Question 1. Pseudocode with Dynamic Memory Allocation

1. Intialise the queues (Job dispatcher Q, RT job dispatch Q, Normal job dispatch Q, **Level 0 Q**, **Level 1 Q** and **Level 2 Q**)
2. Create arena (i.e. head of doubly linked list of mab)
3. Populate Job dispatcher Q from input file
4. Ask the user to enter an integer value for **time_quantum**
5. While there is a currently running process or any of the queues are not empty
    i. Unload all arrived processes from Job dispatcher Q into RT or Normal job dispatch Q
    ii. If there is a process running
        a. If the priority value is **0** (**Level 0** process running)
            A. **SIGINT** to terminate process
            B. Calculate turnaround_time and wait_time for process
            C. Free process struct memory and mab
            D. Flag that this iteration involved freeing of a process
        b. If the priority value is **1** (**Level 1** process running)
            A. Decrease <remaining_cpu_time> by **time_quantum**
            B. If time has been exhausted
                - **SIGINT** to terminate process
                - Calculate turnaround_time and wait_time for process
                - Free process struct memory and mab
                - Flag that this iteration involved freeing of a process
            C. Else
                - **SIGTSTP** to suspend process
                - Set priority to **2**
                - Enqueue to tail of **Level 2 Q**
        c. If the priority value is **2** (**Level 2** process running)
            A. Decrease <remaining_cpu_time> by 1
            B. If the time has been exhausted
                - **SIGINT** to terminate process
                - Calculate turnaround_time and wait_time for process
                - Free process struct memory and mab
                - Flag that this iteration involved freeing of a process
            C. If there is another process in **Level 0 Q**
                - **SIGTSTP** to suspend process
                - Enqueue to head of **Level 2 Q**
                - Dequeue process from the head of **Level 0 Q**
                - Start and set as the currently running process

D. If there is another process in **Level 1 Q**
- **SIGTSTP** to suspend process
- Enqueue to head of **Level 2 Q**
- Dequeue process from the head of **Level 1 Q**
- Start and set as the currently running process

E. Else
- If time has been exhausted
  - **SIGINT** to terminate process
  - Calculate turnaround_time and wait_time for process
  - Free process struct memory and mab
  - Flag that this iteration involved freeing of a process

iii. Unload all jobs that can be admitted from RT or Normal Qs into their respective Level Qs (i.e. check if there is free memory and add to level Q)

iv. If no process was freed this iteration

a. If there is no process running and at least one of the Level Qs aren't empty (we schedule the next job with highest priority)

A. If **Level 0 Q** is not empty
- Dequeue process from the head of **Level 0 Q**
- Start and set as the currently running process

B. Else if **Level 1 Q** is not empty
- Dequeue process from the head of **Level 1 Q**
- Start and set as the currently running process

C. Else if **Level 2 Q** is not empty
- Dequeue process from the head of **Level 2 Q**
- If the process is a suspended process
  - Send **SIGCONT** to resume it
  - and set as the currently running process
- Else
  - Allocate memory to memory linked list
  - Start and set as the currently running process

b. If there is a current process

A. Sleep for **decrease_time** (different for each level process)

B. Increase timer by **decrease_time**

c. Else

A. Sleep for 1

B. Increase timer by 1

v. Unflag the freed variable

vi. Go back to 4.

6. Calculate the average turnaround and wait times

7. Terminate dispatcher

Question 2. Example and Explanation of 2 ‘Job Dispatch List Files’ and their Outputs

Example 1. Testing for the “First Fit” property & more

Process	<arrival time>	<cpu_time>	<priority>	<memory>
1	0	5	1	700
2	1	5	1	400
3	2	5	1	150
4	3	5	1	150
5	4	5	0	1024
6	5	5	1	1024

With a **time_quantum** of 5 or greater, the following things are tested in this example:

- The process is allocated to the first memory fit
- Pre-emptive properties from part 1 still hold
- After the process is finished, it will set the memory to free and then merge the adjacent free nodes together
- It prioritises process 5 before processes 2, 3 and 4 (i.e. it first considers the **Level 0 Q** over the **Level 1 Q**)

However, with a **time_quantum** of 4 or less, the following things are tested in this example:

- RT Q won’t be allocated if memory is full
- **Level 2 Q** is tested to ensure that it preempts the node back to the head of the queue
- Suspending processes doesn’t break the code from part 1 (i.e. all of part 1 still works)
- Prioritises process 5 before processes 2, 3 and 4 (i.e. it first considers the **Level 0 Q** over the **Level 2 Q**)

Expected Outcomes:

- **Time Quantum: 5**
  - Order of jobs finishing → 1,5,2,3,4,6
  - Average Turnaround Time → 15 seconds
  - Average Wait Time → 10 seconds
- **Time Quantum: 4**
  - Order of jobs finishing → 1,5,2,3,4,6
  - Average Turnaround Time → 17 seconds
  - Average Wait Time → 12 seconds

Example 2. Testing for Correctness in the Memory Linked List

Process	<arrival time>	<cpu_time>	<priority>	<memory>
1	0	5	1	512
2	0	5	1	511
3	1	9	0	512
4	2	16	0	128
5	4	5	1	400
6	47	3	0	1024

With a **time_quantum** of 3, the following things are tested:

- A block of size 1 of offset 1023 is at the very end of linked list is valid
- Suspending a process doesn't free memory (i.e. as long as the process is still on the queue its memory is stored)
- If the head of linked list is freed the processes will still be able to find "first fit" correctly
- All of example 1 is also tested (i.e. It prioritises 0 over all other jobs and sets and merges correctly)
- When no processes are running and no jobs are on any queue apart from the job dispatch queue, the dispatcher will sleep until arrival time
- When process is terminated, the dispatcher will immediately allocate job appropriately and run it
- All pre-emptive properties hold and are correct

With a **time_quantum** of 5, the following things are tested:

- The same as above, but the order of jobs starting are different

Expected Outcomes:

- **Time Quantum: 3**
  - Order of jobs starting → 1,2,3,4,5,6
  - Order of jobs finishing → 1,3,4,2,5,6
  - Average Turnaround Time → 15 seconds
  - Average Wait Time → 10 seconds
- **Time Quantum: 5**
  - Order of jobs starting → 1,3,4,2,5,6
  - Order of jobs finishing → 1,3,4,2,5,6
  - Average Turnaround Time → 20 seconds
  - Average Wait Time → 12.83 seconds