Maoting Ren
mren

1-1. Virtual Initialization

It is obviously that we should use some technologies to mark if the position we are accessing has been assigned our own value. Assume we have a N-length array named "**data**", then we can use two extra arrays with the same length and a counter to solve this problem. Let's define them like below:

```
unsigned int index[N];
unsigned int pos[N];
int number = 0;
```

When we modify the ith element, we should do the following things:

```
data[i] = value;
index[i] = number;
pos[index[i]] = i;
number++;
```

When we access to the **i**th element, if **index[i] < number** and **pos[index[i]] = i**, it means we have modified this value, then we can return its value **data[i]**, else we should return the default value ***v***. Now I will show you how it works:

1. At beginning, we access to the **i**th element (and we know that the array is full of random data now), we search **index[i]** first, since **index** and **pos** contain unsigned value and **number = 0**, so **index[i] < number** will never be true.

2. After we have modified some elements of **data** (assume that **number = k** now, which means we have modified **k** elements), then we access to the **i**th element, which have not been modified yet. We see **index[i]** first, we know the value of **index[i]** is random, but if unfortunately this random value is less than **k**, then we have to see the value of **pos[index[i]]**, and we know it can't be **i** because it have been assigned another value before, so in this condition we return a default value ***v***.

3. And now we access to the ith element which is an exist value, it will satisfy the condition **index[i] < number** and **pos[index[i]] = i**, so we return its value **data[i]**.

1-2. Enumerating Subsets by Incrementing a Binary Counter.

We can use the following routine to enumerate subsets

```
    void increment(int data[], int n)
    {
1       int i = 0;
2       while(i < n && data[i] == 1)
3       {
4           data[i] = 0;
5           i = i + 1;
6       }
7       if(i < n)
8           data[i] = 1;
    }
```

1. Firstly, I will use accounting method to explain this amortized analysis.

Let's charge an amortized cost of 2 units to set a bit from 0 to 1, thus at anytime each bit that equal 1 have 1 unit of credit on it. When we reset the bits from 1 to 0, the credit on it will pay for the cost, which mean the cost of reset is 0. And we know each time there will be only one bit set to 1, so the total amortized cost is O(n).

2. Secondly, I will use potential method to show how to perform this amortized analysis
Let's define the potential of counter after invocate **i**th **increment** to be $\mathbf{b_i}$, the number of 1s in the counter after the **i**th operation. Assume **i**th **increment** operation reset $\mathbf{t_i}$ bits, so the actual cost will be $\mathbf{t_i} + 1$, since it will also set one bit to 1.
If $\mathbf{b_i} = 0$, it means that this time it reset all **n** bits, so $\mathbf{b_{i-1}} = \mathbf{t_i} = \mathbf{n}$
if $\mathbf{b_i} > 0$, $\mathbf{b_i} = \mathbf{b_{i-1}} - \mathbf{t_i} + 1$, and at any time $\mathbf{b_i} <= \mathbf{b_{i-1}} - \mathbf{t_i} + 1$
and each operation's difference of potential is: $p(\mathbf{b_i}) - p(\mathbf{b_{i-1}}) <= \mathbf{b_{i-1}} - \mathbf{t_i} + 1 - \mathbf{b_{i-1}} = 1 - \mathbf{t_i}$
As we have known, the actual cost is $\mathbf{t_i} + 1$ each operation, so amortized cost is:
$c' = c + p(\mathbf{b_i}) - p(\mathbf{b_{i-1}}) <= (\mathbf{t_i} + 1) + (1 - \mathbf{t_i}) = 2$.
Therefore we can know the total amortized cost is O(n).

1-3. Enumerating Permutations.
To find the next permutation we should do the following steps:
1. find the first ascend order pair(A, B) which arr[A] < arr[B] from right to left.
2. then find the first element that big than A from B to the end, suppose it is in position C.
3. swap value of A and C, swap(arr[A], arr[C])
4. reverse the element in [B, end] interval.

```
nextPermutation(int arr, int n)
{
    int i=n-1, j, k;
    for(i = n-1; i> 0; arr[i-1] > arr[i]);
    if(i == 0)      return;
    for(j=i+1; j < n; j++);
    swap(arr[i-1], arr[i]);
    reverse(arr[i...n-1]);
}
```

This algorithm is O(1) amortized cost over all permutation.

1-4. In-Place Matrix Transposition.
To achieve the goal of In-Place Matrix Transposition, we should move the element to its correct position directly without extra memory. And this mean we put the element at position A to position B, and then put the element at position B to position C, do these operations until the next position is A, just like you see, we encounter a circle. But some elements may have been moved to their correct positions already, so we should not move them again. To find if this element have been moved before, we can check its circle, if any element position is less than this position, it mean this element has been dealt already, so we can ignore it.
To determine where should one element move to, we can use this formula:

(index%n)*m + (index/n)

For example, we have a matrix like this:

0 1 2 3 4
5 6 7 8 9

They should like below after transposition:

0 5
1 6
2 7
3 8
4 9

If we store them in an array, they are in the below sequence:

0 1 2 3 4 5 6 7 8 9
0 5 1 6 2 7 3 8 4 9

1. The first circle is 0 → 0, and we don't move this element.
2. The second circle is 1 → 2 → 4 → 8 → 7 → 5 → 1, so we move all elements in this circle to their correct position.
3. The third circle is 2 → 4 → 8 → 7 → 5 → 1 → 2, as we can see, in this circle there is one element's position 1 is less than 2, so we know this element have been moved before, so ignore it.
After we do this operation to all elements, we will put all elements in their right position.

Although this algorithm don't use extra memory, it's time complexity is O(N*N), so in order to make it be O(N*log(N)), we should use “domination radius” algorithm. It means when traversal the circle, we should simultaneous walk two directions, but one direction. So we use two pointers, and one walk to left, another walk to right. The pointer walk to right will following its next position, the pointer walk to left will walk following its previous position. If in someplace the two pointer meet, then this element hasn't been dealt before, else if they find some position less than this element's position, it indicate this element have been dealt before, we will ignore this position.