MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

Week 09 Tutorial

Tutorial n. 7

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

March 8, 2022

Question 1

An array A contains $n-1$ unique integers in the range $[0, n-1]$ (both ends inclusive), that is, there is one number from this range that is not in A. Remember that inclusive range $[0, n-1]$ contains n integers, so when we say an array contains $n-1$ unique integers of that range, it basically means there is one integer missing.

An example of such an array could be $[4, 2, 0, 6, 1, 5]$. It contains 6 elements in the range $[0, 6]$, and obviously element 3 is missing.

Design an algorithm for finding that missing number that fulfills two requirements:

1. with time complexity of $O(n)$
2. You are only allowed to use $O(1)$ additional space besides the array A itself. This means, for example, you are not allowed to use $O(n)$ space; that is, you can not use a space that is proportional to n.

Question 1

An array A contains $n-1$ unique integers in the range $[0, n-1]$ (both ends inclusive), that is, there is one number from this range that is not in A. Remember that inclusive range $[0, n-1]$ contains n integers, so when we say an array contains $n-1$ unique integers of that range, it basically means there is one integer missing.

An example of such an array could be $[4, 2, 0, 6, 1, 5]$. It contains 6 elements in the range $[0, 6]$, and obviously element 3 is missing.

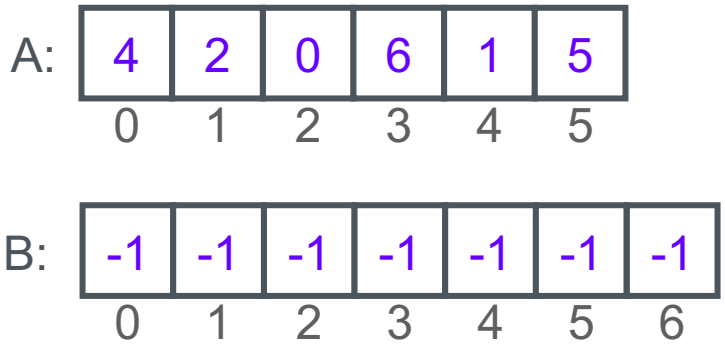
Design an algorithm for finding that missing number that fulfills two requirements:

1. with time complexity of $O(n)$
2. You are only allowed to use $O(1)$ additional space besides the array A itself. This means, for example, you are not allowed to use $O(n)$ space; that is, you can not use a space that is proportional to n.

The range 0 to n-1 (inclusive) contains n elements
Example: range [0, …, 6] contains 7 elements: [0, 1, 2, 3, 4, 5, 6]

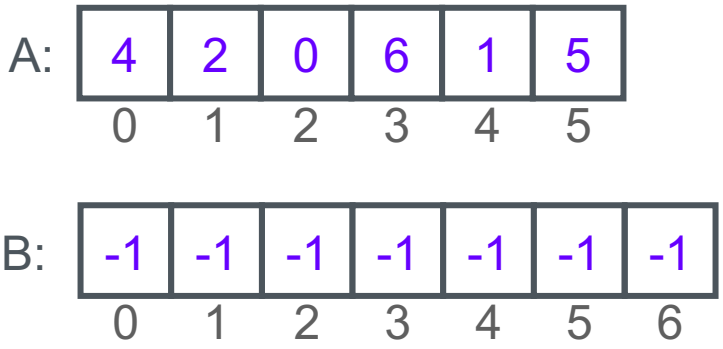
Solution to Q1

- We will see three solutions to the problem among which **only one** solution meets the requirements of the question.
- First solution:
  - Define another array **B** of size **n** and initialize it to -1.
    - It has n element to hold the range [0, n − 1]
    - We initialize to -1, a number that we know is not in the range.
    - This can be done in O(n) time.



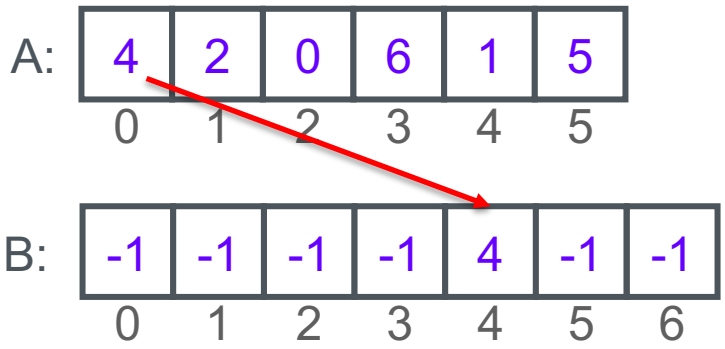
Solution to Q1

- We will see three solutions to the problem among which **only one** solution meets the requirements of the question.
- First solution:
  - Define another array **B** of size **n** and initialize it to -1.
    - It has n element to hold the range [0, n − 1]
    - We initialize to -1, a number that we know is not in the range.
    - This can be done in O(n) time.
  - Read over array A (input array) and insert each element A[i] into index A[i] of array B. (B[A[i]] = A [i]
    - Can be done in O(n) time.



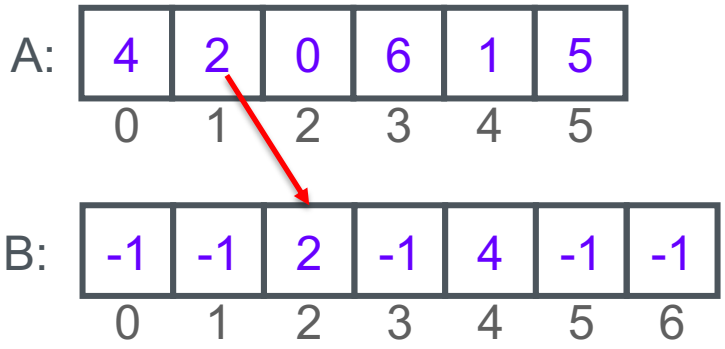
Solution to Q1

- We will see three solutions to the problem among which **only one** solution meets the requirements of the question.
- First solution:
  - Define another array **B** of size **n** and initialize it to -1.
    - It has n element to hold the range [0, n − 1]
    - We initialize to -1, a number that we know is not in the range.
    - This can be done in O(n) time.
  - Read over array A (input array) and insert each element A[i] into index A[i] of array B. (B[A[i]] = A [i]
    - Can be done in O(n) time.



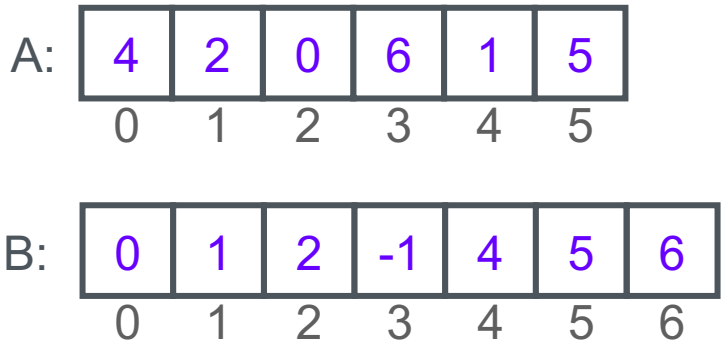
Solution to Q1

- We will see three solutions to the problem among which **only one** solution meets the requirements of the question.
- First solution:
  - Define another array **B** of size **n** and initialize it to -1.
    - It has n element to hold the range [0, n − 1]
    - We initialize to -1, a number that we know is not in the range.
    - This can be done in O(n) time.
  - Read over array A (input array) and insert each element A[i] into index A[i] of array B. (B[A[i]] = A [i]
    - Can be done in O(n) time.



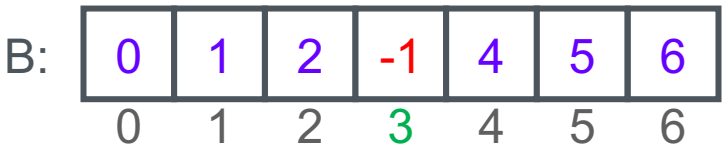
Solution to Q1

- We will see three solutions to the problem among which **only one** solution meets the requirements of the question.
- First solution:
  - Define another array **B** of size **n** and initialize it to -1.
    - It has n element to hold the range [0, n − 1]
    - We initialize to -1, a number that we know is not in the range.
    - This can be done in O(n) time.
  - Read over array A (input array) and insert each element A[i] into index A[i] of array B. (B[A[i]] = A [i]
    - Can be done in O(n) time.



Solution to Q1

- We will see three solutions to the problem among which **only one** solution meets the requirements of the question.
- First solution:
  - Define another array **B** of size **n** and initialize it to -1.
    - It has n element to hold the range [0, n − 1]
    - We initialize to -1, a number that we know is not in the range.
    - This can be done in O(n) time.
  - Read over array A (input array) and insert each element A[i] into index A[i] of array B. (B[A[i]] = A [i]
    - This can be done in O(n) time.



  - Read over array B to find the element with value -1. The index of that element is the missing value in the range.
    - This can be done in O(n) time.

Solution to Q1

- First solution:

Algorithm findMissingElement1(*A*):
Input: An array *A* of non-negative integers in range [0, …, n-1]
Output: An integer i indicating the missing element in the range.

```
Let B be an array of n numbers initialized to -1          O(n)
for i ← 0 to n − 1 do
        B[A[ i ]] ← A [ i ]                               O(n)
for i ← 0 to n − 1 do                                     O(n)
        if B[i] = -1 then
                return i
```

  - Overall time complexity of the algorithm is: O(n + n + n) which is O(n) ✓
  - The space complexity is O(n)
    - extra space for B which is proportional to the size of A ✗

Solution to Q1

- Second solution: (using **sorting** as a means of solving a problem)

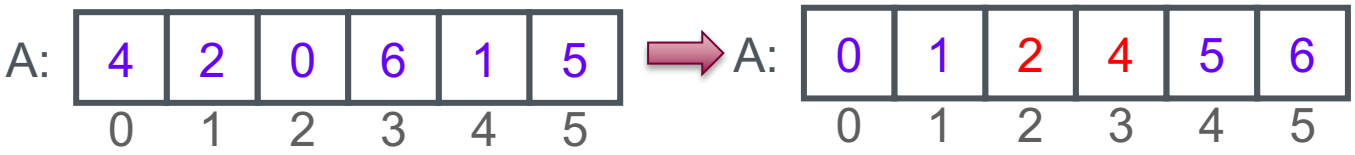
Algorithm findMissingElement2(*A*):
Input: An array *A* of non-negative integers in range [0, …, n-1]
Output: An integer i indicating the missing element in the range.

sort array A. O(nlogn)

for i ← 0 **to** n − 2 **do**

  if A[i+1] - A[i] > 1 **then** O(n)

    return A[i]+1



- We assume that the best sorting algorithm takes **O(nlogn)** time and performs in-place sorting with a constant extra space
- Overall time complexity of the algorithm is: O(nlogn + n) which is O(nlogn) ✗
- The space complexity is O(1) ✓

Solution to Q1

- Third solution:
- We know that $0 + 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$
- So, $0 + 1 + 2 + \cdots + n - 1 = \frac{n-1(n-1+1)}{2} = \frac{n(n-1)}{2}$

Algorithm findMissingElement3(*A*):
Input: An array *A* of non-negative integers in range [0, …, n-1]
Output: An integer i indicating the missing element in the range.

```
S1 ← n(n-1)/2          // desired sum
S2 ← 0                 // actual sum
for i ← 0 to n − 1 do
    S2 ← S2 + A[i]
return S1 - S2
```

O(1)

O(n)

A:

4	2	0	6	1	5
0	1	2	3	4	5

S1 = (7*6)/2=21
S2 = 18
S1-S2 is 3

- Overall time complexity of the algorithm is: O(1 + n) which is O(n) ✓
- The space complexity is O(1) ✓

Question 2

Suppose we have two different sorting algorithms A and B that given an input array, they can sort that array. The number of primitive operations executed by algorithms A and B is $8nlogn$ and $2n^2$, respectively for an input size of n. Assuming that the primitive operations all take the same time, for which values of n does algorithm B beat algorithm A? (Note: logarithm should be considered as base 2)

Solution to Q2

- Algorithm B has $2n^2$ primitive operations
- Algorithm A has $8nlogn$ primitive operations
- Algorithm B beats algorithm A when:
  - $2n^2 < 8nlogn \Rightarrow n^2 < 4nlogn \Rightarrow \mathrm{n} < 4logn$ (n is non-negative)
  - Also, we can write this way:
    - $2^{\frac{n}{4}} < \mathrm{n}$
    - So we can check for which values of n this happens
      - 2 ≤ n< 16
      - at n = 16 and onward the above

      does not hold anymore.
      - So, the cross-over of functions

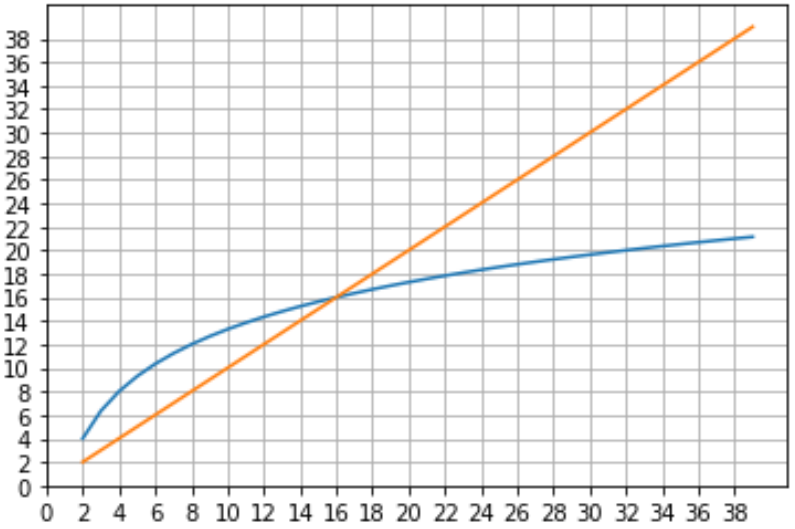
      happens at n = 16.

Solution to Q2

- Algorithm B has $2n^2$ primitive operations
- Algorithm A has $8nlogn$ primitive operations
- Algorithm B beats algorithm A when:
  - $2n^2 < 8nlogn \Rightarrow n^2 < 4nlogn \Rightarrow \boxed{n < 4logn}$ (n is non-negative)
  - Also, we can write this way:
    - $2^{\frac{n}{4}} < n$
    - So we can check for which values of n this happens
      - $2 \leq n < 16$
      - at n = 16 and onward the above does not hold anymore.
      - So, the cross-over of functions happens at n = 16.

      The blue is **4logn** for algorithm A

      orange is **n** for algorithm B



Question 3

Arrange the following functions by their order of asymptotic growth rate from smaller to larger.

- $4nlogn + 2n$
- 2^n
- 2^{10}
- 4^{logn}
- $2n^2 + 10n$
- 2^{logn}
- $3n + 100logn$
- $nlogn$
- $4n$
- n^3

Solution to Q3

- We can sort as follows:
  - 2^{10} which is constant, hence $O(1)$
  - $2^{logn} = n$ which is $O(n)$
  - $3n + 100logn$ which is $O(n)$, it has a constant factor of 3 which means it grows faster than the previous one.
  - $4n$ which is $O(n)$, it has a constant factor of 4 which means it grows faster than the previous one.
  - $nlogn$ which is $O(nlogn)$
  - $4nlogn + 2n$ which is $O(nlogn)$ but its constant factor is larger so it grows faster than the previous one.
  - $4^{logn} = 2^{logn^2} = n^2$ which is $O(n^2)$
  - $2n^2 + 10n$ which is $O(n^2)$ but it has a larger constant factor than the previous one
  - n^3 which is $O(n^3)$
  - 2^n which is $O(2^n)$

- $4nlogn + 2n$
- 2^n
- 2^{10}
- 4^{logn}
- $2n^2 + 10n$
- 2^{logn}
- $3n + 100logn$
- $nlogn$
- $4n$
- n^3

Question 4

Which one is correct? Use the definition of the Big-Oh notation to solve this problem.

- 2^{n+1} is $O(2^n)$
- 2^{2n} is $O(2^n)$

Solution to Q4

Which one is correct? Use the definition of the Big-Oh notation to solve this problem.

- 2^{n+1} is $O(2^n)$
- 2^{2n} is $O(2^n)$

- 2^{n+1} is $O(2^n)$. This is correct. Because $2^{n+1} = 2 \times 2^n$ and obviously we have $2 \times 2^n \leq c \times 2^n$ for $c = 2$ and $n_0 = 0$.
- 2^{2n} is **not** $O(2^n)$. Because $2^{2n} = (2^n)^2$ and obviously it will have a greater growth rate than 2^n. That is, for no constant c, $(2^n)^2$ may be less than or equal to $c \times 2^n$.

Question 5

What is the expected output of the following code fragment? Also, give a high-level description of the task that the algorithm does when it is given an arbitrary positive integer n. Note that **ArrayStack** is an array-based implementation of Stack data structure that you already have seen during the lectures.

```
int main(){
    ArrayStack<int> A;
    int n = 43;
    while (n > 0){
        A.push(n % 2);
        n = n / 2;
    }
    while (!A.empty()){
        cout << A.top();
        A.pop();
    }
    cout << endl;
}
```

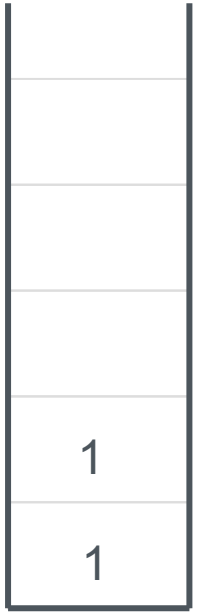
Solution to Q5

- n = 43 43 / 2 = 21 r = 1

```
int main(){
    ArrayStack<int> A;
    int n = 43;
    while (n > 0){
        A.push(n % 2);
        n = n / 2;
    }
    while (!A.empty()){
        cout << A.top();
        A.pop();
    }
    cout << endl;
}
```

Solution to Q5

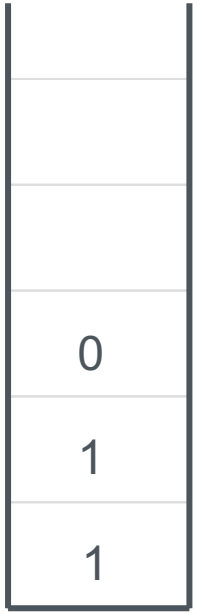
- n = 43 43 / 2 = 21 r = 1
- n = 21 21 / 2 = 10 r = 1




```
int main(){
    ArrayStack<int> A;
    int n = 43;
    while (n > 0){
        A.push(n % 2);
        n = n / 2;
    }
    while (!A.empty()){
        cout << A.top();
        A.pop();
    }
    cout << endl;
}
```

Solution to Q5

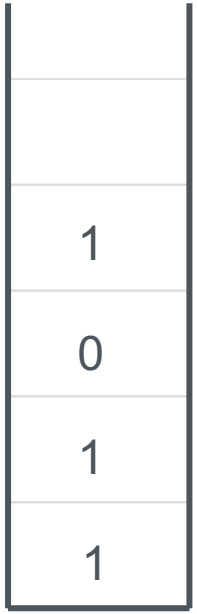
- n = 43 43 / 2 = 21 r = 1
- n = 21 21 / 2 = 10 r = 1
- n = 10 10 / 2 = 5 r = 0




```
int main(){
    ArrayStack<int> A;
    int n = 43;
    while (n > 0){
        A.push(n % 2);
        n = n / 2;
    }
    while (!A.empty()){
        cout << A.top();
        A.pop();
    }
    cout << endl;
}
```

Solution to Q5

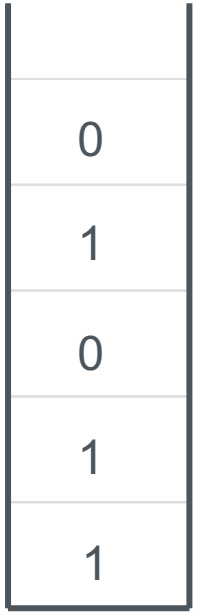
- n = 43 43 / 2 = 21 r = 1
- n = 21 21 / 2 = 10 r = 1
- n = 10 10 / 2 = 5 r = 0
- n = 5 5 / 2 = 2 r = 1




```
int main(){
    ArrayStack<int> A;
    int n = 43;
    while (n > 0){
        A.push(n % 2);
        n = n / 2;
    }
    while (!A.empty()){
        cout << A.top();
        A.pop();
    }
    cout << endl;
}
```

Solution to Q5

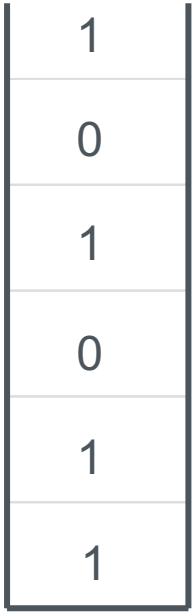
- n = 43 43 / 2 = 21 r = 1
- n = 21 21 / 2 = 10 r = 1
- n = 10 10 / 2 = 5 r = 0
- n = 5 5 / 2 = 2 r = 1
- n = 2 2 / 2 = 1 r = 0




```
int main(){
    ArrayStack<int> A;
    int n = 43;
    while (n > 0){
        A.push(n % 2);
        n = n / 2;
    }
    while (!A.empty()){
        cout << A.top();
        A.pop();
    }
    cout << endl;
}
```

Solution to Q5

- n = 43 43 / 2 = 21 r = 1
- n = 21 21 / 2 = 10 r = 1
- n = 10 10 / 2 = 5 r = 0
- n = 5 5 / 2 = 2 r = 1
- n = 2 2 / 2 = 1 r = 0
- n = 1 1 / 2 = 0 r = 1




```
int main(){
    ArrayStack<int> A;
    int n = 43;
    while (n > 0){
        A.push(n % 2);
        n = n / 2;
    }
    while (!A.empty()){
        cout << A.top();
        A.pop();
    }
    cout << endl;
}
```

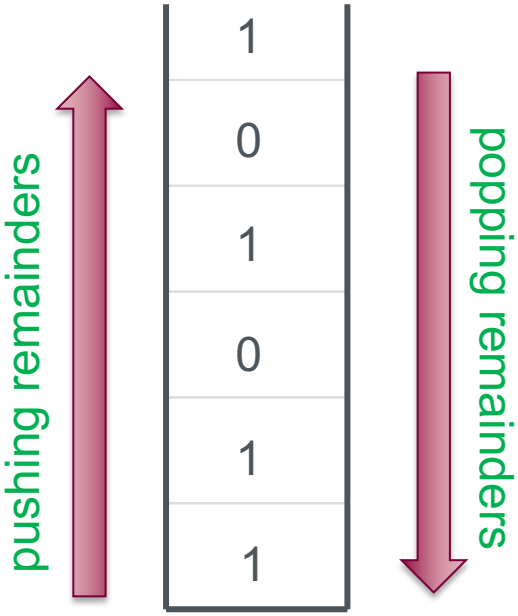
Solution to Q5

- n = 43 43 / 2 = 21 r = 1
- n = 21 21 / 2 = 10 r = 1
- n = 10 10 / 2 = 5 r = 0
- n = 5 5 / 2 = 2 r = 1
- n = 2 2 / 2 = 1 r = 0
- n = 1 1 / 2 = 0 r = 1
- n = 0 => exit the while loop

```
int main(){
    ArrayStack<int> A;
    int n = 43;
    while (n > 0){
        A.push(n % 2);
        n = n / 2;
    }
    while (!A.empty()){
        cout << A.top();
        A.pop();
    }
    cout << endl;
}
```

The output is:

101011



Solution to Q5

- n = 43 43 / 2 = 21 r = 1
- n = 21 21 / 2 = 10 r = 1
- n = 10 10 / 2 = 5 r = 0
- n = 5 5 / 2 = 2 r = 1
- n = 2 2 / 2 = 1 r = 0
- n = 1 1 / 2 = 0 r = 1
- n = 0 => exit the while loop

```
int main(){
    ArrayStack<int> A;
    int n = 43;
    while (n > 0){
        A.push(n % 2);
        n = n / 2;
    }
    while (!A.empty()){
        cout << A.top();
        A.pop();
    }
    cout << endl;
}
```

The output is:

101011

Given a positive integer **n**, and a stack that can hold integer values, the algorithm divides n by 2, sets the dividend to the original integer n, and pushes the remainder onto the stack. The algorithm tries this until n becomes zero. The sequence of remainders if printed reversely in the second loop (remember that a stack is used), shows the binary representation of original **n**.

We can use the same method to compute the representation of any given n for any base b, just by changing:

n % 2 to n % b

n = n / 2 to n = n / b

Question 6

We have seen two implementations of Stack, one using the array and the other using the singly linked list. We also have talked about the fact that we can implement the array-based Stack in a way that when it becomes full, it can perform a resize operation and make its internal array bigger. For example, it can double the array's size and copy the values from the old array to the new array. We also discussed the fact that the Stack implemented using the linked list does not need any resize and never gets full (logically). Why is sometimes the array-based implementation and resorting to resize operation **preferable** over the linked list-based implementation?

Solution to Q6

- Stacks are an important part of many computer science algorithms.
- Having a **fast stack** is often **preferable** to having a **stack with a flexible size** when the efficiency of algorithms is essential.
- There is indeed **no difference** in the asymptotic time complexity of their operations.
  - However, the linked list-based implementation has overheads of working with pointers, dynamic memory allocation, etc., making it slower than the array-based implementation for frequently-used operations.
  - Arrays can be accessed randomly with the index, and this is a very fast operation and makes stack operations fast.
  - We can **skip** the **overhead of resizing the array and relocating the stack elements** as it does not frequently happen if we choose a reasonably large array size.

Questions?