❓ What is Task Scheduling?

Task scheduling is the process of assigning a set of tasks to resources (like machines or processors) over time so that certain objectives are achieved. Common objectives include:

.Minimizing total completion time (makespan)
.Balancing the load across resources
.Maximizing efficiency or resource utilization

It's a general problem in operations research and computer science, used in areas like CPU scheduling, manufacturing, and project management.

❓ **What is List Scheduling?**

List scheduling is a specific greedy algorithm for task scheduling, where:

1. Tasks are placed in a list (any order, often arbitrary or based on priority).

2. Each task is assigned to the machine that becomes available first (the one with the [1]minimum current load).

3. The algorithm proceeds until all tasks are scheduled.

It's a fast, approximate method for parallel machine scheduling and usually provides a reasonable makespan, though not always optimal.

❓ ***Hands on (Program) below, Addresses Task Scheduling:***

The program defines a set of tasks with durations and a fixed number of machines.

```
#===============

import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.animation import FuncAnimation
from IPython.display import HTML, display

# --- Example data ---
tasks = [4, 6, 3, 5, 2, 7, 4, 3]   # task durations
num_machines = 3                   # number of machines
```

[1] YouTube Handle : @csquickrevisionshorts

```
# --- Scheduling using List Scheduling (Greedy) ---
machines = [[] for _ in range(num_machines)]
end_times = [0] * num_machines
states = []

for t in tasks:
    m = end_times.index(min(end_times))  # choose machine that gets free first
    machines[m].append((end_times[m], t))  # store (start_time, duration)
    end_times[m] += t
    # snapshot of current state
    state = [[(s, d) for (s, d) in mach] for mach in machines]
    states.append(state)

# --- Statistics ---
makespan = max(end_times)
avg_load = sum(tasks) / num_machines
efficiency = (sum(tasks) / (num_machines * makespan)) * 100

print("📊 SCHEDULING STATISTICS")
print("------------------------")
print(f"Total tasks: {len(tasks)}")
print(f"Machines: {num_machines}")
print(f"Makespan (total time): {makespan}")
print(f"Average load per machine: {avg_load:.2f}")
print(f"Scheduling efficiency: {efficiency:.2f}%")

# --- Visualization setup ---
fig, ax = plt.subplots(figsize=(8, 4))
colors = ['skyblue', 'lightgreen', 'lightcoral', 'khaki', 'plum', 'orange']

def draw_state(state):
    ax.clear()
    ax.set_xlim(0, makespan + 2)[2]
    ax.set_ylim(-1, num_machines)
    ax.set_title("Task Scheduling Visualization (List Scheduling)", fontsize=12)
    ax.set_xlabel("Time")
    ax.set_ylabel("Machine ID")

    for m_id, mach in enumerate(state):
        for i, (start, dur) in enumerate(mach):
            ax.add_patch(patches.Rectangle((start, -m_id - 0.4), dur, 0.8,
                                   color=colors[i % len(colors)], ec='black'))
            ax.text(start + dur / 2, -m_id, f"T{i+1}", ha='center', va='center', fontsize=8)
        ax.text(-1.5, -m_id, f"M{m_id+1}", va='center', fontsize=9)
```

[2] YouTube Handle : @csquickrevisionshorts

```
def update(frame):
    draw_state(states[frame])

ani = FuncAnimation(fig, update, frames=len(states), interval=1000, repeat=False)
display(HTML(ani.to_jshtml()))

#@=====

#--- Save the animation ---
from matplotlib.animation import PillowWriter

# Save as MP4
ani.save("task_scheduling_animation.mp4", writer='ffmpeg', fps=1)

# Save as GIF (alternative)
ani.save("task_scheduling_animation.gif", writer=PillowWriter(fps=1))

print("✅ Animation saved as MP4 and GIF")

#============
```

❓ Understanding the code

The code implements a List Scheduling algorithm, a classical greedy method in machine scheduling problems. The main steps are:

1. Input Definition:

A set of tasks with known durations: tasks = [4, 6, 3, 5, 2, 7, 4, 3].

A fixed number of machines: num_machines = 3.

2. Initialization:

‘machines’ stores tasks assigned to each machine along with their start times.

end_times tracks when each machine becomes free.

‘states’ stores snapshots of the schedule after each task assignment, used for visualization.

[3]

3. Task Assignment (Greedy):

[3] YouTube Handle : @csquickrevisionshorts

For each task t, the machine with the earliest finishing time is selected (min(end_times)).

The task is scheduled starting at the current end time of that machine, and the machine's end time is updated.

After each assignment, a snapshot of the current state is saved.

4. Statistics Calculation:

Makespan: The total time taken to finish all tasks (max(end_times)).

Average load per machine: Total task time divided by number of machines.

Efficiency: Ratio of total task time to the total machine time used (sum(tasks) / (num_machines * makespan) × 100).

5. Visualization:

Using matplotlib and FuncAnimation, each task is drawn as a colored rectangle on a timeline for its machine.

Animates task assignment dynamically over time.

Labels indicate task IDs (T1, T2, ...) and machine IDs (M1, M2, ...).

❓ **Problem Solved**

This code solves the Parallel Machine Scheduling Problem, specifically:

Objective: Minimize the makespan (total completion time) when assigning a set of tasks to multiple identical machines.

Approach: Uses List Scheduling (Greedy algorithm).

This problem is NP-hard for arbitrary task durations and multiple machines. The greedy method provides a fast, approximate solution with reasonable efficiency.

[4]

[4] YouTube Handle : @csquickrevisionshorts

Statistical Results and Interpretation:

For the input given:

1. Tasks: 8 tasks with durations [4, 6, 3, 5, 2, 7, 4, 3]

2. Machines: 3

3. Makespan: max(end_times) = 13

This is the total time taken for the last machine to finish all assigned tasks.

4. Average load per machine:

sum(tasks)/num_machines = 34 / 3 ≈ 11.33

Represents the ideal average time per machine if perfectly balanced.

5. Efficiency:

sum(tasks)/(num_machines * makespan) × 100 = 34 / (3 * 13) × 100 ≈ 87.18%

Indicates how well the tasks are balanced across machines; 100% would be perfectly balanced.

---[5]

2. *Explanation*

1. Task Scheduling Logic:
The algorithm always assigns the next task to the machine that is currently least loaded. This simple greedy choice reduces idle time for machines and keeps the schedule reasonably balanced, though it does not guarantee an optimal solution.

2. Greedy Heuristic:

[5] YouTube Handle : @csquickrevisionshorts

It is fast, with time complexity O(n × m), where n is the number of tasks and m is the number of machines.

Provides a makespan ≤ (2 - 1/m) × OPT, a known approximation bound for identical machines.

3. Visualization:

The animated rectangles provide a clear Gantt-chart style visualization, showing start and end times of each task per machine.

This helps understand the allocation sequence and identify potential bottlenecks visually.

4. Interpretation of Results:

Makespan > average load → some imbalance exists, which is normal in greedy scheduling.

Efficiency ≈ 87% → the machines are fairly well utilized.

This approach is suitable for offline scheduling, where all task durations are known in advance.

❓ ***Code Output and Explanation:***

Given tasks: [4, 6, 3, 5, 2, 7, 4, 3]
Machines: 3

Step 1: Assign tasks using list scheduling (greedy):

Machine end times: initially [0, 0, 0]

Task 1 (4) → Machine 0 (min end time 0): end time = 0 + 4 = 4 → [4,0,0]
[6]

Task 2 (6) → Machine 1: end = 0 + 6 = 6 → [4,6,0]

Task 3 (3) → Machine 2: end = 0 + 3 = 3 → [4,6,3]

Task 4 (5) → Machine 2 (min end 3): end = 3 + 5 = 8 → [4,6,8]

Task 5 (2) → Machine 0 (min end 4): end = 4 + 2 = 6 → [6,6,8]

[6] YouTube Handle : @csquickrevisionshorts

Task 6 (7) → Machine 0 (min end 6): end = 6 + 7 = 13 → [13,6,8]

Task 7 (4) → Machine 1 (min end 6): end = 6 + 4 = 10 → [13,10,8]

Task 8 (3) → Machine 2 (min end 8): end = 8 + 3 = 11 → [13,10,11]

Explanation :
***

Task 4 (duration = 5)

Current machine end times before Task 4: [4, 6, 3]

Greedy rule: Assign the task to the machine that will finish earliest (i.e., the one with the smallest end time).

Minimum end time: 3 (Machine 3)

Action: Assign Task 4 to Machine 3 → starts at 3, ends at 3 + 5 = 8

Updated end times: [4, 6, 8]

➡️ Explanation: Even though Machine 1 has only processed 1 task so far, its end time is 4. Machine 3 is free earlier at time 3, so it gets Task 4. This is the key principle of list scheduling: always pick the machine that becomes free first, not necessarily the one with fewer tasks.

Task 5 (duration = 2)

Current end times: [4, 6, 8]

Minimum end time: 4 (Machine 1)[7]

Action: Assign Task 5 to Machine 1 → starts at 4, ends at 4 + 2 = 6

Updated end times: [6, 6, 8]

➡️Explanation: Now Machines 1 and 2 both have end time 6. But the algorithm always picks the first occurrence of the minimum, which here is Machine 1.

[7] YouTube Handle : @csquickrevisionshorts

Task 6 (duration = 7)

Current end times: [6, 6, 8]

Minimum end time: 6 (Machine 1)

Action: Assign Task 6 to Machine 1 → starts at 6, ends at 6 + 7 = 13

Updated end times: [13, 6, 8]

➡️Explanation: Even though Machine 2 also has 6 as the current end time, Machine 1 is selected because it appears first in the end_times list.

Task 7 (duration = 4)

Current end times: [13, 6, 8]

Minimum end time: 6 (Machine 2)

Action: Assign Task 7 to Machine 2 → starts at 6, ends at 6 + 4 = 10

Updated end times: [13, 10, 8]

➡️Explanation: Machine 2 becomes free earlier than Machine 3 (8), so it gets the task.

Task 8 (duration = 3)

Current end times: [13, 10, 8]

Minimum end time: 8 (Machine 3)

Action: Assign Task 8 to Machine 3 → starts at 8, ends at 8 + 3 = 11

Updated end times: [13, 10, 11][8]

➡️Explanation: Machine 3 finishes its previous task first (at 8), so it gets Task 8.

[8] YouTube Handle : @csquickrevisionshorts

✅ Key Principle

From Task 4 onwards, machine selection is always based on the current end times:

1. Identify the machine that will become free earliest.

2. If multiple machines have the same end time, pick the first in the list.

3. Assign the task to that machine and update its end time.

This ensures that tasks are distributed dynamically to minimize idle time, which is why the makespan is kept reasonably low even though the solution is not always perfectly balanced.

➡️ Statistics:

Makespan: max([13, 10, 11]) = 13

Average load per machine: sum(tasks)/num_machines = 34 / 3 ≈ 11.33

Efficiency: (sum(tasks)/(num_machines*makespan))*100 = (34 / (3*1[9]3))*100 ≈ 87.18%

[9] YouTube Handle : @csquickrevisionshorts