cpt_s 350

Homework 2 prime

11641327 Yu-Chieh Wang

2020/2/11

Three constraints of “one-pass”, “in-place”, and “linear time”.

1. Two pointers algorithm

   (a) The constraints we put on the two pointers.

   In the algorithm, there are only two pointers (in-place) i and j runs on an array. Both pointers i and j only move one direction (one-pass). Initially, the pointer i is at the far left of the array, and it can only move to the right. Also, the pointer j is at the far right of the array, and it can only move to the left. When two pointers meet at the same position, they stop moving because the array has finished sorting (linear time).

   (b) The process of the algorithm.

   There is an array called A with two kinds of value ‘b’ and ‘p’ which mean babies with brown and purple hair (Each baby has only one hair color, so each cell in the array only has one kind of value). Also, Its relative position is 0 to n from left to right. In addition, there are two pointers i and j which are going to run on the arrow. At the beginning, the pointer i equals to 0, and the pointer j equals to the length to the arrow, which means pointer i is at the far left of the array, and the pointer j is at the far right of the arrow. For example:

(relative address)	0	1	2	3	4	5	…	n-4	n-3	n-2	n-1	n
(value)	b	b	p	p	b	p	…	p	b	b	p	p
(pointer)	↑ i											↑ j

Next, we want to sort the array so that all “b”s are on the left side and all “p”s are on the right side. Therefore, the pointer i should check the values from left to right are all “b”s until it meets the other pointer j, and the pointer j should check the values from right to left are all “p”s until it meets the pointer i.

(relative address)	0	1	2	3	4	5	…	n-4	n-3	n-2	n-1	n
(value)	b	b	b	b	b	b	…	b	p	p	p	p
(pointer)								↑ j	↑ i			

Then, The sorting process is as follow:

Step 1: we only run the pointer i to check if A[i]==“b”. if so, we move i to the right position (i+=1). Otherwise, i stops moving, and keep arrows “p”.

Step 2: we only run the pointer j to check if A[j]==“p”. If so, we move j to the left position (i-=1). Otherwise, j stops moving, and keep arrows “b”.

Step 3: we check if i<j first. If so, we finish sorting the array. Otherwise, we swap the values of A[i] and A[j], and then move the pointer i to the right (i+=1) and pointer j to the left (j-=1). Finally, we go back to the step 1 to keep check the array.

(c) The psuedo-code of the algorithm.

```
i=0
j=len(A)
while i<j {
        if A[i]==‘b’ {
                i+=1 }
        else if A[j]==‘p’ {
                j-=1 }
        else {
                swap(A[i], A[j])
                i+=1
                j-=1
        }
}
```

2. Three pointers algorithm

(a) The constraints we put on the three pointers.

In the algorithm, there are totally three pointers (in-place) i, j, and k runs on an array. All pointers i, j, and k only move one direction (one-pass). Initially, pointer i is at the far left of the array, and it can only move to the right. Also, pointer j is at the far right of the array, and it can only move to the left. In addition, the pointer k is in the same position as the pointer i and can only move to the right when we need it, or it will keep equal to -1. When both i and j cannot move anymore and k check the values from position i to j are all “p”s, we finish sorting (linear time).

(b) The process of the algorithm.

There is an array called A with three kinds of value ‘b’, ‘p’, and ‘B’ which mean babies with brown, purple, and black hair (Each baby has only one hair color, so each cell in the array only has one kind of value). Also, Its relative position is 0 to n from left to right. In addition, there are three pointers i, j and k which are going to run on the arrow.

At the beginning, the pointer i equals to 0, and the pointer j equals to the length to the arrow, which means pointer i is at the far left of the array, and the pointer j is at the far right of the arrow. For example:

(relative address)	0	1	2	3	4	5	…	n-4	n-3	n-2	n-1	n
(value)	b	p	B	B	p	b	…	p	b	b	b	B
(pointer)	↑ *i*											↑ *j*

Next, we want to sort the array so that all “b”s are on the left side, all “B”s are on the right side, and all “p”s are in the middle.

Therefore, the pointer i should check the values from left to right are all “b”s until it meets “p”, and the pointer j should check the values from right to left are all “p”s until it meets “p”. Also, the pointer k will check if there is any other value (“b” or

"B") except "p" from A[i] to A[j].

(relative address)	0	1	2	3	4	5	…	n-4	n-3	n-2	n-1	n
(value)	b	b	b	b	b	p	…	p	p	B	B	B
(pointer)						↑ i, k			↑ j			

Then, The sorting process is as follow:

Step 1: we check if k still equals to -1. If so, we have other value except "p" from A[i] to A[j], so we need to keep sort the array. Otherwise, we finish sorting the array.

Step 2: we only run the pointer i to check if A[i]=="b". If so, we move i to the right position (i+=1). Otherwise, i stops moving, and keep arrows other value.

Step 3: we only run the pointer j to check if A[j]=="p". If so, we move j to the left position (i-=1). Otherwise, j stops moving, and keep arrows other value.

Step 4: when both values of A[i] and A[j] equal to "p", we run the pointer k from position i to j to check if we have other value from A[i] to A[j]. If so, we swap the values of A[i] and A[k], and then reset the k=-1. Finally, we go back to the step 1 to keep check the array.

(c) The psuedo-code of the algorithm.

```
i=0
j=len(A)
k=-1
while i<j {
        if k!=-1{
                break }
        if A[i]==‘b’ {
                i+=1 }
        else if A[j]==‘B’ {
                j-=1 }
        else if A[i]==‘p’ and A[j]==‘p’ {
                for (k=i; k<j; k++) {
                        if A[k]!=‘p’{
```

```
                        swap(A[i], A[k])
                        k=-1
                        break
                }
        }
    }
    else {
        swap(A[i], A[j])
    }
}
```