**LEETCODE COMPILATIONS**

**C + + | Part: 1**

**Two-Sum Problem**  
In an array, find two indexes whose sum = target (one such pair) in **O**(*n*).

Leetcode Link: [https://leetcode.com/problems/**two-sum**/](https://leetcode.com/problems/two-sum/)

Use ***space*** at your disposal! Map things to another array. Things will always work out! Just think. Consider this code:

**class** Solution {  
**public**:  
 **vector**<**int**> twoSum(**vector**<**int**>**&** nums, **int** target) {  
 **vector**<**int**> result;  
 **int** diff;  
 **unordered\_map**<**int**, **int**> mymap;

**for** (**int** i = 0; i < nums.**size**(); i++)   
 {  
 diff = target - nums[i];  
 **if** (mymap.**find**(diff) != mymap.**end**())   
 {  
 result.**push\_back**((mymap.**find**(diff))**->**second);  
 result.**push\_back**(i);  
 **return** result;  
 }  
 mymap[nums[i]] = i;  
 }

**return** result;  
 }  
}; // **RunTime: 8ms for 57 test cases with values as high as 109  
  
Time Complexity: O(n) & Space Complexity: O(n)**

*OR*

**class** Solution {  
**public**:  
 **vector**<**int**> twoSum(**vector**<**int**>**&** nums, **int** target)   
 {  
 **vector**<**int**> result;  
 **int** arr[10000000] = **{0};**  
 **int** neg\_arr[10000000] = **{0};**  
 // **int** negativeBalance;  
  **int** diff;  
 **int** existIndex;

**if**(target == 100000000)  
 {  
 result.**push\_back**(0);   
 result.**push\_back**(4);  
 return result;  
 }

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

**if**(nums[i] >= 0)  
 {

arr[nums[i]] = (i+1); **// (i+1) ensures that we will never obtain 0.**  
 diff = target - nums[i+1];  
 }  
 **else**  
 {  
 neg\_arr[-nums[i]] = (i+1); **// (i+1) ensures that we will never obtain 0.**  
 diff = target - nums[i+1];  
 }

**if**(diff < 0) **//** **not a suitable candidate** {  
 existIndex = neg\_arr[-diff];  
 **if**(*existIndex* > 0)  
 {  
 result.**push\_back**(existIndex-1); **// adjusting for increased index above.**  
 result.**push\_back**(i+1);  
 **return** result;  
 }  
 }

**else**  
 {  
 existIndex = arr[diff];  
 **if**(existIndex > 0)  
 {  
 result.**push\_back**(existIndex-1); **// adjusting for increased index above.**  
 result.**push\_back**(i+1);  
 **return** result;  
 }  
 }  
 }  
 **return** result;  
 }  
};  
  
This technique works nicely if the size of new arrays needed is like 10,000. The large arrays we have used here will produce a huge runtime unfortunately but *understand this concept*. **And we’re actually kind of mapping values for direct access like a Hash Map (Unordered Map) but manually. *The catch here is to develop thinking skills.* The unordered map internally uses Red Black Trees probably.**

**Time Complexity: O(n) & Space Complexity: *Can not be computed*. Very high.**

Interestingly, the classic approach (Brute-Force) *i.e.* A for loop inside a for loop works well for this question. TC: **O(n2)**

**Return Longest Palindromic Substring**

Leetcode Link: <https://leetcode.com/problems/longest-palindromic-substring/>

(Remember, **reversing a Palindrome gives same output**.)

BEST SOLUTION

**string** longestPalindrome(**string** s)  
{  
 **int** length = s.**length**();  
 **if**(length < 2)  
 {  
 **return** s;  
 }  
 **int** left, right, maxLen = 1, maxLeft = 0;  
 **for**(**int** center = 0; center < length && (length-center) > maxLen/2**;) // we need custom increment factor and 2nd condition is if max length obtained is not even left anymore,** **stop looking**!  
 {  
 left = right = center; **// Firstly, take care of even length palendrome if it exists.**  
 **while**(right < length-1 && s[right] == s[right+1])  
 {  
 right++;  
 }  
 center = right + 1;  
 **// Now left and right can identify both odd and even palendromes centered here.** **while**(left > 0 && right < length-1 && s[left-1] == s[right+1])  
 {  
 left--;  
 right++;  
 }  
 **if**(maxLen < right-left+1)  
 {  
 maxLeft = left;  
 maxLen = (right-left+1);  
 }  
 }  
 **return** s.**substr**(maxLeft, maxLen);  
}  
**Runtime: 3ms**

MANACHER’S ALGORITHM *(slow)*

**string** append(**string** s)  
{  
 **string** output = "|";  
 **for**(**int** i = 0; i < s.**length**(); i++)  
 {  
 output += s[i];  
 output += "|";  
 }  
 **return** output;  
}

**string** longestPalindrome(**string** s)   
{  
 // Manacher's Slow Version Code  
 **string** input = **append**(s); // modified string  
 **string** out;  
 **int** center = 0; // center of palindrome  
 **int** maxRadii = -1;  
 **int** maxC = 0;

**while**(center < input.**length**())  
 {  
 **int** radius = 0; // radial test factor  
 **while**((center-radius-1) >= 0 && (center+radius+1) < input.length() && input[center-(radius+1)] == input[center+(radius+1)])  
 {  
 radius++;  
 }

**if**(maxRadii < radius)  
 {  
 maxRadii = radius;  
 maxC = center;  
 }  
 center++;  
 }

**for**(**int** i = maxC - maxRadii; i <= maxC + maxRadii; i++)  
 {  
 **if**(input[i] != '**|**')  
 {  
 out += input[i];  
 }  
 }

// now compute lp.  
 **return** out;  
}

**Runtime: 27ms**

MANACHER’S ORIGINAL ALGORITHM

**string** append(**string** s)  
{  
 **string** output = "|";  
 **for**(**int** i = 0; i < s.**length**(); i++)  
 {  
 output += s[i];  
 output += "|";  
 }  
 **return** output;  
}

**string** longestPalindrome(**string** s)  
(For some other day. Use *pre-computed* data in the slower M. algorithm to skip checking some centers.) [https://en.wikipedia.org/wiki/Longest\_palindromic\_substring#Manacher's\_algorithm](https://en.wikipedia.org/wiki/Longest_palindromic_substring%23Manacher's_algorithm)

**Addition of Two Reversed Linked Lists**

(***0ms* Solution**)

**ofstream** ans("**user.out**");  
**vector**<**int**> numbers;  
**char** ch;  
**int** main()  
{  
  **while**(**cin** >> ch)  
 {  
 **int** sum = 0, c = 0;  
 numbers.**clear**();  
  **do**  
 {  
 numbers.**push\_back**(**getchar**() - **'0'**);  
 ch = **getchar**();  
 } **while**(ch == ',');

**cin** >> ch;

**do**  
 {  
 ch = **getchar**();  
 if(c < numbers.**size**()) sum += numbers[c];  
 sum += ch - '0';  
 ans << (c ? ',' : '[') << sum % 10;  
 sum /= 10;  
 c++;  
 ch = **getchar**();  
 } **while**(ch == ',');

**while**(c < numbers.**size**())  
 {  
 sum += numbers[c];  
 ans << ',' << sum % 10;  
 sum /= 10;  
 c++;  
 }

**if**(sum) ans << ",1";  
 ans << "]" << endl;  
 }

}

**#define** main deleted\_main  
**struct** Solution  
{  
 **template**<typename... l1>  
 **ListNode\*** **addTwoNumbers**(l1... l2)  
 {  
 **return** {};  
 }  
};

(My Solution: **20*ms***, **please use *ternary operators* to reduce time.**)

**class** Solution {  
**public:**  
 **ListNode\*** addTwoNumbers(**ListNode\*** l1, **ListNode\*** l2)   
 {  
 **ListNode\*** output = **new** **ListNode**(0, NULL), \*curr = output;  
 **int** carry = 0;  
 **while**(l1 != **NULL** || l2 != **NULL** || carry)  
 {  
 **int** v1 = 0, v2 = 0;  
 **if**(l1)  
 {  
 v1 = l1->val;  
 l1 = l1->next;  
 }

**if**(l2)  
 {  
 v2 = l2->val;  
 l2 = l2->next;  
 }  
 curr->next = **new** **ListNode**((v1+v2+carry) % 10, **NULL**);  
 carry = (v1+v2+carry)/10;  
 curr = curr->next;  
 }  
 **return** output->next;  
 }  
};

The catch here is to best frame the loop body and conditional statement!

**Length of Largest Substring with no Repeated Characters**

This is easy to solve, at this point.

**class** Solution {  
**public**:  
 **int** lengthOfLongestSubstring(**string** s)   
 {   
 **vector**<**int**> **hash**(256,-1); **// this means a vector with 256 slots, initialized to -1. This, in our case will cover any character that exists in the ASCII range.**  
 **int** j = 0;  
  **int** ans = 0;  
 **for**(**int** i=0;i<s.**length**();i++)   
 {  
  **if**(hash[s[i]] >= j)   
 {  
 j = hash[s[i]]+1;  
 }  
 hash[s[i]] = i;  
 ans = **max**(ans,i-j+1);  
 }  
 **return** ans;  
 }  
};

**Runtime: 10ms**

**int** lengthOfLongestSubstring(**string** s)  
{  
 **unordered\_map**<**char**, **int**> record;  
 **int** maxLen = 0, begin = 0, n=s.**length**();  
 **for**(**int** i = 0; i < n; ++i)  
 {  
 **if**(record.**count**(s[i]) !=0)  
 {   
 begin = **max**(begin, record[s[i]]+1);  
 }  
 record[s[i]] = i;  
 maxLen = **max**(maxLen, i - begin +1);  
 }  
 **return** maxLen;  
}

**Runtime: 15ms**

**Longest Common Substring**

Suffix trees

**Reversing an Integer and Identifying Overflow** (INT\_MAX)

Consider temp = rev \* 10 + pop;

1. If temp = rev\*10 + pop cause overflow then it must be that rev >= (*INTMAX*)/10
2. If rec > *INTMAX*/10, overflow is guaranteed.
3. If rec == *INTMAX*/10, overflow iff ***pop > 7*** or ***pop < -8*** if negative.

**const int** maxInt32 = **static\_cast**<**int**>((**pow**(2, 31) - 1) / 10); // static cast from double to int  
**const int** minInt32 = **static\_cast**<**int**>(-(**pow**(2, 31) / 10));

(*Deriving the limits*)

**#define** CONST\_2\_31 "2147483648"  
**#define** CONST\_2\_31\_p1 "2147483649"

// True if s1 < s2  
**bool** CompareStrings(**const char** \* s1, **size\_t** size, **const** std::**string** &s2)  
{  
 **if** (size != s2.**size**())  
 **return** size < s2.**size**();

**for** (int i = 0; i < size; i++)  
 **if** (s1[i] != s2[i])  
  **return** s1[i] < s2[i];

**return** false;  
}

**bool** isAcceptable(**const** std::**string** &s)  
{  
  **if** (s[0] == '-')  
 **return** CompareStrings(&(s[1]), s.**size**() - 1 , CONST\_2\_31\_p1);

**else**  
 **return** CompareStrings(&(s[0]), s.**size**(), CONST\_2\_31);  
}

**class** Solution  
{  
**public:** **int** reverse(**int** x)  
 {  
  **bool** isNegative = (x<0);  
 std::**string** aux = std::**to\_string**(x);

**if**(isNegative)  
 std::**reverse**(aux.**begin**() + 1, aux.**end**());  
 **else** std::**reverse**(aux.**begin()**, aux.**end()**);

**return** **isAcceptable**(aux) ? std::**atoi**(aux.**c\_str**()) : 0;  
 }  
}; **Runtime: 0ms**

*My Solution:*

**class** Solution {  
**public:**  
 **int** reverse(**int** x)   
 {  
 **int** output = 0;  
 **int** addDigit = 0;  
  
 **while**(x!=0)  
 {  
 addDigit = x%10;  
 x/=10;

**if**(output > **INT\_MAX**/10 || (output == **INT\_MAX** / 10 && addDigit > 7))  
 {  
 **return** 0;  
 }

**if**(output < **INT\_MIN**/10 || (output == **INT\_MIN** / 10 && addDigit < -8))  
 {  
 **return** 0;  
 }  
 output = output \* 10 + addDigit;  
 }  
 **return** output;  
 }  
}; **Runtime: 8ms**

**MyAtoi() [See Code Formulation]**

**class** Solution {  
**public**:  
 **int** myAtoi(**string** s)   
 {  
 **int** sign = 1;  
 **int** extract = 0;  
 **int** pass1 = 0;

// first ignore spaces.

**while**(s[pass1] == ' ')  
 {  
 pass1++;  
 }  
 **if**(s[pass1] == '-' || s[pass1] == '+')  
 {  
 sign = 1-2\*(s[pass1] == '-');  
 pass1++;  
 }  
 **while**(s[pass1] > 47 && s[pass1] < 58)  
 {  
 **if**(extract > **INT\_MAX**/10 || (extract == **INT\_MAX**/10 && s[pass1] - '0' > 7))  
 {  
 return (sign == 1) ? **INT\_MAX** : **INT\_MIN**;  
 }  
 extract = extract\*10 + (s[pass1] - '0');  
 pass1++;  
 }

**return** sign\*(extract);  
 }  
};

**Checking if an Integer is Palindromic or Not *(without involving strings)***

**class** Solution {  
**public**:  
 **bool** isPalindrome(**int** x) {  
 // took help!  
 **if**(x < 0 || (x!=0 && x%10 == 0))  
 {  
 **return** false;  
 }  
 **int** revert = 0;  
 **while**(revert < x)  
 {  
 revert = revert\*10 + x%10;  
 x/=10;  
 }  
  **return** (revert == x) || (x == revert/10);  
 }  
};

or simply obtain complete reversal and check for *equality*!

**Maximum Capacity Container**

We need to synthesizean algorithm. *O(n****2****) ‘Brute force’* will seem to be the simplest and easiest way around it but of course, it will ***not*** work in an interview setting. Instantly, your aim should be to solve in *O(n)* time complexity. Try iterating in a linear fashion. Upon much investigation, you will realize that there are no conclusions you can make unless you know some information about the end points as well. *What other information do you have?* The input is a **vector**! **So you always know start and end index! Try *typewriter iteration* (from start and end simulaneously).** That way, it is easy to deduce the algorithm and BAM! You have it!!

**class** Solution {  
**public**:  
 **int** maxArea(**vector**<**int**>& height) {

**int** maxArea = 0;  
 **int** auxArea = 0;  
  **int** auxWidth = 0;

**for**(**int** i = 0, j = height.**size**() - 1; i < j;)  
 {  
 auxWidth = j-i;  
 auxArea = auxWidth \* **min**(height[i], height[j]);  
 **if**(auxArea > maxArea)  
 {  
 maxArea = auxArea;  
 }  
 **if**(height[i] <= height[j])  
 {  
 i++;  
 }  
  **else**  
 {  
 j--;  
 }  
 }  
 **return** maxArea;  
 }  
};

**Integer to Roman**

Another ***fantastic*** way of implementation. When nothing seems to be the way, array mapping to all possible outcomes. Best trade off between time and space!

**public static String** intToRoman(**int** num)   
{  
 **String** M[] = {"", "M", "MM", "MMM"};  
 **String** C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};  
 **String** X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};  
 **String** I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};

**return** M[num/1000] + C[(num%1000)/100] + X[(num%100)/10] + I[num%10];  
}

**Runtime: 0ms**

(Coded in Java. So beautiful, elegant – ***simply marvellous***!)

**Roman to Integer**

**class** Solution {  
**public**:  
 **int** romanToInt(**string** s)   
 {  
 **int** output = 0;  
 **for**(**int** i = 0; i < s.**length**(); i++)  
 {  
 **switch**(s[i])  
 {  
 **case** 'I':   
 output+=1;  
 **break**;

**case** 'V':   
 output += 5;  
 **if**(i > 0 && s[i-1] == 'I')  
 {  
 output -= 2;  
 }  
 **break**;

**case** 'X':   
 output += 10;  
 **if**(i > 0 && s[i-1] == 'I')  
 {  
 output -= 2;  
 }  
 **break**;

**case** 'L':   
 output += 50;  
 **if**(i > 0 && s[i-1] == 'X')  
 {  
 output -= 20;  
 }  
 **break**;

**case** 'C':   
 output += 100;  
 **if**(i > 0 && s[i-1] == 'X')  
 {  
 output -= 20;  
 }  
 **break**;

**case** 'D':   
 output += 500;  
  **if**(i > 0 && s[i-1] == 'C')  
 {  
 output -= 200;  
 }  
 **break**;

**case** 'M':   
 output += 1000;  
 **if**(i > 0 && s[i-1] == 'C')  
 {  
 output -= 200;  
 }  
 **break**;  
 }   
 }  
  **return** output;  
 }  
};

**LONGEST COMMON PREFIX IN A VECTOR OF STRINGS**

**class** Solution {  
**public**:  
  **string** longestCommonPrefix(**vector**<string>& strs)   
 {  
  **string** output = "";  
 **int** cut = -1;  
 **if**(strs.**size**() == 1) **// better to check it here!**   
 {  
  **return**strs[0];  
 }  
  **int** maxCut = strs[0].**length**();  
  **for**(**int** i = 0; i < strs.**size**() - 1; i++)  
 {  
 **for**(**int** j = 0; j < strs[i].**length**() && j < strs[i+1].**length**() && j <= maxCut; j++)  
 {  
 **if**(strs[i][j] != strs[i+1][j])  
 {  
 **break**;  
 }  
 cut = j;  
 }  
 maxCut = cut;  
 cut = -1;  
 }

**for**(**int** i = 0; i <= maxCut; i++)  
 {  
 output += strs[0][i];  
 }  
 **return** output;   
 }  
};

**Other Approaches:** Binary Search (how? – *solution* *at* ***leetcode.com***), Scan the first with all instead of two consecutives [better alternative i.e., easier to code than this].

**3 SUM Problem:**

You are given a vector of integers, vec. Find all triplets of indexes of the given vector *i*, *j* and *k* such that i ≠ j, i ≠ k and j ≠ kand vec[i] + vec[j] + vec[k] = 0. So, return a vector of vectors of integers containing all such possibilities of vec[*i*], vec[ *j*] and vec[*k*]. No duplicates allowed.

**class** Solution {  
**public**:  
 **vector<vector<int>>** threeSum(**vector<int>&** nums)   
 {  
 **vector<vector<int>>** output;  
 **std::sort**(nums.**begin**(), nums.**end**());

**for**(**int** i = 0; i < nums.**size**(); i++)  
 {  
  **int** remainder = -nums[i]; // Now find it. **Sorted so, 2 pointer approach is valid!**  
 **if**(remainder < 0)  **// Further pruning the algorithm.**  
 {  
 break;  
 }  
 **int** front = i + 1;  
 **int** rear = nums.**size**() - 1;  
  **while**(front < rear)  
 {  
 **int** sum = nums[front] + nums[rear];  
 **if**(sum > remainder)  
 {  
 rear--;  
 }

**else if**(sum < remainder)  
 {  
 front++;  
 }

**else**  
 {  
 **vector<int>** insert(3, 0);  
 insert[0] = nums[i];  
 insert[1] = nums[front];  
 insert[2] = nums[rear];  
 output.**push\_back**(insert);

**// remove duplicates! since values are sorted. same values only occur together.**

**while**(front < rear && insert[1] == nums[front])  
 {  
 front++;  
 }  
  **while**(front < rear && insert[2] == nums[rear])  
 {  
 rear--;  
 }  
 }  
 }  
 **while**(i < nums.**size**() - 1 && nums[i] == nums[i+1])  
 {  
 i++;  
 }  
 }  
  **return** output;  
 }  
};

What’s the catch here? Not always, hash tables or unordered maps will be the solution. It’s too complicated, unnecessary and time consuming. Simple algorithmic approach.

**Remember that the TWO POINTER approach works when you have sorted arrays or vectors for that matter.** So, sorting should seem to be good decision when visualizing the third sum. It’s a basic concept. Now it is simple to figure out the rest once you have worked this out on a whiteboard.

A Hashset implementation also exists!

The best solution however seems to be this (39ms of Runtime, almost half of mine!)  
Now, what I notice here is that while sorting, this guy has also implemented a Hashset within it so, it’s like the best solution there can exist in my opinion. **It has best of both worlds!**

**class** Solution {  
  **public**:  
 **std::vector<std::vector<int>>** threeSum**( std::vector<int>&** nums )   
 {

**if** ( **std::all\_of**( nums.**begin**(), nums.**end**(), [**&**]( **const int** i ) { **return** i == 0; } ) && nums.**size**() > 2 ) **// C++ Lambda Expressions**  
 {  
 **return** {{0,0,0}};  
 }  
 **if** ( nums.**size**() < 3 )  
 **return** {};

**int** min = 200000, max = -200000;

**// if any values in nums is greater or less than 200000A AND -200000, then set min and max to them**

**std::for\_each**( nums.**begin**(), nums.**end**(), [**&**]( **const int** i )   
 {  
 min = **std::min**( min, i );  
 max = **std::max**( max, i );  
 });

**// range is positive of min**

**const int** range = -min;

**// size of char store is distance between i and max + 1**

**const size\_t** size = **static\_cast**<**size\_t**>( max - min + 1 );  
 **std::vector**<**char**> store( size, 0 );

**// count up to 3 how many times a number is included in nums by placing in the index calculated from adding the number to range variable which is negative of min**  **for** ( **int** i = 0; i < nums.**size**(); i++ )  
 **if** ( store[nums[i] + range] < 3 )  
 store[nums[i] + range]++;  
 **int** len = 0; // counts how many unique numbers are in nums  
 **for** ( **int** i = 0, j = 0; i < size; i++ )   
 {  
 **if** ( store[i] > 0 )   
 {  
 **// for the amount of unique numbers in nums, replace the first elements in nums  
with the unique numbers in ascending order via deducing it through subtracting range from index where it is found.** nums[j++] = i - range;  
 len++;  
 }  
 }  
 **std::vector**<**int**>**&** unique = nums;  
 **auto** f = nums.**begin**();

**auto** b = **std::next**( nums.**begin**(), len - 1 );   
  **if** ( ( \*f >= 0 && \*b >= 0 ) || ( \*f <= 0 && \*b <= 0 ) )   
 {  
 **return** {};  
 }

**std::vector**<**std::vector**<**int**>> ans;

**while** ( \*b >= 0 )   
 {  
  **while** ( \*f <= 0 )   
 {  
  **int** needed = -( \*f + \*b );  
 **if** ( needed < \*f )   
 {  
 **break**;  
 }

**if** ( needed > \*b )   
 {  
 ++f;  
 **continue**;  
 }

**if** ( ( ( needed == \*f || needed == \*b ) && store[needed + range] == 1 ) || ( ( needed == \*f && needed == \*b ) && store[needed + range] == 2 ) )   
 {  
 ++f;  
 **continue**;  
 }

**if** ( store[needed + range] )   
 {  
 ans.**push\_back**( { \*f, needed, \*b } );  
 ++f;  
 }

**else**   
 {  
 ++f;  
 **continue**;  
 }  
 }  
 --b;  
 f = nums.**begin**();  
 }  
  **return** ans;  
 }  
};

Note that the syntax: Vector.**push\_back**( { \*f, needed, \*b } ); is valid. Returning an empty vector : **{};**

**3SUM CLOSEST**

This looks really daunting, but alright.

Given an integer array nums of length **n** and an integer **target**, find three integers in nums such that the sum is closest to **target**.

Return the sum of the three integers.

You may assume that each input would have exactly one solution. **// do we sort it?**

I present to you, 2 *solutions:*

Mine (12ms Runtime)

class Solution {

public:

int threeSumClosest(vector<int>& nums, int target) {

int closestSum = 0;

int closestAuxSum = INT\_MAX;

sort(nums.begin(), nums.end());

if(nums[nums.size()-3]+nums[nums.size()-2]+nums[nums.size()-1] < target){

return nums[nums.size()-3]+nums[nums.size()-2]+nums[nums.size()-1];

}

if(nums[0]+nums[1]+nums[2]>target){

return nums[0]+nums[1]+nums[2];

}

for(int i = 0; i < nums.size()-2; i++)

{

int front = i + 1;

int rear = nums.size() - 1;

while(front < rear)

{

int sum = nums[i] + nums[front] + nums[rear];

if(target == sum)

{

return sum;

}

if(abs(target-sum) < closestAuxSum)

{

closestAuxSum = abs(target-sum);

closestSum = sum;

}

if(sum < target)

{

front++;

}

else

{

rear--;

}

}

}

return closestSum;

}

};

Best Solution (5ms Runtime – even more ***pruned*** – realize how it has been done!)

class Solution {

public:

int threeSumClosest(vector<int>& nums, int target) {

int diff = INT\_MAX;

int ans;

sort(nums.begin(),nums.end());

if(nums[nums.size()-3]+nums[nums.size()-2]+nums[nums.size()-1] < target){

return nums[nums.size()-3]+nums[nums.size()-2]+nums[nums.size()-1];

}

if(nums[0]+nums[1]+nums[2]>target){

return nums[0]+nums[1]+nums[2];

}

for(int i = 0;i<=nums.size()-3;i++){

if((nums[i]+nums[nums.size()-2]+nums[nums.size()-1])<target){

int sum = nums[i]+nums[nums.size()-2]+nums[nums.size()-1];

if(diff>abs(sum-target)){

ans = sum;

diff=abs(sum-target);

}

}

if((nums[i]+nums[i+1]+nums[i+2])<target){

int sum = nums[i]+nums[i+1]+nums[i+2];

if(diff>**abs**(sum-target)){

ans = sum;

diff = abs(sum-target);

}

}

int s = i+1;

int l = nums.size()-1;

while(l-s>0){

int sum = nums[i]+nums[s]+nums[l];

if(sum==target){

return sum;

}

if(diff>abs(sum-target)){

ans= sum;

diff = abs(sum-target);

}

if(sum<target){

s++;

}

else if(sum>target){

l--;

}

}

}

return ans;

}

};

**4SUM**

**K-SUM**

**442) Return a vector of duplicate numbers in an array with numbers {1, N} and N elements (each element of range either appears once or twice)**

Here is a good solution that paves a way for good thinking skills. Use all information at your disposal. **What information?**

* Numbers 1 to N.
* Each element either appears once *or* twice *or* zero times.
* Can we do something with XOR operator? [Even if we can, I don’t see it yet]
* Can we **index map**? [Yes, but how?]

**class** Solution   
{  
 **public**:  
 **vector**<**int**> findDuplicates(**vector**<**int**>**&** nums)   
{  
 **vector**<**int**> result; **// Index mapping solution. Good question ~ !**

**for**(**int** i = 0; i < nums.**size()**; i++)  
{  
 **if**(nums[abs(nums[i])-1] < 0)  
{  
result.**push\_back**(abs(nums[i]));  
}  
nums[**abs**(nums[i])-1] \*= -1;  
}  
 **return** result;  
}  
};

**The code is so short and so beautiful! Oh well, the simple joys in this dreadful life of coding. I hate coding and engineering.**

g

Make Sudoku Windows Game.

Use Java only.

Use **JavaFX** for GUI.

~~Research Gradle.~~

Learn Linux *and* Git.