Yubo Securities
Questions from CodeBytes

Question 1 : Wiggle sort (also known as wave sort)

- non strictly wiggle case
- strict wiggle case

Question 2 : K sum

- 2 sum in $O(N)$
- 3 sum in $O(N^2)$
- 4 sum in $O(N^3)$

Question 3 : Moving median (with two data structures)

Question 4 : LRU cache (with two data structures)

Question 5 : Lowest common ancestor (tree traversal)

Question 6 : City traffic (tree traversal)

Question 7 : Parallel sum

Question 8 : Trapping water

Question 9 : Convex hull

Question 1 : Wiggle sort
Given a vector of integers $x_0, x_1, x_2, ..., x_{N-1}$, sort them such that they are :

- non-strictly wiggle i.e. $x_0 \le x_1 \ge x_2 \le x_3 \ge x_4 \le ...$ *or* $x_0 \ge x_1 \le x_2 \ge x_3 \le x_4 \ge ...$ *for odd or even N*
- strictly wiggle i.e. $x_0 < x_1 > x_2 < x_3 > x_4 < ...$ *or* $x_0 > x_1 < x_2 > x_3 < x_4 > ...$ *for odd or even N*

There are 8 cases in total. In order to facilitate discussion, lets call :

$x_0 \le x_1 \ge x_2 \le x_3 \ge x_4 \le ...$ *sine wave*
and $x_0 \ge x_1 \le x_2 \ge x_3 \le x_4 \ge ...$ *cosine wave*

Solution to non-strictly wiggle sort
It is easier, as there must be at least one solution, even if there are many duplicated numbers or all-equal numbers.

Method 1 in *O(NlogN)*
For sine wave
- sort the vector into $y_0 \ge y_1 \ge y_2 \ge y_3 \ge y_4 \ge y_5 \ge ...$
- swap every two consecutive numbers as : $(y_1 \le y_0) \ge (y_3 \le y_2) \ge (y_5 \le y_4) \ge ...$ *regardless of odd/even N*

For cosine wave
- sort the vector into $y_0 \le y_1 \le y_2 \le y_3 \le y_4 \le y_5 \le ...$
- swap every two consecutive numbers as : $(y_1 \ge y_0) \le (y_3 \ge y_2) \le (y_5 \ge y_4) \le ...$ *regardless of odd/even N*

Method 2 in *O(NlogN)*
- sort the vector into $y_0 \le y_1 \le y_2 \le y_3 \le y_4 \le y_5 \le ...$
- starting from median, pick in the following pattern :

$(((((y_{med} \le y_{med+1}) \ge y_{med-1}) \le y_{med+2}) \ge y_{med-2}) \le y_{med+3}) \ge ...$ *for sin wave regardless of odd/even N*
$(((((y_{med} \ge y_{med-1}) \le y_{med+1}) \ge y_{med-2}) \le y_{med+2}) \ge y_{med-3}) \le ...$ *for cos wave regardless of odd/even N*

Method 3 in *O(N)*
Let's solve the sine wave problem in dynamic programming way (please generalize it to cosine wave latter).

- decompose original problem into : $x_0 \le x_1 \ge x_2$ *and subproblem* $[x_2, x_3, x_{4,...}]$
- the first part can be solved by either :

logic 1 : $med(x_0,x_1,x_2) \le \max(x_0,x_1,x_2) \ge \overbrace{\min(x_0,x_1,x_2)}^{remark}$
logic 2 : *if* $(x_1 < x_0)$ *then swap* (x_0, x_1)
if $(x_{1,new} < x_2)$ *then swap* (x_1, x_2)
remark : *this number must be* $\le x_2$ *so that it can concatenate with subproblem*

- the second part can be solved by iterative / recursive call
- as a result, this is one dimensional linear scan

Here is the implementation of *logic 2*, consider even position is enough :

```
void wiggle_sort(std::vector<int>& vec)
{
      for(std::uint32_t n=1; n<vec.size(); ++n)
      {
            if (vec[n] < vec[n-1]) std::swap(vec[n],vec[n-1]);
            if (vec[n] < vec[n+1]) std::swap(vec[n],vec[n+11]);
      }
}
```

Solution to strictly wiggle sort

It is possible to have no solution (consider all equal numbers). The function should return false in that case. All solutions in non-strictly wiggle sort do not work (as they cannot guarantee a strict relation between adjacent numbers).

Method in *O(N)*

- no sorting is needed, instead put the numbers into a histogram i.e. `std::unordered_map<int,long>`
- where key is the input numbers, value is the count, so that we can find median in *O(N)*
- put the numbers in 3 subsets :
- *set M = { all numbers equal to median }*
- *set L = { all numbers lower than median }*
- *set H = { all numbers higher than median }*

- then the solutions for different cases are :

sine wave		*high*		*high*		*high*	...		*med*		*med*	
	med		*med*		*med*		...	*low*		*low*		*(low) for odd only*
cosine wave	*med*		*med*		*med*		...	*high*		*high*		*(high) for odd only*
		low		*low*		*low*	...		*med*		*med*	

A solution exists when :

$$num(medians) \le \frac{N-1}{2} \qquad \textit{for odd } N$$

$$num(medians) \le \frac{N}{2} \qquad \textit{for even } N$$

Question 2 : K sum

Given a vector :

- find all unique pairs so that pair-sum equals to a target
- find all unique triples so that triple-sum equals to a target
- find all unique quadruple so that quadruple-sum equals to a target

Solution

Research found that there is a lower bound on the computation time for this type of questions :

- 2 sum in $O(N^1)$
- 3 sum in $O(N^2)$
- 4 sum in $O(N^2)$
- K sum in $O(N^{upper(K/2)})$

For 2 sum, it is a single *1D* scan of vector. For each element x_n :

- check if $target - x_n$ exists in histogram, if yes, all the pair in output
- insert x_n into the histogram

For 3 sum, it is a single *2D* scan of vector. For each element x_n x_m :

- check if $target - x_n - x_m$ exists in histogram, if yes, all the triple in output
- insert x_n and x_m into the histogram

For 4 sum, it is one *2D* scan of vector plus one *1D* scan of historgam.

- For each element x_n x_m, insert into histogram like this :

```
std::unordered_map<int, sdt::vector<std::pair<std::uint32_t, std::uint32_t>>> hist;
hist[vec[n]+vec[m]].insert(std::make_pair(n,m));
```

- For each histogram key, check if `target-key` exists, if yes, add all combo to output (note : this is another 2D loop).

Question 3 : Moving median

Given a vector and window size, find moving median. This is a typical question which can be solved efficiently with two containers (instead of one). The first two attempts failed. Only the last one can work.

Solution 1

- A map acting as histogram, which :
  - keeps number of instances for every integer in region of interest
  - keeps a median-iterator pointing to median, hence it can return median in *O(1)*
- A list acting as sorting of in time domain, it keeps iterators to map. When we ***move*** to the next element :
  - pop one element from front of list in *O(1)*, erase it from the map in *O(1)* ***and***
  - push ***new*** element to back of list in *O(1)*, insert it to the map in *O(logN)* ***and***
  - update median-iterator by moving either one step forward or one step backward in *O(1)* ***as it is sorted***

Solution 2

- An unordered-map acting as histogram, which :
  - keeps number of instances for every integer in region of interest
  - keeps a median-iterator pointing to median, hence it can return median in *O(1)*
- A list acting as sorting of in time domain, it keeps number by value. When we ***move*** to the next element :
  - pop one element from front of list in *O(1)*, erase it from the unordered-map in *O(1)* ***and***
  - push ***new*** element to back of list in *O(1)*, insert it to the unordered-map in *O(1)* ***and***
  - update median-iterator by scanning histogram in *O(M)* ***as it is unsorted***
    where M is number of bin, and most likely N = M

Question 4 : LRU cache

Implement a LRU cache for hashmap. *I put this problem here intentionally, because question 3/4 are in fact the same concept.*

Solution

For detail solution, please refer to Lighthouse interview. The solution is very similar to moving median, it can be done by 2 containers as well. The main difference is that the ***region of interest*** becomes a ***time interval of interest***.

- A list to keeping all elements in region of interest, whenever we ***visit*** to next element :
  - pop one element from front of list (and the map as well) ***and***
  - push ***current*** element to back of list (and the map as well)
- A map to keeping key-value pairs in region of interest
  - for *O(1)* search for requested key-value pair

Question 5 : Lowest common ancestor (tree traversal)
Given a binary tree (not sorted) and 2 nodes, find lowest common ancestor.

Solution
Try to do it without using call stack (recursion). We have to instantiate a stack ourselves and perform a region growing. If the region growing is a depth first search, the first common ancestor must the lowest common ancestor. Therefore what we need to propagate are just 3 flags :

```
struct propagate_info
{
      bool is_A_found_under_this_subtree;
      bool is_B_found_under_this_subtree;
      K    key_of_1st_common_ancestor;
};
```

Question 6 : City traffic (tree traversal)
Given a non-cyclic non-directed graph and given a node on it, find the values of each subtree of the specific node, where

value(subtree) = sum(value(sub-subtree))

Question 7 : Parallel sum
Please refer to *Dynamic programming document* - Equal partition sum problem

Question 8 : Trapping water
Please refer to *Hackerrank document* - Trapping water problem

Question 9 : Convex hull
Given set of 2D points, find the convex hull.

Solution
Let's build the convex hull iteratively, starting from triangle. We maintain the convex hull as a list of points, so that when we walk through the points in sequence, we are either walking clockwisely or anticlockwisely. Clockwiseness of 3 points can be checked by :

$$\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} > 0$$

Given a convex hull *(a,b,c,d,...)* and a new point *(x,y)*, we can check whether the point lies inside/outside convex hull by :

$$\text{if} \quad \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ x & y & 1 \end{vmatrix} = \begin{vmatrix} b_x & b_y & 1 \\ c_x & c_y & 1 \\ x & y & 1 \end{vmatrix} = \begin{vmatrix} c_x & c_y & 1 \\ d_x & d_y & 1 \\ x & y & 1 \end{vmatrix} = \quad \ldots \qquad \text{then ignore the new point and goto next iteration}$$

else *insert the new point into the convex hull (probably removing existing points in current convex hull)*

The insertion of new point and deletion of existing points are in fact one single step.

- group the edges into two subsets according to the corresponding sign of cross product
- remove the edge subset with smaller size, replace it with new vertex, done.

This iterative method is also a typical example of dynamic programming.