DATA STRUCTURES AND ALGORITHMS

Name : Hrudhay Reddy Garisa

Table of Contents

Data Structures and Algorithms Questions

Question 1

Q.12	Which one of the following sequences when stored in an array at locations $A[1], \ldots, A[10]$ forms a max-heap?
(A)	23, 17, 10, 6, 13, 14, 1, 5, 7, 12
(B)	23, 17, 14, 7, 13, 10, 1, 5, 6, 12
(C)	23, 17, 14, 6, 13, 10, 1, 5, 7, 15
(D)	23, 14, 17, 1, 10, 13, 16, 12, 7, 5

Answer to Question 1

What is a Max-Heap?

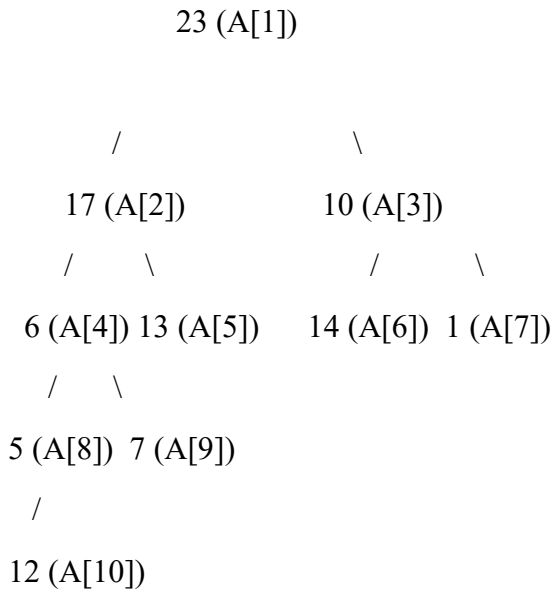
A **max-heap** is a special type of **complete binary tree** where:

1. **The value at each node is greater than or equal to the values of its children.**
2. **The tree is complete**, meaning all levels are fully filled except possibly the last level, which is filled from left to right.

If the max-heap is stored as an array starting from index **1**, then:

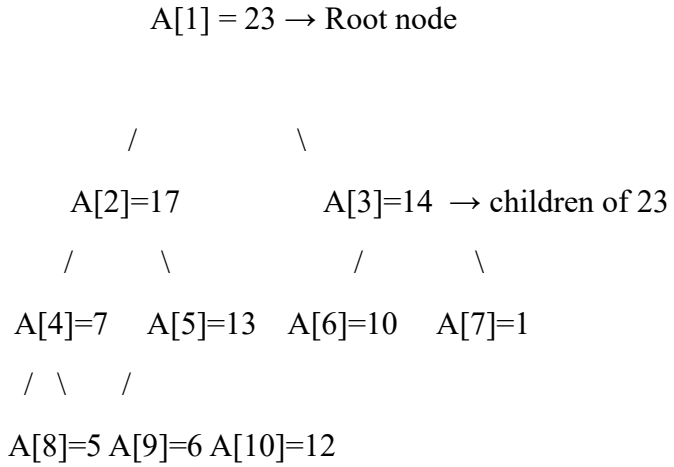
- For any node at index i:
    - Left child is at index 2*i
    - Right child is at index 2*i + 1
    - Parent is at index i//2

Lets Check Option A : 23, 17, 10, 6, 13, 14, 1, 5, 7, 12



Heap **property violated here** because **14 > 10**, and 14 is a child of 10 . So **Option (A) is NOT a max-heap**.

Option (B): 23, 17, 14, 7, 13, 10, 1, 5, 6, 12



Example of parent and child nodes relationship :

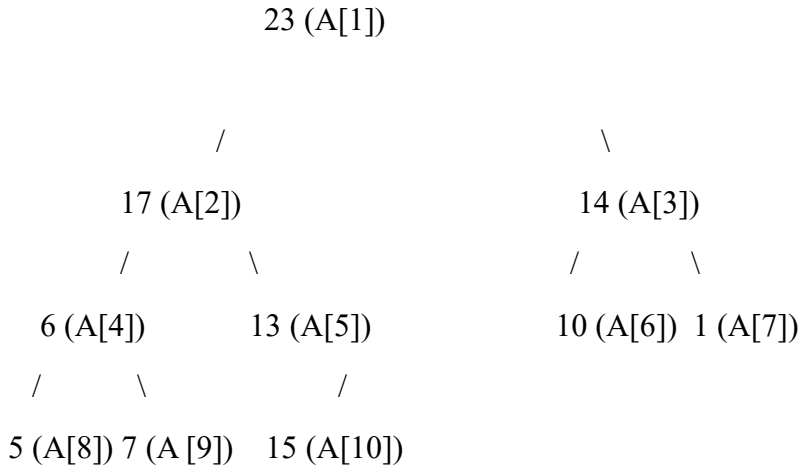
- Each **index** represents a **node**
- The **value at that index** is the **value of the node**
- Parent-child relationship is based on array indices:
- For **node at index i**:
    - **Left child** = node at index 2*i
    - **Right child** = node at index 2*i + 1

For instance :

- Node at index 2 → value = 17
- Left child: index 4 → value = 7
- Right child: index 5 → value = 13

All parent nodes are ≥ their children . This is a **valid max-heap**!

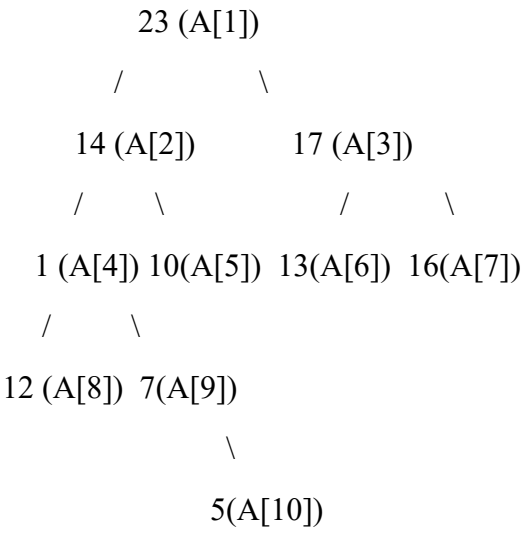
Option (C) : 23, 17, 14, 6, 13, 10, 1, 5, 7, 15



1. **Node A[4] = 6 has child A[9] = 7 → violation of max heap**
2. **Node A[5] = 13 has child A[10] = 15 → violation of max heap**

multiple violations exist in Option (C), making it **clearly not a max-heap**.

Option (D): 23, 14, 17, 1, 10, 13, 16, 12, 7, 5



$1 \geq 12, 7$ FALSE . **Violation here**: **1 < 12** and **1 < 7**

So again: **Option (D) is NOT a max-heap** .

Question 2

Q.13	Let `SLLdel` be a function that deletes a node in a singly-linked list given a pointer to the node and a pointer to the head of the list. Similarly, let `DLLdel` be another function that deletes a node in a doubly-linked list given a pointer to the node and a pointer to the head of the list. Let n denote the number of nodes in each of the linked lists. Which one of the following choices is TRUE about the worst-case time complexity of `SLLdel` and `DLLdel`?
(A)	`SLLdel` is $O(1)$ and `DLLdel` is $O(n)$
(B)	Both `SLLdel` and `DLLdel` are $O(\log(n))$
(C)	Both `SLLdel` and `DLLdel` are $O(1)$
(D)	`SLLdel` is $O(n)$ and `DLLdel` is $O(1)$

Answer to Question 2

The question is about analysing the **worst-case time complexity** of deleting a node from:

- A **singly linked list (SLL)** – SLLdel

- A **doubly linked list (DLL)** – DLLdel

A **singly linked list** is a linear data structure where each element (called a **node**) points to the **next node** in the sequence.

Structure of a Node: [Data | Next]

- **Data**: stores the value
- **Next**: stores a pointer/reference to the **next node**

For Example: [10|] → [20|] → [30|] → null

Each node links to the next one, but **not backwards**.
To reach a node, you must start from the **head** and follow the next pointers.

Limitations:

- No direct way to go backward
- To delete a node, you **need access to the previous node**

Lets say for example : [10] → [20] → [30] → [40] → null

You want to delete [30], and you're **only given**:

- A pointer to the **head** of the list
- A pointer to the **node to delete**: [30]

Why can't you just delete [30]?

To delete a node in a singly linked list, you need to **update the .next of the previous node**.

In this case, you need to set:

20.next = 40

But you don't have a direct way to access the previous node ([20]) from [30], because SLL only has .next.

So What Do You Have to Do?

You must:

1. Start from the **head** node
2. Traverse the list
3. Keep checking each node's .next
4. Stop when you find a node where: current.next == nodeToDelete

Once you find that node (current = 20), then you can do:

current.next = current.next.next;

What makes it O(n)?

Let’s say you want to delete the node with value x, and the list has n elements:

[1] → [2] → [3] → ... → [x] → ...

- Worst case: x is at the **end**
- You must check each node’s .next to find the one before x
- You might have to scan **all n nodes**

So it's a **linear scan**, and that makes it **O(n)**

Why Not O(1)?

O(1) means **constant time** — the operation takes the same time no matter how many elements there are.

To get O(1) deletion, you'd need:

- Direct access to the **previous node**
- Or the ability to **jump straight to the node before the one to delete**

In SLL, you **can’t go backward**, and there are **no shortcuts**, so you **can’t do it in O(1)** unless that previous node is already provided.

Why Not O(log n)?

O(log n) usually happens with **binary search** or **tree-based structures**, where you can eliminate half the data at each step — like:

- Binary Search Tree (BST)
- Balanced trees (e.g., AVL, Red-Black Trees)
- Binary search on arrays

But in an SLL:

- You **can’t jump** to the middle
- You **can’t divide the list**
- You must go **one node at a time**

So you can't get that logarithmic reduction — it's strictly linear traversal.

Doubly Linked List (DLL)

A **doubly linked list** is like a singly linked list, but each node has **two** pointers:

- One to the **next** node
- One to the **previous** node

Structure of a Node:

[Prev | Data | Next]

For Example: null ← [10] ↔ [20] ↔ [30] → null

Now you can traverse **both forward and backward** through the list.

Advantages:

- Can go backwards easily
- Can delete a node directly if you have a pointer to it (since you can access both neighbors)

Why is the time complexity of DLLdel (deleting a node from a Doubly Linked List) = O(1)?

Let's break it down step by step

Doubly Linked List (DLL) — Recap

Each node has 3 parts: [prev | data | next]

So:

- You can go **forward** (node.next)
- And **backward** (node.prev)

Goal: Delete a Node (say, B)

Example DLL: null ←→ [A] ←→ [B] ←→ [C] ←→ null

Each connection:

- A.next = B, B.prev = A
- B.next = C, C.prev = B

You Are Given a Pointer to Node B

To delete B, you need to:

```
B.prev.next = B.next;  // A.next = C
B.next.prev = B.prev;  // C.prev = A
```

Now B is completely skipped.

Why Is This O(1)?

Because you already have **direct access** to B and its neighbors (B.prev and B.next) via pointers.

You don't need to:

- Traverse from head
- Search for the node
- Count anything

All pointer updates are done in **constant time**, regardless of list size.

So the answer is Option (D).

Question 3

Q.20	An algorithm has to store several keys generated by an adversary in a hash table. The adversary is malicious who tries to maximize the number of collisions. Let k be the number of keys, m be the number of slots in the hash table, and $k > m$. Which one of the following is the best hashing strategy to counteract the adversary?
(A)	Division method, i.e., use the hash function $h(k) = k \mod m$.
(B)	Multiplication method, i.e., use the hash function $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$, where A is a carefully chosen constant.
(C)	Universal hashing method.
(D)	If k is a prime number, use Division method. Otherwise, use Multiplication method.

Answer to Question 3

Key Terms:

- **Hash table**: A data structure that stores data using a key → hash function → index.
- **Collision**: When two keys end up at the same position in the hash table.
- **Adversary**: A malicious actor who tries to choose keys that cause many collisions.

Problem Setup:

- The attacker can **choose the keys**.
- k = number of keys (greater than number of slots m).
- We need a hashing method that makes it **hard for the attacker to force collisions**.

What are the options?

- **(A) Division method**: h(k) = k mod m
  - Simple and fast.
  - Easy for attacker to guess collisions if m is known.
- **(B) Multiplication method**: h(k) = floor(m * (kA mod 1))
  - More complex, less predictable.
  - Harder for attacker to predict collisions, if A is well-chosen.
- **(C) Universal Hashing**: Randomized hashing method.
  - **Best defense** against attackers.
  - Uses a randomly chosen function from a family of hash functions.
  - Since the attacker doesn't know the function, it's **very hard to create collisions**.
- **(D) Mix of division and multiplication depending on whether k is prime**.
  - Irrelevant to the security from attacker; not robust.

Correct Answer: (C) Universal hashing method

Why?

- Since the adversary **knows** the hash function (or can guess it), methods like division/multiplication are **predictable**.
- **Universal hashing** introduces **randomness** — the hash function is selected at random from a family of functions.
- Because the adversary doesn't know **which** hash function will be used, they **cannot force collisions effectively**.

In Simple Words:

You want to store keys in a way that an attacker **can't easily make them crash into each other** (collide).
The best way is to pick your hash function **randomly from a secure pool** — so the attacker can't predict how to cause damage.

That's why **Universal Hashing** is the best answer here.

Question 4

Q.35 The integer value printed by the ANSI-C program given below is _____.

```
#include<stdio.h>

int funcp(){
        static int x = 1;
        x++;
        return x;
}

int main(){
        int x,y;
        x = funcp();
        y = funcp()+x;
        printf("%d\n", (x+y));
        return 0;
}
```

Answer to Question 4

Step-by-step Execution:

1. **First Call: x = funcp();**
    - static int x = 1; → initializes x to 1 **only once**.
    - x++ → x becomes 2.
    - returns 2 → So, x = 2
2. **Second Call: y = funcp() + x;**
    - funcp() again:
        - x is already 2 (static retains its value)
        - x++ → becomes 3
        - returns 3
    - y = 3 + x → y = 3 + 2 = 5
3. **Final Output:**
    - x + y = 2 + 5 = 7

Answer: 7

Question 5

Q.36

Consider the following program:

```
int main()
{
   f1();
   f2(2);
   f3();
   return(0);
}
```

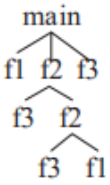
```
int f1()
{
  return(1);
}
```

```
int f2(int X)
{
   f3();
  if (X==1)
    return f1();
  else
    return (X*f2(X-1));
}
```

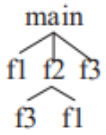
```
int f3()
{
  return(5);
}
```

Which one of the following options represents the activation tree corresponding to the main function?

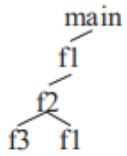
(A)



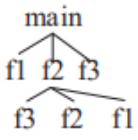
(B)



(C)



(D)



Answer to Question 5

What is an Activation Tree?

An **activation tree** (also known as a **function call tree**) is a diagram that shows **how functions are called** during the execution of a program. Each node in the tree represents a function call. A function A that calls function B will have B as its child node in the tree.

It helps visualize **the order and nesting of function calls**, especially when functions call other functions recursively.

You are asked:

"Which option represents the **activation tree** of this program — i.e., which diagram shows how the function calls happen during execution?"

Let's Simulate What Happens

Here's what happens step-by-step in main():

1. f1() is called (simple, no further calls).
2. f2(2) is called:
    - It calls f3() first.
    - Then X != 1, so it calls f2(1):
        - f3() is called again.
        - X == 1, so it calls f1() again.
3. f3() is called again at the end.

If you look at option **(A)** :

It shows:

main

```
├── f1
├── f2
│   ├── f3
│   └── f2
│       ├── f3
│       └── f1
└── f3
```

This perfectly matches the flow we just described! Final Answer: **Option (A)**

Question 6

Q.46	Let A be a priority queue for maintaining a set of elements. Suppose A is implemented using a max-heap data structure. The operation EXTRACT-MAX(A) extracts and deletes the maximum element from A. The operation INSERT(A,key) inserts a new element key in A. The properties of a max-heap are preserved at the end of each of these operations. When A contains n elements, which one of the following statements about the worst case running time of these two operations is TRUE?
(A)	Both EXTRACT-MAX(A) and INSERT(A,key) run in $O(1)$.
(B)	Both EXTRACT-MAX(A) and INSERT(A,key) run in $O(\log(n))$.
(C)	EXTRACT-MAX(A) runs in $O(1)$ whereas INSERT(A,key) runs in $O(n)$.
(D)	EXTRACT-MAX(A) runs in $O(1)$ whereas INSERT(A,key) runs in $O(\log(n))$.

Answer to Question 6

Question Summary:

You're given:

- A **priority queue A** implemented using a **max-heap**
- Two operations:
    - Extract-Max(A): removes and returns the **maximum element**
    - Insert(A, key): inserts a **new element**
- The question is asking about the **worst-case time complexity** of these two operations when A contains n elements.

Key Concepts: Max-Heap Operations

A **max-heap** is a binary tree where:

- Each parent is ≥ its children
- It's a **complete binary tree** stored as an array

1. Extract-Max(A):

- Removes the root (maximum element).
- To maintain the heap structure:
    - Replace root with last element

- Then **"heapify down"** (i.e., swap down with the larger child)

- This may go all the way from root to leaf.

Worst-case time:
O(log n) ← because height of a binary heap with n nodes is log n

---

2. Insert(A, key):

- Add new key at the end (maintains completeness)
- Then **"heapify up"** (bubble it up while parent < child)
- This may go from the bottom to the root.

Worst-case time:
Also O(log n)

Why Option (B) is correct: (B) Both Extract-Max(A) and Insert(A, key) run in O(log n).

This exactly matches what we just discussed — both operations involve traversing the **height** of a binary heap, which is log n in a complete binary tree.

Why Option (A) is incorrect: (A) Both Extract-Max(A) and Insert(A, key) run in O(1).

- O(1) = constant time
- That would only be true if:
  - We removed the root without reheapifying (which would break the heap)
  - We inserted at the end without restoring heap order

Heap structure would be violated . **Not possible in a valid max-heap**

Why Option (C) is incorrect: (C) Extract-Max(A) runs in O(1) whereas Insert(A, key) runs in O(n).

- Extract-Max is **not O(1)** → we must heapify-down → takes **O(log n)**
- Insert is **not O(n)** → heapify-up only involves log n levels, not n

Incorrect on both counts

Why Option (D) is incorrect: (D) Extract-Max(A) runs in O(1) whereas Insert(A, key) runs in O(log n).

- Insert(A, key) **is correctly O(log n)**
- But **Extract-Max(A) is NOT O(1)** — it's O(log n) due to heapify-down

Half-correct ≠ correct answer

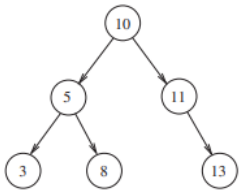
Question 7

Q.47

Consider the C function `foo` and the binary tree shown.

```
typedef struct node {
  int val;
  struct node *left, *right;
} node;

int foo(node *p) {
  int retval;
  if (p == NULL)
    return 0;
  else {
    retval = p->val + foo(p->left) + foo(p->right);
    printf("%d ", retval);
    return retval;
  }
}
```



When `foo` is called with a pointer to the root node of the given binary tree, what will it print?

(A) 3 8 5 13 11 10

(B) 3 5 8 10 11 13

(C) 3 8 16 13 24 50

(D) 3 16 8 50 24 13

Answer to Question 7

What is the question asking?

You are given:

1. A **binary tree** with integer values
2. A **C function named foo**
3. You are asked:

If foo() is called with a pointer to the **root node** of the binary tree (value = 10), **what will be printed?**

In other words:

- Understand the **order in which nodes are visited**
- Understand **what values get printed and when**

Let's Understand the C Code :

```
typedef struct node {
   int val;
   struct node *left, *right;
} node;
```

A basic structure for a binary tree node:

- val is the value stored in the node
- left and right are pointers to the left and right children

Function: foo(node *p)

```
int foo(node *p) {
   int retval;
   if (p == NULL)
      return 0;
   else {
      retval = p->val + foo(p->left) + foo(p->right);
      printf("%d ", retval);
      return retval;
   }
}
```

Let's break this down:

1. **Base Case**: if (p == NULL) return 0;

If the current node is NULL (empty), return 0.

2. **Recursive Step**:

retval = p->val + foo(p->left) + foo(p->right);

This is the key line!

- It **recursively sums**:
    - the value of the **current node**
    - the result of **left subtree**
    - the result of **right subtree**

So it's doing a **post-order traversal** where:

- You process the **left subtree**
- Then the **right subtree**
- Then calculate and print the total for the **current node**

3. **Print and Return**:

   printf("%d ", retval);

   return retval;

- It prints the **sum of the subtree rooted at p**
- Then returns this sum to the parent call

The Tree Structure

```
        10
      /    \
     5      11
    / \      \
   3   8      13
```

Let's label:

- Root: 10
- Left subtree: 5 → 3, 8
- Right subtree: 11 → 13

How foo() Executes (Post-order traversal)

Let's simulate the recursion:

Node 3:

- Left & right are NULL → returns 3
- Prints: 3

Node 8:

- Left & right are NULL → returns 8
- Prints: 8

Node 5:

- Left = 3, Right = 8 → total = 5 + 3 + 8 = **16**
- Prints: 16

Node 13:

- Left & right NULL → returns 13
- Prints: 13

Node 11:

- Left = NULL, Right = 13 → total = 11 + 0 + 13 = **24**
- Prints: 24

Node 10 (root):

- Left = 16, Right = 24 → total = 10 + 16 + 24 = **50**
- Prints: 50

Final Output Order: 3 8 16 13 24 50

This matches **Option (C)**.

Summary

What they're asking:

"What gets printed when the function foo() is called on the root node?"

What the code does:

- **Post-order traversal**
- Calculates the **sum of subtree values**
- **Prints the subtree sum at each node**, bottom-up

Correct Answer: (C) 3 8 16 13 24 50

Question 8

Q.57

Consider the following two-dimensional array `D` in the C programming language, which is stored in row-major order:

```
int D[128][128];
```

Demand paging is used for allocating memory and each physical page frame holds 512 elements of the array `D`. The Least Recently Used (LRU) page-replacement policy is used by the operating system. A total of 30 physical page frames are allocated to a process which executes the following code snippet:

```
for (int i = 0; i < 128; i++)
   for (int j = 0; j < 128; j++)
         D[j][i] *= 10;
```

The number of page faults generated during the execution of this code snippet is _____.

Answer to Question 8

What is demand paging?

- It's a **virtual memory** concept.
- **Pages of a program are not loaded into memory until they're actually used.**
- If the program tries to access something **not currently in memory**, the **operating system raises a "page fault"**.
- Then the OS **loads the required page from disk into RAM**.

Analogy:

Imagine a huge book. Instead of carrying the entire book, you only carry the chapters you're currently reading. If you need another chapter, you go fetch it.

So, in your program — **only the parts of the array that are actively being used get loaded into RAM**.

"Each physical page frame holds 512 elements of the array"

What is a physical page frame?

- It's a **fixed-size block** of physical memory (RAM).
- Think of RAM being divided into equally-sized **slots**.
- Each of these slots is called a **page frame**.

What does "holds 512 elements" mean?

- The array is made of ints (integers).

- Each frame (slot in memory) can store **512 integer values** from the array.
- So, if you load part of the array into memory, **512 integers (elements) come in at once.**

For Example:

If your array is D[128][128], that's 16,384 integers total.
Each page frame holds 512 integers.
So, to store the entire array, you'd need:
16,384 / 512 = 32 page frames.

But your OS only gives you **30 page frames** to work with. So at any time, **2 pages won't fit**, and **LRU (Least Recently Used) policy** is used to evict old ones.

Putting it all together:

- The array D[128][128] is **too big to fit entirely in RAM**.
- The OS uses **demand paging** to load only parts of the array **when needed**.
- Each **page fault** happens when a part of the array is accessed but **not currently in memory**.
- And each time, **a block of 512 integers is loaded** into a **page frame**.

How does RAM work here?

RAM = Main Memory

- RAM is **fast memory** where your program runs.
- But it is **limited** in size (e.g., 8 GB, 16 GB).
- So it can't store **everything** at once.

What Happens:

1. The program starts. The OS puts some **initial pages** into RAM.
2. The program accesses **new data or code**.
3. If it's not already in RAM → **Page fault** occurs.
4. OS takes that missing page from **disk** (virtual memory) → loads it into **RAM**.
5. If RAM is full, the OS may **remove (evict)** an old page from RAM using a policy like LRU (Least Recently Used).

What is a Page Fault?

A **page fault** occurs when:

The program **tries to access a page** that is **not in RAM**.

Here's what happens in a page fault:

1. The CPU tries to access memory (e.g., a variable).
2. The **Memory Management Unit (MMU)** checks the page table.
3. It finds that the required page is **not in RAM**.
4. **Page Fault triggered** — like a red flag!
5. OS **pauses the program**, goes to the **disk**, and **loads that page** into RAM.
6. If RAM is full → OS may **remove an old page** (like closing a tab in a browser).
7. The page is loaded, and the program **resumes exactly where it left off**.

Analogy:

Imagine you're reading a book on a tablet with limited memory.

- You start reading Chapter 1 — it's loaded in memory.
- You scroll to Chapter 3 — the tablet goes: "Oh! That's not loaded yet."
- It downloads Chapter 3 from the internet (like fetching a page from disk).
- It might remove Chapter 1 if memory is full.

That "download" = **page fault**.

Now the actual answer is 4096.

What the Official Answer is Assuming:

They're assuming:

Each page holds 512 elements

Each row has 128 elements

So: 4 rows fit per page (since 512/128 = 4)

Total rows = 128 → so:

128 / 4 = 32 pages are needed to store all rows

Now look at the access:

D[j][i] → access 1 element from each row

So in 1 column, we access 128 rows = 32 pages (again)

So: in each of the 128 outer loop iterations, 32 page accesses are made.

Because only 30 pages fit in memory:

In every column, 2 pages must be evicted (as only 30 can stay in RAM)

So every column iteration causes 2 new page faults

But this is not how the answer key got 4096.

Here's how they did it:

Each page holds **512 elements**.

Each **access** is to a single element, and there are **16,384 accesses** (128 × 128).

So:

- Each **access** may trigger a page fault, but once a page is loaded, other accesses to it may not.

Because the access pattern jumps across rows (D[j][i]), and each page stores **4 rows**, every **i iteration** accesses 128 rows → thus **32 pages**.

So:

- Each i (128 total) → accesses **32 pages**
- So: **total page accesses = 128 × 32 = 4096**
- All are **page faults**, because LRU can't retain them all (only 30 pages)

Answer = 4096 page faults

Final Summary:

- Each column access = 128 elements → 32 different pages
- 128 such columns → **128 × 32 = 4096**
- That's the total number of page faults when every access triggers a page fault (due to poor locality)
- **Answer = 4096**

Question 9

Q.59	Consider a sequence a of elements $a_0 = 1$, $a_1 = 5$, $a_2 = 7$, $a_3 = 8$, $a_4 = 9$, and $a_5 = 2$. The following operations are performed on a stack S and a queue Q, both of which are initially empty. I: `push` the elements of a from a_0 to a_5 in that order into S. II: `enqueue` the elements of a from a_0 to a_5 in that order into Q. III: `pop` an element from S. IV: `dequeue` an element from Q. V: `pop` an element from S. VI: `dequeue` an element from Q. VII: `dequeue` an element from Q and `push` the same element into S. VIII: Repeat operation VII three times. IX: `pop` an element from S. X: `pop` an element from S. The `top` element of S after executing the above operations is _____.

Answer to Question 9

You are given:

- A sequence of elements:
  a = [1, 5, 7, 8, 9, 2]
  (These are a_0 to a_5)
- Two data structures:
  - **Stack** S (Last-In-First-Out)
  - **Queue** Q (First-In-First-Out)
- Initially both S and Q are empty.

You are told to:

- Perform a sequence of operations (I through X)
- After all operations, tell **what is the top element of the stack S**

Step-by-step execution

Step I: Push a_0 to a_5 into S

Sequence: 1, 5, 7, 8, 9, 2 → push into **Stack S**

Since stack is **LIFO**, top is the last pushed.

After step I:
S = [1, 5, 7, 8, 9, 2] ← 2 is on top

Step II: Enqueue a_0 to a_5 into Q

Sequence: 1, 5, 7, 8, 9, 2 → enqueue into **Queue Q**

Since queue is **FIFO**, front is the first enqueued.

After step II:
Q = [1, 5, 7, 8, 9, 2] ← 1 is at front

Step III: Pop from S

S.pop() → removes 2
Now: S = [1, 5, 7, 8, 9]

Step IV: Dequeue from Q

Q.dequeue() → removes 1
Now: Q = [5, 7, 8, 9, 2]

Step V: Pop from S

S.pop() → removes 9
Now: S = [1, 5, 7, 8]

Step VI: Dequeue from Q

Q.dequeue() → removes 5
Now: Q = [7, 8, 9, 2]

Step VII: Dequeue from Q and Push to S

Dequeue → 7, Push to S
Now:

- Q = [8, 9, 2]
- S = [1, 5, 7, 8, 7]

Step VIII: Repeat VII three times

1. Dequeue 8, Push to S → S = [1, 5, 7, 8, 7, 8], Q = [9, 2]
2. Dequeue 9, Push to S → S = [1, 5, 7, 8, 7, 8, 9], Q = [2]
3. Dequeue 2, Push to S → S = [1, 5, 7, 8, 7, 8, 9, 2], Q = []

Step IX: Pop from S

S.pop() → removes 2
Now: S = [1, 5, 7, 8, 7, 8, 9]

Step X: Pop from S

S.pop() → removes 9
Now: S = [1, 5, 7, 8, 7, 8]

Final Answer:

Top element of S = 8

Answer: 8