Problem 0

1. All by myself

2. 許睿尹、張程凱

3. 張程凱

4. All by myself

5. 李沛宸、許睿尹、張程凱

Problem 1

1. 7

2.

```
Global Variable cnt = 0;
MergeSort(arr, left, right)
	if(left >= right)
		return;
	mid = (left + right)/2
	MergeSort(arr, left, mid)
	MergeSort(arr, mid+1, right)
	Merge(arr, left, mid, right)
	return
Merge(arr, left, mid, right)
	len1 = mid – left + 1
	len2 = right – mid
	leftArr[len1] = arr[left]~arr[mid]
	rightArr[len2] = arr[mid+1]~arr[right]
	i, j, k = 0
	while(i < len1 && j < len2)
		if(leftArr[i] <= rightArr[j])
			arr[k] = leftarr[i]
			i++
		else
			arr[k] = rightarr[j]
			cnt += len1 – i
			j++
		k++
	While(i < len1)
		arr[k] = leftarr[i]
		i++
		k++
	While(j < len2)
```

```
            arr[k] = rightarr[j]
            j++
            k++
```

The answer would be (n*(n-1)/2) - cnt

I only use Merge Sort in this question, and the time complexity of Merge Sort is O(nlogn), so the total time complexity is O(nlogn).

3(a). 4, 5, 2, 3

3(b).

For the worst case, each time we run the outer while loop we can only remove one pair, and there are at most n/2 pairs that need to be removed, so we need to run the outer while loop for at most n/2 times. Since the time complexity of Ordered(P) is O(n), and the time complexity for the inner while loop is O(n), the total time complexity is O(n^2+n^2) = O(n^2).

4.

Every time when P[i].difficulty > P[i+1].difficulty is true, we will remove 2 problems, and we know that at least one of them is misplaced. And since there are k misplaced problems, we will remove at most 2k problems when using Remove-out-of-order-pairs, which is the HOWMAGIC! property.

5.

```
if(!Ordered(P))
    i = 0
    while(i+1 < len(P))
        if(P[i].difficulty > P[i+1].difficulty)
            remove(P[i])
            remove(P[i])
            if(i > 0)
                i -= 1
        else
            i += 1
```

We only traverse the array once and the time complexity of Ordered(P) is n, so the total time complexity is O(n).

6.

(1) We use the function in problem 5 to remove the out-of-order pairs and store them in a new array. The time complexity of this part is O(n).

(2) We use Merge Sort to sort the new array. The time complexity of this part is O(klogk).

(3) Combine the two arrays in order and we will get the answer. The time complexity of this part is O(n).

So the total time complexity is O(n + klogk).

Problem 2

1.
Hash the one-dimensional 1*h pattern into a one-dimension 1*h array.
Use a three-dimensional array value[i][j]k] to store the hash value of map[i][j][k](i stands for the index of map, j stands for the column, k stands for row).
Use Robin Carp to find all maps that consist of the desired pattern.

2.
The time complexity of hashing the one-dimensional pattern into a one-dimensional array is O(h).
The time complexity of creating the three-dimensional hash value array is O(K*M*N).
The time complexity of using Robin Carp to find the answer is O(K*M*N).
The time complexity of printing the answer is O(K*M*N).
So the total time complexity for the worst case is O(K*M*N+h).

3.
Hash the two-dimensional g*h pattern into a one-dimensional 1*h array (hash the column).
Use a three-dimensional array value[i][j][k] to store the value of hashing map[i][j from n to n+g-1][k] (i stands for the index of map, j stands for the column, k stands for row, constraint: n >= 1, n+g-1 <= N).
Use Robin Carp to find all maps that consist of the desired pattern.

4.
The time complexity of hashing the two-dimensional pattern into a one-dimensional array is O(g*h).
The time complexity of creating the three-dimensional hash value array is O(K*M*N).
The time complexity of using Robin Carp to find the answer is O(K*M*N).
The time complexity of printing the answer is O(K*M*N).
So the total time complexity for the worst case is O(K*M*N+g*h).

5.
Hash the two-dimensional N*M target map into a one-dimensional 1*M array (hash the column).
Hash those two-dimensional N*M maps into a one-dimensional 1*M array and store them in a two-dimensional K*M array Value[K][M].
Use Robin Carp to find the target.

6.
The time complexity of hashing the two-dimensional target map into a one-dimensional array is O(M*N).
The time complexity of hashing those two-dimensional maps and storing the values is O(K*M*N).
The time complexity of using Robin Carp to find the target is O(K*M).
So the total time complexity for the worst case is O(K*M*N).

Problem 3

1.

Origin	One	Ten	Hundred
4	73	4	4
73	4	504	9
184	184	9	47
76	504	47	73
299	76	73	76
47	47	76	184
9	299	184	299
504	9	299	504

2.

```
For i from 0 to 6
	counting sort(arr, 10^i)
```

3.

```
Standard = arr[1]
Push arr[1]
answer_index = 1
For i from 2 to N
	if arr[i] <= standard
		if arr[i] <= stack -> top
			push arr[i]
		else
			while(stack -> top < arr[i])
				tmp = pop stack
				answer[answer_index] = tmp
				answer_index+=1
			push arr[i]
	else
```

```
        standard = arr[i]
        while(stack -> top != NULL)
            tmp = pop stack
            answer[answer_index] = tmp
            answer_index+=1
        push arr[i]
while(stack -> top != NULL)
    tmp = pop stack
    answer[answer_index] = tmp
    answer_index+=1
```

In this function, every element would be stored in the stack once, and we use the while loop to pop out those elements, so the while loops would run N times in total. And since the For loop runs N times, the total time complexity is O(N).
Since there are only a few variables and a stack in the function, the space complexity is O(N).

4.

```
k = 2
For i from 1 to k
    Flag[i] = 0
For i from 1 to N
    For j from 1 to k
        if Flag[j] == 0
            PUSH(arr[i], stack[j])
            break
        else if arr[i] <= stack[j] -> top
            PUSH(arr[i], stack[j])
            break
For i from 1 to N
    answer[i] = min(stack[1~k] -> top)
    Pop min
```

The answer array would be the answer.

The time complexity for the first For loop is O(k), for the double For loop is O(N*k), and for converting stacks to answer array is O(N*k).
So the total time complexity is O(N*k) = O(N).
Since every element needs to be stored in one of the stacks, the space complexity for a stack is O(N), and there are k stacks and several variables, so the total space complexity is O(k*N) = O(N).

5.
k = K, others remain the same as problem 4.