PDSA Quiz 1 Revision Session - Kaziranga House

25 October 2025 15:00

Time Complexities

Q1 Time Complexities and their Order

Consider the following operations and their time complexity.

- Operation 1 complexity: $O(n \log n)$
- Operation 2 complexity: $O(n)$
- Operation 3 complexity: $O(\log n)$

The overall complexity of executing all three operations sequentially is:

OPTIONS:

- $O(n)$
- $O(\log n)$
- $O(n \log n)$
- $O(n^2)$

Max is the answer.

Data structures & Algorithms

Eg Queues, Stacks

W) - 4

Data Structures
1) Graphs
2) Lists → Stacks, Queues.
3) Dict

Algorithms
1) Searching →
2) Sorting
3) Traversing.

Op1 → O(n log n)
→ O(n)
→ O(log n)
→ Max

Time Complexities: Judges of how well a algorithm performs.
↳ Execution Time
↳ Computation Power

Q Arrange in descending Order

1) $3n + \log n \rightarrow n$
2) $(\log n)^2 \rightarrow (\log n)^2$
3) $\log(\log(n)) \rightarrow \log(\log n)$
4) $100 \log n \rightarrow \log n$
5) $6n \log n \rightarrow n \log n$

n = 100

100 ②
$2^2 = 4$ ③
$\log_{10}(2) = 0.3$ ⑤
$\log n = 2$ ④
$100(2) = 200$ ①

5, 1, 2, 4, 3 ←

5, 2, 3, 1, 4 ←

5, 1, 2, 4, 3

Op1 Op2

$n + 2\log(n)$

Op1

$2n \log(n)$

* When, have > 1 operations take the max.

Q2 Binary Search

When using the **binary search algorithm** on a sorted list, consider the **maximum number of comparisons** it might take in the worst-case scenario to find an item or confirm it's absent.

If the **total number of items in this sorted list is doubled**, how does this **maximum number of comparisons** change?

OPTIONS:

- It doubles
- It remains the same
- It becomes half
- It increases by 1

* Binary Search always work on Sorted arrays.

n

[1, 2, 3, 10, 21, 61, 70,] = 3

[1, 2, ..., 61, 70, 84, ..., 176]

14 elements

Doubled the size => Max compare increases by 1

Quad. the size => Increase by 2.

Size $\times n$ => Increase by $\log_2 n$

Working of Binary Search

[2, 6, 7, 10, 15, 21, 67]

3 Compare.

search for 67

1) Go to the middle and compare
2) If middle < go right else go left

Q3 Sorting Algos

Consider the following function for **Insertion sort**.

```
def insertionsort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        j = i
        while(j > 0 and L[j] < L[j-1]):
            (L[j],L[j-1]) = (L[j-1],L[j])
            j = j-1
```

Insertion Sort

$O(n^2) \rightarrow$ n = 1,00,000 $\Rightarrow 10^5$
$n^2 \rightarrow 10^{10}$
$O(n \log n)$

```
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        j = i
        while(j > 0 and L[j] < L[j-1]):
            (L[j],L[j-1]) = (L[j-1],L[j])
            j = j-1
    return(L)
```

Under which scenario does the insertion sort exhibit its worst-case time complexity of $O(n^2)$, and why does this specific input trigger that behavior?

OPTIONS:

- When the input array is already sorted; the algorithm still performs redundantchecks.
- When the input array is sorted in reverse (descending) order; each elementneeds to be compared against and shifted past all previously sorted elements.
- When the input array contains many duplicate elements; handling duplicatesrequires extra comparisons.
- When the input array elements are randomly distributed; the lack of ordermaximizes the average number of shifts required.

1) 1, 4, 9, 27, 35 → O(n)

2) 35, 27, 9, 4, 1 → $O(n^2)$ ✓

1, 2, 3, 4

3) 1, 4, 2, 2, 5

4) 27, 4, 35, 1, 9 → $O(n^2)$

Insertion Sort

$n^2 \to 10^{-}$

$O(n \log n)$

27, 35, 4, 1, 9

27, 4, 35, 1, 9

4, 27, 35, 1, 9

35

1, 4, 27, 35, 9

1, 4, 9, 27, 35

Steps

1) Stand from 2nd element

2) Check the left/previous element

3) Swap if current < prev.

Q4 Merge Sort

Merge Sort's stability (preserving the relative order of equal elements) is an important property for certain applications. How is stability typically ensured during the critical `merge` step of the algorithm?

OPTIONS:

- Stability is automatically guaranteed by the divide-and-conquer approach itself.
- The algorithm uses a secondary key comparison if the primary keys are equal.
- During the merge step, when comparing an element from the left subarray (`L[i]`) and the right subarray (`R[j]`), if `L[i] == R[j]`, the element from the *right* subarray (`R[i]`) must be chosen first to be placed into the merged array.
- During the merge step, when comparing an element from the left subarray (`L[i]`) and the right subarray (`R[j]`), if `L[i] == R[j]`, the element from the *left* subarray (`L[i]`) must be chosen first to be placed into the merged array. ✓

B, C, F, D, A, E

B | F, C D | A, E

B F | C D A | E

F C A E

1st

Merge :- 1) Take the left element, compare with the 1st right element.

2) if less append, else append the right element

Stable Algorithm

3, 1, 2, 5, 2

↓ Stable

1, 2, 2, 3, 5

* Merge sort is a Stable algorithm

(2, 5) (2, 7)

In this case give priority to left.

2-2

5-2

2, 2

B-A

B-D

C-D

F-D

F-E

A, B, C, D, E, F

Q5 Quick Sort

Which of the following scenarios would guaranties that Quick Sort exhibits its **worst-case complexity** of $O(n^2)$?

OPTIONS:

- Choosing a median value as the pivot for each recursive call.
- Choosing either the smallest or the largest element as the pivot at eachrecursive call. ✓
- Choosing either the first or the last position element as the pivot at eachrecursive call.
- Partitioning the array into two roughly equal halves at each recursive call.

Solve → MoM (Week 8)

43, 32, 22, 78, 63, 57, 91, 13

s → s l

R1 → s keep moving Right until you find an element > Pivot

43, 32, 22, 13, 63, 57, 91, 78

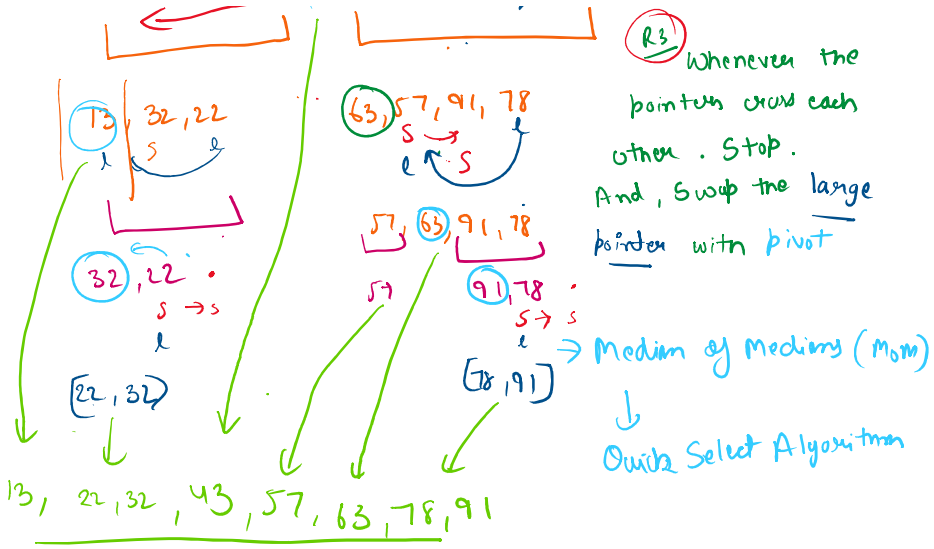
l s

R2 → l keep moving left until to find an element < Pivot

13, 32, 22, 43, 63, 57, 91, 78

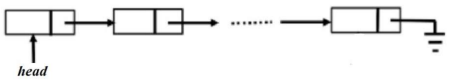
43

R3 Whenever the



Q6 Linked Lists

You are given a linked list where each node of the linked list contains data and a reference to the next node. The linked list has a `head` pointer that points to the first node of the linked list.

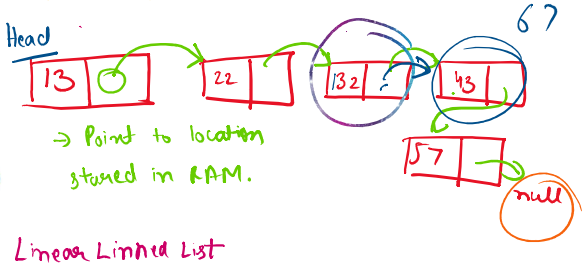


Given a pointer `p_current_node` that points to an arbitrary node in the middle of a large linked list, which of the following actions is impossible to perform directly using *only* `p_current_node` without traversing from the list's head?

OPTIONS:

- ○ Modify the data stored within `p_current_node`.
- ○ Delete the node immediately following `p_current_node` (assuming `p_current_node` is not the tail).
- ○ Insert a new node just after the `p_current_node`. ✓
- ○ Modify the data of the node that immediately *precedes* `p_current_node` in the list. ✓

Node Pointer [13,22,32,43,57]



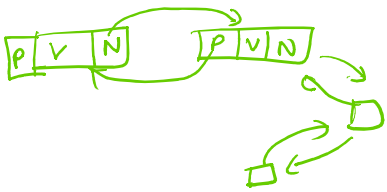
1) Do able
2) Change the pointer
3) 32 → 43
   32 → 67 → 43
4) We can't go back.

x = 10

Linear Linked List

Circular Linked List

Double Linked List



Q7 Stack and Queue

Start with an empty stack `S` and an empty queue `Q`. Perform the following sequence of operations in the given order:

1. `Q.Enqueue(10)`
2. `Q.Enqueue(20)`
3. `S.Push(5)` → Push → Put in Stack
4. `Q.Enqueue(30)` Pop → Take out of Stack
5. `x = Q.Dequeue()`, then `S.Push(x)`
6. `S.Push(15)`
7. `y = S.Pop()`, then `Q.Enqueue(y)` → Put in Q
8. `Q.Enqueue(40)`
9. `z = Q.Dequeue()`, then `S.Push(z)` take out Q
10. `S.Pop()` (the returned value is discarded)
11. `w = Q.Dequeue()`, then `S.Push(w)`

After all these operations are completed, what is the value of the element at the **top** of stack `S`?

Stack

LIFO.

Queue

FIFO

[10, 20, 30, 15, 40]

x = 10
y = 15
z = 20
w = 30

Q8 BFS

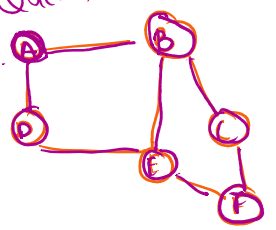
When BFS is performed on an unweighted, undirected graph starting from a source node `s`, it defines a BFS tree where edges connect nodes to their "discoverers." Consider an edge `(u,v)` from the original graph. If this edge `(u,v)` is **not** part of the BFS tree (i.e., it's a non-tree edge), what is the relationship between the levels (shortest distances from `s`) of `u` and `v`? Let `level(x)` be the distance of node `x` from `s`.

OPTIONS:

- ○ `level(u)` and `level(v)` must either be the same or differ by exactly 1 (i.e., `|level(u) - level(v)| <= 1`).
- ○ One of `u` or `v` must be `s`, and the other is at `level 1`.

~~A, B, D, E, C, F~~

BFS (Queue)



→ Put children of current in Queue
→ When take out, put in the children in Q.

OPTIONS :

- ○ `level(u)` and `level(v)` must either be the same or differ by exactly 1 (i.e., `|level(u) - level(v)| <= 1`).
- ○ One of `u` or `v` must be `s`, and the other is at `level 1`.
- ○ `level(u)` and `level(v)` must differ by more than 1 (e.g., `level(v) >= level(u) + 2`).
- ○ `level(u)` and `level(v)` must be the same, and `u` and `v` must have been discovered from different parent nodes in the BFS tree.

Q9 Hashing

Consider a hash table of size M=10 (indices 0 through 9). The hash function used is $h(k) = k(mod 10)$.

Collisions are resolved using **linear probing** with a probe step of 1 (i.e., if $h(k)$ is occupied, try $(h(k)+1)(mod M)$, then $(h(k)+2)(mod M)$, and so on).

The following keys are inserted into the initially empty hash table in this exact order:

12, 42, 35, 28, 63, 55

After all these keys have been inserted, at what index in the hash table will the key **23** be stored?

OPTIONS :

- ○ 3
- ○ 4
- ○ 6
- ○ 7

Q10 BFS Traversal and DFS Traversals

Consider the following undirected graph represented by its adjacency list:

- 0: [1, 2]
- 1: [0, 4, 5]
- 2: [0, 4]
- 3: [4, 5]
- 4: [1, 2, 3, 5]
- 5: [1, 3, 4]

A Breadth-First Search (BFS) is performed on this graph starting from source node **0**. If multiple unvisited adjacent vertices are available from the current vertex being processed, the BFS algorithm always explores the vertex with the smaller label/ID first.

Which of the following edges from the original graph will **NOT** be part of the BFS traversal tree?

OPTIONS :

- ☐ (1,4)
- ☐ (2,4)
- ☐ (3,4)
- ☐ (3,5)
- ☐ (4,5)
- ☐ (1, 5)

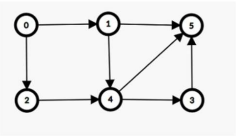
Q11 DFS

Consider a complete directed graph K5 (a graph with 5 nodes where every distinct pair of nodes is connected by a unique edge). If you perform a **Depth-First Search (DFS)** starting from an arbitrary node, say Node 1, what is the maximum number of edges that will be classified as "back edges" during this single DFS traversal, assuming the traversal explores all reachable nodes from Node 1?

Note: A back edge connects a node to an ancestor in the DFS tree (an already visited node that is still on the recursion stack).

Q12 Topological Ordering

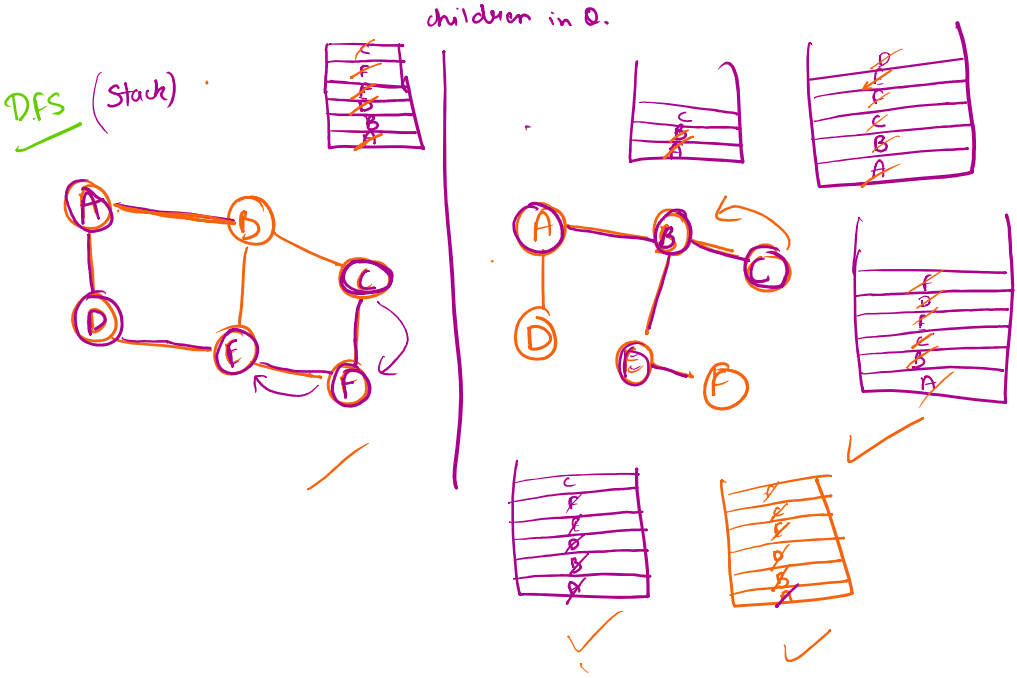
Consider the following directed graph.



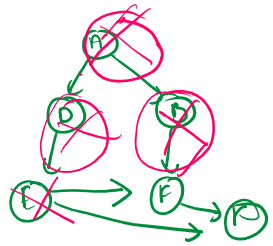
The number of topological orderings of the vertices of the given graph is ___

out, put in the children in Q.

DFS (Stack)



Topological Ordering



→ Take the node without any parent/ incoming

→ Remove it → Repeat.

A, B, D, E, F, K

A, D, E, B, F, K

A, D, B, E, F, K