Volant Trading *2016 Nov 09*

6+3

Find repeating number in an array

1Given a vector of shuffled consecutive integers, all numbers lie in range *[1,N]*, vector size is *N+1*, if all numbers appear once except one numbers which appear twice. Find the repeating number in *O(N)* time and *O(1)* space. The solution is to sum up all numbers, then subtract it with *N(N+1)/2*. 2The question is slightly modified that vector size becomes *N+2*, two numbers are duplicated, find them in *O(N)* time and *O(1)* space. The solution is similar, now we have both the sum and the sum of square, setting up two simultaneous equations in two unknowns *x'* and *x"*.

$$\sum_{n=1}^{N} x_n = x'+x''+N(N+1)/2 \qquad \textit{(linear)}$$
$$\sum_{n=1}^{N} x_n^2 = x'^2+x''^2+N(N+1)(N+1/2)/3 \qquad \textit{(quadratic)}$$

Another method is to set up two equations with sum and product.

$$\sum_{n=1}^{N} x_n = x'+x''+N(N+1)/2 \qquad \textit{(linear)}$$
$$\prod_{n=1}^{N} x_n = x'x''N! \qquad \textit{(quadratic)}$$

The third method is to use bitwise XOR. XOR returns *0* for same inputs, hence *A XOR A XOR B = B*. Thus we have :

$$x'\hat{}x'' = \underbrace{(1\hat{}2\hat{}3\hat{}...\hat{}(N-1)\hat{}N)}_{setA}\hat{}\underbrace{(x_1\hat{}x_2\hat{}x_3\hat{}...\hat{}x_{N-1}\hat{}x_N)}_{setB} \qquad \textit{(recall : bitwise XOR in C++ is ^)}$$

x'^x" indicates the different bits in *x'* and *x"*. We pick one different bit, says the k^{th} bit, suppose the k^{th} bit of *x'* is 1, then the k^{th} bit of *x"* is 0, we perform XOR among all elements in set A and B having the k^{th} bit equals to 1 (i.e. x" is excluded in filtered set A and set B) thus the result of XOR is x', the process can be repeated to find *x"*.3The question is modified such that size of vector becomes *N*, there exists one duplicated number and one missing number, find them in *O(N)* time and *O(1)* space. The sum-and-sum-of-square method still applies :

$$\sum_{n=1}^{N} x_n = x'-x''+N(N+1)/2 \qquad \textit{(linear)}$$
$$\sum_{n=1}^{N} x_n^2 = x'^2-x''^2+N(N+1)(N+1/2)/3 \qquad \textit{(quadratic)}$$

The sum-and-product method does also apply.

$$\sum_{n=1}^{N} x_n = x'-x''+N(N+1)/2 \qquad \textit{(linear)}$$
$$\prod_{n=1}^{N} x_n = (x'/x'')N! \qquad \textit{(quadratic)}$$

Now we generalize the problem to non-consecutive numbers, vector size is *N*, numbers are picked from *[1,M]*, such that *N≥M*, all numbers appear once, except one number appears twice. Find the number. 4Naïve solution is to do a nested for loop, which needs $O(N^2)$ in time and *O(1)* in space. 5A better solution is to do an inplace sort, then scan for duplication, it needs *O(NlogN)* in time and *O(1)* in space. 6An even better solution is to do pigeonhole sorting which needs *O(N)* in time and *O(N)* in space (geeksforgeeks call this the count array method). However, can we do that with *O(N)* in time and *O(1)* in space? Yes! It is like the pigeonhole sorting, but using the same array as count array, and instead of counting, it keeps a boolean flag indicating whether there exists duplications, the boolean flag is embedded into integer as a negative sign, i.e. negative integer denotes integers that happen before, if $x_m<0$, it means number *m* happens before, $\exists n<m$ *s.t.* $x_n = m$.

```
void check_duplication(std::vector<int>& x) // There is no constness. It works when x.size() >= M.
{
	for(unsigned long n=0; n!=x.size(); ++n)
	{
		// (step 1) convert vector value to histogram index
		int index = abs(x[n])-1;

		// (step 2) check histogram
		if (x[index] < 0) { std::cout << "\n" << x[index] << " is repeated"; continue; }

		// (step 3) update histogram
		x[index] = -x[index];
	}
}
```

Loop detection in singly linked list

Implement a reverse function for a singly linked list, recall that a singly linked list does not have a tail.

```
template<typename T> struct node
{
	node(const T& x) : label(x), next_ptr(nullptr) {} // Important !
	T label;
	node* next_ptr;
};

template<typename T> struct singly_linked_list
{
	singly_linked_list() : head_ptr(nullptr) {}

	void push_back(const T& x)
	{
		if (head_ptr == nullptr) { head_ptr = new node<T>(x); return; }
		get_last_ptr()->next_ptr = new node<T>(x);
	}
	node<T>* get_ptr(unsigned long N)
	{
		node<T>* this_ptr = head_ptr;
		for(unsigned long n=0; n!=N; ++n) { if (this_ptr != nullptr) this_ptr = this_ptr->next_ptr; }
		return this_ptr;
	}
	node<T>* get_last_ptr() // assume head_ptr != nullptr, i.e. non-empty list
	{
		node<T>* this_ptr = head_ptr;
		while(this_ptr->next_ptr != nullptr) this_ptr = this_ptr->next_ptr;
		return this_ptr;
	}
	void print() const
	{
		node<T>* this_ptr = head_ptr;
		while(this_ptr != nullptr) { std::cout << this_ptr->label; this_ptr = this_ptr->next_ptr; }
	}

	node<T>* head_ptr;
};

template<typename T> void reverse(singly_linked_list<T>& list)
{
	node<T>* prev_ptr = list.head_ptr;		if (prev_ptr == nullptr) return;
	node<T>* this_ptr = prev_ptr->next_ptr;		if (this_ptr == nullptr) return;

	head_ptr->next_ptr = nullptr;
	while(this_ptr != nullptr)
	{
		node<T>* next_ptr = this_ptr->next_ptr;
		this_ptr->next_ptr = prev_ptr;
		prev_ptr = this_ptr;
		this_ptr = next_ptr;
	}
	list.head_ptr = prev_ptr;
}
```

If there exists loop in singly linked list, write a loop-detection function with O(N) time. Method 1 : add a boolean flag in the node, reset on construction, loop-detection function then iterates through the list, setting the flag in each node, check if any node has been set before, in case it does, there is a loop. However, if we are not allowed to do so, we need method 2 : instantiate a node-to-integer mapping, count the number of visits for each node, this requires O(N) space. Can we do it with O(N) in time and O(1) in space? Here comes method 3, known as Floyd Cycle Detection algorithm. There are two pointers, a fast one which increments by 2, and a slow one which increments by 1, both start iterating from the beginning of list, there exists loop if they meet. For testing, I add this function :

```
template<typename T> struct singly_linked_list
{
	void create_loop(unsigned long n)
	{
		node<T>* p0 = get_last_ptr();
		node<T>* p1 = get_ptr(n);
		p0->next_ptr = p1;
	}
};
```

Suppose A is the starting point, B is where the loop starts, while C is where fast and slow pointers meet, length of loop is L. How can we measure L and length of AB? L can be found by adding a counter in the slow pointer, which increments as slow pointer iterates, counting starts when two pointers meet for the first time, counter stops when two pointers meet again. How about length of AB? There are 3 methods, for each method, we need two pointers with the same speed.

Method 1 : Exhaustive search $O(N^2)$
Pinning pointer 1 at *C-1*, while pointer 2 starts iterating from *C*, through one complete loop, and back to *C* again, if two pointers do meet, then the loop starting point *B* must lies before *C-1*. Thus we pin pointer 1 at *C-2*, repeat the procedures until they never meet, the location of pointer 1 is then the node prior to the loop start point.

Method 2 : Bisection *O(NlogN)*
The method is similar, however, pointer 1 doesn't scan from *C-1*, *C-2*, etc, bisection is used instead.

Method 3 : Fastest approach *O(N)*
Place pointer 1 at *A* and pointer 2 at *C*, both of them iterate with same speed, they will meet again at *B*. Here is the proof.

$$
\begin{aligned}
2(AB+BC) &= AB+L+BC \\
2(AB+BC) &= AB+2BC+CB \qquad \textit{note : } BC \neq CB \textit{ , they are different sides of the loop} \\
AB &= CB
\end{aligned}
$$

From youtuber JOMA Tech
This is called Floyd's Tortoise and Hare algorithm. It can be used to detect duplicated number in array question too.

The 3 steps in solving duplicated number questions :

- sort number and simple scan — O(NlogN) in time
- use hash map — O(N) in time but O(N) in memory
- use Floyd Tortoise and Hare — O(N) in time O(1) in memory (with value as pointer to next node)