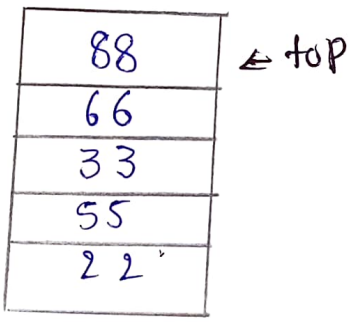
① Perform the following operations using stack. Assume the side of the stack is a 5 and having a value of 22, 55, 33, 66, 88. in the stack from a position to size-1. Now perform the following operation.

1) Insert the elements in the stack, 2, pop[3,3] pop[], 3) pop[], 4) push(90), 5) push[36], 6) push[11], 7) push [88], 8) pop[], 9) pop[], Draw the diagram of stack & illustrate the above operations & identify where the top is?

A) Stack of the stack : 5

Elements in stack (from bottom to top): 22, 55, 33, 66, 88

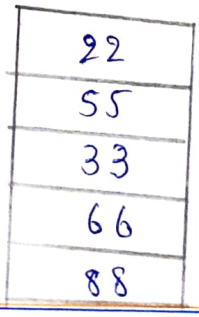
Top of stack : 88



operations:-

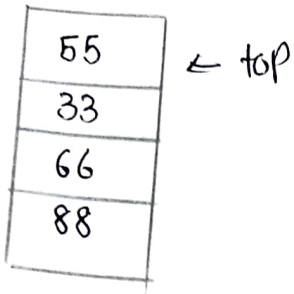
1) Invert the elements in the stack:

- The operation will reverse the order of elements in the stack.
- After inversion, the stack will look like:



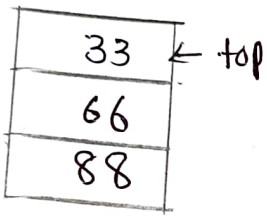
2) pop():

- Remove the top element (22).



3) pop():-

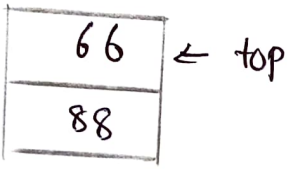
- Remove the top element (55).



4) pop():-

- Remove the top element (33).

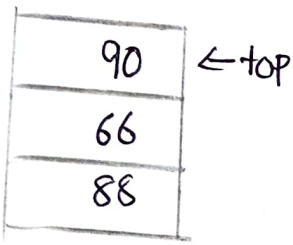
Stack after pop:



5) push(90):-

- push the element 90 onto the stack

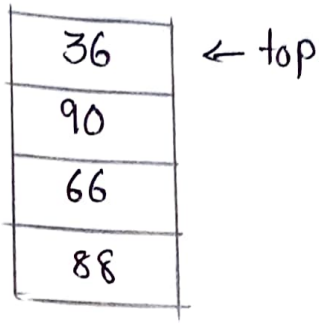
Stack after push



6) push (36):-

- push the element 36 onto the stack.

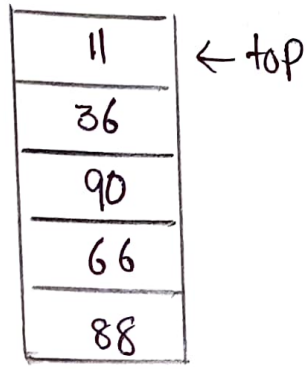
stack after push:



7) push(11):

- push the element 11 onto the stack.

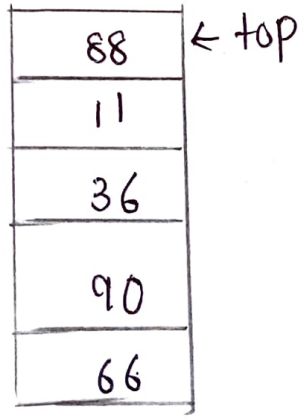
stack after push:



8) push (88):-

- push the element 88 onto the stack.

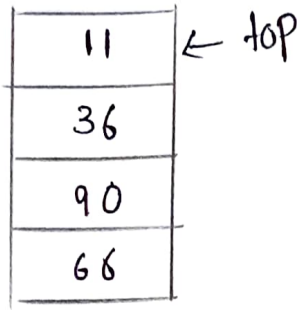
stack after push.



9) pop():

• Remove the top element (88).

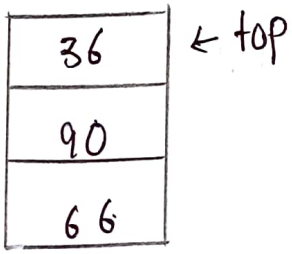
stack after pop:



10) pop():

• Remove the top element (11).

stack after top:



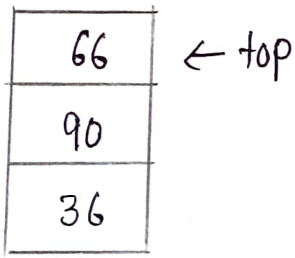
Final stack state:-

size of stack : 5

Elements in stack (from bottom to top):

36, 90, 66

TOP of stack : 66



Develop an algorithm to detect duplicate elements in an unsorted array linear using search. Determine the time complexity & discuss how you would optimize this process.

Algorithm:

1) Initialization:

create an empty set or list to keep track of elements that have already been seen.

2) linear search:

Iterate through each element of the array:

- for each element, check it it is already in the set of seen elements
- If it is, a duplicate has been found.
- If it is found, add it to the set of seen elements.

3) Output:

Return the list of duplicates, or simply indicate that duplicates exit.

C code :-

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
int arr[] = {4,5,6,7,8,5,4,9,0};
int size = sizeof(arr)/sizeof(arr[0]);
```

```
bool seen [1000] = {false}
for (int i=0; i<size ; i++)
if (seen [arr[i]])
printf ["Duplicate found : %d \n", arr[i]);
else
    seen [arr[i]] = true;
return 0;
}.
```

Time complexity:

the. linear search complexity:-

The time complexity for this algorithm is O(n), where 'n' is the no. of elements in the array. This is because each element is checked only once, and operations (checking for membership & adding to a set) are O(1) on the average.

Space Complexity:

The space complexity is O(n) due to the additional space used by the 'seen' & 'duplicates' sets, which may store up to 'n' elements in the worst case.

Optimization:

Hashing:

the use of a set for checking duplicates is already efficient because sets provide average O(1) time complexity for membership and insertions.

sorting:

If we are allowed to modify the array, another approach is to sort the array first & then perform a linear scan to find duplicates.

sorting would take O(n log n) time, & the subsequent scan would take O(n) time. this approach uses less space (O(1) additional space if sorting in-place).