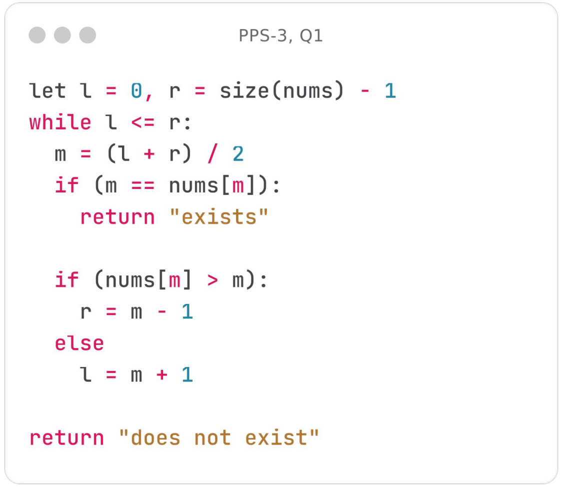
Tutorial 6

Algorithm Analysis and Design
IIIT-H (Monsoon ‘24)

Q1

Given a sorted array of distinct integers $A[1 \dots n]$, we want to determine whether there exists an index i such that $A[i] = i$. Design a divide-and-conquer algorithm that runs in time $O(\log(n))$.

Soln:

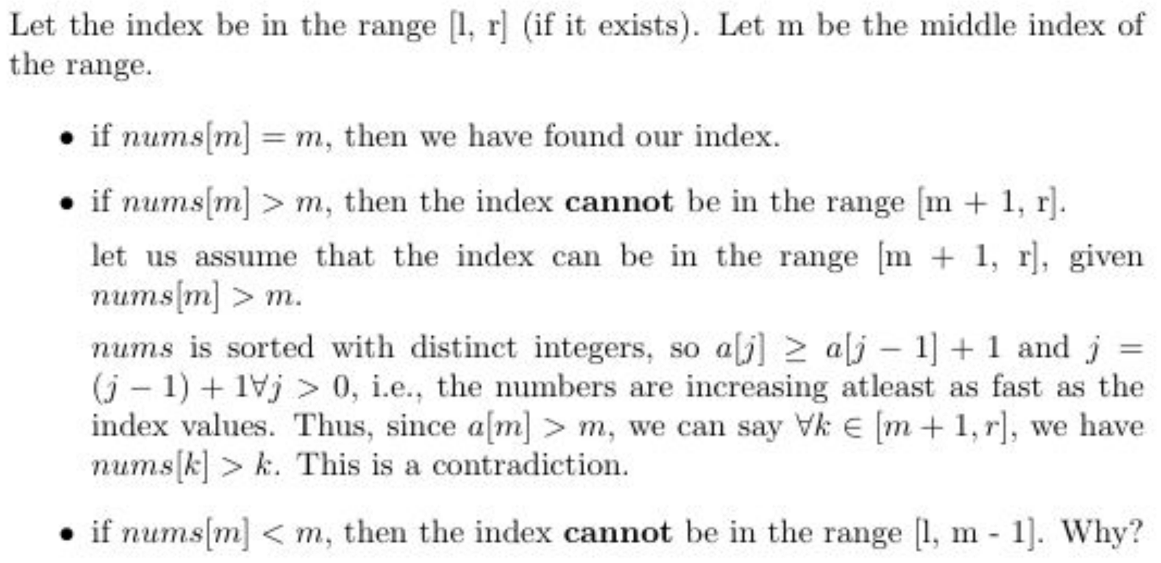


```
let l = 0, r = size(nums) - 1
while l <= r:
  m = (l + r) / 2
  if (m == nums[m]):
    return "exists"

  if (nums[m] > m):
    r = m - 1
  else
    l = m + 1

return "does not exist"
```

(Rough) Proof of Correctness



Let the index be in the range [l, r] (if it exists). Let m be the middle index of the range.

- if $nums[m] = m$, then we have found our index.
- if $nums[m] > m$, then the index **cannot** be in the range [m + 1, r].

  let us assume that the index can be in the range [m + 1, r], given $nums[m] > m$.

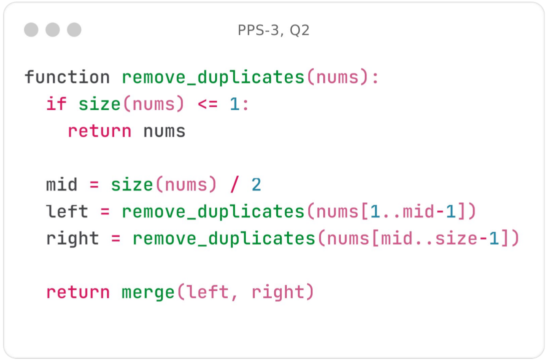
  $nums$ is sorted with distinct integers, so $a[j] \geq a[j-1] + 1$ and $j = (j-1) + 1 \forall j > 0$, i.e., the numbers are increasing atleast as fast as the index values. Thus, since $a[m] > m$, we can say $\forall k \in [m+1, r]$, we have $nums[k] > k$. This is a contradiction.
- if $nums[m] < m$, then the index **cannot** be in the range [l, m - 1]. Why?

Time Complexity?

Q2

You are given an array of n elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time $O(n \log n)$.

Soln:

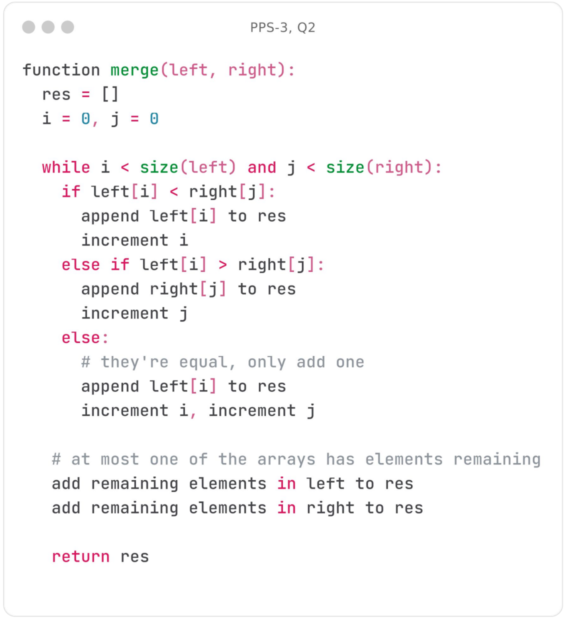


PPS-3, Q2

```
function remove_duplicates(nums):
  if size(nums) <= 1:
    return nums

  mid = size(nums) / 2
  left = remove_duplicates(nums[1..mid-1])
  right = remove_duplicates(nums[mid..size-1])

  return merge(left, right)
```



PPS-3, Q2

```
function merge(left, right):
  res = []
  i = 0, j = 0

  while i < size(left) and j < size(right):
    if left[i] < right[j]:
      append left[i] to res
      increment i
    else if left[i] > right[j]:
      append right[j] to res
      increment j
    else:
      # they're equal, only add one
      append left[i] to res
      increment i, increment j

   # at most one of the arrays has elements remaining
   add remaining elements in left to res
   add remaining elements in right to res

   return res
```

Proof of Correctness?

Do it yourself, but here are some considerations:

- How are we assured that the merge step removes duplicates across arrays? How do we know that if the same element exists in L and R, our i or j will increment such that these elements will eventually be compared?
- Why do we only worry about removing duplicates across arrays in the merge step? How do we know there aren't duplicates within L and R?

Time Complexity

$$T(N) = 2T(\frac{N}{2}) + O(N) \rightarrow O(N \cdot log(N))$$

Q3

A k-way merge operation. Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

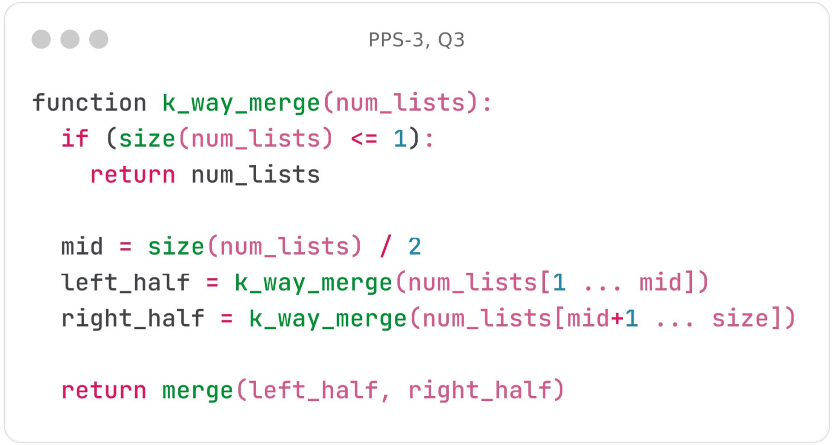
(a) Here's one strategy: Using the merge procedure from Section 2.3, merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of k and n?

(b) Give a more efficient solution to this problem, using divide-and-conquer.

1. There are $k-1$ merge steps to perform. The first merge involves $n+n = 2\cdot n$ elements. The second will involve $2 \cdot n + n = 3 \cdot n$ elements, and so on, until the $(k-1)^{th}$ merge involving $k \cdot n$ elements.

   The total time complexity is $O(2n + 3n + 4n + ... + kn)$. Evaluate the arithmetic mean to get $O(\frac{k-1}{2}(2(2n) + (k-2)n) = O(nk^2)$.

2.



PPS-3, Q3

```
function k_way_merge(num_lists):
  if (size(num_lists) <= 1):
    return num_lists

  mid = size(num_lists) / 2
  left_half = k_way_merge(num_lists[1 ... mid])
  right_half = k_way_merge(num_lists[mid+1 ... size])

  return merge(left_half, right_half)
```

Time Complexity:

$$T(k) = 2T(\frac{k}{2}) + O(nk) \rightarrow O(nk \cdot log(k))$$

DIY

You are given an infinite array A[·] in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are not given the value of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in O(log n) time. (If you are disturbed by the fact that the array A has infinite length, assume instead that it is of length n, but that you don't know this length, and that the implementation of the array data type in your programming language returns the error message ∞ whenever elements A[i] with i > n are accessed.)

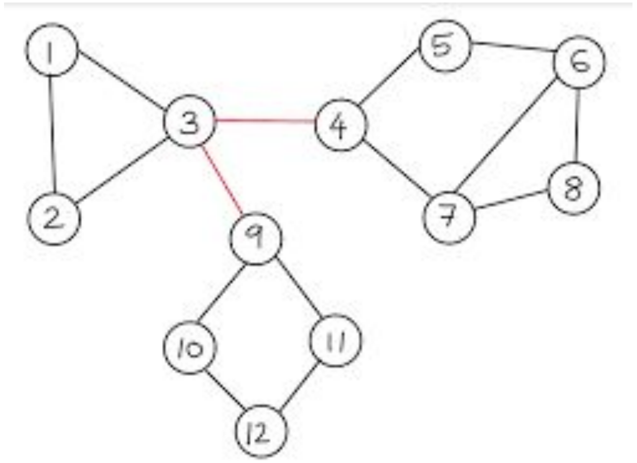
(Source: Algorithms by Dasgupta et al., Exercise 2)

Miscellaneous Doubts (from previous topics)

Bridge Finding

“What does the low array do?”

We are given an *undirected* graph. A **bridge** is defined as an edge which, when removed, makes the graph disconnected (or more precisely, increases the number of connected components in the graph).



Source: CF Tutorial

Formulating our problem

Let's say we are in the DFS, looking through the edges starting from vertex **v**. The current edge **(v, to)** is a bridge if and only if none of the vertices to **to** and its **descendants** in the DFS traversal tree has a back-edge to vertex **v** or any of its ancestors. This condition means that there is no other way from **v** to **to** except for edge **(v, to)**.

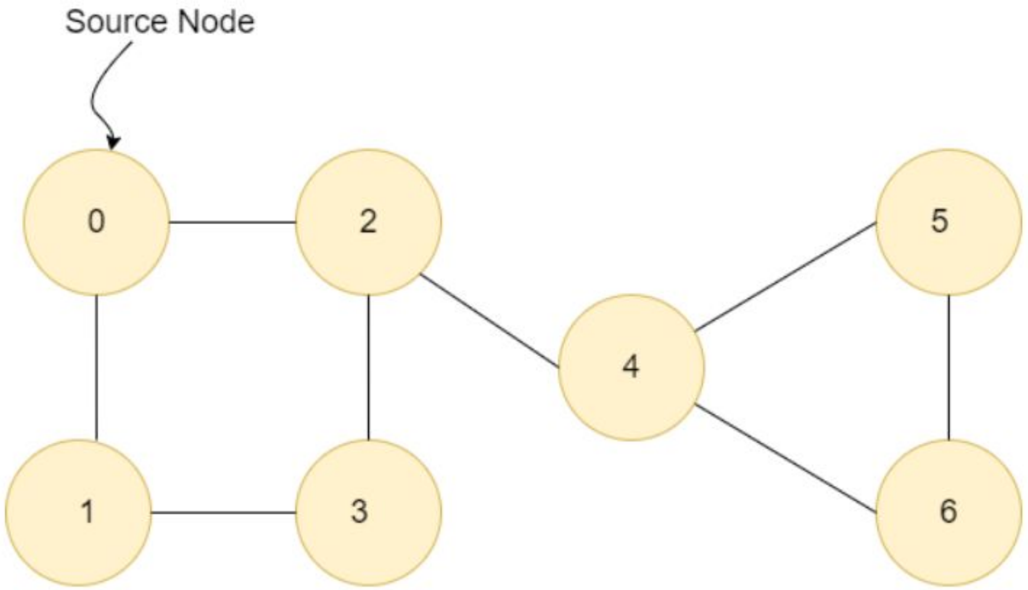
To help us determine this, we maintain two arrays, **low** and **tin**. Here

- **tin[v]** denotes the entry time for node *v*, the time the dfs first discovers the node.
- **low[v]** denotes the *lowest possible entry time that is reachable from the node **v** or its descendents.* Basically, it is the lowest entry time among itself and all of its adjacent nodes <u>*apart from its parent*</u>*.*

Now, if the current edge is **(v, to)**, it is a bridge if and only if **low[to] > tin[v].** This is because if **low[to] <= tin[v]**, then *to* can reach *v* or one of its ancestors through some immediate adjacent node or that of one of its descendents.

Dry Run

Source of image: [javatpoint]



Resources

- Simple dry run, if your understanding is weak: https://www.youtube.com/watch?v=qrAub5z8FeA
- More terse yet complete, and the source for most of this explanation: https://cp-algorithms.com/graph/bridge-searching.html

After understanding bridges, articulation points should not be too difficult to understand: https://cp-algorithms.com/graph/cutpoints.html

Questions?