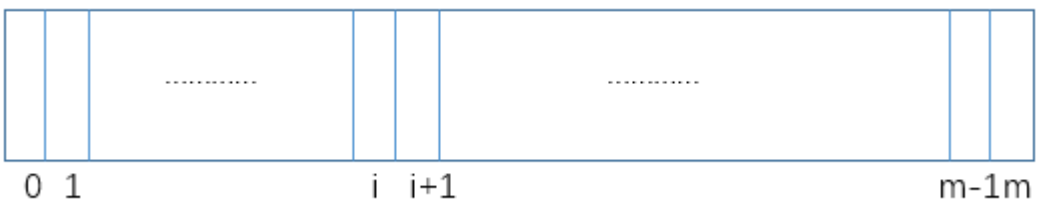
1. 该系统允许有万分之一以下的判断失误率。
2. 使用的额外空间不要超过30GB。

如果将这100亿个URL通过数据库或哈希表保存起来，就可以对每条URL进行查询，但是每个URL有64B，数量是100亿个，所以至少需要640GB的空间，不满足要求2。

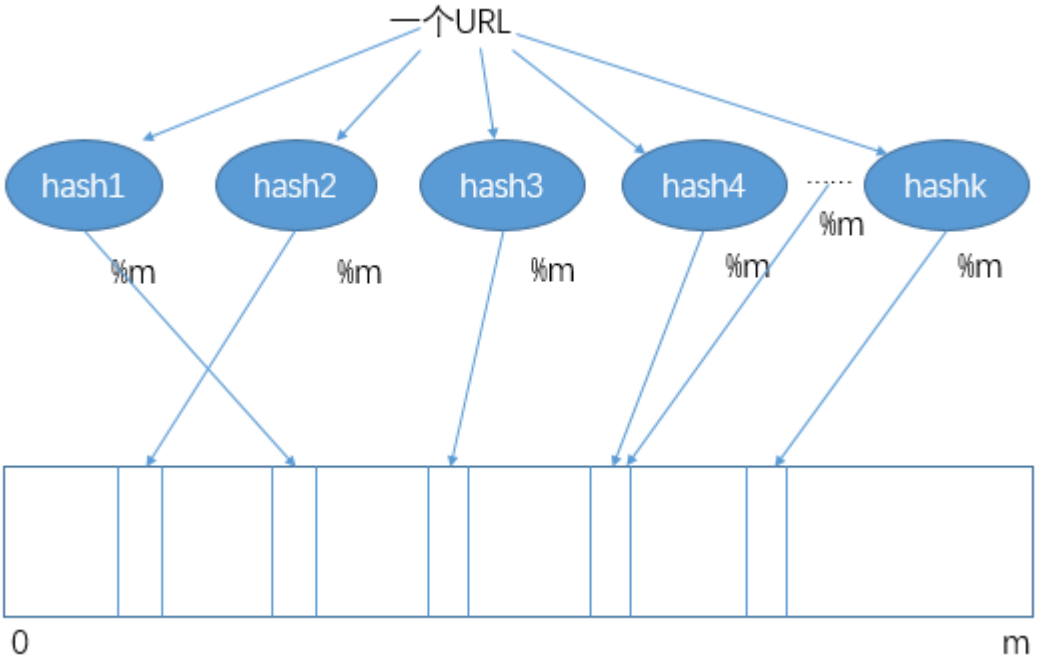
> 如果面试者遇到网页黑名单系统、垃圾邮件过滤系统，爬虫的网页判重系统等题目，又看到系统容忍一定程度的失误率，但是对空间要求比较严格，那么很可能是面试官希望面试者具备布隆过滤器的知识。一个布隆过滤器精确地代表一个集合，并可以精确判断一个元素是否在集合中。注意，只是精确代表和精确判断，到底有多精确呢？则完全在于你具体的设计，但想做到完全正确是不可能的。布隆过滤器的优势就在于使用很少的空间就可以将准确率做到很高的程度。该结构由`Burton Howard Bloom`于1970年提出。

那么什么是布隆过滤器呢？

假设有一个长度为`m`的bit类型的数组，即数组的每个位置只占一个bit，如果我们所知，每一个bit只有0和1两种状态，如图所示：



再假设一共有`k`个哈希函数，这些函数的输出域S都大于或等于m，并且这些哈希函数都足够优秀且彼此之间相互独立（将一个哈希函数的计算结果乘以6除以7得出的新哈希函数和原函数就是相互独立的）。那么对同一个输入对象（假设是一个字符串，记为URL），经过k个哈希函数算出来的结果也是独立的。可能相同，也可能不同，但彼此独立。对算出来的每一个结果都对m取余（%m），然后在bit array 上把相应位置设置为1（我们形象的称为涂黑）。如图所示



我们把bit类型的数组记为`bitMap`。至此，一个输入对象对`bitMap`的影响过程就结束了，也就是`bitMap`的一些位置会被涂黑。接下来按照该方法，处理所有的输入对象（黑名单中的100亿个URL）。每个对象都可能把`bitMap`中的一些白位置涂黑，也可能遇到已经涂黑的位置，遇到已经涂黑的位置让其继续为黑即可。处理完所有的输入对象后，可能`bitMap`中已经有相当多的位置被涂黑。至此，一个布隆过滤器生成完毕，这个布隆过滤器代表之前所有输入对象组成的集合。

那么在检查阶段时，如何检查一个对象是否是之前的某一个输入对象呢（判断一个URL是否是黑名单中的URL）？假设一个对象为a，想检查它是否是之前的输入对象，就把a通过k个哈希函数算出k个值，然后把k个值都取余（%m），就得到在[0,m-1]范围伤的k个值。接下来在`bitMap`上看这些位置是不是都为黑。如果有一个不为黑，说明a一定不再这个集合里。如果都为黑，说明a在这个集合里，但可能误判。

再解释具体一点，如果a的确是输入对象，那么在生成布隆过滤器时，`bitMap`中相应的k个位置一定已经涂黑了，所以在检查阶段，a一定不会被漏过，这个不会产生误判。会产生误判的是，a明明不是输入对象，但如果在生成布隆过滤器的阶段因为输入对象过多，而`bitMap`过小，则会导致`bitMap`绝大多数的位置都已经变黑。那么在检查a时，可能a对应的k个位置都是黑的，从而错误地认为a是输入对象（即是黑名单中的URL）。通俗地说，布隆过滤器的失误类型是“宁可错杀三千，绝不放过一个”。

布隆过滤器到底该怎么生成呢？只需记住下列三个公式即可：

- 对于输入的数据量n（这里是100亿）和失误率p（这里是万分之一），布隆过滤器的大小m：`m = -(n*lnp)/(ln2*ln2)`，计算结果向上取整（这道题m=19.19n，向上取整为20n，即需要2000亿个bit，也就是25GB）
- 需要的哈希函数的个数k：`k = ln2 * m/n = 0.7 * m/n`（这道题`k = 0.7 * 20n/n = 14`）
- 由于前两步都进行了向上取整，那么由前两步确定的布隆过滤器的真正失误率p：`p = (1 - e^(-nk/m))^k`

一致性哈希算法的基本原理

题目

工程师常使用服务器集群来设计和实现数据缓存，以下是常见的策略：

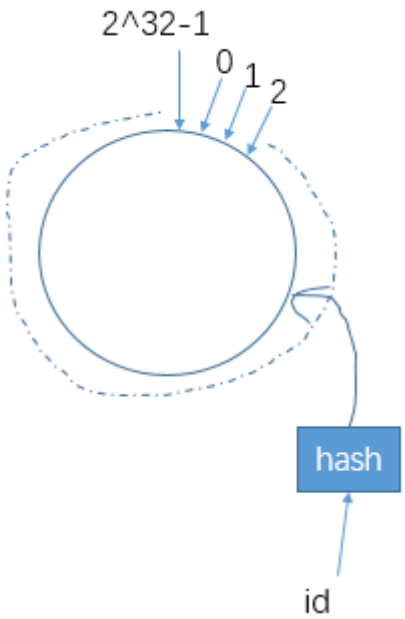
1. 无论是添加、查询还是珊瑚数据，都先将数据的id通过哈希函数换成一个哈希值，记为key
2. 如果目前机器有N台，则计算 `key%N` 的值，这个值就是该数据所属的机器编号，无论是添加、删除还是查询操作，都只在这台机器上进行。

请分析这种缓存策略可能带来的问题，并提出改进的方案。

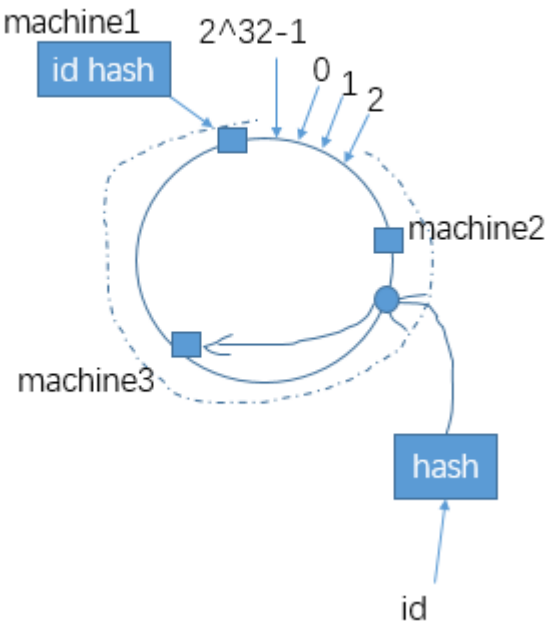
解析

题目中描述的缓存从策略的潜在问题是，如果增加或删除机器时（N变化）代价会很高，所有的数据都不得不根据id重新计算一遍哈希值，并将哈希值对新的机器数进行取模啊哦做。然后进行大规模的数据迁移。

为了解决这些问题，下面介绍一下一致性哈希算法，这时一种很好的数据缓存设计方案。我们假设数据的id通过哈希函数转换成的哈希值范围是2^32，也就是0~(2^32)-1的数字空间中。现在我们可以将这些数字头尾相连，想象成一个闭合的环形，那么一个数据id在计算出哈希值之后认为对应到环中的一个位置上，如图所示



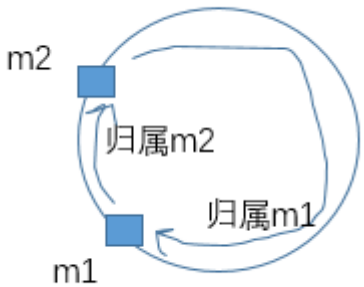
接下来想象有三台机器也处在这样一个环中，这三台机器在环中的位置根据机器id（主机名或者主机IP，是主机唯一的就行）设计算出的哈希值对2^32取模对应到环上。那么一条数据如何确定归属哪台机器呢？我们可以在该数据对应环上的位置顺时针寻找离该位置最近的机器，将数据归属于该机器上：



这样的话，如果删除`machine2`节点，则只需将`machine2`上的数据迁移到`machine3`上即可，而不必大动干戈迁移所有数据。当添加节点的时候，也只需将新增节点到逆时针方向新增节点前一个节点这之间的数据迁移给新增节点即可。

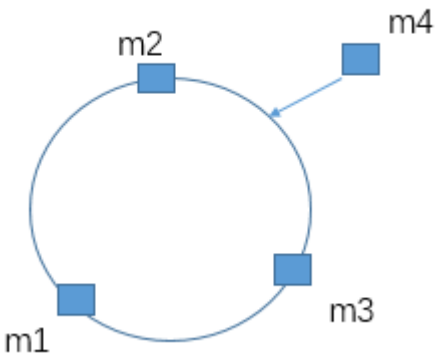
但这时还是存在如下两个问题：

- 机器较少时，通过机器id哈希将机器对应到环上之后，几个机器可能没有均分环

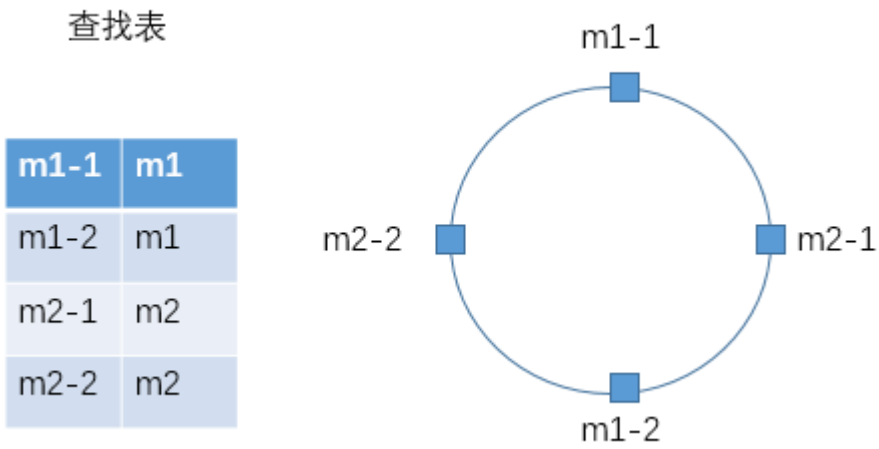


  那么这样会导致负载不均。

- 增加机器时，可能会打破现有的平衡：



为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一台机器通过不同的哈希函数计算出多个哈希值，对多个位置都放置一个服务节点，称为虚拟节点。具体做法：比如对于`machine1`的IP`192.168.25.132`（或机器名），计算出`192.168.25.132-1`、`192.168.25.132-2`、`192.168.25.132-3`、`192.168.25.132-4`的哈希值，然后对应到环上，其他的机器也是如此，这样的话节点数就变多了，根据哈希函数的性质，平衡性自然会变好：



此时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，比如上图的查找表。当某一条数据计算出归属于`m2-1`时再根据查找表的跳转，数据将最终归属于实际的m1节点。

> 基于一致性哈希的原理有很多种具体的实现，包括Chord算法、KAD算法等，有兴趣的话可以进一步学习。

RandomPool

设计一种结构，在该结构中有如下三个功能：

- inserrt(key)：将某个key加入到该结构中，做到不重复加入。
- delete(key)：将原本在结构中的某个key移除。
- getRandom()：等概率随机返回结构中的任何一个key。

要求：insert、delete和getRandom方法的时间复杂度都是`O(1)`

> 思路：使用两个哈希表和一个变量`size`，一个表存放某`key`的标号，另一个表根据根据标号取某个`key`。`size`用来记录结构中的数据量。加入`key`时，将`size`作为该`key`的标号加入到两表中；删除`key`时，将标号最大的`key`替换它并将`size--`；随机取`key`时，将`size`范围内的随机数作为标号取`key`。

```java
import java.util.HashMap;

public class RandomPool {
    public int size;
    public HashMap<Object, Integer> keySignMap;
    public HashMap<Integer, Object> signKeyMap;

    public RandomPool() {
        this.size = 0;
        this.keySignMap = new HashMap<>();
```

```
        this.signKeyMap = new HashMap<>();
    }

    public void insert(Object key) {
        //不重复添加
        if (keySignMap.containsKey(key)) {
            return;
        }
        keySignMap.put(key, size);
        signKeyMap.put(size, key);
        size++;
    }

    public void delete(Object key) {
        if (keySignMap.containsKey(key)) {
            Object lastKey = signKeyMap.get(--size);
            int deleteSign = keySignMap.get(key);
            keySignMap.put(lastKey, deleteSign);
            signKeyMap.put(deleteSign, lastKey);
            keySignMap.remove(key);
            signKeyMap.remove(lastKey);
        }
    }

    public Object getRandom() {
        if (size > 0) {
            return signKeyMap.get((int) (Math.random() * size));
        }
        return null;
    }

}
```

小技巧

对数器

概述

有时我们对编写的算法进行测试时，会采用自己编造几个简单数据进行测试。然而别人测试时可能会将大数量级的数据输入进而测试算法的准确性和健壮性，如果这时出错，面对庞大的数据量我们将无从查起（是在操作哪一个数据时出了错，算法没有如期起作用）。当然我们不可能对这样一个大数据进行断点调试，去一步一步的分析错误点在哪。这时 **对数器** 就粉墨登场了，**对数器** 就是通过随机制造出几乎所有可能的简短样本作为算法的输入样本对算法进行测试，这样大量不同的样本从大概率上保证了算法的准确性，当有样本测试未通过时又能打印该简短样本对错误原因进行分析。

对数器的使用

1. 对于你想测试的算法
2. 实现功能与该算法相同但绝对正确、复杂度不好的算法

3. 准备大量随机的简短样本的
4. 实现比对的方法：对于每一个样本，比对该算法和第二步中算法的执行结果以判断该算法的正确性
5. 如果有一个样本比对出错则打印该样本
6. 当样本数量很多时比对测试依然正确，可以确定算法a已经正确

对数器使用案例——对自写的插入排序进行测试：

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

//1.有一个自写的算法，但不知其健壮性（是否会有特殊情况使程序异常中断甚至崩溃）和正确性
void insertionSort(int arr[], int length){
    if(arr==NULL || length<=1){
        return;
    }
    for (int i = 1; i < length; ++i) {
        for (int j = i - 1; j >= 0 || arr[j] <= arr[j + 1]; j--) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}

//2、实现一个功能相同、绝对正确但复杂度不好的算法（这里摘取大家熟知的冒泡排序）
void bubbleSort(int arr[], int length) {
    for (int i = length-1; i > 0; i--) {
        for (int j = 0; j < i; ++j) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}

//3、实现一个能够产生随机简短样本的方法
void generateSample(int arr[], int length){
    for (int i = 0; i < length; ++i) {
        arr[i] = rand() % 100-rand()%100;//控制元素在-100~100之间，考虑到零正负三种情况
    }
}

//4、实现一个比对测试算法和正确算法运算结果的方法
bool isEqual(int arr1[],int arr2[],int length) {
    if (arr1 != NULL && arr2 != NULL) {
        for (int i = 0; i < length; ++i) {
            if (arr1[i] != arr2[i]) {
                return false;
            }
        }
```

```
        return true;
    }
    return false;
}

void travels(int arr[], int length){
    for (int i = 0; i < length; ++i) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void copy(int source[], int target[],int length){
    for (int i = 0; i < length; ++i) {
        target[i] = source[i];
    }
}

int main(){

    srand(time(NULL));
    int testTimes=10000;
    //循环产生100000个样本进行测试
    for (int i = 0; i < testTimes; ++i) {
        int length = rand() % 10;   //控制每个样本的长度在10以内，便于出错时分析样本（因为简短）
        int arr[length];
        generateSample(arr, length);

        //不要改变原始样本，在复制样本上改动
        int arr1[length], arr2[length];
        copy(arr, arr1, length);
        copy(arr, arr2, length);
        bubbleSort(arr1,length);
        insertionSort(arr2, length);

//        travels(arr, length);
//        travels(arr1, length);

        //5、比对两个算法，只要有一个样本没通过就终止，并打印原始样本
        if (!isEqual(arr1, arr2, length)) {
            printf("test fail!the sample is: ");
            travels(arr, length);
            return 0;
        }
    }

    //6、测试全部通过，该算法大概率上正确
    printf("nice!");
    return 0;
}
```

打印二叉树

有时我们不确定二叉树中是否有指针连空了或者连错了，这时需要将二叉树具有层次感地打印出来，下面就提供了这样一个工具。你可以将你的头逆时针旋转90度看打印结果。`v`表示该结点的头结点是左下方距离该结点最近的一个结点，`^`表示该结点的头结点是左上方距离该结点最近的一个结点。

```
package top.zhenganwen.algorithmdemo.recursive;

public class PrintBinaryTree {

    public static class Node {
        public int value;
        public Node left;
        public Node right;

        public Node(int data) {
            this.value = data;
        }
    }

    public static void printTree(Node head) {
        System.out.println("Binary Tree:");
        printInOrder(head, 0, "H", 17);
        System.out.println();
    }

    public static void printInOrder(Node head, int height, String to, int len) {
        if (head == null) {
            return;
        }
        printInOrder(head.right, height + 1, "v", len);
        String val = to + head.value + to;
        int lenM = val.length();
        int lenL = (len - lenM) / 2;
        int lenR = len - lenM - lenL;
        val = getSpace(lenL) + val + getSpace(lenR);
        System.out.println(getSpace(height * len) + val);
        printInOrder(head.left, height + 1, "^", len);
    }

    public static String getSpace(int num) {
        String space = " ";
        StringBuffer buf = new StringBuffer("");
        for (int i = 0; i < num; i++) {
            buf.append(space);
        }
        return buf.toString();
    }

    public static void main(String[] args) {
        Node head = new Node(1);
        head.left = new Node(-222222222);
```

```
        head.right = new Node(3);
        head.left.left = new Node(Integer.MIN_VALUE);
        head.right.left = new Node(55555555);
        head.right.right = new Node(66);
        head.left.left.right = new Node(777);
        printTree(head);

        head = new Node(1);
        head.left = new Node(2);
        head.right = new Node(3);
        head.left.left = new Node(4);
        head.right.left = new Node(5);
        head.right.right = new Node(6);
        head.left.left.right = new Node(7);
        printTree(head);

        head = new Node(1);
        head.left = new Node(1);
        head.right = new Node(1);
        head.left.left = new Node(1);
        head.right.left = new Node(1);
        head.right.right = new Node(1);
        head.left.left.right = new Node(1);
        printTree(head);

    }

}
```

递归的实质和Master公式

递归的实质

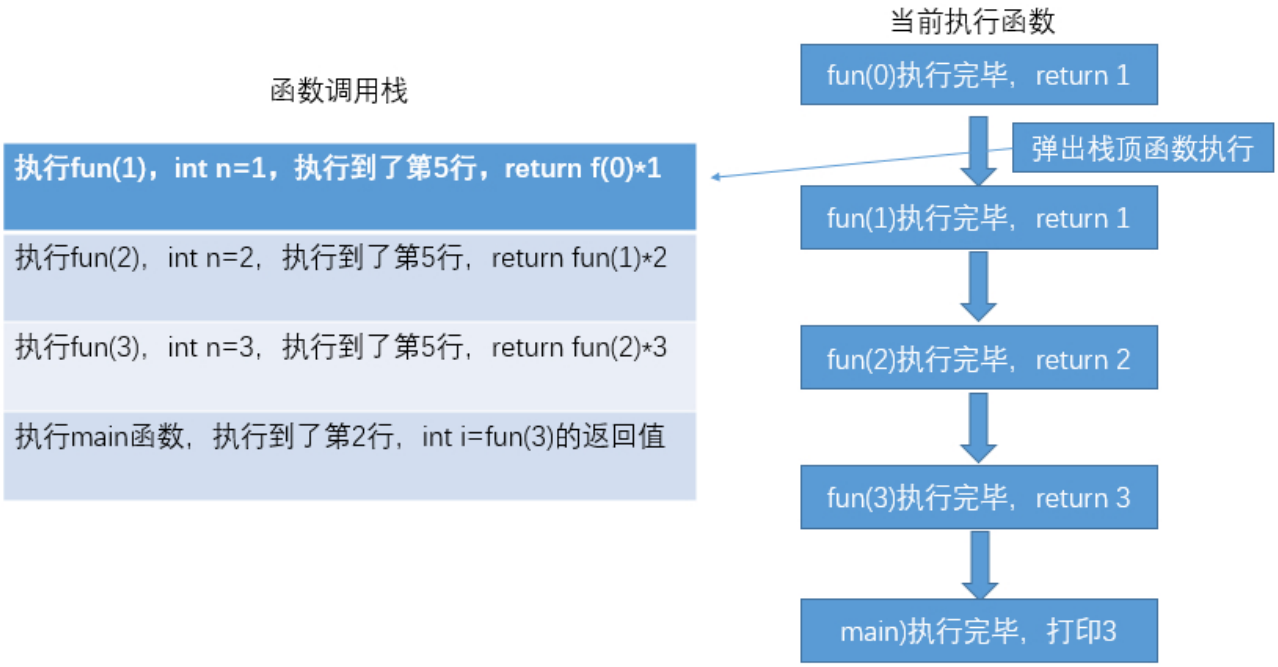
递归的实质就是系统在帮我们压栈。首先让我们来看一个递归求阶乘的例子：

```
int fun(int n){
    if(n==0){
    return 1;
    }
  return n*fun(n-1);
}
```

课上老师一般告诉我们递归就是函数自己调用自己。但这听起来很玄学。事实上，在函数执行过程中如果调用了其他函数，那么当前函数的执行状态（执行到了第几行，有几个变量，各个变量的值是什么等等）会被保存起来压进栈（先进后出的存储结构，一般称为函数调用栈）中，转而执行子过程（调用的其他函数，当然也可以是当前函数）。若子过程中又调用了函数，那么调用前子过程的执行状态也会被保存起来压进栈中，转而执行子过程的子过程……以此类推，直到有一个子过程没有调用函数、能顺序执行完毕时会从函数调用栈依次弹出栈顶被保存起来的未执行完的函数（恢复现场）继续执行，直到函数调用栈中的函数都执行完毕，整个递归过程结束。

例如，在`main`中执行`fun(3)`，其递归过程如下：

```c
int main(){
  int i = fun(3);
  printf("%d",i);
  return 0;
}
```



很多时候我们分析递归时都喜欢在心中模拟代码执行，去追溯、还原整个递归调用过程。但事实上没有必要这样做，因为每相邻的两个步骤执行的逻辑都是相同的，因此我们只需要分析第一步到第二步是如何执行的以及递归的终点在哪里就可以了。

> 一切的递归算法都可以转化为非递归，因为我们完全可以自己压栈。只是说递归的写法更加简洁。在实际工程中，递归的使用是极少的，因为递归创建子函数的开销很大并且存在安全问题（stack overflow）。

Master公式

包含递归的算法的时间复杂度有时很难通过算法表面分析出来， 比如 **归并排序**。这时Master公式就粉墨登场了，当某递归算法的时间复杂度符合 `T(n)=aT(n/b)+O(n^d)` 形式时可以直接求出该算法的直接复杂度：

- 当（以b为底a的对数）`log(b,a) > d` 时，时间复杂度为 `O(n^log(b,a))`
- 当 `log(b,a) = d` 时，时间复杂度为 `O(n^d * logn)`
- 当 `log(b,a) < d` 时，时间复杂度为 `O(n^d)`

> 其中，`n` 为样本规模，`n/b` 为子过程的样本规模（暗含子过程的样本规模必须相同，且相加之和等于总样本规模），`a` 为子过程的执行次数，`O(n^d)` 为除子过程之后的操作的时间复杂度。
>
> 以归并排序为例，函数本体先对左右两半部分进行归并排序，样本规模被分为了左右各 `n/2` 即 `b=2`，左右各归并排序了一次，子过程执行次数为 `2` 即 `a=2`，并入操作的时间复杂度为 `O(n+n)=O(n)` 即 `d=1`，因此 `T(n)=2T(n/2)+O(n)`，符合 `log(b,a)=d=1`，因此**归并排序的时间复杂度**为 `O(n^1*logn)=O(nlogn)`

未完待续。。。