**Algorithm Notes**

目录

[1 Array 1](#_Toc2267414)

[1.1 HashTable 1](#_Toc2267415)

[1 Two Sum 1](#_Toc2267416)

[1.2 Two Pointer 1](#_Toc2267417)

[15 3 sum 2](#_Toc2267418)

[16 3 sum closest 2](#_Toc2267419)

[18 4 sum 2](#_Toc2267420)

[1.3 backtracking 3](#_Toc2267421)

[39 combination sum 3](#_Toc2267422)

# 1 Array

## 1.1 HashTable

利用hash数据结构记录数组中元素的出现，然后对相应的元素直接检查获取。

### 1 Two Sum

通过hash表记录各个元素的出现，然后在遍历元素的过程中，检查目标和与该元素的差是否出现，**值得注意的是：相对于完整的遍历存储好再检查，可以边遍历边检查。**

## 1.2 Two Pointer

在**有序**的前提下寻找符合特定条件的两个元素时，相对于暴力的两层遍历叠加，可以使用一头一尾两个指针，分别从前后逼近查找。

**核心代码：**

for(int i=0;i<nums.size();i++){

int res = target-nums[i];

if(num\_map.find(res)!=num\_map.end()){

result[0] = num\_map[res];

result[1] = i;

return result;

}else{

num\_map[nums[i]] = i;

}

}

### 15 3 sum

三个数的求和，指定其中一个（依序便利），对另外两个数使用两个指针

核心代码：

while(l<r){

if(nums[l]+nums[r]<-nums[i])l++;

else if(nums[l]+nums[r]>-nums[i])r--;

else{

tmp[0] = nums[i];

tmp[1] = nums[l];

tmp[2] = nums[r];

res.push\_back(tmp);

while(l<r && nums[l]==tmp[1])l++;

while(l<r && nums[r]==tmp[2])r--;

}

}

### 16 3 sum closest

同15，不同的地方在于最近的元素需要记录最小的差

### 18 4 sum

同15，只是外面有两层遍历，相应的需要两层去重

if(i!=0 && nums[i]==nums[i-1])continue;

…

if(j!=i+1&&nums[j]==nums[j-1])continue;

## 1.3 backtracking

对每个可能的分支遍历时，达到一定条件便回溯到之前的位置继续遍历其他分支。值得注意的是：为了节省空间和时间，结果集可以在遍历的过程中通过函数参数传递从而达到只使用一个结果集的目的。

**关键点：**

* **参数传递**
* **有进有出**

### 39 combination sum

**Input:** candidates = [2,3,6,7], target = 7,

**A solution set is:**

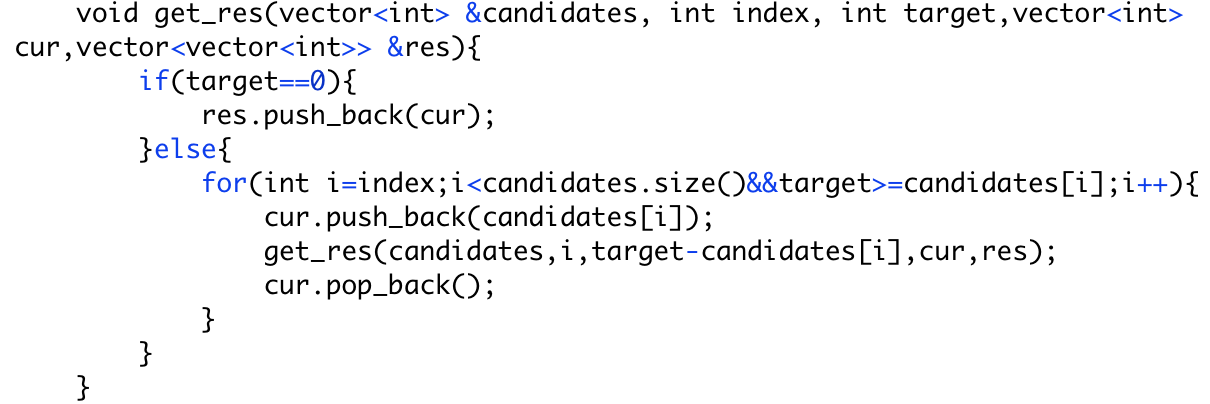
[

[7],

[2,2,3]

]

核心代码：



**注意：**

* **cur有push\_back也有pop\_back, 满足回溯要求的有进有出**
* **为了减少不必要的操作（如减去大于target的candidates元素），可以提前判断（在循环中），减少时间复杂度。**