Assignment 3

June 8, 2020 12:33 AM

Problem 1

- Intuition :

Binary Search can be used to solve this problem. If A[mid] < mid, we narrow our search to the right half. If A[mid] > mid, we narrow our search to the left half. If A[mid] == mid, the index is found. Otherwise, we return -1 to indicate that there is no such index.

e.g. When there is no such index

⓪	①	②	③	④	⑤	⑥	⑦	⑧	⑨
[1,	3,	5,	7,	9,	11,	12,	20	23,	90]
l				m					r

l m r

l r
m

r l → there is no such index

e.g. When such index exists

⓪	①	②	③	④	⑤	⑥	⑦	⑧	⑨
[-3,	-1,	0,	3,	5,	16,	29,	40,	50,	66]
l				m					r

l m r

lm r

l r
m A[m] == m ; target found.

- Pseudo Code

```
FIND_INDEX ( A[], int index){
    // corner cases
```

```
if(A ==null || A.length == 0 ) return -1;

int left = 0;
int right = A.length -1;
int mid;

while (left ≤ right){
    mid = left + (right - left) /2;
    if (A[mid] == mid ){
      return mid;
    }

    if (A[mid] < mid){
       left = mid +1;
    } else{
       right = mid -1;
    }

  }
  return -1;
}
```

- Time Complexity Analysis

This is a classical binary search approach; every time, the problem size is reduced by half, and the additional operations for each subproblem takes O(1). (just comparisons)

the recurrence relation is: $T(n) = T(n/2) + 1$

$\Rightarrow O(n) = \log N$

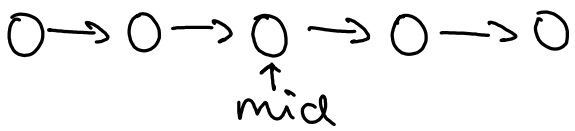
Problem 2

- Intuition:

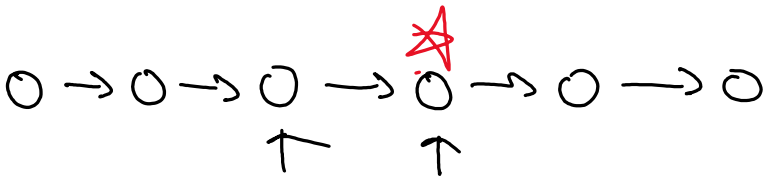
This problem is equivalent to "randomly sorting" a linked list. we can first use merge sort to sort a linked list, and modify the algorithm such that instead of merging by

value (i.e. always taking the smaller value first), we merge in a random fashion

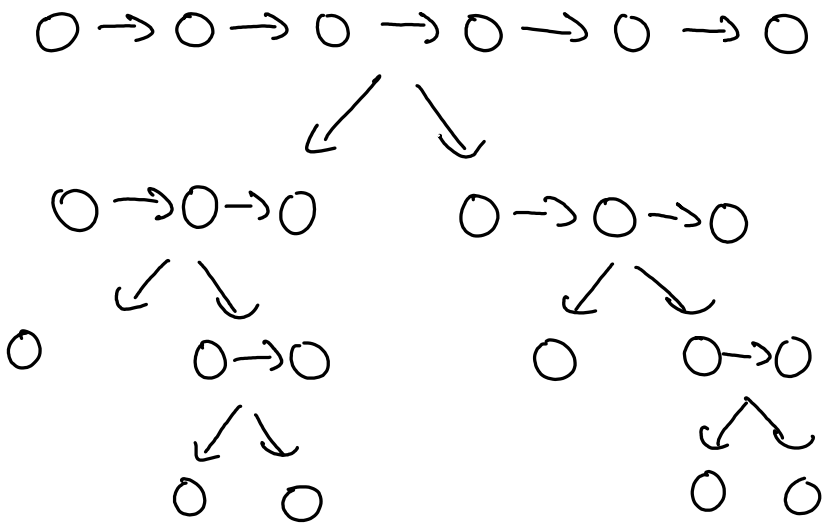
In order to perform mergesort on a linked list, need to find the middle node of the list. It is easy when there is an odd number of nodes:



But what if there is an even number of nodes?



the middle node can be either of these two; in this problem, we select the rightmost middle node.



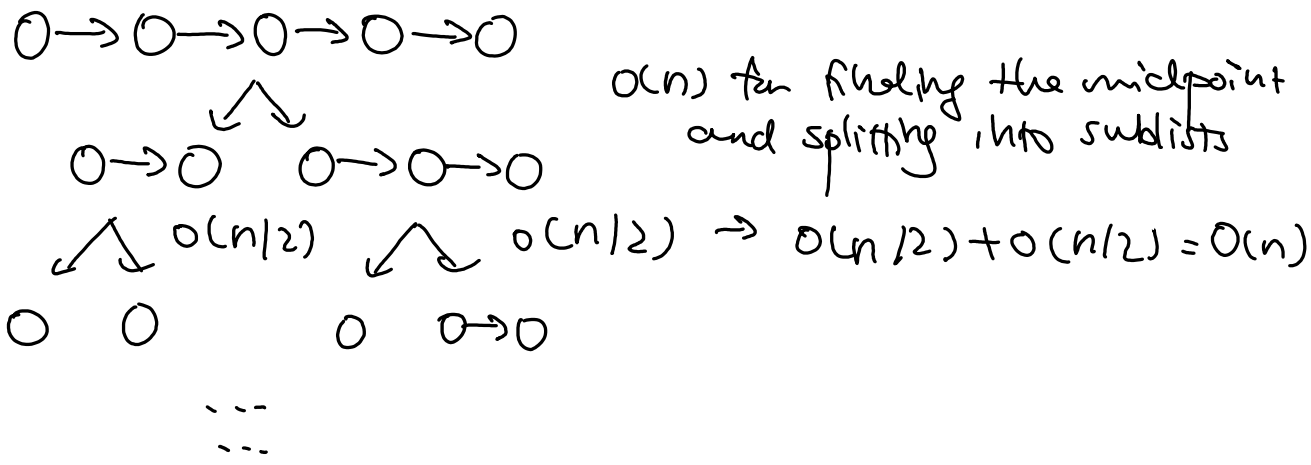
Divide until there's only one node left in each list. Then, we begin the merging process. Instead of taking the node with the smaller value first, we can use a random number generator to decide whether the next node to be appended will come from the first list or the second list. And for lists with an odd number of nodes, depending on which of the lists is longer, we append the left-out node to the end of the resulting linked list and return the list

- Pseudocode

```
SORT_LIST(ListNode head){
  // base cases
  if(head == null || head.next == null) return head;

  // Find midpoint
  ListNode slow = head;
  ListNode fast = head;
  // store the new left end
  ListNode temp = head;

  while (fast != null && fast.next != null){
     temp = slow;
     slow = slow.next;
     fast = fast.next.next;
  }
  temp.next = null;

  ListNode left = SORT_LIST(head);
  ListNode right = SORT_LIST(slow);

  return MERGE_RANDOM(left, right);

}


MERGE_RANDOM(ListNode left, ListNode right){
   ListNode res = new ListNode(0);
   ListNode cur = res;

   while(left != null && right != null){   // rand will be
      int rand = (int)(Math.random*2);      either 0 or 1
      if(rand == 0){
         cur.next = left.next;
         left = left.next;
      } else {
         cur.next = right.next;
```

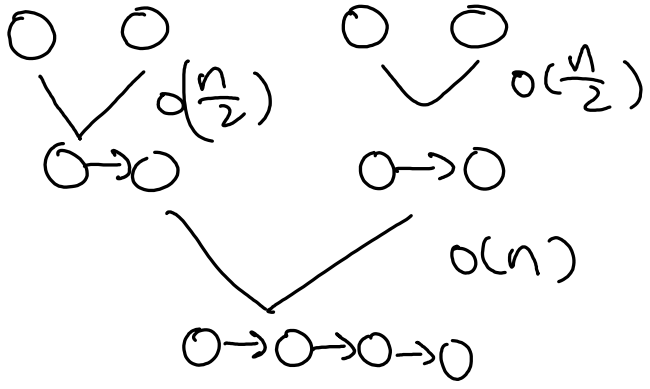
```
            right = right.next;
        }
        cur = cur.next;
    }

    if (left != null) cur.next = left;
    if (right != null) cur.next = right;
    return res.next;
}
```

- Time complexity analysis



There is a total of log n levels, work done on each level takes O(n), ∴ the time complexity for dividing the nodes: O(nlogn)

For merging, need to use two pointers to iterate through n nodes.



A total of logN levels; each merging takes O(n); the merging process also takes O(nlogN) Overall time complexity

O(nlogn)

- Space Complexity analysis:

At any time, the deepest level possible of the call stack is O(logN), in addition to this, we used constant space for creating new ListNodes, ∴ the space complexity is O(log N).

Problem 3

- Intuition:

We can use two pointers, i and j, initially pointing to the 1st element of each array (A and B) respectively when the number that i points to is smaller, we advance i (and k--), and when the number that j points to is smaller, we advance j (and k--). After k moves, we will be at the k^{th} smallest element. This approach is not ideal because its time complexity is O(n+m); the worst case happens when k is large.

Optimization: Since the arrays are sorted, can make use of binary search.

Suppose

A = [1, 3, 5, 7, 9]

B = [2, 4, 6, 8, 10, 12] and k = 6.

A and B merged:

[1, 2, 3, 4, 5, (6), 7, 8, 9, 10, 12]

should return 6

(truncating down)

Check the middle element of each array first

check the middle element in each array first:

A = [1, 3, 5, 7, 9] (① ② ③ ④ ⑤)

B = [2, 4, 6, 8, 10, 12] (① ② ③ ④ ⑤ ⑥)

the sum of their indexes is 3+2 = 5, which is smaller than 6. Now, we have to compare A[mid] with B[mid]; since the arrays are sorted and A[mid] < B[mid] it means that all other numbers to the left of A[mid], in A are smaller than B[mid] as well, and since we still haven't reached the kth element when we reached B[mid], it means that all values to the left of A[mid], in A can be discarded. And k = 3.

A = [~~1, 3, 5~~, 7, 9] (④ ⑤)

B = [2, 4, 6, 8, 10, 12] (① ② ③ ④ ⑤ ⑥)

Sum of indices = 7
7 > 3; since we already passed k, and A[mid] > B[mid], all numbers to the right of A[mid] are even larger, we can discard all numbers to the right of A[mid].

A = [7, 9] (④ ⑤)

B = [2, 4, 6, 8, 10, 12] (① ② ③ ④ ⑤ ⑥)

Sum of indices: 7
7 > 3 and A[mid] > B[mid]; now we stop considering array A

B = [2, 4, 6, 8, 10, 12] (① ② ③ ④ ⑤ ⑥)

Return B[k], which is 6.

- Pseudocode

```
FIND_KTH_SMALLEST (A[], B[], int Astart, int Aend,
                                int Bstart, int Bend,
                                int k) {
```

```
int Alen = Aend - Astart;
int Blen = Bend - Bstart;

// base cases
if (k <= 0 && k > (Alen + Blen)){
    error: out of bound; invalid k
}

if (Alen == 0) return B[Bstart + k];
if (Blen == 0) return A[Astart + k];

int Amidind = Alen / 2;
int Bmidind = Blen / 2;
int Amid = A[Astart + Amidind];
int Bmid = B[Bstart + Bmidind];

if ((Amidind + Bmidind) < k){
    return (Bmid < Amid) ?
     FIND_KTH_SMALLEST(A[], B[],
                       Astart, Aend,
                       Bstart + Bmidind + 1, Bend,
                       k - (Bmidind + 1)) :

     FIND_KTH_SMALLEST(A[], B[]
                       Astart + Amidind + 1, Aend,
                       Bstart, Bend,
                       k - (Amidind + 1));

}else{
    return (Bmid < Amid) ?
    FIND_KTH_SMALLEST(A[], B[],
                      Astart, Astart + Amidind,
                      Bstart, Bend, k) :
    FIND_KTH_SMALLEST(A[], B[],
                      Astart, Aend,
                      Bstart, Bstart + Bmidind, k);
```

}

- Time Complexity Analysis:

Every time half an array is discarded, the problem size is reduced by half, and the rest of the operation takes constant time; this happens for both arrays, ∴ the overall time complexity is $O(\log n + \log m)$.

Problem 4

- Intuition:

Usually, we run binary search to check whether a number is present in a sorted integer array. However, in order to do that, we need to know the "right boundary" of the binary search. Therefore, the first step in solving this problem would be finding the portion of the array in which we should be running binary search, and then we can run the classical binary search to find the target index, or return -1 if the target integer is not in the array.

e.g. A = [1, 3, 5, 7, 9, 11, 13, 15, ∞, ∞, ∞, ∞, ∞ ...]

First, we set the left boundary to be 0 and the right boundary to be 1; at any time, if A[right] < target, we know that the target is not in our current range now, so we change the position of right boundary from i to $2i$; if A[right] ≥ target or we encounter a ∞, we have found the boundary.

Suppose target = 15:

A = [1, 3, 5, 7, 9, 11, 13, 15, ∞, ∞, ∞, ∞, ∞, ∞, ∞ ...]

l=0 r=1

l=1 r=2

l=2 r=4

l=4 r=8 ✱ ∞! we have found the

portion to perform binary search:

$A = [1, 3, 5, 7, 9, 11, 13, 15, \infty]$

Then, we run classical binary search on A.

- Pseudocode (assuming that when the number is not found, we return -1, and we represent "∞" with "null")

```
FIND_RANGE (A[], int target){
    int left = 0;
    int right = 1;
    while (A[right] < target){
        left = right;
        right = 2 * right;
    }
    BINARY_SEARCH (A[], 0, right, target);
}

BINARY_SEARCH(A[], int left, int right, int target){
    int l = left;
    int r = right - 1;
    int mid;

    while (l ≤ r){
        mid = left + (right - left) / 2;
        if (A[mid] != null && A[mid] == target ){
            return mid;
        }else if ( A[mid] != null && A[mid] > target){
            right = mid - 1;
        }else {
            left = mid + 1;
        }
    }
    return -1;
}
```

)

- Time complexity Analysis

This algorithm consists of two parts: finding the range and performing binary search. When finding range, every time A[right] < target, we multiply the right index by 2, ∴ in the worst case, it will take log N steps to find the correct bounding. (i.e. in the worst case, after log N jumps, n being the number of integers currently in array, A[right] will either be a number greater than the target, or "∞". So we know that we have found the boundary. Classical binary search takes O(log n), ∴ the overall time complexity is O(log N).

When n = 8 and target = 27:

[1, 3, 7, 9, 10, 18, 27, 30, ∞, ∞, ∞, ∞, ...]

Init: l r

1st jump: l r

2nd jump: l r

3rd jump: l r

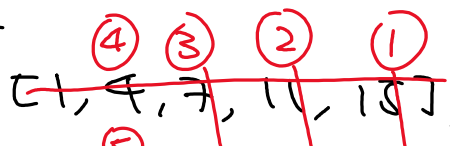
Found range for binary search in $\log_2 8$ jumps.

Problem 5

- Intuition:

We can use elimination to narrow down the search range until we find the target or conclude that the target is not in the 2D matrix (return [-1, -1]).

e.g. [



① Suppose we start from this index; A[index] > target, therefore we can eliminate all values in the same

[2, 5, 8, 12, 19],
[3, 6, 9, 16, 22],
[10, 13, 14, 17, 24],
[18, 21, 23, 26, 30]
]

target = 5

column, as they will be even larga than the target

② look at the next column; A[index] is still larger than the target, ∴ we can eliminate this column too.

③ Same for this column.

④ Because A[index] < target and all elements to the right of 4 are already eliminated, the 1st row can be eliminated completely since the value(s) to 4's left side are even smaller

⑤ Found 5.

- Pseudocode returns the coordinates of the target if it is in the matrix, otherwise, return [-1, -1].

```
SEARCH_MATRIX (matrix[][], int target) {
  // corner cases
  if(matrix == null || matrix.length == 0 ||
     matrix[0] == null || matrix[0].length == 0) {
     return new int[] {-1, -1};
  }

  int row = 0;
  int col = matrix[0].length - 1;   // always start from the top
                                    // right corner
  if (matrix[row][col] == target ){
     return new int[] {row, col};
  }

  if (matrix[row][col] > target){ // all elements in the current
     col--;                       // column can be eliminated
  }else{      // all elements in the current row can be eliminated
     row++;
  }

  return new int[] {-1, -1}; // if not found
}
```

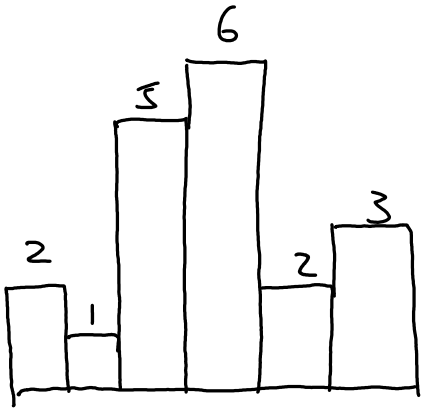
- Time Complexity Analysis

O(m+n) there are m rows and n columns in the matrix and every time we can check & eliminate one row or one column; in the worst case, the number of checks can be # rows + # columns, hence the time complexity is O(m+n).

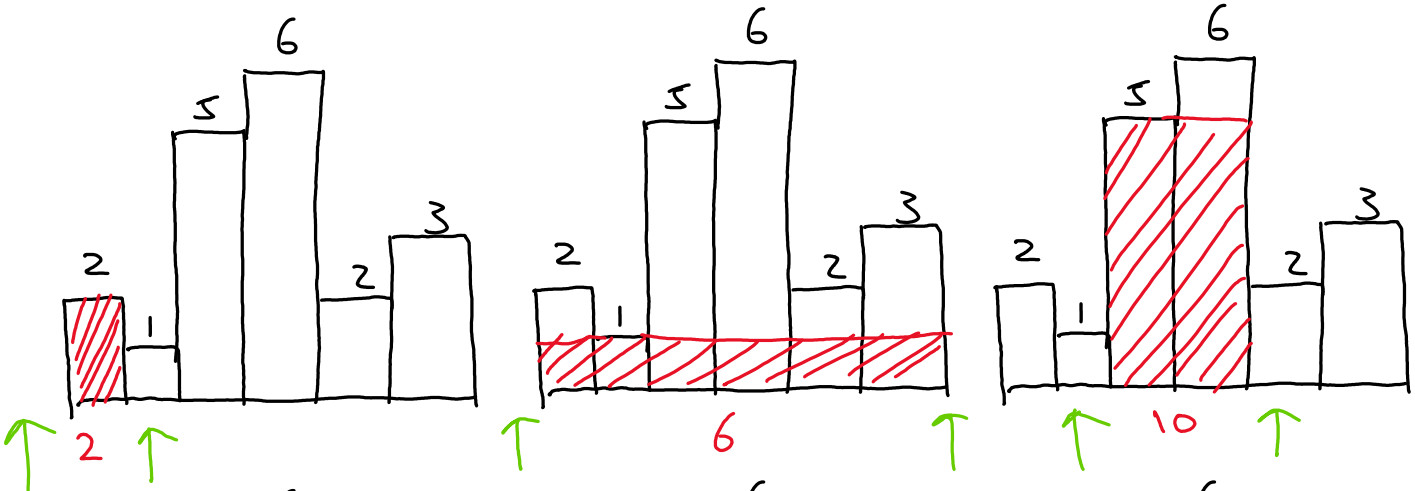
Problem 6

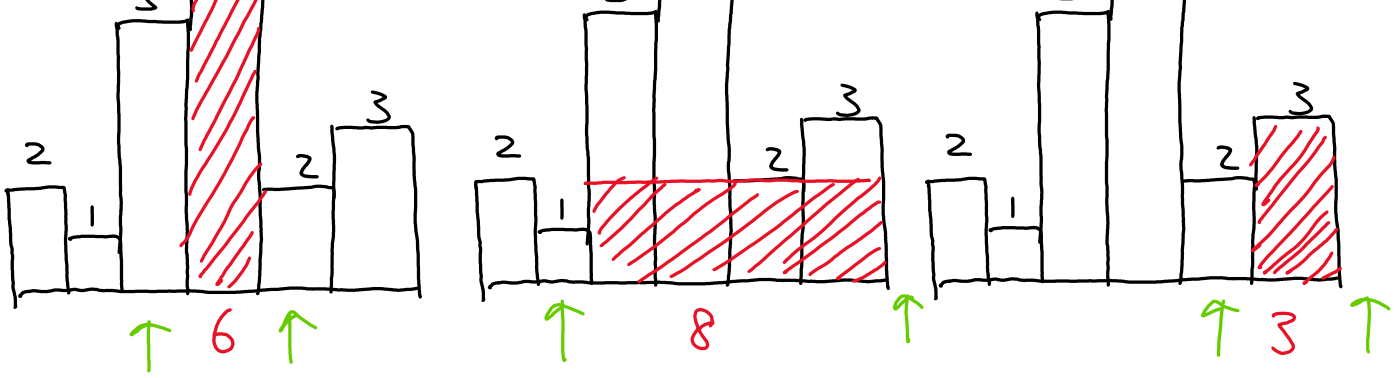
- Intuition

e.g. [2, 1, 5, 6, 2, 3]



We can look at each bar separately; calculate the maximum area of rectangle based on its height; the maximum value among those areas would be the solution to the problem.



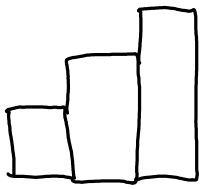


In order to calculate these areas, we must find these boundaries (indicated by the green arrows). Therefore, every time that we encounter a bar with height less than the previous one, we know that we've hit a boundary. A stack can be used here: we keep adding more bars (i.e. increasing the width) until reaching a bar with smaller height.

- Pseudocode

```
LARGEST_RECTANGLE_IN_HISTOGRAM(heights[]){
  // base cases
  if(heights == null || heights.length == 0) return 0;
  int maxArea = 0;  // update the maximum area to date
                    // when necessary
  Stack<Integer> stack = new Stack<>();
  int right, left, h;

  right: index of right boundary
  left: index of left boundary
  h: index of maximum height of the current rectangle

  for(int i = 0 ... heights.length - 1){   // keep adding bars as long as their heights are increasing
    if(stack.isEmpty() || heights[i] >= heights[stack.peek()]){
      stack.push(i);
    } else{
      right = i;  // right boundary
      h = stack.pop();  // the max height possible
      left = stack.isEmpty() ? -1 : stack.peek();
                                  ↓
                          // the left boundary;
                          // -1 for cases like this
```

```
                // 
                maxArea = Math.max(maxArea,
                                   (right - left - 1) * heights[h]);
                i--; // because for-loop is increased by one each time
                     // and we haven't checked the current index yet
            }
        }

        // At this point, all we have are bars with increasing
        // lengths; they have a common right boundary:
```



right = stack.peek() + 1

```
        right = stack.peek() + 1;
        while (!stack.isEmpty()) {
            h = stack.pop();
            left = stack.isEmpty() ? -1 : stack.peek();
            maxArea = Math.max(maxArea,
                               (right - left - 1) * heights[h]);
        }
        return maxArea;
    }
```

- Time complexity Analysis:

This algorithm iterated through each bar (i.e. each entry in the array); and for each bar, we did constant time operations such as comparing, popping out from the stack, calculating area... ∴ this algorithm runs in linear time O(n).

Problem 7

Problem 2

- Intuition: Suppose that n=7, an array satisfying the requirements could be:

[5, 1, 2, 3, 7, 4, (7), 6] n+1=8 elements
                ↓
            repeated

This could also be the array:

[5, 1, 2, 7, (7, 7, 7), 6]
        repeated three times

↳ Initial thought: Initialize a hashset, iterate through the array, adding each number to the hashset; as soon as we find that the hashset already contains the number, that number is the duplicate we are looking for.

↳ Pro: O(n) time → satisfies the requirement that time should be less than $O(n^2)$.

↳ Con: O(n) space → not constant!

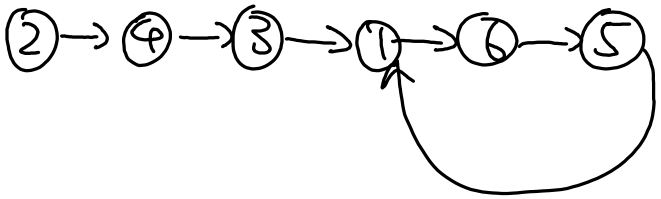
After, I looked up online and learnt this problem can be solved using a similar strategy as detecting the starting node of a loop in a linked list with loop; we use what is called the "Floyd's tortoise and hare cycle detection algorithm".

First of all, we need to construct a sequence according to the following pattern: x, A[x], A[A[x]], A[A[A[x]]] ... where each new element in the sequence is an element in A at the index of the previous element.

e.g.

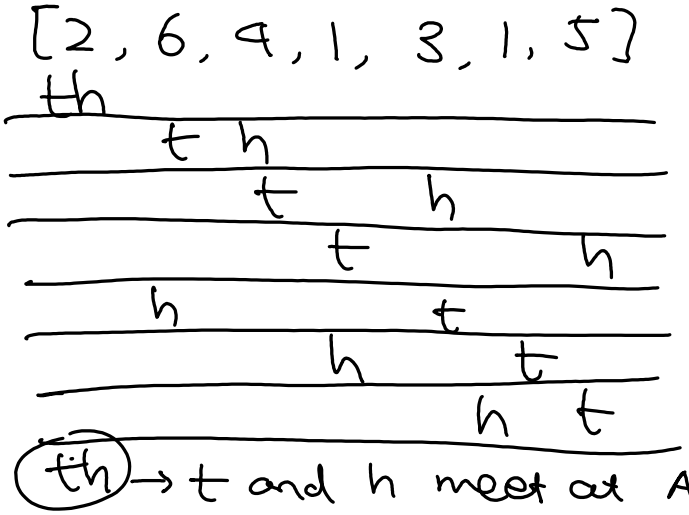
⓪	①	②	③	④	⑤	⑥
2	6	4	1	3	1	5

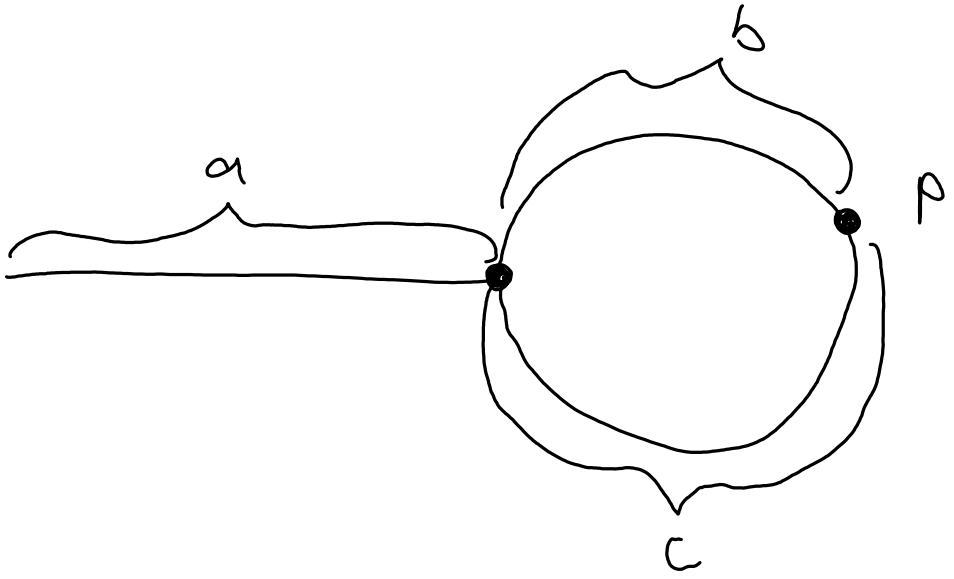
A[0]=2
A[2]=4
A[4]=3
A[3]=1
A[1]=6
A[6]=5
A[5]=1



A cycle appears because A contains duplicates. And the duplicate node is the cycle entrance.

Floyd's algorithm makes use of two pointers - tortoise, which is analogous to a slow pointer, and hare, which is analogous to a fast pointer. We let the hare move 2 times as fast as the tortoise.

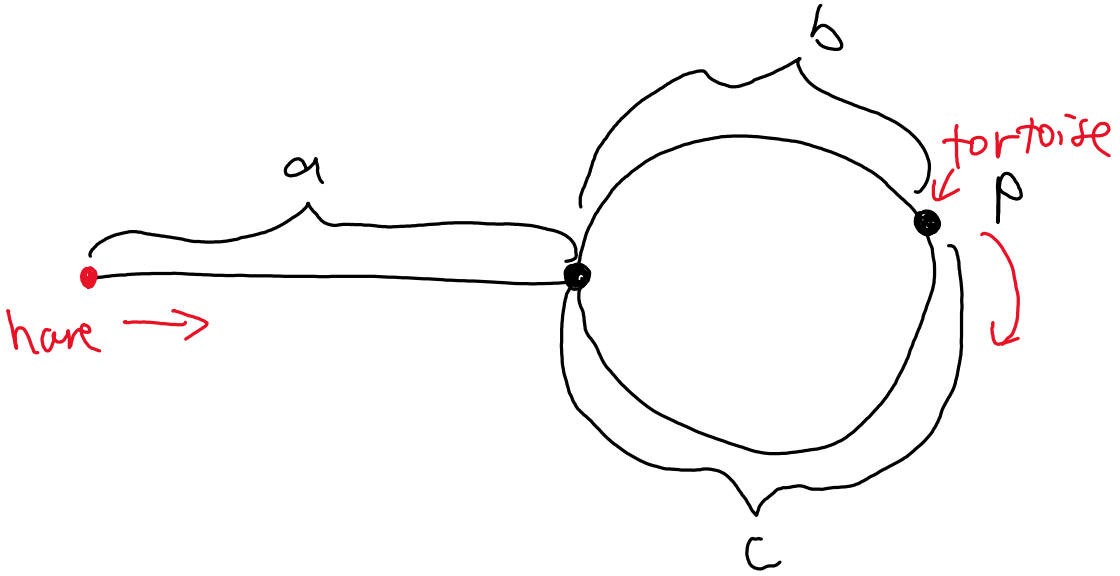




- In a more general case, since the hare moves 2 nums faster than the tortoise, it enters the loop first. By the time that tortoise and hare meet at p, the tortoise has travelled a+b, and the hare has travelled a+2b+c since distance(hare) = 2*distance(tortoise):

$$a+2b+c = 2(a+b)$$

$$\boxed{a = c}$$



Next, we reset the hare to the starting point and leave tortoise at the current intersection point. And now we set speed(hare) = speed(tortoise); then they meet again (after traveling the same distance) at the loop entrance.

- Pseudocode

```
FIND_DUPLICATE_NUM (A[]){
  int t = A[0];
  int h = A[0];

  do {
     t = A[t];
     h = A[A[h]];
  } while (t != h);

  //NOW, tortoise & hare intersect for the 1st time.
```

```
    h = A[0];
    while (t != h){
        h = A[h];
        t = A[t];
    }
    return h;
}
```

- **Analysis**

  Time : O(n) → This is a linear-time algorithm; we used two pointers to traverse the array

  Space : O(1) → Constant space complexity since we only initialized some variables; no new data structures initialized.

Problem 8

- **Intuition:**

$$A = \begin{bmatrix} B & C \\ D & E \end{bmatrix}$$

$$AA = \begin{bmatrix} B & C \\ D & E \end{bmatrix} \begin{bmatrix} B & C \\ D & E \end{bmatrix}$$

According to the Strassen's algorithm, we can break two 2x2 matrices into eight 1x1 matrices and perform the matrix multiplication blockwise.

$$AA = \left[\begin{array}{c|c} B & C \\ \hline D & E \end{array}\right] \left[\begin{array}{c|c} B & C \\ \hline D & E \end{array}\right] = \begin{bmatrix} BB + CD & BC + CE \\ DB + ED & DC + EE \end{bmatrix}$$

In order to compute the result using only 5 multiplications, we need to come up with 5 matrices such that each sum of matrices in AA can be represented as a sum/substraction

among two or more of the 5 matrices.

$$\begin{bmatrix} BB+CD & BC+CE \\ DB+ED & DC+EE \end{bmatrix} = \begin{bmatrix} B^2+CD & C(B+E) \\ D(B+E) & CD+E^2 \end{bmatrix}$$

1st, 2nd, 3rd, 4th, 5th

∴, we only need 5 multiplications to compute the square of a 2×2 matrix.

- Time complexity analysis:

  - 5 subproblems
  - original problem: multiplying two 2×2 matrices
    new subproblems: multiplying two (x) matrices

Master's theorem

$a = 5$

$b = 2$

What is c?

↳ since we know that we are dealing with 2×2 matrices and the size of each submatrix is (x), computing the product of each of the 5 submatrices takes $O(1)$ only, since the calculations only consists of multiplying two numbers together, or adding two numbers and multiplying it against another number in the worst case (e.g. $C(B+E)$). ∴ $c = 1$ (i.e. the additional time spent on each subproblem is constant).

Recurrence relation: $T(n) = 5T\left(\frac{n}{2}\right) + 1$

$\log_2 5 > 2$, $T(n) = O\left(n^{\log_2 5}\right)$

Conclusion: Yes, we can say that the running time is $O(n^{\log_2 5})$.