# 算法记录

/**********************************************************

2017-7-24

1.SUM

Given an array of integers, return indices of the two numbers such that they add up to a specific target. You may assume that each input would have exactly one solution, and you may not use the same element twice.

Example:

Given nums = [2, 7, 11, 15], target = 9,
Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1]

my tip:
这个用到了图
输入输出用到了向量
为了高效用到了图

.h

```
#include<vector>
#include <unordered_map>
**********************************************************/
class Solution {
public:
 vector<int> twoSum(vector<int>& numbers, int target)     //定义了向量作为返回值  target 输入的和
 {
  unordered_map<int, int> hash;                          //定义了一个图
  vector<int> result;
  for (int i = 0; i < numbers.size(); i++)
  {
   int numberToFind = target - numbers[i];
 //find   unordered_map的函数,   如果查找到则返回一个迭代器 如果没有则返回end
   if (hash.find(numberToFind) != hash.end())  //if numberToFind is found in map, return them

   {
    result.push_back(hash[numberToFind] + 1); //+1 because indices are NOT zero based
                                              //直接hash ()   返回的是序号位置   但是为什么呢
    result.push_back(i + 1);
    return result;
   }
   //number was not found. Put it in the map.
   hash[numbers[i]] = i;
  }
  return result;
```

```
 }
};

/**************************************************************************************************
2017-7-6
2.Add Two Numbers

You are given two non-empty linked lists representing two non-negative integers.
The digits are stored in reverse order and each of their nodes contain a single digit.
Add the two numbers and return it as a linked list.
You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8

.h
结构体也可以有构造函数

my tip
输出结果的赋值只从第二个值开始赋值的  不是从第一个开始赋值的  我试着写了从第一个开始赋值
但是不知道为啥结果第一个的结果是0


**************************************************************************************************/
 struct ListNode
{
 int val;
 ListNode *next;
 ListNode(int x) : val(x), next(NULL) {}        //这里应该是构造函数  初始化 val为x  next 为 null
};


ListNode *addTwoNumbers(ListNode *l1, ListNode *l2)
{
 ListNode preHead(0), *p = &preHead;
 int extra = 0;
 while (l1 || l2 || extra) // if *p  is the end
 {
  int sum = (l1 ? l1->val : 0) + (l2 ? l2->val : 0) + extra;     //看l1 或者l2是否为空  计算进位
  extra = sum / 10;
  p->next = new ListNode(sum % 10);
  p = p->next;
  l1 = l1 ? l1->next : l1;
  l2 = l2 ? l2->next : l2;
 }
 return preHead.next;
}
```

/**********************************************************************************************************
2017-8-17

3. Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters.
Examples:
Given "abcabcbb", the answer is "abc", which the length is 3.
Given "bbbbb", the answer is "b", with the length of 1.
Given "pwwkew", the answer is "wke", with the length of 3.
Note that the answer must be a substring, "pwke" is a subsequence and not a substring.

```
#include <vector>
#include"string"
#include<algorithm>    //为了使用max
using namespace std;
```

my tip
巧妙的利用的一个256长度的向量　包含所有的ASCII 例如 a　为 97　则dict[s[i]] --->dict[97] 赋值为a的下标
例如a为第一个字母则为0 dict[97]=0; 如果再遇到a　因为已经遇到了一次 所以 dict[97]>-1; 把start重置为a的下标
这样在计算时用i-a的下标 即为length

**********/

```
class Solution {
public:
int lengthOfLongestSubstring(string s) {

vector<int> dict(256, -1);    //用向量模板定义一个向量 共256个元素 初始化为-1
int maxLen = 0, start = -1;
for (int i = 0; i != s.length(); i++)    //根据向量的特性 这里最好用 !=  而不用 <
{
 if (dict[s[i]] > start) //dict[s[i]] 例如第一个字母为a 即97 dict[97]如果被赋值则大于-1
 start = dict[s[i]];
 dict[s[i]] = i;
 maxLen = max(maxLen, i - start);
}
return maxLen;
}
};
```

# 4. Median of Two Sorted Arrays

To solve this problem, we need to understand "What is the use of median". In statistics, the median is used for dividing a set into two equal length subsets, that one subset is always greater than the other. If we understand the use of median for dividing, we are very close to the answer.

First let's cut A into two parts at a random position i:

```
      left_A             |        right_A
A[0], A[1], ..., A[i-1]  |  A[i], A[i+1], ..., A[m-1]
```

Since A has m elements, so there are m+1 kinds of cutting( i = 0 ~ m ). And we know: len(left_A) = i, len(right_A) = m - i . Note: when i = 0 , left_A is empty, and when i = m , right_A is empty.

With the same way, cut B into two parts at a random position j:

```
      left_B             |        right_B
B[0], B[1], ..., B[j-1]  |  B[j], B[j+1], ..., B[n-1]
```

Put left_A and left_B into one set, and put right_A and right_B into another set. Let's name them left_part and right_part :

```
      left_part          |        right_part
A[0], A[1], ..., A[i-1]  |  A[i], A[i+1], ..., A[m-1]
B[0], B[1], ..., B[j-1]  |  B[j], B[j+1], ..., B[n-1]
```

If we can ensure:

1) len(left_part) == len(right_part)
2) max(left_part) <= min(right_part)

then we divide all elements in {A, B} into two parts with equal length, and one part is always greater than the other. Then median = (max(left_part) + min(right_part))/2.

To ensure these two conditions, we just need to ensure:

(1) i + j == m - i + n - j (or: m - i + n - j + 1)
    if n >= m, we just need to set: i = 0 ~ m, j = (m + n + 1)/2 - i
(2) B[j-1] <= A[i] and A[i-1] <= B[j]

ps.1 For simplicity, I presume A[i-1],B[j-1],A[i],B[j] are always valid even if i=0/i=m/j=0/j=n . I will talk about how to deal with these edge values at last.

ps.2 Why n >= m? Because I have to make sure j is non-nagative since 0 <= i <= m and j = (m + n + 1)/2 - i. If n < m , then j may be nagative, that will lead to wrong result.

So, all we need to do is:

Searching i in [0, m], to find an object `i` that:
    B[j-1] <= A[i] and A[i-1] <= B[j], ( where j = (m + n + 1)/2 - i )
And we can do a binary search following steps described below:

<1> Set imin = 0, imax = m, then start searching in [imin, imax]

<2> Set i = (imin + imax)/2, j = (m + n + 1)/2 - i

<3> Now we have len(left_part)==len(right_part). And there are only 3 situations
     that we may encounter:
    <a> B[j-1] <= A[i] and A[i-1] <= B[j]

Means we have found the object `i`, so stop searching.

<b> B[j-1] > A[i]

Means A[i] is too small. We must `ajust` i to get `B[j-1] <= A[i]`.

Can we `increase` i?

Yes. Because when i is increased, j will be decreased.

So B[j-1] is decreased and A[i] is increased, and `B[j-1] <= A[i]` may be satisfied.

Can we `decrease` i?

`No!` Because when i is decreased, j will be increased.

So B[j-1] is increased and A[i] is decreased, and B[j-1] <= A[i] will be never satisfied.

So we must `increase` i. That is, we must ajust the searching range to [i+1, imax]. So, set imin = i+1, and goto <2>.

<c> A[i-1] > B[j]

Means A[i-1] is too big. And we must `decrease` i to get `A[i-1]<=B[j]`.

That is, we must ajust the searching range to [imin, i-1].

So, set imax = i-1, and goto <2>.

When the object i is found, the median is:

max(A[i-1], B[j-1]) (when m + n is odd)

or (max(A[i-1], B[j-1]) + min(A[i], B[j]))/2 (when m + n is even)

Now let's consider the edges values i=0,i=m,j=0,j=n where A[i-1],B[j-1],A[i],B[j] may not exist. Actually this situation is easier than you think.

What we need to do is ensuring that max(left_part) <= min(right_part). So, if i and j are not edges values(means A[i-1],B[j-1],A[i],B[j] all exist), then we must check both B[j-1] <= A[i] and A[i-1] <= B[j]. But if some of A[i-1],B[j-1],A[i],B[j] don't exist, then we don't need to check one(or both) of these two conditions. For example, if i=0, then A[i-1] doesn't exist, then we don't need to check A[i-1] <= B[j]. So, what we need to do is:

Searching i in [0, m], to find an object `i` that:

(j == 0 or i == m or B[j-1] <= A[i]) and

(i == 0 or j == n or A[i-1] <= B[j])

where j = (m + n + 1)/2 - i

And in a searching loop, we will encounter only three situations:

<a> (j == 0 or i == m or B[j-1] <= A[i]) and

(i == 0 or j = n or A[i-1] <= B[j])

Means i is perfect, we can stop searching.

<b> j > 0 and i < m and B[j - 1] > A[i]

Means i is too small, we must increase it.

<c> i > 0 and j < n and A[i - 1] > B[j]

Means i is too big, we must decrease it.

Thank @Quentin.chen , him pointed out that: i < m ==> j > 0 and i > 0 ==> j < n . Because:

m <= n, i < m ==> j = (m+n+1)/2 - i > (m+n+1)/2 - m >= (2*m+1)/2 - m >= 0

m <= n, i > 0 ==> j = (m+n+1)/2 - i < (m+n+1)/2 <= (2*n+1)/2 <= n
So in situation <b> and <c>, we don't need to check whether j > 0 and whether j < n.

Below is the accepted code:

```
def median(A, B):
    m, n = len(A), len(B)
    if m > n:
        A, B, m, n = B, A, n, m
    if n == 0:
        raise ValueError

    imin, imax, half_len = 0, m, (m + n + 1) / 2
    while imin <= imax:
        i = (imin + imax) / 2
        j = half_len - i
        if i < m and B[j-1] > A[i]:
            # i is too small, must increase it
            imin = i + 1
        elif i > 0 and A[i-1] > B[j]:
            # i is too big, must decrease it
            imax = i - 1
        else:
            # i is perfect

            if i == 0: max_of_left = B[j-1]
            elif j == 0: max_of_left = A[i-1]
            else: max_of_left = max(A[i-1], B[j-1])

            if (m + n) % 2 == 1:
                return max_of_left

            if i == m: min_of_right = B[j]
            elif j == n: min_of_right = A[i]
            else: min_of_right = min(A[i], B[j])

            return (max_of_left + min_of_right) / 2.0
```

## 第 5 节　最短路径算法对比分析

| | Floyd | Dijkstra | Bellman-Ford | 队列优化的 Bellman-Ford |
|---|---|---|---|---|
| 空间复杂度 | $O(N^2)$ | $O(M)$ | $O(M)$ | $O(M)$ |
| 时间复杂度 | $O(N^3)$ | $O((M+N)\log N)$ | $O(NM)$ | 最坏也是 $O(NM)$ |
| 适用情况 | 稠密图<br>和顶点关系密切 | 稠密图<br>和顶点关系密切 | 稀疏图<br>和边关系密切 | 稀疏图<br>和边关系密切 |
| 负权 | 可以解决负权 | 不能解决负权 | 可以解决负权 | 可以解决负权 |

Floyd 算法虽然总体时间复杂度高，但是可以解决负权边，并且均摊到每一点对上，在所有的算法中还是属于较优的。另外，Floyd 算法较小的编码复杂度也是它的一大优势。所以，如果要求的是所有点对间的最短路径，或者如果数据范围较小，则 Floyd 算法比较适合。

Dijkstra 算法最大的弊端是它无法适应有负权边的图。但是 Dijkstra 具有良好的可扩展性，扩展后可以适应很多问题。另外用堆优化的 Dijkstra 算法的时间复杂度可以达到 $O(M\log N)$。当边有负权时，需要使用 Bellman-Ford 算法或者队列优化的 Bellman-Ford 算法。因此我们选择最短路径算法时，要根据实际需求和每一种算法的特性，选择适合的算法。