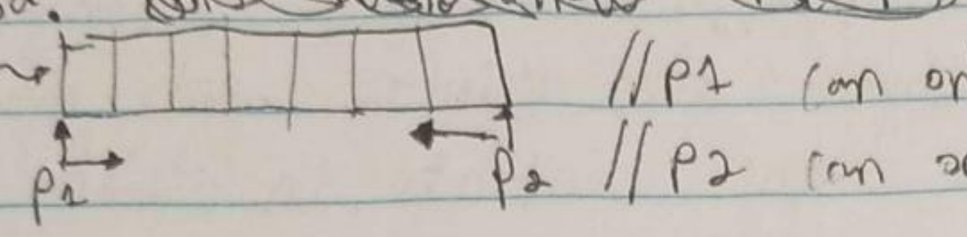
Brenden Nelson
ID #116 35716

350 Hw 2

(1) a) The two pointers are constrained in that they must be "one-pass" and "in-place" and "linear time".

"One-pass" - The pointers MUST move only in one direction. If a pointer moves right, it can only move right & vice versa.



"in-place" - The pointers must also maintain a constant amount of auxillary memory. This means, I can't create any extra memory when using the two pointers. I can't have a 3rd pointer or temp variable to help with swapping elements. This constraint means I'll have to directly swap pointer contents using swap(i,j) to re-organize the array.

"linear time" - the algorithm must run on linear time, or must be upperbound by $O(n)$. This means the algorithm time complexity must be linear. In other words, $\exists c > 0$ such that $T_w(n) \leq c \cdot n$ for all n.

b) The constraint I would consider first is that the algorithm must run on linear time. I consider this one first because that is the driving force behind the whole algorithm. It must run in linear time before I can start thinking about pointer manipulation. Next, I would consider that they must be "in-place". I shouldn't start designing the algorithm or messing with pointers until I can figure out how to maintain constant memory. I can fix the memory size by swapping pointer contents directly, rather than declaring a temp variable.

Lastly, I consider that the pointers must be "one-pass" pointers. I can accomplish this by setting p2 to the starting element and p2 to the ending element. I can move p1 right and p2 left, while performing comparisons. Until they meet on the same element in order to organize the array. This would ensure that the pointers only move in one direction and sweeps the contents in linear time without using any additional memory.

c)
```
*p1, *p2;                    //declaring pointers
p2 = A [last];               // Pointer to last element
p1 = A [first];              //Pointer to first element
   while ( p1 != p2)         // while not pointing to same element
      {if (p2 == "brown")    // p2 might need swapped
         {if (p1 == "purple") // p1 needs swapped w/ p2
            {swap (p1, p2)}  //swapping elements
                             // only 1 element in wrong place
      { p1++                 // advancing to right
                             //only 1 element in wrong place
        p2--                 // advancing to left
      {
```

The above pseudo-code will declare a pointer to the first and last element. Then it will compare the elements to see if they need swapped. If one does, then I move the other pointer until they both need to be swapped. Once both pointers point to elements that should be swapped, I can swap the elements, not the pointers, to correctly organize the array with all brown haired babies followed by all purple haired babies.

(2) This algorithm would be implemented using 3 pointers. I would point P1 at the first element, P2 ~~to~~ P3 at the last element. I would move P1 to the right until it reached a child without brown hair. I would move P2 to the left until it pointed to an element with brown hair. I would then swap those elements and repeat until P1 & P1 point to the same element. Once P1 == P2, I would move P1 right until it reached an element without purple hair and move P2 left until it reached a purple haired element. I would then swap those elements and repeat until P1 == P3. Once P1 == P3, the array should be sorted in the order of brown, purple, black. This algorithm is "one pass" "in place" and linear time complexity and only involves using 3 pointers.

```
*P1, *P2, *P3                    // pointer decleration
P1 = A[first]                    // P1 point at 1st element
P2 = P3 = A[last]                // P2,P3 point at last element
While (P1 != P2)                 // while not sorted //1st part
    {if (P2 == "brown")          // P2 needs swapped
        if (P1 == "purple")      // P1 ready to swap
            Swap (P1,P2)         // swap elements
        P1++                     // moving right
        P2--                     // moving left
    }  // as of now, all brown at front, then purple/black unordered.
While (P1 != P3)                 // while not sorted completely
    { if (P3 == "purple")        // P3 to be swapped
        if (P1 == "black")       // P1 to be swapped
            Swap (P1,P3)         // swapping elements.
        P1++                     // move right
        P3--                     // move left
} // array is fully sorted and P1 == P3.
```