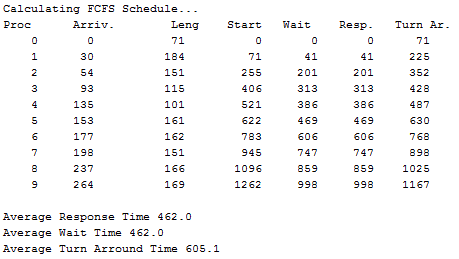
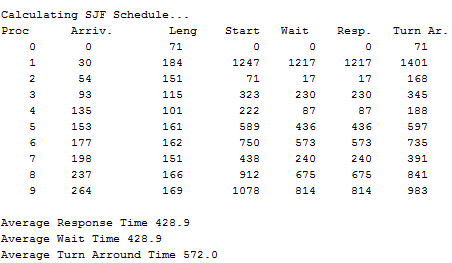
**Task 2.1:**

**2.**





The waiting time in FCFS algorithm is usually greater than SJF, because later processes must wait for all its before processes finished to execute, if there’s a process with great CPU burst, it will hold CPU very long and slow down the computer. In FJS algorithm, CPU will pick a process need the smallest CPU burst to execute. Hence, it decreases the total of waiting time. In the below example, if we change the length of Process 0 ( bigger ), we can see the effect more clearly.

**Task 2.2:**

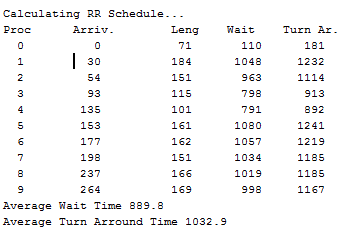
**1.**

The shortest-remaining-job-first algorithm without pre-emption is the same to SJF. CPU will be given to a shortest process in queue and this process will hold CPU till it finish. Even if there’s new process arrives in queue, the currently process will just continue to run.

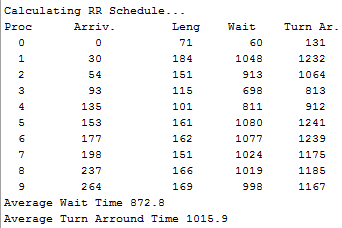
**2.**

When the processes’ length is long, the quantum time need to be big to optimal CPU usage. If the quantum time is too short, the CPU must switch between processes many times, the process will be divide into many part and it make the waiting time bigger. Conversely, if the quantum time is too long when process’s CPU burst is short will response to short interactive request or may be become useless.

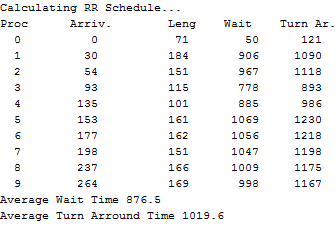
**Quantum = 10 :**

****

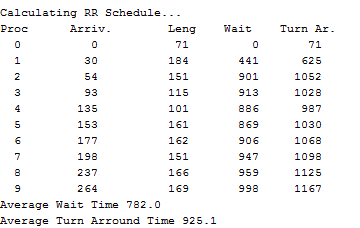
**Quantum = 20 :**



**Quantum = 50 :**



**Quantum = 100 :**



In this example, the average length of Process is big, so the quantum small like 10, 20 is not as better as 100.

**Task2.3:**

The basic idea is straightforward: each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

There’re some problems of priority scheduling.

* If process has higher priority, it will always have a change to run, and process with lower priority might wait forever till the highest process finished.

To prevent high-priority processes from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick (i.e., at each clock interrupt). If this action causes its priority to drop below that of the next highest process, a process switch occurs. Alternatively, each process may be assigned a maximum time quantum that it is allowed to run. When this quantum is used up, the next highest priority process is given a chance to run.

* The other problem is about using multi queues. It is often convenient to group processes into priority classes and use priority scheduling among the classes but round-robin scheduling within each class. For example, consider a process that needed to compute continuously for 100 quanta. If we give all processes a large quantum, it can be poor response time. The process would initially be given one quantum, then swapped out. Next time it would get two quanta before being swapped out. On succeeding runs it would get 4, 8, 16, 32, and 64 quanta, although it would have used only 37 of the final 64 quanta to complete its work. Only 7 swaps would be needed (including the initial load) instead of 100 with a pure round-robin algorithm. Furthermore, as the process sank deeper and deeper into the priority queues, it would be run less and less frequently, saving the CPU for short, interactive processes.

**3. Optional**

- Context: implemented.

The context switching time effects non-preemptive algorithm slightly because the number of switching is small. But when we use pre-emptive algorithm, especially with a small quantum, the context switching will slow the computer critically, because when we use pre-emptive algorithm, the CPU will switch process many time, for every quantum, every termination, interrupt.