LP1 Practical

1. **Problem Statement 1: Two-Pass Assembler**

Design suitable Data structures and implement Pass-I and Pass-II of a two-pass assembler for pseudo-machine. Implementation should consist of a few instructions from each category and few assembler directives. The output of Pass-I (intermediate code file and symbol table) should be input for Pass-II.

🔷 BASIC CONCEPTS

Q1. What is an assembler?

A: An assembler translates **assembly language code** into **machine code (object code)** understood by the CPU.

Q2. Why is it called a "Two-Pass" assembler?

A: Because the assembler scans the source program **twice**:

1. **Pass 1:** Builds symbol and literal tables, generates **Intermediate Code (IC)**.
2. **Pass 2:** Uses IC, SYMTAB, and LITTAB to generate **final machine code**.

Q3. What are the main outputs of Pass 1?

A:

- **Intermediate Code (IC.txt)**
- **Symbol Table (SYMTAB.txt)**
- **Literal Table (LITTAB.txt)**

Q4. What is generated after Pass 2?

A: The **Machine Code (MACHINECODE.txt)** file — it contains the actual translated binary-like output.

Q5. Why is an assembler implemented in two passes instead of one?

A:

- Because during the first pass, **symbol addresses are not yet known**.
- In the second pass, after building SYMTAB and LITTAB, **actual addresses can be resolved**.

🔷 PASS 1 — Functionality Questions

Q6. What is the role of Pass 1?

A:

- Assign addresses to all symbols and literals.
- Build **SYMTAB** and **LITTAB**.
- Generate **Intermediate Code (IC)** for Pass 2.

Q7. What is the function of the Location Counter (LC)?

A: LC keeps track of **the next memory address** to be assigned to instructions and data.

Q8. What is the purpose of SYMTAB (Symbol Table)?

A: SYMTAB stores:

- Each **symbol name**
- Its **memory address**

Example:

```
RESULT 203
```

Q9. What is the purpose of LITTAB (Literal Table)?

A: LITTAB stores:

- Each **literal constant** (like `=5`)
- Its **assigned memory address**

Example:

```
=5 204
```

Q10. What is the function of Intermediate Code (IC)?

A: IC is a **machine-independent representation** generated after Pass 1,

used by Pass 2 to produce machine code.

Example:

```
(IS,04)(1)(S,A)
```

Q11. Why do we add 1 to the literal index when generating IC?

A: Because arrays/lists in Java are **0-indexed**, but literal indices in assembler listings **start from 1**.

Q12. How are labels handled in Pass 1?

A: If the first word of a line is **not a mnemonic**, it is treated as a **label**,

and its address (LC value) is added to SYMTAB.

🔷 PASS 2 — Functionality Questions

Q13. What is the role of Pass 2?

A:

- Reads **IC.txt**, **SYMTAB.txt**, and **LITTAB.txt**.
- Replaces symbolic references with **actual memory addresses**.
- Produces **MACHINECODE.txt**.

Q14. Why do we need SYMTAB and LITTAB again in Pass 2?

A: To **look up the addresses** of symbols and literals while converting IC into machine code.

Q15. How does Pass 2 convert IC to machine code?

A:

- Extracts the opcode (like 04 for MOVER).
- Finds the operand address from SYMTAB or LITTAB.
- Combines both to generate a line like:

```
+ 04 1 203
```

Q16. Why do we use substring functions (like `line.substring(4,6)`) in Pass 2?

A: To **extract opcode and register fields** from the formatted Intermediate Code strings.

Q17. What is the purpose of `String.format("%03d", address)`?

A: It ensures all memory addresses are printed as **3-digit numbers** (e.g., 005, 203).

Q18. Why do we print `(IS,04)` and `(DL,02)` in Intermediate Code?

A: `(IS)` = **Imperative Statement**, `(DL)` = **Declarative Statement**

The number represents its **opcode** (from instruction set).

🔷 INSTRUCTION & DIRECTIVE QUESTIONS

Q19. What is the difference between an Instruction and a Directive?

A:

- **Instruction:** Generates machine code (e.g., MOVER, ADD).
- **Directive:** Controls the assembler, doesn't generate code (e.g., START, END, DS, DC).

Q20. What does the START directive do?

A: Sets the **initial value** of the Location Counter (LC).

Example: `START 200 → LC = 200`

Q21. What does the END directive do?

A: Indicates **end of program**, and triggers **literal address assignment**.

Q22. What does DS mean?

A: DS (Define Storage) → Reserves memory locations for variables.

Example:

```
RESULT DS 1
```

Reserves 1 memory unit.

Q23. What are literals and how are they recognized?

A: Literals are constants written as `=value`.

Assembler automatically allocates memory for them at program end or LTORG.

🔷 CODE-SPECIFIC QUESTIONS

Q24. What does `isMnemonic()` do in your Pass 1 code?

A: Checks if a given token is a **valid mnemonic** (like ADD, MOVER, etc.) or not.

Q25. What is the purpose of the `code()` function in Pass 1?

A: Returns the **numeric opcode** corresponding to a mnemonic.

Example: MOVER → 04, MOVEM → 05, ADD → 01.

Q26. What is the purpose of `(AD,01)` and `(IS,04)` in IC?

A:

- `(AD)` → Assembler Directive (like START, END)
- `(IS)` → Imperative Statement (like MOVER, ADD)

Q27. Why do you use LinkedHashMap for SYMTAB?

A: LinkedHashMap keeps **insertion order**, ensuring symbols are printed in the same order they appear in the source.

Q28. What is the difference between SYMTAB and LITTAB data structures in your code?

A:

- SYMTAB = **Map<Symbol, Address>**
- LITTAB = **List of literals** assigned addresses later.

Q29. Why is `addr` initialized to LC while writing LITTAB?

A: Because literals are **assigned memory addresses starting from the current LC** at the end of Pass 1.

Q30. What happens if a symbol appears but isn't declared (e.g., used before definition)?

A: It's added to SYMTAB with address **0** initially, and will be updated when defined.

🔷 EXAMPLE AND OUTPUT QUESTIONS

Q31. What input program did you use?

```
START 200
MOVER AREG, A
ADD AREG, =5
MOVEM AREG, RESULT
RESULT DS 1
END
```

Q32. What will the generated Intermediate Code (IC.txt) look like?

```
(AD,01)(C,200)
(IS,04)(1)(S,A)
(IS,01)(1)(L,1)
(IS,05)(1)(S,RESULT)
(DL,02)(C,1)
```

```
(AD,02)
```

Q33. What will the SYMTAB.txt contain?

```
A 200
RESULT 203
```

Q34. What will the LITTAB.txt contain?

```
=5 204
```

Q35. What will the MACHINECODE.txt contain?

```
+ 04 1 200
+ 01 1 204
+ 05 1 203
```

🔷 ADVANCED CONCEPTUAL QUESTIONS

Q36. What is the difference between one-pass and two-pass assembler?

A:

- **One-pass:** Generates machine code directly (used in simple systems).
- **Two-pass:** Generates IC first, then machine code (used for complex programs with forward references).

Q37. What are forward references?

A: A reference to a **symbol defined later** in the program.

Two-pass assemblers handle them easily because addresses are known in the second pass.

Q38. What is the purpose of a Pool Table (POOLTAB)?

A: (In extended assembler versions) It indicates where a **new literal pool** starts — used when multiple LTORGs are present.

Q39. What are the classes of assembler statements?

A:

1. **Imperative Statements (IS)** — executable instructions
2. **Declarative Statements (DL)** — memory definitions
3. **Assembler Directives (AD)** — control assembler behavior

Q40. What are the benefits of using Intermediate Code (IC)?

A:

- Simplifies second pass.
- Makes debugging easier.
- Provides separation between syntax processing and code generation.

Symbol Table (SYMTAB)	Stores **symbol names** and their **addresses**.	RESULT 203

Literal Table (LITTAB)	Stores **literals (constants)** and their **assigned addresses**.	=5 204

Location Counter (LC)	Keeps track of the **current memory address** during assembly.	LC = 200 initially (from START)

Directive	A command that instructs the assembler but **doesn't produce machine code**.	START, END, DS, DC

Instruction	A statement that **produces machine code**.	MOVER, ADD, MOVEM

Opcode	The numeric code that represents an operation.	MOVER → 04, ADD → 01

Mnemonic	A short symbolic name for an operation.	ADD, SUB, MOVER, MOVEM

🧩 2. Directives (AD - Assembler Directives)

Directive	Full Form	Function	Example
START	Start of Program	Initializes LC with given address.	`START 200` → LC = 200
END	End of Program	Marks end of program; triggers literal assignment.	`END`
DS	Define Storage	Reserves memory space for variable(s).	`RESULT DS 1`
DC	Define Constant	Allocates space and assigns constant value.	`NUM DC 5`
LTORG	Literal Origin (not in your example but common)	Assigns memory to literals at that point.	

⚙️ 3. Imperative Statements (IS)

Instruction	Meaning	Example	IC Format
MOVER	Move contents of memory to register.	`MOVER AREG, A`	`(IS,04)(1)(S,A)`
MOVEM	Move contents of register to memory.	`MOVEM AREG, RESULT`	`(IS,05)(1)(S,RESULT)`
ADD	Add contents of memory to register.	`ADD AREG, =5`	`(IS,01)(1)(L,1)`
STOP	Stop program execution.	`STOP`	`(IS,00)`

Problem Statement 2: Two-Pass Macro Processor

Design suitable data structures and implement Pass-I and Pass-II of a two-pass macroprocessor. The output of Pass-I (MNT, MDT and intermediate code file without any macro definitions) should be input for Pass-II.

🧩 Conceptual Questions

Q1. What is a macro processor?

A: A macro processor is a program that expands macros in assembly language before actual assembly takes place. It replaces macro calls with the corresponding set of assembly instructions defined by the macro.

Q2. What is a macro?

A: A macro is a sequence of assembly instructions grouped under a single name. It allows repetitive code to be written once and reused multiple times.

Q3. What is the difference between a macro and a subroutine?

Macro	Subroutine
Code is expanded **inline** during assembly.	Control is transferred to subroutine using call instruction.
No runtime overhead.	Involves call and return overhead.
Handled by **assembler**.	Handled by **linker/loader**.
Used for **short repetitive code**.	Used for **larger reusable modules**.

Q4. What is the need for a two-pass macro processor?

A: Because during the first pass, macros are **defined**, and during the second pass, macros are **expanded**.

Both tasks cannot be done in one pass since expansion requires full macro definition, which may appear later in the program.

Q5. What happens in Pass 1 of a macro processor?

A:

Pass 1 performs:

1. Identification of macro definitions.
2. Construction of:
    - **MNT (Macro Name Table)**
    - **MDT (Macro Definition Table)**
3. Produces **Intermediate Code** that excludes macro definitions.

Q6. What happens in Pass 2 of a macro processor?

A:

Pass 2 expands all macro calls found in the intermediate code using **MNT** and **MDT**, replacing formal parameters with actual arguments.

Q13. What do the files MNT.txt, MDT.txt, IC.txt, and ExpandedCode.txt contain?

File	Description
MNT.txt	Macro Name Table (macro name + MDT index).
MDT.txt	Full macro definition (prototype, body, MEND).
IC.txt	Intermediate Code without macro definitions.
ExpandedCode.txt	Final expanded program after Pass II.

Q9. What is ALA (Argument List Array)?

A: A mapping between **formal arguments** and **actual arguments**

FCFS

```
import java.util.*;

class FCFS {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of processes: ");
        int n = sc.nextInt();

        int[] pid = new int[n]; // Process IDs
        int[] at = new int[n];  // Arrival Times
        int[] bt = new int[n];  // Burst Times
        int[] wt = new int[n];  // Waiting Times
        int[] tat = new int[n]; // Turnaround Times
```

```
        int[] ct = new int[n];  // Completion Times

        // Input process details
        for (int i = 0; i < n; i++) {
            System.out.print("Process ID: ");
            pid[i] = sc.nextInt();
            System.out.print("Arrival Time: ");
            at[i] = sc.nextInt();
            System.out.print("Burst Time: ");
            bt[i] = sc.nextInt();
            System.out.println();
        }

        // Sort processes by Arrival Time
        for (int i = 0; i < n - 1; i++) {
            for (int j = i + 1; j < n; j++) {
                if (at[i] > at[j]) {
                    // Swap arrival times
                    int temp = at[i]; at[i] = at[j]; at[j] = temp;
                    // Swap burst times
                    temp = bt[i]; bt[i] = bt[j]; bt[j] = temp;
                    // Swap process IDs
                    temp = pid[i]; pid[i] = pid[j]; pid[j] = temp;
                }
            }
        }

        // Calculate CT, TAT, WT
        int time = 0;
        double avgWT = 0, avgTAT = 0;

        for (int i = 0; i < n; i++) {
            if (time < at[i]) time = at[i]; // CPU idle
            time += bt[i];
            ct[i] = time;
            tat[i] = ct[i] - at[i];
```

```
            wt[i] = tat[i] - bt[i];
            avgWT += wt[i];
            avgTAT += tat[i];
        }

        // Display results
        System.out.println("\nPID\tAT\tBT\tWT\tTAT\tCT");
        for (int i = 0; i < n; i++) {
            System.out.println(pid[i] + "\t" + at[i] + "\t" + bt[i] + "\t" + wt[i] + "\t" +
tat[i] + "\t" + ct[i]);
        }

        // Gantt Chart with time scale
        System.out.println("\nGantt Chart:");
        System.out.print("|");
        for (int i = 0; i < n; i++) {
            System.out.print(" P" + pid[i] + " |");
        }
        System.out.println();

        System.out.print("0");
        for (int i = 0; i < n; i++) {
            System.out.print("   " + ct[i]);
        }
        System.out.println();

        // Averages
        System.out.printf("\nAverage Waiting Time:", avgWT / n);
        System.out.printf("Average Turnaround Time:", avgTAT / n);
    }
}
```

Section 1: Basic Conceptual Questions

Q1. What is CPU Scheduling?

A:

CPU scheduling is the process of selecting which process in the ready queue will be executed next by the CPU, to make efficient use of processor time.

Preemptive Scheduling

> In preemptive scheduling, the CPU can be taken away (preempted) from a process before it finishes if a higher-priority or more urgent process arrives.

Non-Preemptive Scheduling

> In non-preemptive scheduling, once a process starts execution, it cannot be interrupted until it completes.

Q2. What is FCFS scheduling?

A:

FCFS stands for **First Come First Serve**.

It is a **non-preemptive scheduling algorithm** where the process that arrives first is executed first.

It works like a **FIFO (First In First Out)** queue.

Q3. Is FCFS preemptive or non-preemptive?

A:

Non-preemptive.

Once a process starts execution, it runs until completion before another process can start.

Q4. What is the main criterion used in FCFS scheduling?

A:

Arrival Time — the process that arrives first in the ready queue is executed first.

Q5. What is meant by “ready queue”?

A:

The ready queue is the list of processes that are ready to execute and waiting for CPU time.

Q6. What is meant by CPU idle time?

A:

When no process is available to run (i.e., CPU is waiting for the next process to arrive), it remains idle.

⚙️ Section 2: Program-Specific Questions

Q7. What inputs does your program take?

A:

It takes:

- Process ID
- Arrival Time
- Burst Time

Q8. What outputs does your program produce?

A:

- **Waiting Time (WT)**
- **Turnaround Time (TAT)**
- **Completion Time (CT)**
- **Average WT and TAT**
- **Gantt Chart**
- **Burst Time**: “How long the process runs on CPU.”
- **Completion Time**: “When the process finishes — including any waiting before it starts.”

Q9. What is the Gantt Chart?

A:

A Gantt chart is a visual representation of the process execution timeline.

It shows the order and duration of each process's execution.

Example:

```
| P1 | P2 | P3 |
0    5    9    14
```

Q10. How do you calculate the following?

- **Turnaround Time (TAT)** = Completion Time − Arrival Time
- **Waiting Time (WT)** = Turnaround Time − Burst Time
- **Completion Time (CT)** = Time when process finishes execution

---

Q11. What happens if the CPU is idle?

A:

If a process has not yet arrived, the CPU will stay idle until that process arrives.

This is handled in the code using:

```
if (time < at[i]) time = at[i];
```

Q12. What data structures are used?

A:

- Arrays for process details (`pid` , `at` , `bt` , `wt` , `tat` , `ct`)
- Loops and sorting logic for ordering by arrival time.

Q13. Why do you sort by Arrival Time?

A:

To ensure the processes are executed in the order they arrive, as FCFS depends on arrival order.

🧩 Section 3: Analytical / Calculation-Based Questions

Q14. What is Turnaround Time (TAT)?

A:

The total time taken from process arrival to its completion.

It includes waiting + execution time.

Q15. What is Waiting Time (WT)?

A:

The total time a process spends waiting in the ready queue before it starts execution.

Q16. How do you calculate average waiting time and turnaround time?

A:

Average Waiting Time=∑WTn\text{Average Waiting Time} = \frac{\sum WT}{n}

Average Waiting Time=n∑WT

Average Turnaround Time=∑TATn\text{Average Turnaround Time} = \frac{\sum TAT}{n}

Average Turnaround Time=n∑TAT

Q17. Why can FCFS cause high waiting time for some processes?

A:

Because of the **Convoy Effect** — when a long process delays all others behind it.

Q18. What is the Convoy Effect?

A:

It occurs when small, short processes must wait for a long process to finish — reducing CPU efficiency.

Q19. Can FCFS cause starvation?

A:

No, because every process eventually gets CPU time in the order they arrive.

Q23. What happens if two processes have the same arrival time?

A:

The process entered first (or with the smaller PID in the code) will be executed first.

Q24. Why is FCFS not used in modern systems?

A:

Because it leads to **poor average waiting time** and is **not responsive** for short or interactive processes.

- **Problem Statement 2: Two-Pass Macro Processor**

Design suitable data structures and implement Pass-I and Pass-II of a two-pass macroprocessor. The output of Pass-I (MNT, MDT and intermediate code file without any macro definitions) should be input for Pass-II.

```
import java.util.*;

class SJFPreemptive {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
```

```
// Input: Number of processes
System.out.print("Enter number of processes: ");
int n = sc.nextInt();

int[] pid = new int[n]; // Process IDs
int[] at = new int[n];  // Arrival Times
int[] bt = new int[n];  // Burst Times
int[] rt = new int[n];  // Remaining Times
int[] wt = new int[n];  // Waiting Times
int[] tat = new int[n]; // Turnaround Times

// Input: Process details
for (int i = 0; i < n; i++) {
    System.out.print("Process ID: ");
    pid[i] = sc.nextInt();
    System.out.print("Arrival Time: ");
    at[i] = sc.nextInt();
    System.out.print("Burst Time: ");
    bt[i] = sc.nextInt();
    rt[i] = bt[i]; // Initialize remaining time
    System.out.println();
}

int complete = 0, time = 0;
int minRemaining, shortest = -1;
boolean found;
ArrayList<Integer> order = new ArrayList<>();

// SJF Preemptive Scheduling Loop
while (complete < n) {
    minRemaining = Integer.MAX_VALUE;
    found = false;

    for (int i = 0; i < n; i++) {
        if (at[i] <= time && rt[i] > 0 && rt[i] < minRemaining) {
```

```
                minRemaining = rt[i];
                shortest = i;
                found = true;
            }
        }

        if (!found) {
            time++;
            continue;
        }

        order.add(pid[shortest]); // Record execution order
        rt[shortest]--;
        time++;

        if (rt[shortest] == 0) {
            complete++;
            int finish = time;
            tat[shortest] = finish - at[shortest];
            wt[shortest] = tat[shortest] - bt[shortest];
            if (wt[shortest] < 0) wt[shortest] = 0;
        }
    }

    // Output: Process table
    double avgWT = 0, avgTAT = 0;
    System.out.println("\nPID\tAT\tBT\tWT\tTAT");
    for (int i = 0; i < n; i++) {
        avgWT += wt[i];
        avgTAT += tat[i];
        System.out.println(pid[i] + "\t" + at[i] + "\t" + bt[i] + "\t" + wt[i] + "\t" +
tat[i]);
    }

    // Output: Execution order (Gantt-style)
    System.out.println("\nExecution Order:");
```

```
        for (int x : order) {
            System.out.print("P" + x + " ");
        }

        // Output: Averages
        System.out.printf("\n\nAverage Waiting Time: %.2f\n", avgWT / n);
        System.out.printf("Average Turnaround Time: %.2f\n", avgTAT / n);
    }
}
```

1. Basic Conceptual Questions

Q1. What is SJF Scheduling?

A:

Shortest Job First (SJF) is a CPU scheduling algorithm that selects the process with the **smallest burst time** (shortest job) for execution next.

Q2. What is meant by “preemptive” in SJF Preemptive?

A:

Preemptive means the CPU can **interrupt** the currently running process if a **new process arrives** with a **shorter remaining burst time**.

So, the CPU always executes the process with **minimum remaining time**.

Q3. What is another name for SJF Preemptive?

A:

Shortest Remaining Time First (SRTF) scheduling.

Q4. What is the main goal of SJF?

A:

To **minimize the average waiting time** and **turnaround time** of all processes.

Q5. Is SJF optimal?

A:

Yes, **SJF gives the minimum average waiting time**, but only if we know burst times in advance — which is not always possible in real systems.

Q6. What are the parameters required for SJF Preemptive?

A:

- Arrival Time (AT)
- Burst Time (BT)
- Remaining Time (RT)
- Waiting Time (WT)
- Turnaround Time (TAT)

Q7. What is the difference between SJF Preemptive and Non-Preemptive?

Feature	**SJF Preemptive**	**SJF Non-Preemptive**
CPU can be interrupted?	✅ Yes	❌ No
Selection criteria	Shortest remaining burst time	Shortest total burst time
Responsiveness	Higher	Lower
Implementation	More complex	Simpler

Q8. What is meant by “remaining time” in this program?

A:

It represents how much burst time is **still left** for a process to complete.

It decreases by 1 each time unit in the preemptive scheduling loop.

⚙️ 2. Code Understanding Questions

Q9. What does this part of the code do?

```
if (at[i] <= time && rt[i] > 0 && rt[i] < minRemaining)
```

A:

It finds the process that has **arrived**, is **not completed**, and has the **minimum remaining time** at the current clock time.

Q10. What happens if no process has arrived yet?

```
if (!found) {
    time++;
    continue;
}
```

A:

The CPU **remains idle**, and the system clock `time` is incremented until a process arrives.

Q11. What is `order.add(pid[shortest]);` used for?

A:

It records the **execution order** of processes to display a **Gantt chart-style output** later.

Q12. What does this mean?

```
rt[shortest]--;
```

A:

It simulates **one unit of execution time** — reducing the remaining burst time of the running process by 1.

Q13. What happens when `rt[shortest] == 0`?

A:

The process is **completed**.

We calculate its **turnaround time (TAT)** and **waiting time (WT)**, and increment the `complete` counter.

Q14. What are these formulas?

Parameter	Formula	Meaning
Turnaround Time (TAT)	Finish Time − Arrival Time	Total time from arrival to completion
Waiting Time (WT)	TAT − Burst Time	Time spent waiting in the ready queue

---

Q15. Why do we check this condition?

```
if (wt[shortest] < 0) wt[shortest] = 0;
```

A:

It ensures the waiting time is not negative.

Some processes may start immediately on arrival, so their waiting time is 0.

Q16. What does this variable do?

```
int complete = 0;
```

A:

It counts how many processes have finished execution.

When `complete == n`, the scheduling is done.

Q17. Why do we use `ArrayList<Integer> order`?

A:

To **store the sequence** of process execution dynamically (since preemptive scheduling switches frequently).

Q18. What is the time complexity of this code?

A:

$O(n^2)$ approximately, since for every time unit, the program checks all processes to find the one with the shortest remaining time.

📊 3. Output and Gantt Chart Questions

Q19. What does the "Execution Order" output show?

A:

It displays the **sequence of processes** that got CPU time — similar to a Gantt chart.

For example:

```
Execution Order:
P1 P2 P1 P3
```

This means the CPU first ran P1, then switched to P2, then back to P1, then to P3.

Q20. How are average waiting and turnaround times calculated?

A:

Average WT=∑WTin\text{Average WT} = \frac{\sum WT_i}{n}

Average WT=n∑WTi

Average TAT=∑TATin\text{Average TAT} = \frac{\sum TAT_i}{n}

Average TAT=n∑TATi

These are printed using:

```
System.out.printf("Average Waiting Time: %.2f", avgWT/n);
```

Q21. How does SJF Preemptive improve CPU performance?

A:

It ensures shorter processes finish quickly, **reducing average waiting time** and **increasing throughput**.

Q22. Can SJF Preemptive cause starvation?

A:

Yes ❗

If short processes keep arriving, **long processes may never get CPU time** — this is called **starvation**.

Q23. How can starvation be avoided?

A:

By using **Aging** — gradually increasing the priority of waiting processes.

Q24. Why is this algorithm more complex than FCFS?

A:

Because the CPU must **continuously check remaining burst times** and **preempt** when a shorter process arrives — meaning more comparisons and frequent context switching.

🧩 4. Real-world and Conceptual Depth

Q25. What kind of systems use preemptive scheduling?

A:

Preemptive scheduling is used in **real-time** and **multitasking operating systems**, where response time is important.

Q26. What are the advantages of SJF Preemptive?

✅ Minimum average waiting time

✅ Higher throughput

✅ Better response for short processes

Q27. What are the disadvantages?

❌ Requires knowing burst time in advance

❌ High overhead due to frequent preemption

❌ May cause starvation

Problem Statement 5: Priority Scheduling (Non-Preemptive)

Write a program to simulate the **Priority Scheduling (Non-Preemptive)** algorithm.

Each process should have an associated **priority value**, and the scheduler should select the process with the **highest priority** for execution next. Compute and display the **waiting time**, **turnaround time**, and **average times** for all processes.

```
import java.util.*;

class PriorityNonPreemptive {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of processes: ");
        int n = sc.nextInt();

        int[] pid = new int[n];  // Process IDs
        int[] at = new int[n];   // Arrival Times
```

```
        int[] bt = new int[n];   // Burst Times
        int[] pr = new int[n];   // Priorities
        int[] wt = new int[n];   // Waiting Times
        int[] tat = new int[n];  // Turnaround Times
        boolean[] completed = new boolean[n]; // Completion flags

        // Input process details
        for (int i = 0; i < n; i++) {
            System.out.print("Process ID: ");
            pid[i] = sc.nextInt();
            System.out.print("Arrival Time: ");
            at[i] = sc.nextInt();
            System.out.print("Burst Time: ");
            bt[i] = sc.nextInt();
            System.out.print("Priority (lower = higher priority): ");
            pr[i] = sc.nextInt();
            System.out.println();
        }

        int complete = 0, time = 0;
        ArrayList<Integer> order = new ArrayList<>();

        // Priority Scheduling (Non-Preemptive)
        while (complete < n) {
            int idx = -1;
            int highestPriority = Integer.MAX_VALUE;

            for (int i = 0; i < n; i++) {
                if (!completed[i] && at[i] <= time && pr[i] < highestPriority) {
                    highestPriority = pr[i];
                    idx = i;
                }
            }

            if (idx == -1) { // No process has arrived yet
                time++;
```

```
            continue;
        }

        // Execute selected process fully (non-preemptive)
        order.add(pid[idx]);
        time += bt[idx];
        tat[idx] = time - at[idx];
        wt[idx] = tat[idx] - bt[idx];
        completed[idx] = true;
        complete++;
    }

    // Calculate averages
    double avgWT = 0, avgTAT = 0;
    System.out.println("\nPID\tAT\tBT\tPR\tWT\tTAT");
    for (int i = 0; i < n; i++) {
        avgWT += wt[i];
        avgTAT += tat[i];
        System.out.println(pid[i] + "\t" + at[i] + "\t" + bt[i] + "\t" + pr[i] + "\t" +
wt[i] + "\t" + tat[i]);
    }

    avgWT /= n;
    avgTAT /= n;

    // Display Execution Order
    System.out.println("\nExecution Order:");
    for (int x : order) {
        System.out.print("P" + x + " ");
    }

    // Display Averages
    System.out.printf("\n\nAverage Waiting Time:"+ avgWT);
    System.out.printf("\nAverage Turnaround Time:"+ avgTAT);
```

```
    }
  }
```

🔷 Basic Concept

Q1: What is Priority Scheduling?

A: In Priority Scheduling, each process is assigned a priority. The CPU is allocated to the process with the **highest priority (lowest number in this case)**.

Q2: What is the difference between *preemptive* and *non-preemptive* priority scheduling?

A:

- **Preemptive:** If a higher-priority process arrives, it can interrupt the currently running process.
- **Non-Preemptive:** Once a process starts executing, it **cannot be stopped** until it finishes, even if a higher-priority process arrives.

Q3: What does "lower value = higher priority" mean in your code?

A: It means that process with **priority number 1** is treated as higher priority than process with priority 3.

Q4: What is a *burst time*?

A: Burst time is the **amount of CPU time required** by a process to complete execution.

Q5: What is *arrival time*?

A: It is the time at which a process enters the ready queue (becomes available for execution).

🔷 Algorithm Logic

Q6: How does your algorithm choose which process to execute next?

A: It checks all processes that have **arrived (arrival time ≤ current time)** and **not yet completed**, then selects the one with **highest priority (lowest number)**.

Q7: What is the role of the `completed[]` array?

A: It keeps track of which processes have already finished execution, so they are not selected again.

Q8: Why do you increment `time` when no process has arrived yet?

A: Because the CPU is **idle** at that moment. Time moves forward until a process arrives.

Q9: What is the execution order printed at the end?

A: It shows the **sequence of process IDs (PIDs)** executed by the CPU, according to their arrival and priorit

Q20: Can this algorithm be used in real-time systems?

A: Yes, priority scheduling is common in **real-time and embedded systems**, where urgent tasks must execute first.

Q22: What is the CPU utilization here?

A: Depends on input; if there's no idle time, CPU utilization is 100%.

Q23: What is throughput?

A: Number of processes completed per unit time.

Problem Statement 6: Round Robin (RR) Scheduling

Write a program to simulate the **Round Robin (Preemptive)** CPU scheduling algorithm.

The program should take **time quantum** as input and schedule processes in a cyclic order.

Display the **Gantt chart**, **waiting time**, **turnaround time**, and **average values** for all processes.

```
import java.util.*;
```

```
class RoundRobin {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of processes: ");
        int n = sc.nextInt();

        int[] pid = new int[n];     // Process IDs
        int[] at = new int[n];      // Arrival Times
        int[] bt = new int[n];      // Burst Times
        int[] rem = new int[n];     // Remaining Times
        int[] wt = new int[n];      // Waiting Times
        int[] tat = new int[n];     // Turnaround Times
        int[] ct = new int[n];      // Completion Times

        // Input process details
        for (int i = 0; i < n; i++) {
            System.out.print("Process ID: ");
            pid[i] = sc.nextInt();
            System.out.print("Arrival Time: ");
            at[i] = sc.nextInt();
            System.out.print("Burst Time: ");
            bt[i] = sc.nextInt();
            rem[i] = bt[i]; // Initialize remaining time
            System.out.println();
        }

        System.out.print("Enter Time Quantum: ");
        int tq = sc.nextInt();

        int time = 0, completed = 0;
        Queue<Integer> q = new LinkedList<>();
        boolean[] inQueue = new boolean[n];
        ArrayList<String> gantt = new ArrayList<>();

        // Add first arrived process to queue
```

```
        for (int i = 0; i < n; i++) {
            if (at[i] == 0) {
                q.add(i);
                inQueue[i] = true;
                break;
            }
        }

        // Round Robin Scheduling Loop
        while (completed < n) {
            if (q.isEmpty()) {
                time++;
                for (int i = 0; i < n; i++) {
                    if (at[i] <= time && !inQueue[i] && rem[i] > 0) {
                        q.add(i);
                        inQueue[i] = true;
                        break;
                    }
                }
                continue;
            }

            int i = q.poll();
            int exec = Math.min(tq, rem[i]);
            rem[i] -= exec;
            time += exec;
            gantt.add("P" + pid[i]);

            // Check for new arrivals during execution
            for (int j = 0; j < n; j++) {
                if (at[j] <= time && rem[j] > 0 && !inQueue[j]) {
                    q.add(j);
                    inQueue[j] = true;
                }
            }
```

```
            // Re-add process if not completed
            if (rem[i] > 0) {
                q.add(i);
            } else {
                completed++;
                ct[i] = time;
                tat[i] = ct[i] - at[i];
                wt[i] = tat[i] - bt[i];
            }
        }

        // Output results
        double avgWT = 0, avgTAT = 0;
        System.out.println("\nPID\tAT\tBT\tWT\tTAT");
        for (int i = 0; i < n; i++) {
            avgWT += wt[i];
            avgTAT += tat[i];
            System.out.println(pid[i] + "\t" + at[i] + "\t" + bt[i] + "\t" + wt[i] + "\t" +
tat[i]);
        }

        // Gantt Chart
        System.out.println("\nGantt Chart:");
        for (String g : gantt) {
            System.out.print("| " + g + " ");
        }
        System.out.println("|");

        // Averages
        System.out.printf("\nAverage Waiting Time:"+ avgWT / n);
        System.out.printf("\nAverage Turnaround Time:"+ avgTAT / n);
    }
}
```

🔷 Basic Concept

Q1: What is Round Robin scheduling?

A: Round Robin (RR) is a **preemptive CPU scheduling algorithm** where each process is given a **fixed time slice** (called *time quantum*).

If a process doesn't finish within its time quantum, it's **moved to the end of the queue** and the next process runs.

Q2: Is Round Robin preemptive or non-preemptive?

A: It is **preemptive**, because processes are interrupted after each time quantum and CPU is given to the next process.

Q3: What is a *time quantum* (or *time slice*)?

A: Time quantum is the **maximum amount of time** a process can use the CPU continuously before being preempted.

It ensures fair CPU sharing among all processes.

Q4: What happens if a process finishes before the time quantum expires?

A: The process **releases the CPU early**, and the scheduler immediately switches to the **next process in the queue**.

Q5: What happens if the time quantum is very small or very large?

A:

- **Too small:** More context switches → high overhead → less efficiency.
- **Too large:** Acts like **First-Come-First-Serve (FCFS)** → poor response time.

🔷 Program Logic

Q6: What is the purpose of the `Queue` in your program?

A: The queue represents the **ready queue**, which stores the order of processes waiting for CPU time in Round Robin order.

Q7: What is the role of the `inQueue[]` array?

A: It keeps track of which processes are **already in the queue**, so that no process is added twice.

Q8: Why do we use `rem[]` (remaining time) array?

A: Because processes may not complete in one CPU burst.

We need to **store how much burst time is left** after each execution cycle.

Q9: Why is `Math.min(tq, rem[i])` used?

A: To ensure a process executes only for the **smaller of (time quantum or remaining burst time)**.

If remaining time is less than the time quantum, it finishes early.

Q10: What is the significance of the `gantt` list?

A: It stores the **order in which processes are executed** — used to display the Gantt chart later.

Q11: Why do we check `if (q.isEmpty())`?

A: It handles CPU **idle time** — if no process has arrived yet, the scheduler simply increments `time` until one arrives.

Q18: Can starvation occur in Round Robin?

A: ❌ No.

Because each process eventually gets CPU time after a fixed interval (its turn).

Q19: How is context switching handled in Round Robin?

A: After each time quantum, the CPU **saves the state** of the current process and **loads the next one**, leading to a context switch.

Q20: What is the disadvantage of Round Robin scheduling?

A++:

- High number of **context switches** (if quantum is too small).
- **Overhead** due to frequent CPU switching.
- Average waiting time can increase for large numbers of processes.

Q28: In a real OS, where is Round Robin scheduling used?

A: In **time-sharing systems**, interactive environments, and multitasking OS like Windows and Linux.

Q29: What is a *context switch*?

A: The act of saving the state of one process and loading another's state — occurs at every quantum expiration.

Problem Statement 7: Memory Allocation – First Fit

Write a program to simulate **First Fit** memory allocation strategy. The program should allocate each process to the first available memory block that is large enough to accommodate it.

Display the **memory allocation table** and identify any **unused or fragmented memory**.

🔷 1. What is Memory Management?

A: Memory management is the process of efficiently allocating, tracking, and freeing memory in a computer system so that multiple processes can execute without conflicts.

🔷 2. What is Memory Allocation?

A: Memory allocation is assigning parts of the main memory to different processes so they can execute properly.

There are two main types:

- **Static Allocation:** Fixed memory at compile time.
- **Dynamic Allocation:** Memory allocated at runtime (like your program does).

🔷 3. What is the First Fit Algorithm?

A:

In **First Fit**, the operating system scans memory blocks sequentially and allocates the **first available block** that is **large enough** to hold the process.

It stops searching after the first suitable block is found.

✅ Example:

If blocks = [100, 300, 200] and process = 150

→ Allocated to block 2 (since it's the first that fits).

🔷 4. What are the other memory allocation algorithms?

Algorithm	Description
First Fit	Allocates the first available block that fits the process.
Best Fit	Allocates the smallest block that fits the process (minimizes wastage).
Worst Fit	Allocates the largest available block (maximizes future flexibility).
Next Fit	Like First Fit, but continues search from where the last allocation occurred.

🔷 5. Why is it called "First Fit"?

A: Because it picks the **first** memory block from the list that can accommodate the process — no further searching for better matches.

🔷 6. What are the advantages of the First Fit algorithm?

✅ Simple and fast (linear scan).

✅ Efficient for smaller systems with fewer memory blocks.

🔷 7. What are the disadvantages of the First Fit algorithm?

❌ Leads to **external fragmentation** (many small unused gaps).

❌ Not optimal in memory utilization.

❌ May require **compaction** later to defragment memory.

🔷 8. What is fragmentation?

A: Fragmentation refers to the wasted memory space that occurs when total memory is enough to satisfy a process request, but it's not contiguous.

- **External Fragmentation:** Unused memory between allocated blocks.
- **Internal Fragmentation:** Unused memory within an allocated block.

Your code shows **external fragmentation**.

🔷 9. How does your program handle fragmentation?

A: It calculates **remaining space** (`rem[i]`) in each block after allocation and prints it as **fragmentation** in the output table.

🔷 10. What are the key arrays used in your code and their purpose?

Array	Description
`block[]`	Stores initial sizes of memory blocks.
`rem[]`	Stores remaining (unused) memory after allocation.
`process[]`	Stores the memory size required by each process.
`alloc[]`	Stores which block each process is allocated to (or -1 if not allocated).

🔷 11. Explain the logic of allocation in your code.

```
for (int i = 0; i < np; i++) {
    for (int j = 0; j < nb; j++) {
        if (rem[j] >= process[i]) {
            alloc[i] = j;
            rem[j] -= process[i];
            break;
        }
    }
}
```

- Outer loop: goes through each process.
- Inner loop: checks each memory block.

- If block has enough space (`rem[j] >= process[i]`): allocate it and reduce remaining space.
- Then break (move to next process).

🔷 12. What happens if a process cannot fit in any block?

A: It remains **unallocated**.

The code keeps `alloc[i] = -1`, and prints `Not Allocated` in the output.

🔷 13. What output does your program produce?

Example Input:

```
Blocks: 100 500 200 300 600
Processes: 212 417 112 426
```

Output:

```
Process Size  Block Allocated
P1    212    Block 2
P2    417    Block 5
P3    112    Block 2
P4    426    Not Allocated

Block  Initial Size  Remaining  Fragmentation
B1    100          100        100
B2    500          176        176
B3    200          200        200
B4    300          300        300
B5    600          183        183

Total Fragmented/Unused Memory: 959
```

🔷 14. What is the time complexity of First Fit?

A:

O(n × m), where:

- n = number of processes
- m = number of memory blocks

  (because each process might check all blocks)

🔷 15. Why do we use `rem[]` instead of modifying `block[]` directly?

A:

- To preserve the **original block sizes** for display.
- `rem[]` represents current available space dynamically updated after each allocation.

🔷 16. What is internal vs external fragmentation?

Type	Definition	Example
Internal Fragmentation	Wasted space **inside** an allocated block.	Process needs 90, block is 100 → 10 wasted
External Fragmentation	Wasted space **between** blocks that can't be used.	Many small free blocks remain scattered

---

🔷 17. Can First Fit eliminate fragmentation?

A: No. It only allocates memory — fragmentation can only be reduced using **compaction** (merging free blocks).

🔷 18. How can we minimize fragmentation?

- Use **Best Fit** to minimize unused space.
- Perform **compaction** to merge adjacent free spaces.
- Use **paging/segmentation** for dynamic memory allocation.

🔷 19. What is total fragmentation shown in your output?

A: It's the **sum of all remaining spaces (`rem[i]`)** after allocation.

```
totalFrag += rem[i];
```

This gives total unused memory.

🔷 20. Why is `alloc[i]` initialized to -1?

A:

To indicate that a process is **not yet allocated** to any block.

It helps identify unallocated processes at the end.

🔷 21. What is the significance of `break` in the inner loop?

A:

After a process is allocated to a block, the program immediately exits the inner loop to avoid checking further blocks — implementing the **"first fit"** rule.

🔷 22. What is the output section of your program doing?

It prints:

1. Process → Block allocation table
2. Remaining size (fragmentation) for each block
3. Total unused memory across all blocks

🔷 23. What would happen if two processes fit perfectly into one block?

They **cannot**, since a block can only be allocated once at a time.

Only one process can occupy a block until its remaining space is used for smaller processes.

🔷 24. What's the difference between "block size" and "process size"?

Term	Meaning
Block Size	Amount of memory available in each partition.
Process Size	Amount of memory required by a process.

Allocation occurs only when `block ≥ process`.

🔷 25. How can we modify this program to implement Best Fit or Worst Fit?

- **Best Fit:** Find the smallest block that fits.
- **Worst Fit:** Find the largest block that fits.

  Only allocation condition changes inside the inner loop.

🔷 26. What is dynamic partitioning?

A: A memory allocation scheme where memory is divided into **variable-sized blocks** dynamically at runtime, rather than fixed partitions.

Your program models **dynamic partitioning** using the **First Fit** policy.

🔷 27. What does "unused memory" represent in your output?

A: It represents **fragmentation** — free space that couldn't be used for any process because it's too small.

🔷 28. What is the main drawback of First Fit?

A: Over time, small holes appear in memory, leading to **external fragmentation**, reducing overall efficiency.

🔷 29. What real-world systems use similar memory allocation?

Older **Operating Systems** and **Dynamic Memory Managers** (like early UNIX and MS-DOS) used **First Fit** for process allocation in main memory.

Best Fit :→

2. What is the **Best Fit** allocation strategy?

Answer:

In the Best Fit method, each process is allocated to the **smallest memory block** that is **large enough** to accommodate it.

This minimizes internal fragmentation but may lead to external fragmentation

15. What are the advantages of the Best Fit strategy?

Answer:

- Reduces **internal fragmentation** by using the smallest possible block.
- Efficient when process sizes vary significantly.

---

16. What are the disadvantages of the Best Fit strategy?

Answer:

- Slower than First Fit (since it checks all blocks).
- Leads to **external fragmentation** due to scattered unused small blocks.

6. What data structures are used in this program?

Answer:

- `block[]` → to store initial sizes of memory blocks.
- `rem[]` → to store remaining space after allocation.
- `process[]` → to store process sizes.
- `alloc[]` → to record which block each process is assigned to.

---

7. Explain the logic of the allocation loop.

Answer:

For each process:

1. The program scans all blocks.
2. It picks the **block with the least remaining space** that can still fit the process.
3. If found, that block is allocated and remaining memory is updated.

---

8. Why do we use `bestIndex = -1` initially?

Answer:

It indicates that **no suitable block** has been found yet.

If it remains `-1` after checking all blocks, the process cannot be allocated.

---

9. What does the condition

```
if (bestIndex == -1 || rem[j] < rem[bestIndex])
```

mean?

Answer:

It finds the block with the **smallest leftover space** (`rem[j]`) that can fit the process — that's the "best fit".

Problem Statement 9: Memory Allocation – Next Fit

Write a program to simulate **Next Fit** memory allocation strategy. The program should continue searching for the next suitable memory block from the last allocated position instead of starting from the beginning. Display the **memory allocation table** and **fragmentation details**.

4. What is the **Next Fit** memory allocation strategy?

Answer:

Next Fit is a variant of the First Fit algorithm.

It starts searching for the next suitable block **from where the last allocation happened**, not from the beginning of the memory list.

This helps distribute memory usage evenly and reduces search time.

6. What is the advantage of the Next Fit method?

Answer:

- Reduces search time because it doesn't always start from the beginning.
- Distributes memory allocation more evenly across blocks.

---

7. What is the disadvantage of Next Fit?

Answer:

- Can still lead to fragmentation.
- May skip small but usable blocks before the last position.

10. What is the role of `lastPos` variable?

Answer:

`lastPos` stores the **index of the last block allocated**.

The next search for allocation starts **from this position onward** (cyclically).

11. What is the purpose of this line?

```
int j = (lastPos + count) % nb;
```

Answer:

This ensures **cyclic searching** through memory blocks.

If the end of the block list is reached, the search wraps around to the beginning.

---

12. What does the `while (count < nb)` loop do?

Answer:

It checks all memory blocks once (using circular traversal) to find a suitable block for the process.

If no block fits, allocation fails for that process.

++++++++++++++++

Problem Statement 10: Memory Allocation – Worst Fit

Write a program to simulate **Worst Fit** memory allocation strategy. The program should allocate each process to the **largest available memory block**. Display the **memory allocation results** and any **unused space**.

1. What is the Worst Fit memory allocation strategy?

Answer:

In **Worst Fit**, each process is allocated to the **largest available memory block** that can accommodate it.

This is done to leave the **largest remaining hole** possible for future allocations.

4. What is the main advantage of Worst Fit?

Answer:

- Helps reduce the creation of **very small fragments** that cannot be reused later.
- Ensures large blocks remain available for big processes.

---

5. What is the main disadvantage of Worst Fit?

Answer:

- May lead to **poor overall memory utilization** since big blocks get split unnecessarily.

++++++++++

11. **Problem Statement 11: Page Replacement – FIFO**

Write a program to simulate the **First In First Out (FIFO)** page replacement algorithm.

The program should accept a **page reference string** and **number of frames** as input, simulate the process of page replacement, and display the **total number of page faults**.

1. What is Page Replacement?

Answer:

Page replacement is a process in which an operating system decides **which memory page to remove** from RAM to make space for a new page when there is no free frame available.

---

2. What is a Page Fault?

Answer:

A **page fault** occurs when a page referenced by the CPU is **not found in main memory**, so it must be **loaded from secondary storage** (like a disk).

---

3. What is a Frame?

Answer:

A **frame** is a fixed-size block of physical memory in RAM where pages from secondary memory are loaded.

---

4. What is a Page Reference String?

Answer:

It is a sequence of page numbers that represents the **order of page requests** made by a process during execution.

Example:

`1 2 3 2 1` means the process needs page 1, then 2, then 3, then 2 again, then 1.

⚙️ FIFO Algorithm Questions

5. What does FIFO stand for?

Answer:

FIFO stands for **First In, First Out**.

---

6. What is the logic of the FIFO Page Replacement algorithm?

Answer:

The oldest page (the one that entered memory **first**) is replaced first whenever a new page needs to be loaded.

---

7. Why is it called "FIFO"?

Answer:

Because pages are managed like a **queue** — the earliest loaded page is the first to be removed when memory is full.

---

8. How does FIFO decide which page to replace?

Answer:

It maintains a **pointer (index)** to the oldest page in memory. When a page fault occurs, the page at this position is replaced, and the pointer moves to the next frame cyclically.

---

9. What is a Page Hit?

Answer:

When the requested page is **already present in memory**, it is called a **page hit** (no replacement needed)

20. What are the advantages of FIFO?

Answer:

✅ Easy to implement

✅ Uses simple queue logic

✅ Requires minimal overhead

---

21. What are the disadvantages of FIFO?

Answer:

❌ May replace frequently used pages (poor performance)

❌ Suffers from **Belady's Anomaly**

---

22. What is Belady's Anomaly?

Answer:

It is the situation where **increasing the number of frames** results in **more page faults**, which is counterintuitive.

This happens in **FIFO**, but not in optimal or LRU algorithms.

Problem Statement 12: Page Replacement – LRU

Write a program to simulate the **Least Recently Used (LRU)** page replacement algorithm.

The program should display the **frame contents after each page reference** and the **total number of page faults**.

1. What does LRU stand for?

Answer:

LRU stands for **Least Recently Used** — a page replacement algorithm that removes the page that **has not been used for the longest time**.

---

2. What is the main idea behind the LRU algorithm?

Answer:

The LRU algorithm assumes that **pages used recently are likely to be used again soon**, while pages not used for a long time are less likely to be used again — so they are replaced first.

---

3. How is LRU different from FIFO?

Feature	FIFO	LRU
Basis	Order of arrival	Last recent use
Replacement	Oldest page	Least recently used page
Performance	Simpler but less efficient	More efficient (closer to optimal)
Tracking	Queue or pointer	Time of last use or stack/list

4. What is a page fault in this context?

Answer:

A **page fault** occurs when a requested page is **not present in memory** (frame). The system must then load it from disk into memory, possibly replacing an existing page.

5. What is a page hit?

Answer:

When the requested page is **already present in memory**, it is called a **page hit** — no replacement is needed.

6. What is the core principle of LRU?

Answer:

Replace the page that has **not been used for the longest time in the past**.

7. What are the advantages of LRU?

Answer:

- More accurate prediction of future needs (better than FIFO).
- Fewer page faults on average.
- Reflects real program behavior more closely.

8. What are the disadvantages of LRU?

Answer:

- Requires **extra time and space** to track recent usage history.
- Slightly **complex to implement** compared to FIFO.

13. What happens when all frames are full?

Answer:

The algorithm finds the **least recently used (LRU)** page and replaces it with the new page.

---

14. How does the code find the LRU page?

```
for (int f = 0; f < frame.size(); f++) {
    int lastUsed = -1;
    for (int j = i - 1; j >= 0; j--) {
        if (pages[j] == frame.get(f)) {
            lastUsed = j;
            break;
        }
    }
    if (lastUsed < lru) {
        lru = lastUsed;
        indexToRemove = f;
    }
}
```

Answer:

It checks each page currently in memory and finds **when it was last used** (the smallest `j` value).

That page is considered **least recently used** and is replaced.

---

15. What does this line mean?

```
frame.set(indexToRemove, page);
```

Answer:

It replaces the LRU page (at `indexToRemove`) with the new page in memory.

++

Write a program to simulate the **Optimal Page Replacement** algorithm. The program should replace the page that **will not be used for the longest period of time** in the future and display the **page replacement steps** and **page fault count**.

🧠 Conceptual Viva Questions

1. What is the Optimal Page Replacement algorithm?

Answer:

It is a page replacement algorithm that replaces the **page that will not be used for the longest period in the future**.

It gives the **minimum number of page faults** possible for a given reference string.

2. Why is it called "Optimal"?

Answer:

Because it makes the *best possible decision* by using **future knowledge** of page references — it always removes the page that will be needed farthest in the future.

3. Why can't the optimal algorithm be implemented in real operating systems?

Answer:

Because real systems **cannot predict the future** page references.

It is used mainly for **theoretical analysis** and comparison with other algorithms like FIFO or LRU.

13. Explain the logic inside this part:

```
for (int j = 0; j < frames; j++) {
    if (frame[j] == page) {
        hit = true;
        break;
    }
}
```

Answer:

This loop checks whether the **current page already exists** in one of the frames.

If it does, it sets `hit = true` and breaks out — no replacement is needed.

---

14. Why do we use these variables?

```
int replaceIndex = -1;
int farthest = -1;
```

Answer:

- `replaceIndex` → holds the **index of the frame** where the replacement will happen.
- `farthest` → keeps track of how far in the future each page will next be used.

---

15. What does this part do?

```
if (replaceIndex == -1) {
    for (int j = 0; j < frames; j++) {
        int nextUse = Integer.MAX_VALUE;
        for (int k = i + 1; k < n; k++) {
            if (frame[j] == pages[k]) {
                nextUse = k;
                break;
            }
        }
```

```
            if (nextUse > farthest) {
                farthest = nextUse;
                replaceIndex = j;
            }
        }
    }
```

Answer:

This is the **core logic** of Optimal Replacement:

For every page currently in memory, it finds **when it will next be used**.

- The page that will be used **farthest in the future** gets replaced.
- If a page **is never used again**, it is replaced immediately.

---

16. What does this mean?

```
if (nextUse > farthest)
```

Answer:

If this page is used **later than any previously checked page**,

it becomes the **best candidate** for replacement (used farthest in future).

---

17. Why is `Integer.MAX_VALUE` used for `nextUse`?

Answer:

It represents pages that are **never used again** —

this ensures such pages will have the **highest possible next use value**,

and thus will be **selected for replacement**.