>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> CPU Scheduling

```
def fcfs(processes):
    processes.sort(key=lambda x: x[1])
    time = 0
    print("Order: ", end="")
    for p in processes:
        if time < p[1]:
            time = p[1]
        print(p[0], end=" ")
        time += p[2]
    print()

def sjf_preemptive(processes):
    processes.sort(key=lambda x: x[1])
    n = len(processes)
    remaining = [p[2] for p in processes]
    complete = 0
    time = 0
    order = []
    while complete < n:
        idx = -1
        mn = 1e9
        for i in range(n):
            if processes[i][1] <= time and remaining[i] > 0 and remaining[i] < mn:
                mn = remaining[i]
                idx = i
        if idx == -1:
            time += 1
            continue
        order.append(processes[idx][0])
        remaining[idx] -= 1
        time += 1
        if remaining[idx] == 0:
            complete += 1
    print("Order:", " ".join(order))

def priority_non_preemptive(processes):
    processes.sort(key=lambda x: (x[3], x[1]))
    time = 0
    print("Order:", end=" ")
    for p in processes:
        if time < p[1]:
            time = p[1]
        print(p[0], end=" ")
        time += p[2]
    print()

def round_robin(processes, quantum):
    n = len(processes)
    remaining = [p[2] for p in processes]
    time = 0
    queue = []
    arrived = set()
    order = []
    while True:
        for i in range(n):
            if processes[i][1] <= time and i not in arrived:
                queue.append(i)
                arrived.add(i)
        if not queue:
            if all(r == 0 for r in remaining):
                break
            time += 1
            continue
        idx = queue.pop(0)
        order.append(processes[idx][0])
        run = min(quantum, remaining[idx])
        remaining[idx] -= run
        time += run
        for i in range(n):
            if processes[i][1] <= time and i not in arrived:
                queue.append(i)
                arrived.add(i)
        if remaining[idx] > 0:
            queue.append(idx)
    print("Order:", " ".join(order))

processes_fcfs = [["P1", 0, 5], ["P2", 1, 3], ["P3", 2, 8]]
processes_sjf = [["P1", 0, 8], ["P2", 1, 4], ["P3", 2, 2]]
processes_priority = [["P1", 0, 5, 3], ["P2", 1, 3, 1], ["P3", 2, 4, 2]]
processes_rr = [["P1", 0, 5], ["P2", 1, 3], ["P3", 2, 1]]

print("FCFS Scheduling:")
fcfs(processes_fcfs)

print("\nSJF (Preemptive) Scheduling:")
sjf_preemptive(processes_sjf)

print("\nPriority (Non-Preemptive) Scheduling:")
priority_non_preemptive(processes_priority)

print("\nRound Robin Scheduling (Quantum = 2):")
round_robin(processes_rr, 2)
```

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Memory Allocation

last_index = 0

def index(memoryArr, node):
    for i in range(len(memoryArr)):
        if memoryArr[i] == node:
            return i

def first_fit(memoryArr, request):
    if max(memoryArr) < request:
        print("Insufficient Memory")
        return
    for i in range(len(memoryArr)):
        if memoryArr[i] >= request:
            print(request, "KB memory allocated from", memoryArr[i], "KB Block")
            memoryArr[i] -= request
            print(memoryArr)
            return

def next_fit(memoryArr, request):
    global last_index
    if max(memoryArr) < request:
        print("Insufficient Memory")
        return
    n = len(memoryArr)
    count = 0
    i = last_index
    while count < n:
        if memoryArr[i] >= request:
            print(request, "KB memory allocated from", memoryArr[i], "KB Block
(position", i, ")")
            memoryArr[i] -= request
            last_index = i
            print(memoryArr)
            return
        i = (i + 1) % n
        count += 1
    print("No sufficient block found for this request (Next Fit).")

def best_fit(memoryArr, request):
    if max(memoryArr) < request:
        print("Insufficient Memory")
        return
    newArr = []
    for i in memoryArr:
        if i >= request:
            newArr.append(i)
    allocate = min(newArr)
    print(request, "KB memory allocated from", allocate, "KB Block")
    memoryArr[index(memoryArr, allocate)] -= request
    print(memoryArr)

def worst_fit(memoryArr, request):
    if max(memoryArr) < request:
        print("Insufficient Memory")
        return
    newArr = []
    for i in memoryArr:
        if i >= request:
            newArr.append(i)
    allocate = max(newArr)
    print(request, "KB memory allocated from", allocate, "KB Block")
    memoryArr[index(memoryArr, allocate)] -= request
    print(memoryArr)

memoryArr = [100, 500, 200, 300]
request = 200

print("First Fit:")
first_fit(memoryArr.copy(), request)
print("\nNext Fit:")
next_fit(memoryArr.copy(), request)
print("\nBest Fit:")
best_fit(memoryArr.copy(), request)
print("\nWorst Fit:")
worst_fit(memoryArr.copy(), request)
```

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> MutexSemaphore

```
buffer = [None] * 5
mutex = 1
empty = 5
full = 0
in_index = 0
out_index = 0

def producer():
    global mutex, empty, full, in_index

    if empty == 0:
        print("Buffer is full! Producer must wait.")
        return
    empty -= 1
    mutex -= 1

    data = int(input("Enter data to produce: "))
    buffer[in_index] = data
    in_index = (in_index + 1) % 5

    print(f"Produced item: {data}")
    print("Current buffer:", [x for x in buffer if x is not None])

    mutex += 1
    full += 1


def consumer():
    global mutex, empty, full, out_index

    if full == 0:
        print("Buffer is empty! Consumer must wait.")
        return

    full -= 1
    mutex -= 1

    data = buffer[out_index]
    buffer[out_index] = None
    out_index = (out_index + 1) % 5

    print(f"Consumed item: {data}")
    print("Current buffer:", [x for x in buffer if x is not None])

    mutex += 1
    empty += 1

while True:
    print("\n1. Produce\n2. Consume\n3. Exit")
    choice = input("Enter your choice: ")

    if choice == "1":
        producer()
    elif choice == "2":
        consumer()
    elif choice == "3":
        print("Exiting...")
        break
    else:
        print("Invalid choice! Please try again.")
```

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> PageReplacement


size = int(input("Enter The No. of Frames: "))
frames = []

def getIndex(arr, page):
    for i in range(len(arr)):
        if arr[i] == page:
            return i

def optimal(newArr, count):
    index = 0
    newIndex = 0
    for i in range(count):
        if newArr[i] in frames:
            print(frames)
            print()
            continue

        if len(frames) < size:
            frames.append(newArr[i])

        else:
            checkArr = []
            for j in range(i+1, count):
                if newArr[j] in frames and newArr[j] not in checkArr:
                    checkArr.append(newArr[j])
            if len(checkArr) == 0:
                frames[index] = newArr[i]
                index = (index+1)%size
            elif len(checkArr) < size:
                no_page = []
                for k in frames:
                    if k not in checkArr:
                        no_page.append(k)
                newIndex = getIndex(frames, no_page[0])
                frames[newIndex] = newArr[i]
            else:
                newIndex = getIndex(frames, checkArr[-1])
                frames[newIndex] = newArr[i]

        print(frames)
        print()


def least(newArr, count):
    newIndex = 0
    for i in range(count):
        if newArr[i] in frames:
            print(frames)
            print()
            continue

        if len(frames) < size:
            frames.append(newArr[i])

        else:
            checkArr = []
            for j in range(i-1, -1, -1):
                if newArr[j] in frames and newArr[j] not in checkArr:
                    checkArr.append(newArr[j])

            newIndex = getIndex(frames, checkArr[-1])
            frames[newIndex] = newArr[i]

        print(frames)
        print()

def main():
    count = int(input("Enter No. of Pages: "))
    newArr = []
    print("\nEnter PageNo in Sequence Priority >> \n")
    for i in range(count):
        page = int(input("Enter PageNo: "))
        newArr.append(page)

    least(newArr, count)

main()
```