Getting Started

1] Read (and Reread) Carefully and Understand the Problem

Putting the problem in your own words is a powerful way to solidify your understanding of the problem and show a potential interviewer you know what you’re doing.

1. **Ask Questions to Clarify Doubt**: Seek clarifications if anything is unclear. Don't assume details.
2. **Provide Example Cases to Confirm**: Work through examples manually to understand the expected outcomes.
3. **Plan Your Approach**: Outline a logical plan and identify which DS&A could solve the problem. Break problems to sub problems. Always acknowledge the brute force solution if spotted.

Evaluate Input:

- Is it a single array? Or multiple arrays?
- Are there any constraints on the size of the array?
- Can the array contain
  - negative numbers
  - floating-point numbers
  - ...or other data types?
- Is the array presorted?
- Is the array sequential?
  - Can the array elements align with the index if its sequential
- Do they have negative values?

Problem Constraints:

- Are there any time complexity requirements for the solution?
  - O(1): operations in-place and cannot use additional memory (variables, arrays, dictionaries)
- Are there any space considerations for the solution?
  - O(1) means using Hash maps (dictionary)

Output:

- Is it a single value, an array, or something else?
- Are there any specific requirements or constraints on the output format?

Edge Cases:

- Should the algorithm handle edge cases such as
  - empty array?
  - arrays with only one element?
  - arrays with all identical elements?

---

1. **Understand the Problem**:
   - Read and comprehend the problem statement.
2. **Clarify Doubts**:
   - Ask the interviewer for clarification if needed.
3. **Ask Questions**:
   - Gather more information about the problem.
4. **Design a Plan**:
   - Devise an approach to solve the problem.
5. **Break Down the Problem**:
   - Divide the problem into smaller sub problems if necessary.
6. **Choose the Right Data Structures and Algorithms**:
   - Select appropriate tools based on problem requirements.
7. **Write Pseudocode**:
   - Outline the solution logic without worrying about syntax.
8. **Code Implementation**:
   - Write the actual code following best practices.
9. **Test Your Solution**:
   - Verify correctness and robustness with test cases.
10. **Optimize if Necessary**:
    - Improve time or space complexity if possible.
11. **Handle Errors and Edge Cases**:
    - Ensure graceful handling of errors and edge cases.
12. **Review and Debug**:
    - Check for errors and bugs, and troubleshoot as needed.

- **Communicate Your Thought Process**:
  - Explain your approach and reasoning to the interviewer.
- **Be Flexible and Adaptive**:

1. Adapt your approach based on feedback or new insights.
2. **Practice Regularly**:
   - Improve problem-solving skills through practice and mock interviews.

2. If you forget a builtin method, use 'print(dir())' in interactive terminal

This also works on your own methods as well

```
print(dir(list))


[ __DUNDER_METHODS__ 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

For more in depth information use help(list), but remember to press 'q' in interactive prompt when you want to end process.

```
|  append(self, object, /)
 |      Append object to the end of the list.
 |
 |  clear(self, /)
 |      Remove all items from list.
 |
 |  copy(self, /)
 |      Return a shallow copy of the list.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  extend(self, iterable, /)
 |      Extend list by appending elements from the iterable.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  insert(self, index, object, /)
 |      Insert object before index.
 |
 |  pop(self, index=-1, /)
 |      Remove and return item at index (default last).
 |
```

```
|      Raises IndexError if list is empty or index is out of range.
|
|  remove(self, value, /)
|      Remove first occurrence of value.
|
|      Raises ValueError if the value is not present.
|
|  reverse(self, /)
|      Reverse *IN PLACE*.
|
|  sort(self, /, *, key=None, reverse=False)
|      Sort the list in ascending order and return None.
|
|      The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
|      order of two equal elements is maintained).
|
|      If a key function is given, apply it once to each list item and sort them,
|      ascending or descending, according to their function values.
|
|      The reverse flag can be set to sort in descending order.
```

Revision #7
Created 20 February 2024 05:54:24 by victor
Updated 11 March 2024 07:07:09 by victor