# Zigzag

## 问题描述

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

P A H N

A P L S I I G

Y I R

And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of rows:

string convert(string text, int nRows);

convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

字符串 abcdefghijklmnopqrstuvwxyz

如果将其分成n行，返回一个z字形排列

如n=3，则返回

a e i…

b d f h

c g

## 解决方案

原始的序列z字形排列算法：

分配一个足够容纳原序列的n行二维数组

通过一个行列下标和方向来确定下一个增长方向，在遇到超出范围时更改方向和起始下标

直到遇到最后一个元素

当前问题也可以使用这种类似的算法：但是注意调整 (x,y)的下一个值，它总是填补最后一个空位，因此可以使用一个数组来记录每行的最后一个位置；当然，由于这些位置总是固定的，所以其实可以在x,y的变换过程中实现而不必使用一个多余的数组。

输入（str,n）

lastpos[n]={0}

i=0

(x,y)=(0,0)

lastpos[0]=1;

dir=1 (1:向下，-1：向上)

for(;i<str.size;i++)

x+=dir;

if x,y in normal range

fill (x,lastpos[x]++) with str(i)

continue

dir = -dir;

if( x exceeds downside) x-=2;

else if(x exceeds upside) x+=2;

## 到底需要多少列

这是不固定的，因为多数情况下中间的一列需要的比较多，但是无法准确告知到底需要多少。再者，这道题目用动态数组更容易实现。

# 37.Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.

Empty cells are indicated by the character '.'.

You may assume that there will be only one unique solution.



A sudoku puzzle...



...and its solution numbers marked in red.

## 解决方法

整个数组是10\*10，大小是固定的。

对于每行，记录所有可用的数；对于每列，也记录所有可用的数

rowUseable[10][10]

colUseable[10][10]

则i,j的可用的数由 rowUseable[i][10]和colUseable[j][10]共同决定。

在每次填入一个数时，就将i ,j所在的Useable改变，它和栈是同步的。

一个解状态是指rowUseable[i] 或者colUseable[j]仅有一个可用时，并且这个唯一可用的数在两者中都可用。

在解状态下将(i,j)填上。

无解状态是指其他除了可行解和解状态之外的状态。

遍历每个(i,j)，求其row和col的交集 rowcol[i][j][10],

## 使用Exact Cover的算法

在已有的Exact Cover算法的基础上，对数独问题进行求解；最关键的的问题是如何将数独问题转化成精确覆盖问题。

将数独问题的4个约束条件拆分成【是|否】的形式，

有4个数组A,B,C,D 其中A负责检查每一个点是否填上了数，B负责检查每一行的每个数是否被填上，C检查列，D检查宫格。

然后建立待遍历的有效数据集。

## 测试

测试用例

**int** A[9][9]={

{5, 3, 0, 0, 7, 0, 0,0,0 },

{6, 0, 0, 1, 9, 5, 0,0,0 },

{0, 9, 8, 0, 0, 0, 0,6,0 },

{8, 0, 0, 0, 6, 0, 0,0,3 } ,

{4, 0, 0, 8, 0, 3, 0,0,1 },

{7, 0, 0, 0, 2, 0, 0,0,6 } ,

{0, 6, 0, 0, 0, 0, 2,8,0 } ,

{0, 0, 0, 4, 1, 9, 0,0, 5} ,

{0, 0, 0, 0, 8, 0, 0,7,9 } ,

};

输出结果：

<A,3>=4 <A,4>=6 <A,6>=8 <A,7>=9 <A,8>=1 <A,9>=2

<B,2>=7

<B,3>=2

<B,7>=3

<B,8>=4

<B,9>=8

<C,1>=1

<C,4>=3

<C,5>=4

<C,6>=2

<C,7>=5

<C,9>=7

<D,2>=5

<D,3>=9

<D,4>=7

<D,6>=1

<D,7>=4

<D,8>=2

<E,2>=2

<E,3>=6

<E,5>=5

<E,7>=7

<E,8>=9

<F,2>=1

<F,3>=3

<F,4>=9

<F,6>=4

<F,7>=8

<F,8>=5

<G,1>=9

<G,3>=1

<G,4>=5

<G,5>=3

<G,6>=7

<G,9>=4

<H,1>=2

<H,2>=8

<H,3>=7

<H,7>=6

<H,8>=3

<I,1>=3

<I,2>=4

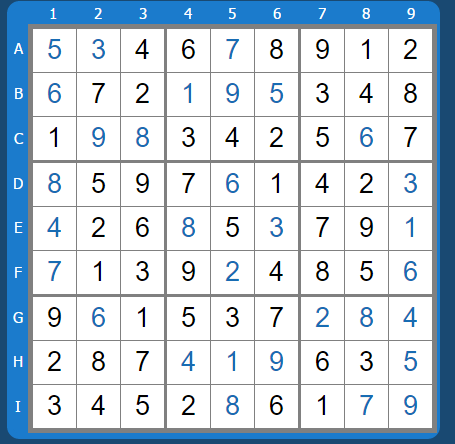
<I,3>=5

<I,4>=2

<I,6>=6

<I,7>=1

与<https://www.sudoku-solutions.com/>上的结果对照



# 41. First Missing Positive

## Problem

Given an unsorted integer array, find the first missing positive integer.

For example,  
Given [1,2,0] return 3,  
and [3,4,-1,1] return 2.

Your algorithm should run in *O*(*n*) time and uses constant space.

//========

class Solution {

public:

int firstMissingPositive(vector<int>& nums) {

}

};

## 方案

连续数字集 0 1 2 –

给定数组A = [ 0,9,7,2,4,6]

获取所有的连续子数组以及相应的missing positive

则最终的missing positive就是第一个不连续的missing positive.

将数组分成两部分，A1,A2;

令 f(A)表示A的missing positive（从第一个非0元素开始）

f(A)

f(A,i,b) 表示数组A中前i个数中大于等于b的第一个丢失整数

f(A,i+1,b1)=

如果 A[i+1] = f(A,i,b1)，则为 f(A,i,b1+1)

如果A[i+1] > f(A,i,b1)，则为f(A,i,b1)，且可以更新 f(A,i+1,b1+1) = A[i+1]

如果A[i+1] < f(A,i,b1) 则为f(A,i,b1), 且可以更新f(A,i,A[i+1])

状态用 [n][|MAX|]表示，则其大小是动态的。不符合条件。

f(A,i) 表示 数组中0~i的第一个丢失整数

C(A,i) 表示数组中 0~i的大于

f(A,i+1) =

如果 A[i+1]=f(A,i) , 则 f(A,i+1)

## 使用快速选择

实际上这道题目可以使用快速选择来完成。

因为题目可转换为：

从1,2,…n搜索数组A，返回第一个没有找到的数

于是使用快速选择就是

第k次将A划分为 <=k 和 >k两部分，然后如果第一部分有k，则进行基于第二部分的k+1查找，直到查找的结果中第一部分没有找到元素

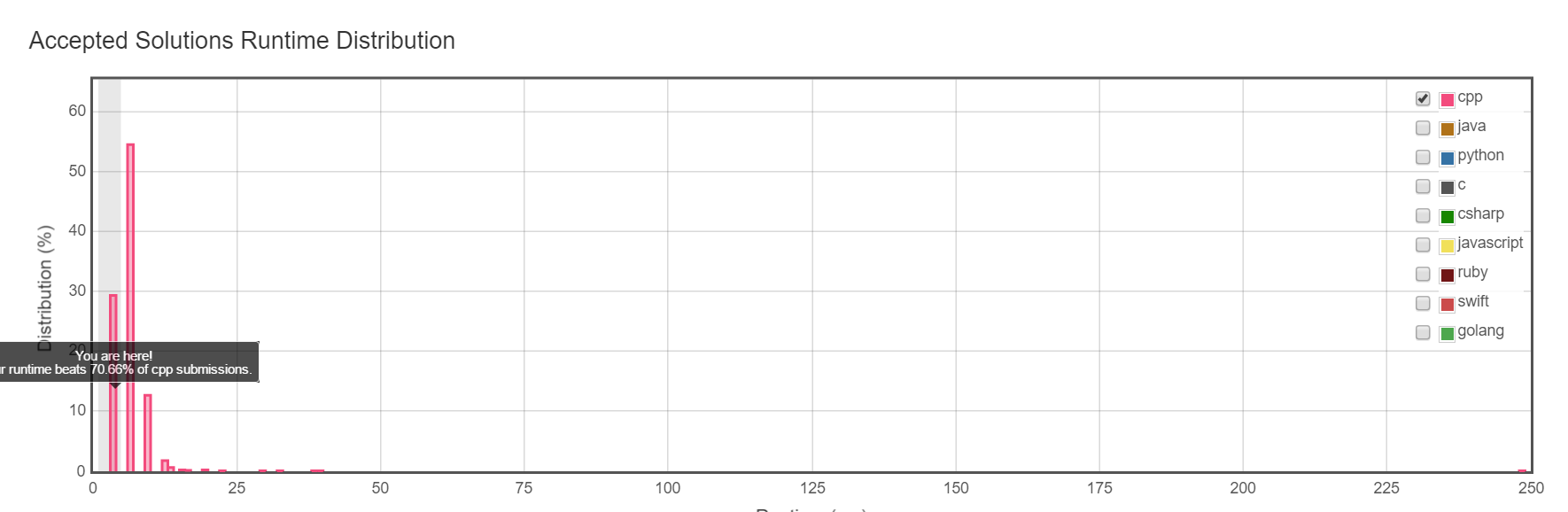
## 快速选择解决的问题

快速选择通过分区将原来的数组分为两个确定的部分，从而避免了后面的信息会被重复使用造成性能丢失。

## 性能分析

因为快速k选择的复杂度是O(n)，所以这个选择的复杂度也是O(n)

## 提交结果



156个测试结果，耗时：3ms

# 42 Trapping Rain Water

Hard

## 问题

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example,   
Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.



The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

class Solution {

public:

int trap(vector<int>& height) {

}

};

## 解决方案

解决此题的关键是找到两个可以容纳水的板子；为了确定两块这样的板子，可以使用如下算法：

1. 设置一个栈，当前总量sum=0，设置数列下标i=0
2. 准备初始化一个栈压入和弹出的过程，subsum=0（记录弹栈过程中的和）
3. 开始遍历数列，压入数列中前面非严格递减的数的下标，i递增，直到遇到i递减
4. 弹出栈中的下标并将值加入subsum中，直到遇到第一个较大的数，然后压入当前的下标，下标增加
5. 重复4过程直到不能进行为止
6. 将 sum 加上 subsum的值
7. 如果4因为栈空而停止，转入3
8. 如果4因为输入不足而停止，结束

上述算法错误。

## 解决方案2

因为积水的高度主要取决于两端的高度，所以从左右两端开始处理数组能获得没有后续影响的确定性。

但如果仅仅从一端开始处理，比如对于数组 7 6 3 2 1 4 4 5 1 6, 如果仅仅从7开始处理，然后每遇到一个小于7的节点，都将差值加入其中，但是这么做的原因是数组中后续必须存在大于等于7的元素，否则累计可能无效；同理，如果从右侧6开始，至少在3之前，无法确定其是否有效；

但我们知道，如果从两个中的一个开始，若一个无效，则另一个必有效；因为如果在一端是最大的，则从这一端开始是无效的，但从另外一端开始一定能遇到终结节点，所以是有效的。

所以第一个算法可以理解为尝试性算法：从左端开始，开始累加；如果左端无效，则从右端开始累加

然而更近一步，为了保证每次选择的开始都有可能的结尾，从两端中最大者开始。

# 546. Remove Boxes

Given several boxes with different colors represented by different positive numbers.   
You may experience several rounds to remove boxes until there is no box left. Each time you can choose some continuous boxes with the same color (composed of k boxes, k >= 1), remove them and get k\*k points.  
Find the maximum points you can get.

**Example 1:**  
Input:

[1, 3, 2, 2, 2, 3, 4, 3, 1]

Output:

23

Explanation:

[1, 3, 2, 2, 2, 3, 4, 3, 1]

----> [1, 3, 3, 4, 3, 1] (3\*3=9 points)

----> [1, 3, 3, 3, 1] (1\*1=1 points)

----> [1, 1] (3\*3=9 points)

----> [] (2\*2=4 points)

**Note:** The number of boxes n would not exceed 100.

class Solution {

public:

int removeBoxes(vector<int>& boxes) {

}

};

类型：DP

## 解决方案

使用Knuth的DLX的简化版：一个双向链表，类似的过程。

直观上的一点是，如果存在k个连续a，则选择k个比选择比k个少得到更多的点

## 动态规划方案

使用一个解的记录方案 C[i][j][k] 表示 数组中i~j的以A[i+k]结尾的解；

解的形式是{值，最后一个元素被消去时的个数}

f(A,i,j,k) A中i~j以k作为最后一个数消除时的最大值；k是基于i的偏移值；假设形成的最优解为C[i][j][k]

则C[i][j+1][k]的值为

求C[i][j][0~ (j-i)]的值

C[i][j+1][0]值为1+C[i][j][0]

C[i][j+1][1]值为1+C[i][j][1]

…

C[i][j+1][(j+1-i)]值为

[i,j+1,j+1-i]可能产生一个合并；这个合并可以是[i,j]内任意有效的一个元素；即可能要求与在 [i,j]内的每一个以A[j+1]结尾的值进行合并（使用一个D[max{Ai}][i]来记录n在位置i是否出现出现，但是这样同样会需要向前的顺序搜索）。

最大值并不由最大值本身唯一决定，还由以往的数目参与构成。

空间耗费：O(|A|^2 \* max{Ai})

或者如果采用map结构，空间会降低，但是效率也会减损。

问题的性质： 如果一个点在第i次进行消除，则i这个点之前和之后的范围独立

【下面的算法是原来的想法，是错误的】

*if A[j+1]*

*if A[j+1]=A[j]*

*C[i][j+1].lastnum = C[i][j].lastnum + 1*

*C[i][j+1].val = C[i][j].val +2\*C[i][j].lastnum + 1*

*else if A[j+1]在i~j中从未出现过*

*C[i][j+1].lastnum=1*

*C[i][j+1].val = C[i][j].val + 1*

*else if A[j+1]在i~j中出现过*

*逆向查找A[j+1]的第一个出现的位置j’,则C[j+1]=max{*

*不将A[j+1]与A[j’]合并构成的最优解{ C[i][j].val + 1 ,1}，*

*将A[j+1] 与 A[j’] 合并构成的最优解{C[i][j’].val + 2\*C[i][j’].val+1 + C[j’+1][j].val, 1*

*}*

*初始时 C[0][0]= {1 ,1} C[0][1]…*

## 测试用例

[1, 3, 2, 2, 2, 3, 4, 3, 1]

----> [1, 3, 3, 4, 3, 1] (3\*3=9 points)

----> [1, 3, 3, 3, 1] (1\*1=1 points)

----> [1, 1] (3\*3=9 points)

----> [] (2\*2=4 points)

[0，8]

将0,7合并的值 另外的值[0,7]

[0,7]

将5,7合并的值 另外的值[0,6]

[0,6]

[0,5]的值 + 1

[0,5]

将1,5合并的值[0,1],[2,5] 另外的值[0,4] {12，1}

[2,5]

[0,4]

[0,3]的值update1 {11,3}

[0,3]

[0,2] update1 {6,2}

[0,2]

[0,1] update2 {3,1}

[0,1]

[0,0] update2 {2,1}

[0,0]

{1,1}

[1, 3, 2, 2, 2, 3, 4, 3, 1]

//合并顺序需要记录吗？不需要

0,5

1 222 33 = 14,1

0,6

2,4

{9,3}

1,6

1,5

1,1

{1,1}

1,5

1,1 + 2,4 = {13,1} 222 33 //最后一个合并的数

0,8

0,0+1,7

1,7

1,5 + 6,6 = 13 + 3 + 1, {17,1}

6,6

{1,1}

## 测试用例2

8, 6, 2, 6, 6, 3, 2, 3, 2, 2

0 1 2 3 4 5 6 7 8 9

0，9

0，8 eq

0,8

0,6+7,7 m

0,6

0,5, no-c

0,5

0,4

0,4

0,3 eq

# 523. Continuous Subarray Sum

## Problem (level : medium)

Given a list of **non-negative** numbers and a target **integer** k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to the multiple of **k**, that is, sums up to n\*k where n is also an **integer**.

**Example 1:**

**Input:** [23, 2, 4, 6, 7], k=6

**Output:** True

**Explanation:** Because [2, 4] is a continuous subarray of size 2 and sums up to 6.

**Example 2:**

**Input:** [23, 2, 6, 4, 7], k=6

**Output:** True

**Explanation:** Because [23, 2, 6, 4, 7] is an continuous subarray of size 5 and sums up to 42.

**Note:**

1. The length of the array won't exceed 10,000.
2. You may assume the sum of all the numbers is in the range of a signed 32-bit integer.

**class** Solution {

**public**:

**bool** **checkSubarraySum**(vector<**int**>& nums, **int** k) {

}

}

## Solution Candidate 1（Slower DP Solution）

for each problem, k is not changed for every subproblem.

let f(i,j) be mod of i~j’s summation of k, e.g. f(i,j)=sum(i to j) % k

1. j=i+1,then if f(i,j)=A[i] + A[j] % k
2. j>i+1, f(i,j)=(f(i,j-1) + A[j] % k)

another transition form using i is

1. i=j-1 …
2. i<j-1 f(i,j) = (A[i] + f(i+1,j))%k

the the problem becomes that if there exists such sub array in i,j that it contains f(i’,j’)=0.

let g(i,j) be such problem,

then

1. i==j g(i,j)=false
2. j=i+1 g(i,j) = f(i,j)==0
3. j>i+1 g(i,j) = g(i,j-1)==true or exists i<=k<=j-1, f(k,j)=0

g->g | f

g->f+

depends on f’s complexity

a simulation process:

[23, 2, 4, 6, 7],k=6

no necessary to calculate g, we calculate f,when we found i,j,return it.

This solution is very slow, only beats 4.88% cpp submissions,75/75 test cases passed but 115ms cost.

Its space complexity is O(n^2),time complexity is O(n^2) also.

## Very Faster DP Solution

实际上对于一个数组 A[0~n],

我们有递推式 sum(i,j) = sum(0,j) – sum(0,i)， 这里可以使用到之前推出的关于状态转移方程的“常数消除技术”，有sum(i,j)=sum\_0(j) – sum\_0(i) ，从而将空间复杂度和时间复杂度一举消到O(n).

对于本问题，如果sum(i,j)%k=0,即意味着 sum\_0(j)=sum\_0(i) + n\*k, 从而两者对于k同余，即

(sum\_0(j)≡sum\_0(i)) mod k

并且这是充要条件。

所以问题转化为计算sum\_0(i)%k，并且当发现已经存在这样一个数且其下标与当前下标之差大于1的时候，则问题结束，返回true。

这个算法总是能够发现最内层的子列。

# 516. Longest Palindromic Subsequence

## problem（level medium）

Given a string s, find the longest palindromic subsequence's length in s. You may assume that the maximum length of s is 1000.

**Example 1:**  
Input:

"bbbab"

Output:

4

One possible longest palindromic subsequence is "bbbb".

**Example 2:**  
Input:

"cbbd"

Output:

2

One possible longest palindromic subsequence is "bb".

public class Solution {

public int longestPalindromeSubseq(String s) {

}

}

## 解决方案

更多地考虑问题所涉及的类型和相关的性质。

回文串S是给定串A的子串

如果S的两端必然相等，对S的首部进行分类：A[0] A[1]…A[n-1]

回文串的内部子串仍然是回文串，并且最长回文串的内部子串一定也是最长子串，并且这个回文串一定来自于外部串两端之间的位置

设S两端的位置在A中分别为Start(S),End(S) (前者<=后者), 设T(S)是S去掉两端的子串（如果|S|=1,则T(S)=””）

设f(i,j)为在i,j之间寻找A的最长回文串，则

1. A[i]=A[j], -->f(i+1,j-1) 并 两端的字符
2. A[i]!=A[j]
   1. 包含A[i] ，令j’为从j向下搜索的遇到的第一个j’,且有A[j’]=A[i]
   2. 包含 A[j] ,令i’为i想上搜索遇到的第一个i’，且A[j]=A[i’]
   3. 两者都不包含

**复杂度分析**

初始时，i=-1,j=n,且A[-1]=A[n]=””

# 517. Super Washing Machines

## problem

You have **n** super washing machines on a line. Initially, each washing machine has some dresses or is empty.

For each **move**, you could choose **any m** (1 ≤ m ≤ n) washing machines, and pass **one dress** of each washing machine to one of its adjacent washing machines **at the same time**.

Given an integer array representing the number of dresses in each washing machine from left to right on the line, you should find the **minimum number of moves** to make all the washing machines have the same number of dresses. If it is not possible to do it, return -1.

**Example1**

**Input:** [1,0,5]

**Output:** 3

**Explanation:**

1st move: 1 0 <-- 5 => 1 1 4

2nd move: 1 <-- 1 <-- 4 => 2 1 3

3rd move: 2 1 <-- 3 => 2 2 2

**Example2**

**Input:** [0,3,0]

**Output:** 2

**Explanation:**

1st move: 0 <-- 3 0 => 1 2 0

2nd move: 1 2 --> 0 => 1 1 1

**Example3**

**Input:** [0,2,0]

**Output:** -1

**Explanation:**

It's impossible to make all the three washing machines have the same number of dresses.

**Note:**

1. The range of n is [1, 10000].
2. The range of dresses number in a super washing machine is [0, 1e5].

class Solution {

public:

int findMinMoves(vector<int>& machines) {

}

};

## Solution

一个实例：7 9 0 2 4 5 8 5 6 7 7

顺序扫描，求出和，然后取平均值，得到需要变化的值V；

总体来说，每一次可以将数组分成数个独立的区域，因为我们知道如果[0,i] 的平均值为V，

则[0,i]就构成一个独立子问题。

所以原问题可以用f(i,j)来刻画：数组A[i~j]的均值为V,求将A[i~j]移动成为均值V的最小步数。

如果i,j中间存在j’,使得 [i,j’]均值为V，则如果移入或者移出任何大于0的值，都需要再平衡这个数所以f(i,j) = min { f(i,j’), f(j’+1,j) }

如果这样的j’不存在，设法移动一次从而形成一个有效的j’。因为对于一个不平均的区域，如果它的值大，就需要移出；如果它的值小，就需要移入

或者

对i,j整体进行考虑； 每一步设置moved来表示这一步是否移动；对于i,j之内的每一个值k，依次计算d[k]=A[k]-V, 如果发现d[k]!=0, moved=true;

总的移入或者移出大小为d[k], 如果d[k]>0 && d[k] + A[k+1] – V >= 0,则

计算d[k]数组，向前扫描，如果d[k]>0，如果i’尚未分发现，记录发现的i’，将作为下一个f的起始参数, 则将d[k]加到d[k+1]上，处理d[k+1];

如果d[k]==0, 向前继续

如果d[k]<0,则将右侧的元素向左移动，直到令d[k]=0，然后下一个i’

## Solution 2（Algorithm Verified）

设f(i,j,a)表示在数组A[i~j]中为了使每个数达到均值以及从左边界移出一个数a或者移入-a所需的最小步数；

则对于一个有可能达到均值的数组，有转移方程：（考虑方向）

1. a=0,则f(i,j,a)=f(i+1,j,a+V-A[i])
2. a>0, (至少需要a步才能将前面的数平衡，且需要保证每个点在一次移动中只能向前或者向后移动)
   1. 如果A[i]<a,需要先移动a-A[i]个数到A[i]这个边界上，然后才能完全将前面的平衡,则先将A[i]个平衡掉，然后等待后面的数平衡。 f(i,j,a) = max { f(i+1,j,a+V -A[i]), a};(即在移动A[i]个的时候，同时寻求将所需的数移入A[i+1])
   2. 如果A[i]>=a 平衡前面的数需要a步 显然A[i]依然需要移出a个到前面，但是如果移出之后如果有剩余，需要移出到后面，次数就是两次之和； 如果移出之后仍然不够，需要移入，但这个移入可以和前面的移入同时进行；因此分两种情况：
      1. a+V-A[i] > 0（移出之后不够）时 f(i,j,a) = max{ a, f(i+1,j,a+V-A[i]) }
      2. a+V-A[i]<=0 （移出之后剩余） f(i,j,a) = max{A[i]-V, f(i+1,j,a+V-A[i])}

A[i]-V = a + (A[i]-V-a) 总的所要移动A[i]的次数

1. a<0, f(i,j,a) = max{ f(i+1,j,a+V-A[i]),|a|}

模拟：

0 3 0 V=1

f(0,2,0) = f(1,2,-1)=1 + max { |-1| , f(2,2,0) }

1 0 5 V=2

f(0,2,0) = f(1,2,1)=max{f(2,2,3),0} + 1

f(2,2,3) =3+max{0,f(3,2,0)}

1 2 3 4 5 6 7 V=4

f(0,0)=f(1,3)=max{2,f(2,5)}=5， 剩余a-A[i]个尚未移入

f(2,5)=max{3,f(3,6)}=5

f(3,6)=max{4,f(4,6)}=5

f(4,6)=max{5,f(5,5)}=5

f(5,5)=[diff>0]max{5,f(6,3)}=5

f(6,3)=[diff>=0]max{3,f(7,0)}=3

7 – 3—>6

6 - 5-->1~5

2 2 3 4 5 6 6

3 2 3 4 5 6 5

4 2 3 4 5 6 4 🡪 7移动3个到左边界

4 3 3 4 5 5 4

4 4 3 4 5 4 4

4 4 4 4 4 4 4

**[0,0,10,0,0,0,10,0,0,0] V=2**

f(0,0)=max(2,f(1,2))

f(1,2)=max(2,f(2,4))

f(2,-4)=

0 0 6 V=2

0 1 5

1 1 4

2 1 3

2 2 2

f(0,0)=max(2,f(1,2))

f(1,2)=max(

7 9 0 2 4 5 8 5 6 7 7 V=7

0 1 2 3 4 5 6 7 8 9 10

f(0,10,0)=f(1,10,0)=f(2,10,-2)=f(3,10,0)=f(4,10,7)=f(5,10,4)+1

## 求精确平均数的算法

数组A的平均数

设f(i,V,diff)为求A中i~n-1的平均数时若以V为平均数，积累的总和差为diff时，求出的平均数；

则

f(i,V,diff) = f(i+1,V,diff + A[i] - V)

每当发现diff % （i +1）== 0，时，将V+diff/i,diff置为0

# 44. Wildcard Matching & 10. Regular Expression Matching

Implement wildcard pattern matching with support for '?' and '\*'.

**Wildcard**

'?' Matches any single character.(在RE中可以用.来表示)

'\*' Matches any sequence of characters (including the empty sequence).(在RE中可以用.\*来表达)|

**RE**

'.' Matches any single character.

'\*' Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).

The function prototype should be:

bool isMatch(const char \*s, const char \*p)

**Wildcard**

Some examples:

isMatch("aa","a") → false

isMatch("aa","aa") → true

isMatch("aaa","aa") → false

isMatch("aa", "\*") → true

isMatch("aa", "a\*") → true

isMatch("ab", "?\*") → true

isMatch("aab", "c\*a\*b") → false

**RE**

Some examples:

isMatch("aa","a") → false

isMatch("aa","aa") → true

isMatch("aaa","aa") → false

isMatch("aa", "a\*") → true

isMatch("aa", ".\*") → true

isMatch("ab", ".\*") → true

isMatch("aab", "c\*a\*b") → true

class Solution {

public:

bool isMatch(string s, string p) {

}

};

## 解决方案

尝试解决一个通用的问题：识别正则表达式

1. 从正则表达式构造NFA
2. 从NFA构造DFA或者直接识别NFA

## 从正则表达式构造NFA

扫描表达式串，确定状态。正则表达式的描述可以由某种文法确定。

扫描正则表达式的自动机

prev代表前一个字符

0 开始状态

1 读入一个任意字符

2 出错状态

如果遇到\*,就构造一个 a\*状态来接收

构造的NFA由下面的部分组成：

1. 状态序列号
2. 状态的输入转换表（ 一个输入字符到到其他状态的转换，是一个list，为了提高查找效率，可以定义为set）

## 快速解决此问题

由于此问题简单，所以可以不用构造NFA。

识别的基本办法是回溯。

使用深度优先的方法进行搜索。正则表达式中 .\*,a\*形成的子树具有 (., 数字) 或者 （a，数字n）的形式，并且next子树就是 (a,n+1)。

当遇到可能存在多棵子树时，将其当前状态加入栈中:(a,0),并且同时记录a所在的正则表达式的下标。如果一次匹配停留在接受状态，则返回true，否则，如果回退栈为空，返回false；否则，回退到栈顶部开始的第一个与下一个输入匹配的状态，使得两个下标都指向上一个位置，然后从这个位置开始匹配。

* 算法思路（下面的算法不清晰，没有使用递归）

parse input pattern as a stream of match states

curChIndex=0

curMatchIndex=0

while(true)

{

if(curChIndex exceeds range)

if curMatchIndex exceeds range

return matched

else if laterMatch can be empty

return matched

else if not traceback()

return not matched

else if curMatchIndex exceeds range and not traceback()

return not matched

if curMatch.isSingle()

if curMatch.matched(ch)

both curMatch,ch move to next

else if not traceback()

return not matched

else curMatch.isMulti()

if curMatch.matched(ch)

incre ch, keep curMatch

else

push {curMatchIndex+1,curChIndex} to saved

move curMatch to next

}

traceback{

if not saved.empty() //

pop top to curMatchIndex and curChIndex,

return tracebacked successfully

else

return tracebacked failed

}

* 转移方程（verified）

设f(i,j) 表示从string的i出开始，pattern的j处开始匹配，则有

1. string[i] == pattern[j] 或者 pattern[j]==’.’
   1. 如果pattern[j]数量任意，则为 f(i,j) =f(i+1,j) f(i+1,j+1) f(i,j+1)
   2. 如果如果pattern[j]数量为1，则f(i,j)=f(i+1,j+1)
2. string[i]!=pattern[j]
   1. pattern[j]数量任意 f(i,j) = f(i,j+1)
   2. pattern[j]数量不任意, false
3. 如果 i>=nstring,如果pattern[j]数量任意，则为f(i,j)=f(i,j+1) 否则 f(i,j)=false
4. 如果i>=ntring,j>=nstate,返回true
5. 如果i<nstring && j>=nstate,false

# （HOJ 1005）Fast Food

The fastfood chain McBurger owns several restaurants along a highway. Recently, they have decided to build several depots along the highway, each one located at a restaurant and supplying several of the restaurants with the needed ingredients. Naturally, these depots should be placed so that the average distance between a restaurant and its assigned depot is minimized. You are to write a program that computes the optimal positions and assignments of the depots.

To make this more precise, the management of McBurger has issued the following specification: You will be given the positions of n restaurants along the highway as n integers d1 < d2 < ... < dn (these are the distances measured from the company's headquarter, which happens to be at the same highway). Furthermore, a number k (k <= n) will be given, the number of depots to be built.

The k depots will be built at the locations of k different restaurants. Each restaurant will be assigned to the closest depot, from which it will then receive its supplies. To minimize shipping costs, the total distance sum, defined as

http://acm.hit.edu.cn/hoj/static/img/pic/1005_a.gif

must be as small as possible.

Write a program that computes the positions of the k depots, such that the total distance sum is minimized.

**Input**

The input file contains several descriptions of fastfood chains. Each description starts with a line containing the two integers n and k. n and k will satisfy 1 <= n <= 200, 1 <= k <= 30, k <= n. Following this will n lines containing one integer each, giving the positions di of the restaurants, ordered increasingly.

The input file will end with a case starting with n = k = 0. This case should not be processed.

**Output**

For each chain, first output the number of the chain. Then output a line containing the total distance sum.

Output a blank line after each test case.

**Sample Input**

6 3

5

6

12

19

20

27

0 0

**Sample Output**

Chain 1

Total distance sum = 8

解释：

给定n递增的数，将其分成k个部分，每个部分选择一个中心点，使得每个部分的中心点距离和最后加起来最小。

5 7 13 14 17 20 23 27 30

k=3

## 解决方案

5,6,12,19,20,27 k=3

brute-force search

what computations does it need?

如果选择了一个点，需要确定这个点到周围点的距离，通过D[i][j]可以快速将这个距离算出来。

对于k，如果选择a1,a2,..,ak几个点，则首先需要确定哪些点属于a1,哪些点属于a2.

对于an am (m=n+1)，只需要找到这样一个分界点b, D[b][an] >= D[b][am] 即可。这样的分界点是很容易确定的。

如果用C(n,k)是候选集的大小，为了比较a1,a2,…,ak 与 b1,b2,…,bk的大小，先计算sum(a1,…,ak) 再计算 sum(b1,b2,…,bk) 作差。

有公式：（n = length）

f(i,j,k) = min[z’ from i to j-k+1,z from i to z’] { f(z’+1,j,k-1) + sum(z with i to z’) } (k>1)

f(i,j,1) =min sum[z from i to j](z with i to j)

if j is omitted

f(i,k) = min[z’ from i to n-k,z from i to z’] { f(z’+1,k-1) + sum(z with i to z’) } (k>1)

## 解决方案2

sum(z,i,z’) 还可以由下面的公式计算

sum(z,i,z’) =

if not sum(z,i,z’-1) sum again

else sum(z,i,z’-1) + D[z][z’]

重复计算sum(z,i,z’)的值开销很大

转化成计算f(j,k) = max{ f(z’+1,k-1) + f(j,z’,1)}

f(a,j,k) = max { f(a,z’,1) + f(z’+1,j,k-1) }

f(a,z’,1) 由 a,z’的中点决定,要么是中点左边，要么是中点右边

## 解决方案3

f(a,j,k) a~j with k requirements

f(a,b,1) and f(b+1,j,k-1)

or

f(a,n-1,k)

f(a,j,1)的优化

3 4 7 8 9

[ . . . . . .]

G[i] 与 G[j]的差异

G[i] = sum{ |A[x] – A[i]| }

G[j] = sum {|A[x]-A[j]|}

A[i] - A[0] + A[i]-A[1]…+A[i]-A[i]+A[i+1]-A[i]…

如果前面有n个，后面有m个

则总和就是 sumA(j+1,j+m) – sumA(j-n,j-1) + (n-m)\*A[j] 是关于 m,n,j的函数。

由于n+m+1=N,m=N-n-1

而如果设开始 为a，则 j = a+n

所以有

X(n)=sum A(a+n+1,N-n-1) – sumA(a,n) + (2n+1-N)A[a+n] 实际上是关于n的函数。只要表达式中的整数部分越小，而负数部分越大则整个表达式就越小

我们可以求出所有的 sumA(j,n)，然后计算X(n)只需要O(n)时间进行比较和选择，而不是O(n)时间选择，O(n)时间求和。

然而由于 g(a,j)是可以提前计算出来的，其复杂度越低越好O(n^2)；所以

## DP的理论：常数特化和表达式依赖

在上面的状态转移方程中，存在

f(a,j,k) = min { f(a,b,1) + f(b+1,j,k-1) }

注意到 f(a,j,k)3个参数是变量，而右侧的f(a,b,1)中k=1是常数，因此对于整个问题而言，可以对f(a,b,1)进行特化，从而将空间需求从O(n^3)降至O(n^2);特化后可以令 f(a,j,1)=g(a,j)

**表达式依赖**即从所有的状态转移方程中，寻找出最后真正需要进行计算的表达式；我们可以设想一个带有递归或者子问题形式的状态转移方程相当于一个带有左递归或右递归的文法，譬如

f(a,k) = min {g(a,b) + f(b+1,k-1) } ----> f->fg | gg

从而可以到到 f=g+

所以我们知道算法的效率最后实际上取决于g的效率。

# 354. Russian Doll Envelopes

## problem

You have a number of envelopes with widths and heights given as a pair of integers (w, h). One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

What is the maximum number of envelopes can you Russian doll? (put one inside other)

**Example:**  
Given envelopes = [[5,4],[6,4],[6,7],[2,3]], the maximum number of envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7]).

class Solution {

public:

int maxEnvelopes(vector<pair<int, int>>& envelopes) {

}

};

## Solution

按照主键-次键排序数组之后，查找最长递增子序列长度。

## 在pair上查找最长递增子序列

f(i,j)-> i~n-1以j结尾的最长长度

1. f(i+1,j) = f(i,j)
2. f(i+1，i+1)=max[A[j’] is in both dimension less than A[i+1]] {f(i,j’) + 1}

## 基于长度查找

设f(i)为序列上长度为i的子序列, last[i]为其最后一个元素下标

则 f(i+1)= f(i) append with minimum A[j] , j>last[i]

or

and last[i+1]=j

## 二分查找

或者返回元素的一个位置，或者返回元素应当插入的位置

f(i,j) let mid=(i+j)/2

while(true) mid==j <==> left==right

1. mid==j || a[mid]==e return mid
2. a[mid]<e return f(mid+1,j)
3. a[mid]>e return f(i,mid)

**while**(left < right)

{

**int** mid=(left+right)/2;

**if**(e < subarr[mid])right=mid;

**else** **if**( e > subarr[mid]) left=mid+1;

**else** { right=mid;**break**;}

}

## 最长递增子序列问题

原来的想法是保留每一个长度中以最小元素结尾的数组，即在遇到小元素的更新相应的数组；

后来发现这种更新每个数组只会用到最后一个元素而且前面的元素更新不影响后面的元素，所以将所有的元素合并在一个数组中进行更新，最后形成的就是数组的长度就是最长的递增子序列，并且可以看到每一个长度的递增子序列的最小末尾数。

# 368. Largest Divisible Subset

## problem

Given a set of **distinct** positive integers, find the largest subset such that every pair (Si, Sj) of elements in this subset satisfies: Si % Sj = 0 or Sj% Si = 0.

If there are multiple solutions, return any subset is fine.

**Example 1:**

nums: [1,2,3]

Result: [1,2] (of course, [1,3] will also be ok)

**Example 2:**

nums: [1,2,4,8]

Result: [1,2,4,8]

class Solution {

public:

vector<int> largestDivisibleSubset(vector<int>& nums) {

}

};

## solution

对于数组A,如果Ai =n\*A0，Aj =k\*A0。 如果将数组排序，则每一个后面的数都能整除前面的数。并且只需要满足这样的关系就可以了。并且如果一个数是另一个数的n倍(n>=1)，则将这后者加入到前者的整除集合中，一定能够形成更大的整除集合。

所以形成的基本解法是：将数组排序，最后一个数构成的可除集合就是它自身；向下扩展，每一个数如果能被现有的可除集合中的第一个数整除，将其加入到那些集合的首部；这样做的复杂度是O(nlog(n) + n^2)

转移方程：

f(i,j,k)

*因为同一个基数可能出现不同的mod集，这些mod之间至少有两个元素不相关。这样就有可能形成多个候选分支，就需要进行剪枝。*

*考虑转移方程f(i,j,k)表示求i~j中以A[k]为基数的最大可除集；如果一个更大的数在A[k]的集合中，我们将不关注它。*

*f(i+1,j,k) 如果A[i+1] % A[k]=0，则将其并入解集；f(i+1,j,k)=f(i,j,k) 并 A[i+1]*

*若A[k] % A[i+1] =0,则f(i+1,j,i+1) = f(i,j,k) 并 A[i+1]*

*如果两者不存在余数关系 则 f(i+1,j,k)=f(i,j,k), f(i+1,j,i+1)={}寻找所有比A[i+1]小的数且能被其整除的集合，A[i+1]并入；如果没有找到，则f(i+1,j,i+1)={A[i+1]}.*

*我们使用 subset[A[k]]来记录以某个数为基数的可除集，如果一个数已经出现在某个集合中了，它就不会再作为subset的键。*

*或者我们可以使用排除法先排除那些不可能作为基准的元素（可能需要一个矩阵 mod[i][j]）,每一行中除了对角线上的元素，如果所有的值都不为0，说明该行的元素是一个基准元素。*

*然后对i列j行的元素进行统计，如果mod[i][j]=0,则i的下一个元素可能变成j，将其记录在案。*

*实际上可以统计j的mod集，依次转换成下一个元素的mod集。*

*复杂度O(n^2)，每个数与其他数进行模运算的次数精确是n*

*而在上一个算法中它等于当前基元素个数相同，多数情况下小于n。*

## Solution 2 using searching tree

*我们发现上面那个算法其实发生了不必要的重复比较，我们可以用类似于二叉查找树的思想来建立一棵树。*

*节点结构*

*Node{*

*next,previous,father,son*

*data*

*insert(int num)*

*{*

*}*

*insert(Node base,Node inode)*

*{*

*if(base.data==inode.data)*

*base.insertFather(inode);*

*else if(base.data%inode.data==0)…*

*}*

*}*

*统计阶段：*

*f(node)=1 if node->son=NULL else 1+max{f(node->son)}*

* 我发现这个想法是浅显易懂的
* 但是错误是难以发现的
  + 1.son链表不一定有序
  + 2.插入一个较小的值时，还需要考虑当另外一个数也被该值替换时，链表中产生重复
* 为什么没有提前发现这些错误
  + 在编写程序之前虽然我已经知道了这些规则，但是在实现时却并没有明确地将它们表达出来
* 怎么改进
  + 列出限制条件并保证更新不会破坏条件
  + 结果就是正确的

## 问题:关于思考方向

有没有一种方法来统一评价某个转移函数对于一个问题的解是最优的？

譬如，对于一个已存在的O(n^2)的解，有没有可不可能找到小于这个复杂度的其他解？

其实这个问题主要与问题本身的结构有关，我们将问题分解为一系列必须子问题和可选子问题，则复杂度就是必须子问题复杂度之和与最小可选子问题复杂度之和。

对于此问题，每个数至少要与另外的一个数作模运算，复杂度是O(n)；

# 521. Longest Uncommon Subsequence I

## problem

Given a group of two strings, you need to find the longest uncommon subsequence of this group of two strings. The longest uncommon subsequence is defined as the longest subsequence of one of these strings and this subsequence should not be **any** subsequence of the other strings.

A **subsequence** is a sequence that can be derived from one sequence by deleting some characters without changing the order of the remaining elements. Trivially, any string is a subsequence of itself and an empty string is a subsequence of any string.

The input will be two strings, and the output needs to be the length of the longest uncommon subsequence. If the longest uncommon subsequence doesn't exist, return -1.

**Example 1:**

**Input:** "aba", "cdc"

**Output:** 3

**Explanation:** The longest uncommon subsequence is "aba" (or "cdc"),   
because "aba" is a subsequence of "aba",   
but not a subsequence of any other strings in the group of two strings.

**Note:**

1. Both strings' lengths will not exceed 100.
2. Only letters from a ~ z will appear in input strings.

class Solution {

public:

int findLUSlength(string a, string b) {

}

};

## solution

后来的解法证明，这道题目的意思是如果能够通过删去得到；显然长度长的一定是最长子串；而长度相同的又不可能删除某一个，所以如果两者不同肯定就是其中一者；相同就是-1.

*brute force：*

*求出a,b的所有子串，选择两个集合的非公共部分中长度不大于两者的最长子串*

*DP：*

*LUS为空意味着两个子串集合非公共部分小于两者长度的子串为空*

*f(i,j)的子问题有 f(i+1,j) f(i,j+1)*

*f(i,j)意味着a[i~n-1] 与b[j~m-1]的LUS问题，*

*f(i,j+1)不存在这样的子串，说明较小的串是较大的串的完全子串；两者的字符集相同*

*如果ai是bj+1的子串，则ai肯定也是bj的子串；*

*如果bj+1是ai的子串，则*

*f(i+1,j)*

# 547. Friend Circles

There are **N** students in a class. Some of them are friends, while some are not. Their friendship is transitive in nature. For example, if A is a **direct** friend of B, and B is a **direct** friend of C, then A is an **indirect** friend of C. And we defined a friend **circle** is a group of students who are direct or indirect friends.

Given a **N\*N** matrix **M** representing the friend relationship between students in the class. If M[i][j] = 1, then the ith and jth students are **direct** friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.

**Example 1:**

**Input:**

[[1,1,0],

[1,1,0],

[0,0,1]]

**Output:** 2

**Explanation:**The 0th and 1st students are direct friends, so they are in a friend circle.   
The 2nd student himself is in a friend circle. So return 2.

**Example 2:**

**Input:**

[[1,1,0],

[1,1,1],

[0,1,1]]

**Output:** 1

**Explanation:**The 0th and 1st students are direct friends, the 1st and 2nd students are direct friends,   
so the 0th and 2nd students are indirect friends. All of them are in the same friend circle, so return 1.

**Note:**

1. N is in range [1,200].
2. M[i][i] = 1 for all students.
3. If M[i][j] = 1, then M[j][i] = 1.

class Solution {

public:

int findCircleNum(vector<vector<int>>& M) {

}

};

## Solution

无向图 M

实际上

i的direct集 OK

i的indirect集 j’s direct for all j in direct[i]， j’s indirect is not considered

实际上集合内的元素具有传递性

考虑f(i)与f(i+1)

如i+1可能属于f(i)的某一个集合，也有可能不属于任何一个集合

如果 如果M[i+1][j]=1,则将i+1加入 i+1所在的circle集合，因为按照定义应当如此；除此之外，i+1还是direct[i+1]的indirect，所以也将它加入到那些集合之中

indirect的性质：

如果i是j的indirect，则 存在 j‘!=i, j’!=j, M[j’,i]=1,M[j,j’]=1; 如果i是j的indirect，则j也是i的indirect

如何寻找indirect?

M[k,i]=1 且 M[k,j]=1 且 M[i,j]=1

circle的性质：

任意i,j

1. M[i,j]=1
2. M[i,j]!=1 i,j是indirect

最大的circle性质：

1. 至少包含一对 direct
2. 如果一对direct被加入到集合中，则通过加入它们之间共同是direct或者indirect
3. 如果一对indirect在circle中，则通过加入它们的共同的direct可以获得更大的集合

~~图算法中的环问题；使用深度优先搜索能够发现这些环。~~并不一定是环

1---2---3这样的结构本身也是

1---2---3---4 就不是，因为1,4不是friend关系

因此主要取决于一颗连通的树上的的任意两点之间的距离，如果最大距离<=2，就是否则就不是。

并且还存在一个选择问题：如何从一棵连通树上选取一个最大距离<=2的节点距离集合？

因为实际上我们可以从一棵生成的树结构上精细化形成一个解集。

## 选择算法(通过回溯)

如下： 对所有距离大于2的点对，试图删除其中一个来形成一个解；直到发现不能删除为止

## 算法

遍历生成图的遍历树，计算每个点距离为2以内的点的集合；返回最大值

~~这个算法的思想是：不必求所有点之间的距离；只需要按照2来扩展即可。~~

~~但是有个问题：一个点到根的点是2，但是到其他的点的距离不一定为2~~

~~distance1[i]-->easy~~

~~distance2[j]-->use set to save this~~

~~sum={0}~~

~~对与G中的每个点i,j(i!=j)~~

~~如果G[i][j]=1;~~

~~sum[i]++;~~

~~对于G中的每个点j,k~~

~~若G[j][k]==1 && G[i][k]!=1&&!=2则G[i][k]=2; sum[i]++;~~

~~//上面这个算法用来统计~~

* ~~赋值不会影响遍历条件~~

还是有必要计算所有的点对之间的距离，求集合个数

## 图算法：最短路径系列

* Dijkstra单源最短路径
* Floyd算法

## 并查集算法

其实一开始我已经想错了，只要所有的连通的子集个数就行了。

用一个精确的数据结构—father数组

某个元素所属的集合属于这个元素的父节点的father

* find(int i)

while(prev[i]!=i)i=prev[i];

return i

* union(int i,int j)

{

prev[find[i]]=prev[find[j]];

}

# 548. Split Array With Equal Sum

## Problem

Given an array with n integers, you need to find if there are triplets (i, j, k) which satisfies following conditions:

1. 0 < i, i + 1 < j, j + 1 < k < n - 1
2. Sum of subarrays (0, i - 1), (i + 1, j - 1), (j + 1, k - 1) and (k + 1, n - 1) should be equal.

where we define that subarray (L, R) represents a slice of the original array starting from the element indexed L to the element indexed R.

**Example:**

**Input:** [1,2,1,2,1,2,1]

**Output:** True

**Explanation:**

i = 1, j = 3, k = 5.

sum(0, i - 1) = sum(0, 0) = 1

sum(i + 1, j - 1) = sum(2, 2) = 1

sum(j + 1, k - 1) = sum(4, 4) = 1

sum(k + 1, n - 1) = sum(6, 6) = 1

**Note:**

1. 1 <= n <= 2000.
2. Elements in the given array will be in range [-1,000,000, 1,000,000].

## Thought

*~~sum[i1~j1] = sum[i2~j2] = sum[i3~j3]=sum[i4~j4]~~*

*~~find such i,j,k~~*

*~~i,j,k are dividing points~~*

*~~对不同的i,j,k进行取值，就能知道存不存在；~~*

*~~由于n太大而无法实施；~~*

*~~O(n^3)~~*

*~~calculate sum[i~j] is O(n)~~*

*~~sum[j1] - sum[i1]~~*

*~~sum[j2] - sum[j1+2]~~*

*~~sum[j3] – sum[j2+2]~~*

*~~sum[j4] – sum[j3+2]~~*

*~~f(i,j,k) 表示第一个分界点在0~i-1之间，第二个分界点在i+1~j-1之间，第三个分界点在j+1~k-1之间,第四个分界点在k+1,n-1之间~~*

*~~i,j,k不作为分界点~~*

*~~f(i,j,k)~~*

*~~g(i,j+1)~~*

* *~~如果A[j+1]=0维持，~~*
* *~~A[j+1]!=0, 或者这个元素必须能够接收到最后一个组内； 需要调整分组~~*

*~~h(i,j,m) i~j以和的分成m个分组~~*

*~~k作为分组大小，将数组a~b m分~~*

*~~h(a,b,k,m)~~*

*~~和的大小取决于第一个分组；后面就是平分的问题~~*

*~~以不同sum[0~i]作为和值，试图将i+1~n 3分；这就变成了一个确定的3分问题；~~*

* *~~3分问题~~*

*~~对于i~j要求和为k的3分， 设置指针 mi=i,mj=j，依次向下递减直到发现第一个和为k；~~*

*~~[ k ][…][k]~~*

*~~计算中间部分的和如果也为k，则返回相等；否则继续继续缩小中间部分的范围；重复上面过程；直到mi>=mj~~*

*~~如果求得一个解，可以将其记录下来。~~*

* *~~3分问题~~*

*~~求开始和末尾的和是否相同的所有下标。~~*

*~~反复运用二分问题即可；~~*

*~~f(i,j) 表示 0~i i<j j~n-1的二分问题；~~*

*~~如果A[i] + sum(0~i-1)=A[j]+sum(j~n-1) 则产生一个解，并移入到下一个f(i+1,j-1)~~*

*~~否则 如果解就是 {将A[i]加到前面的和上，f(i+1,j) } {或者将A[j]加到 后面的和上,f(i,j-1)}~~*

*~~前缀和和后缀和相同，只能取这些值相等的解；（剪枝）~~*

*~~对前缀和和后缀和分别排序，然后分别比较每个前缀和和后缀和，如果相等~~*

*~~中缀和 中间的某一个点（不包含其自身）到达两端任意一个点的和相同的所有组合及其值和下标范围~~*

* 4分问题（verfied）

对于每一对i,j 如果

sum[0~i]=sum[j~n-1],且 sum[i+1~j-1]=2\*sum[0~j] 则对 i+1~j-1 做以sum[0~i]为和的2分；

实际上只需要查找从i+1开始是否存在和为sum[0~i]的一个值即可

可以对所有的sum[i~j]做一个统计，耗时 n^2\*logn（使用map结构），将每一组下标存储起来；

遍历map结构，如果有一个sum含有至少4组下标，检查下标之间是否存在一个划分

# 552. Student Attendance Record II

## problem

Given a positive integer **n**, return the number of all possible attendance records with length n, which will be regarded as rewardable. The answer may be very large, return it after mod 109 + 7.

A student attendance record is a string that only contains the following three characters:

1. **'A'** : Absent.
2. **'L'** : Late.
3. **'P'** : Present.

A record is regarded as rewardable if it doesn't contain **more than one 'A' (absent)** or **more than two continuous 'L' (late)**.

**Example 1:**

**Input:** n = 2

**Output:** 8

**Explanation:**

There are 8 records with length 2 will be regarded as rewardable:

"PP" , "AP", "PA", "LP", "PL", "AL", "LA", "LL"

Only "AA" won't be regarded as rewardable owing to more than one absent times.

**Note:** The value of **n** won't exceed 100,000.

class Solution {

public:

int checkRecord(int n) {

}

};

## Thought(Easy Easy)

这是一道组合数学的题目

\*LL\*

\*A\*A\*

都不可以

f(i)表示规为i的子问题

f(i+1)=

A[i+1]=L h(i) 不以LL结尾

A[i+1]=A g(i) g表示没有A的组合,没有3个L的组合

A[i+1]=P f(i) 无影响

h(i)

g(i)

A[i]=L

i>=2

f(i+1) = f(i-1)\*2 + 2^i + f(i)

i<2时

f(2) = 8

f(1) = 3

# 附录：C++ std::array<type,size>

提供与C语言数组相同的性能，但是具有更多的方法；不过，遗憾的是，不支持动态数组。

#include <string>

#include <iterator>

#include <iostream>

#include <algorithm>

#include <array>

int main()

{

// construction uses aggregate initialization

std::array<int, 3> a1{ {1, 2, 3} }; // double-braces required in C++11 (not in C++14)

std::array<int, 3> a2 = {1, 2, 3}; // never required after =

std::array<[std::string](http://en.cppreference.com/w/cpp/string/basic_string), 2> a3 = { [std::string](http://en.cppreference.com/w/cpp/string/basic_string)("a"), "b" };

// container operations are supported

[std::sort](http://en.cppreference.com/w/cpp/algorithm/sort)(a1.begin(), a1.end());

[std::reverse\_copy](http://en.cppreference.com/w/cpp/algorithm/reverse_copy)(a2.begin(), a2.end(),

[std::ostream\_iterator](http://en.cppreference.com/w/cpp/iterator/ostream_iterator)<int>([std::cout](http://en.cppreference.com/w/cpp/io/cout), " "));

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << '**\n**';

// ranged for loop is supported

for(const auto& s: a3)

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << s << ' ';

}

# 附录：C++ new多维数组

C++中不可以new多维数组，因为C++的new操作符只能分配一维的数组。

可以使用 new int[10]{1,2,3}来提供初始化元素。

注意赋值： int a[]=new int[9]; 这种做法是错误的，因为new返回的是指针。

int main()

{

int \*pos=new int[7] {0,1,-1};

int \*pos2=new int[7];

int \*pos3=new int[7]{-1};

int \*pos4=new int[7](-2); //有编译错误，但如果将-2去掉，没有编译错误。如果加上 -fpermissve，则 -2可以初始化所有元素。

std::cout << toString(pos,7) << std::endl;

std::cout << toString(pos2,7) << std::endl;

std::cout << toString(pos3,7) << std::endl;

std::cout << toString(pos4,7) << std::endl;

return 0;

}

参考：<http://en.cppreference.com/w/cpp/language/value_initialization>

#include <string>

#include <vector>

#include <iostream>

struct T1

{

int mem1;

[std::string](http://en.cppreference.com/w/cpp/string/basic_string) mem2;

}; // implicit default constructor

struct T2

{

int mem1;

[std::string](http://en.cppreference.com/w/cpp/string/basic_string) mem2;

T2(const T2&) { } // user-provided copy constructor

}; // no default constructor

struct T3

{

int mem1;

[std::string](http://en.cppreference.com/w/cpp/string/basic_string) mem2;

T3() { } // user-provided default constructor

};

[std::string](http://en.cppreference.com/w/cpp/string/basic_string) s{}; // class => default-initialization, the value is ""

int main()

{

int n{}; // scalar => zero-initialization, the value is 0

double f = double(); // scalar => zero-initialization, the value is 0.0

int\* a = new int[10](); // array => value-initialization of each element

// the value of each element is 0

T1 t1{}; // class with implicit default constructor =>

// t1.mem1 is zero-initialized, the value is 0

// t1.mem2 is default-initialized, the value is ""

// T2 t2{}; // error: class with no default constructor

T3 t3{}; // class with user-provided default constructor =>

// t3.mem1 is default-initialized to indeterminate value

// t3.mem2 is default-initialized, the value is ""

[std::vector](http://en.cppreference.com/w/cpp/container/vector)<int> v(3); // value-initialization of each element

// the value of each element is 0

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << s.size() << ' ' << n << ' ' << f << ' ' << a[9] << ' ' << v[2] << '**\n**';

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << t1.mem1 << ' ' << t3.mem1 << '**\n**';

delete[] a;

}

Possible output:

0 0 0 0 0

0 4199376

# 附录：Algorithm X

## exact cover

（数学上）给定S是X的一系列子集，一个exact cover是S的一个子集S\*,X中的每一个元素都被包含在S\*中，且S\*之间相交为空。

（计算机）一个决策问题，确定一个exact cover是否存在。NP-complete。

Algorithm X， 用于找到所有的exact cover的解。

## Algorithm X

使用一个0 1矩阵A，目标是选择A中一个行的子集，以使1恰好出现在每一列一次。

算法步骤：

如果A没有列，则终止

选择一列c（选择1的个数最少的那一列，选择的原则是使子问题的分支最少，确定性）

选择一行r，使得A[r,c]=1

将r加入到解中

对任意A[r,j]=1

对任意 A[i,j]=1

将 行i从A中删除

将列j从A中删除

递归在减少的A上使用这个算法

## 正确性证明

设f(A)表示求关于问题A的精确覆盖，每一次选择一列来确定，选择这一列要使变化的可能最小。

然后选择一行来进行测试，如果选择了那一行，则那一行的1所在的列都会被设置，删除那一行的所有为1的列以及与与那些列相交的行（因为那些值都是可以排除的）。

直到选择完所有的行。

## 思想评述

实际上Knuth在此算法中完成了一般的、通用的利用 DLX 进行状态保存的方法。过去，我们可以很轻易地在某些结构中通过标记-1/+1来表明数组元素是否可用；而DLX则通过数据结构将其移除然后回复， 这正是对应了递归—回溯的过程。

实际上Knuth解决的不是关于精确覆盖的算法问题，他是提出了一种用于递归-回溯的通用数据结构。

## 例子

For example, consider the exact cover problem specified by the universe *U* = {1, 2, 3, 4, 5, 6, 7} and the collection of sets {\displaystyle {\mathcal {S}}} = {*A*, *B*, *C*, *D*, *E*, *F*}, where:

* *A* = {1, 4, 7};
* *B* = {1, 4};
* *C* = {4, 5, 7};
* *D* = {3, 5, 6};
* *E* = {2, 3, 6, 7}; and
* *F* = {2, 7}.

This problem is represented by the matrix:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| ***A*** | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| ***B*** | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| ***C*** | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| ***D*** | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| ***E*** | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| ***F*** | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

## 实现

[Donald Knuth](https://en.wikipedia.org/wiki/Donald_Knuth)'s main purpose in describing Algorithm X was to demonstrate the utility of [Dancing Links](https://en.wikipedia.org/wiki/Dancing_Links). Knuth showed that Algorithm X can be implemented efficiently on a computer using [Dancing Links](https://en.wikipedia.org/wiki/Dancing_Links) in a process Knuth calls *"DLX"*. DLX uses the matrix representation of the [exact cover](https://en.wikipedia.org/wiki/Exact_cover) problem, implemented as [doubly linked lists](https://en.wikipedia.org/wiki/Doubly_linked_list) of the 1s of the matrix: each 1 element has a link to the next 1 above, below, to the left, and to the right of itself. (Technically, because the lists are circular, this forms a torus). Because exact cover problems tend to be sparse, this representation is usually much more efficient in both size and processing time required. DLX then uses [Dancing Links](https://en.wikipedia.org/wiki/Dancing_Links) to quickly select permutations of rows as possible solutions and to efficiently backtrack (undo) mistaken guesses.[[1]](https://en.wikipedia.org/wiki/Knuth%27s_Algorithm_X#cite_note-knuth-1)

使用Dancing Links实现可以加快猜解并撤销的速度。

## 参考

N皇后问题是一个更加广泛的精确覆盖算法问题。

<https://arxiv.org/pdf/cs/0011047.pdf> Knuth原始的论文

# 附录：Clustering

聚类：将一组对象分组，每一组内的元素之间相似度大于组间任何元素的相似度。

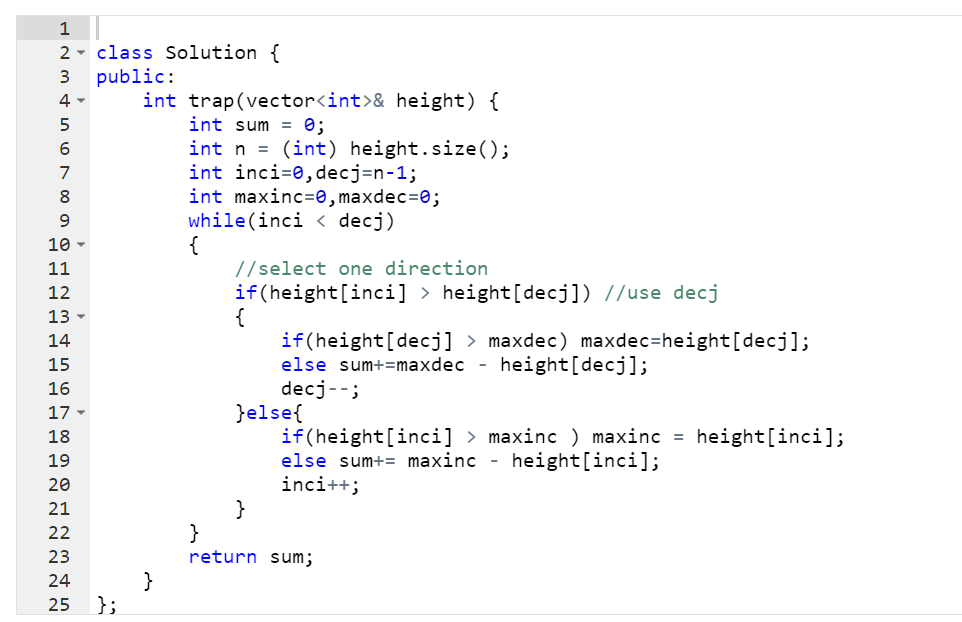
## Nearest-neighbor chain algorithm

最近邻

# 附录：一个关于改进程序的观察

这是在第42个问题中同一算法的两个版本：

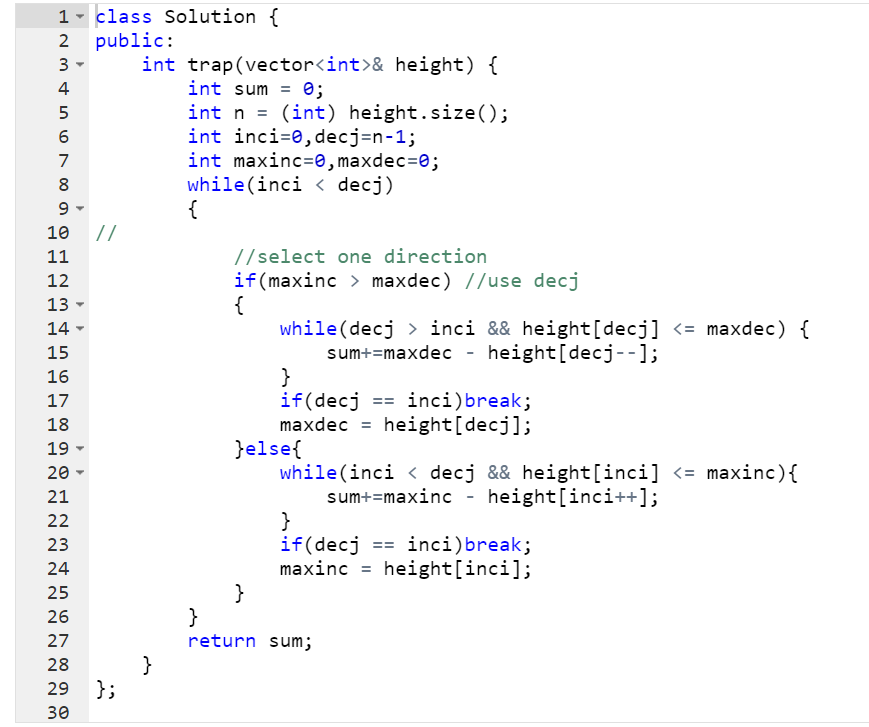
1. 原始版本



运行时间

15 ms， 用例数 315， 22.31%

1. 所谓的改进版本



运行时间： 12ms 用例同上 35.45%

# 附录：一个关于抓取Leetcode数据绘制图表的计划