CSE 344 System Programming Midterm Report

In this assignment, we're asked to simulate a mess hall of a university. We have 3 actors, and if we include bonus part we have 4 actors. We have **supplier** that supplies foods to cooker to cook them. Supplier will be put items on kitchen which has finite size. We have **cooker** that take foods from kitchen cooks them and serve it to counter to students take them. **Cooker** is important in this simulation because it interacts with other access and at some point it needs to make decision since we don't fill **counter** with same items since **counter** also has finite size. **Students** tries to take foods from counter but not one by one. They take if all three type of food is available. So this is the place that cooker must decide. After a student grab its meal, it goes into another place to wait. This time students race for empty table. Once a student finds and empty table, it sits, eats and if there is chance to turn back counter, turn back counter and tries to same process again.

If we look problem, we have similarities with classical IPC problems. For example, **supplier** fills kitchen that can be interpreted as buffer, this is just like producer consumer problem with many consumers. Consumers is here **cookers**. If buffer is empty then consumer must wait just like **cooker** waits for kitchen**,** if kitchen is full then **supplier** must wait just like producer of classic problem. It's same for counter, since we have finite place for foods at counter. Also when we distinct graduate and undergraduate student, just like reader-writer problem. If there is a writer, no other reader is allowed to enter.

First, I started creating supplier. Supplier will supply 3 type of food, L times to M student(or U + G student). We read foods from a file. This file must contain 3LM food which every LM stands for one type of food. So at beginning I need to make sure there this number food available. I opened file and readed all file one by one and record which food is we encounter. After that I compare the numbers with LM. If there is an inequality then I exit since there will be deadlock because some **students** cannot be served. If file is pass that test, now we can proceed to serving foods to kitchen. For a successful operation file must contain exactly 3LM food and LM for each type. Also input format must be one line that contains food types by **'P', 'C'** or **'D'** and no other line must added. For example input file like: *SSSDDDCCCSDCDSC*. One simple line with no empty space between food types. We print message as food types changes. We also asked to print stats of kitchen's current situation. For that we need a shared object since counter also will be try to access these stats of counter. So we must restrict that shared memory named **kitchenStats**. This object will hold in order, number of items in kitchen, and other will hold number of soup, main course and desert in kitchen. To restrict access I used binary semaphore(mutex) **mutex0**. Everytime I tried to access kitchen stats and print them I need this mutex lock. For example before going delivering soup on kitchen. After that I need to make sure that there is empty place in kitchen before deliver some type of food. This is just like Producer-Consumer (empty) semaphore. Before start process, I initialized a semaphore called **sem0** with kitchen size. Of course, at the beginning we have empty kitchen. This semaphore holds number of empty space in kitchen. Every time supplier that tries to serve some food on kitchen, makes sure there is empty room at kitchen by applying wait operation on **sem0**. Since there is no other **supplier**, I don't need worry about context switching after applying wait. Now, we used one place from kitchen, we can update kitchen stats. For every type of meal, I used a semaphore in order **sem7, sem8** and **sem9**. These semaphores will help us when we wait on some special type of food at counter that needs to immediately served, because it is the missing one. For that purpose, every time I added some type of food to kitchen I applied post operation on them semaphores which will be waited on counter. After that, just like Producer-Consumer (full) semaphore, I posted **sem1**, which initialized with 0 at the beginning. This semaphore indicates fullness of kitchen and **cooker**, wait for that semaphore. By that semaphore, cooker understand that there is a food, and I can take it. But this not happens always since we need to make sure that 3 distinct item at counter for **students** take it. After done supplying, **supplier** finishes its job and prints its message and waits because supplier process is **parent** of all others. It needs to wait until all childs terminate.

Secondly, I created **cooker**. **Cooker** crucial part of problem. Cooker must take foods from kitchen by providing, when the counter is full there must be 3 distinct food type food available. Also it need to put foods taken from kitchen to counter with no exceed to counter size. In my design, I restricted access to cooker place. This means only one cooker is active at instance. Reason of doing that is, let’s think a scenario. If we have one place on counter and we are 0 soup at counter. Then cooker must wait for that because if we put some food other than soup we will cause a deadlock, no **students** can be served. So when a **missing food case** occurs, then current cooker must wait and no other cooker can go kitchen. By that only one **cooker** active at instance. I provided this lock to counter by **mutex2**. This **mutex2** restricts the access of cooker. After that I need to access counter’s current situation. Just like kitchen stats I created a shared object that stores counter stats which is **counterStats** array. Once a cooker access its region, first it must check current situation of counter to prevent deadlocks or to terminate itself. Since counter stats is also a shared object, we need to restrict access to this object. By **mutex1**, I restricted access to counter stats and only one process can access it at a time. Counter will provide us to following information: How many items in counter, number of items in counter for each type of food, number of student in counter that waits for food and total served food. When cooker access counter stats it checks if cookers reached to total serviced food count. If this is the case **cooker** break its endless loop and exists. Otherwise, we need to check is there any situation that can cause deadlock. For that purpose we check first how many empty space in counter. If there are one or two empty space, then we need make sure that, there is distinct type of foods. The reason for 2 is for example we filled 4 sized counter with SS and there are two place left on counter. Then we need C and D for empty places. So, if we do that only for 1 place left, then we can fill the third place with S again and there will be again a deadlock. After that we look counter stats for each type of food. If one of them 0, then we need to wait until that specific food comes. This waiting can be done by **sem7, sem8** and **sem9** as we saw in supplier. If there is missing one, we need to make sure that specific food type in kitchen, so cooker can take it and put it to the counter. By applying wait to this semaphores, if there is no food for that specific type of food then we will wait and no other food can be put on counter since we locked all cookers. In worst case, there is no food until the last place of kitchen filled with waited food type. By the constraints kitchen size 2LM+1 provide us there is at least one type of food will be available for cooker to take it before filling kitchen. After that specific food comes, we explicitly say to current cooker to take this type of food. In other cases, we simply look food types one by one and if there is one we simply take it. After we decide what kind of food type we take from kitchen, we can now take it and we simply apply wait operation on **sem1** which indicates fullness of kitchen and we can decrease it. After that we can apply wait on **sem7, sem8** and **sem9** since we decide food type and we take it. At above, while waiting on **sem7, sem8** and **sem9** after wait is done we apply post on them again to don’t corrupt this normal process. After take food, we can update kitchen stats by accessing shared object **kitchenStats**, with **mutex0**. Now we have food at our cooker's hands but we need to make sure that there is empty room at counter. Also since we take food from kitchen we can post **sem0**, since we take it there is one more empty room at kitchen. Just like emptyness and fullness of kitchen we must wait for empty room at counter. **sem2** indicates emptyness of counter. After taking food from kitchen we apply wait on that semaphore. If semaphore 0, cooker will wait for empty room. After wait return, we update **counterStats** for type of food and and current counter stats and total delivered number. In cooker, we also semaphores for each type of food **P, C** and **D** in order **sem4, sem5** and **sem6**. This semaphores posted in **cooker** when food type putted on counter. After we will see that **student** that enter counter apply wait on them in order and makes sure that counter has 3 distinct item that he or she can take and eat. After this values adjusted, since we filled counter we can post fullness semaphore for counter and this is **sem3**. By posting that, student can sure about there is food on counter it can take. After all we release access to cooker by posting **mutex2**. And when serving done, cooker prints message and exits.

For a **student** process starts with updating counter stats. It enters counter and increases number of students in counter by accessing using **mutex1**. After that I locked counter access for every student. This means that, when one student enter taking food stage, no other student can enter and take food from counter. Of course any student can be served any time but one student tries to take 3 type of food from counter at a time. For that purpose I locked every student with **mutex3**, this restricts access of students to counter. There is no meaning accessing them randomly since if there is missing one all students will be wait even if they access randomly and I want to make access to counter and applying wait on **sem3**, which indicates fullness of counter, be atomic and no race condition will occur. After a student access to counter it try to take foods one by one from starting soup to desert. For that purpose, I used **sem4, sem5** and **sem6** which we posted in **cooker**. By doing that if there is a missing food type on counter we wait for that and suspend process until it is available. By doing that of course all students will wait because it can not enter food taking stage and this is logical since if there is missing type of food, there is no meaning other students to enter food taking stage. After all **sem4 and sem5 and sem6**, waited now we can take all 3 food type for that particular student. Student take them at once and apply 3 wait on counters fullness semaphore **sem3**, to indicate we decreased fullness of counter by 3 and then of course we update counter stats since we take some items on counter, and we will exit counter so, we can decrease number of students at counter. Since we take items on counter, we can increase emptyness of counter since there is empty rooms for new foods. We done that by posting 3 times, since we take 3 type of food, to **sem2**. And then we release lock for food taking stage so other students can take.

After a student take its foods on hands another race begins. Now we race for empty tables. Of course to access table's current situation we need to create a shared object **tableStats**, that shows us current situations of tables and since this is shared object we need restrict access to it and we done that by **mutex5**. We must wait for empty tables. To wait for them we use semaphore **sem10** which initialized by table number. When a student can decrease it, this means student finds a empty table to sit and eat. Since we access table, we need to update table stats and we need store the table number of student sit. After eating, if student has remaining right, then student goes again to eat. When student finished eating it sits up from table he or she sat and we update table stats for that. Since we have one more table for sit and eat we post **sem10**, to allow waiting students sit and eat.

I forked all student and cooker process at supplier process. Supplier is main process and others are child processes. So I waited them synchronously at supplier by first blocking SIGCHLD. After that I used sigsuspend with a emptyMask to wait SIGCHLD and when SIGCHLD comes, I reaped their status with waitpid(-1) in SIGCHLD handler until there is no live child remain.

In case of Ctrl-C or any exit case, I exited gracefully by doing followings:

- I constructed a global dynamic array which holds pid of all child processes. After every fork, I added them to this array to terminate them later. **addChild** method adds pid to this global dynamic array.
- I putted a global semaphores and mutextes pointer, to destroys mutexes and semaphores later in **destroySemaphoresAndMutexes**. To destroy them properly I assigned shared memory pointer of semaphores and mutexes to this global variable and then I put a counter values **mutexNum** and **semaphoreNum**, after every unnamed semaphore initialization, I incremented them. In **destroySemaphoresAndMutexes**, in a loop, I destroyed them sem_destroy. This destroy done before shared memory unlinked and is done in parent since to not effect semaphore waits on that values. After terminated all childs this semaphores will be destroyed.
- Also I have a function called **destoryShareMem** which unlinks shared memories for all child and parent.
- I registered **destoryShareMem** for all processes and **destroySemaphoresAndMutexes** for parent process. So in case of exit, this functions will run resources will be freed.
- In case of Ctrl-C:
    - If child catches **SIGINT** it then using **sendChild** function it directs this signal to parent. Parent catches it and starts termination. First it sends **SIGTERM** to all childs that pid's stored in global array we constructed above. In childs **SIGTERM** handled by **termChild** which simply calls exit and this call pulses atexit functions. And atexit function will start and free shared memories. After parent in handler reap status of all childs, it simply exit itsel by destroy semaphores and lastly unlinking shared memories.
    - If parent catches **SIGINT**, same procedure will execute except, there will be no signal sending from child to parent. Then all resources freed up. The last call will be **destroySemaphoresAndMutexes** and in there we close input file if it is opened and free childs dynamic array.

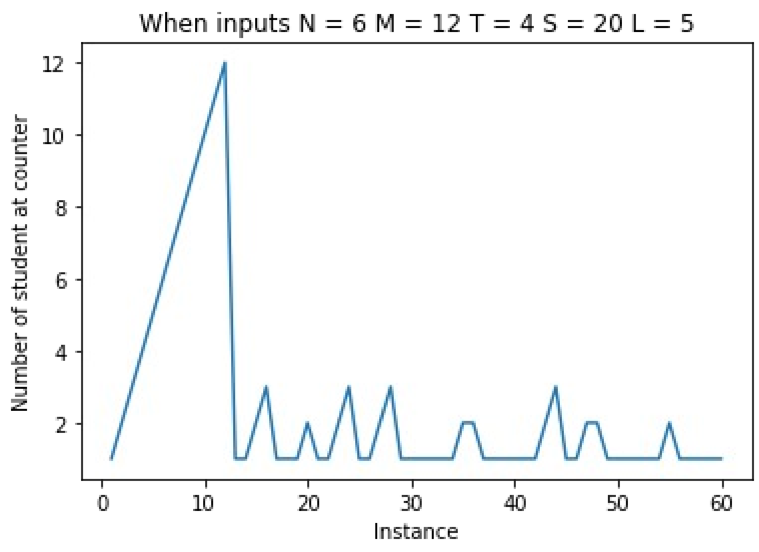
NOTE:Before start showing graph, since simulating issues I putted sleep when student take food to simulate eating. I made test and results of graphs according to this fact. But in normal code I didn't use sleep and I commented that. It works fine without sleep but students sits and eats consequently and could not use table size effectively. This is just done for simulation purpose and no didn't used in real program.

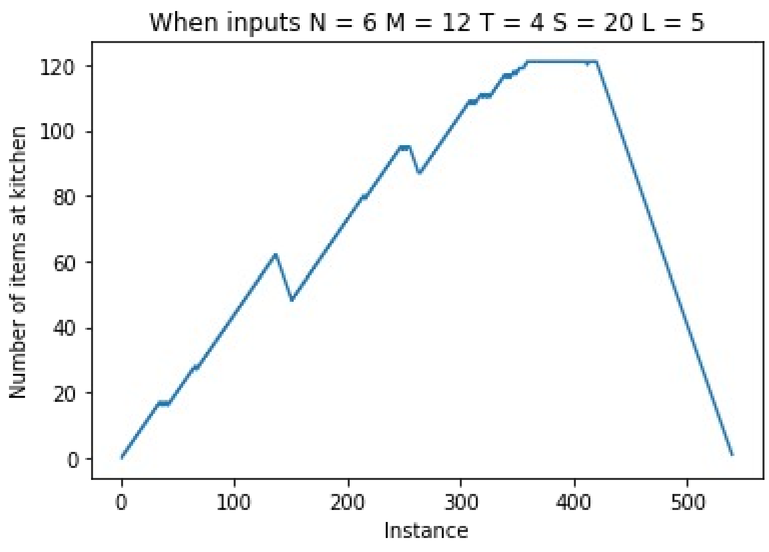
To run program you just simply compile program with *make* command and then you run program as like

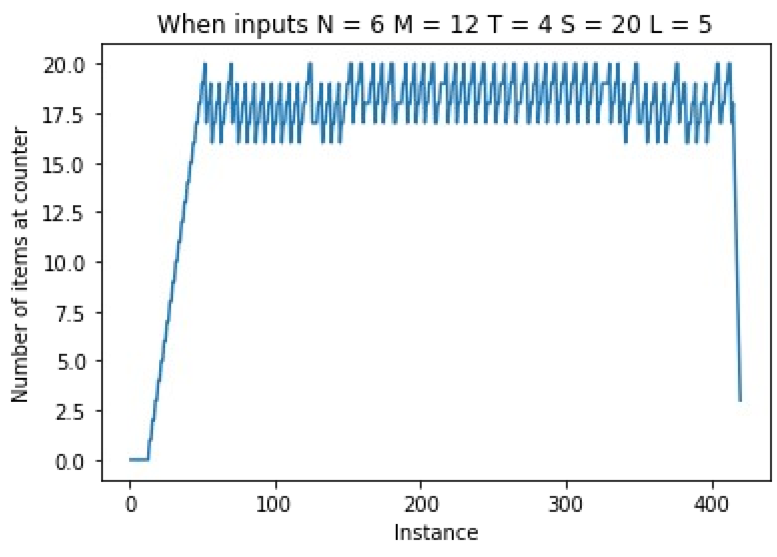
./program -N [cookNum] -M [studentNum] -T [tableNum] -S [counterSize] -L [turnTime] -F [filePath]

Graphs

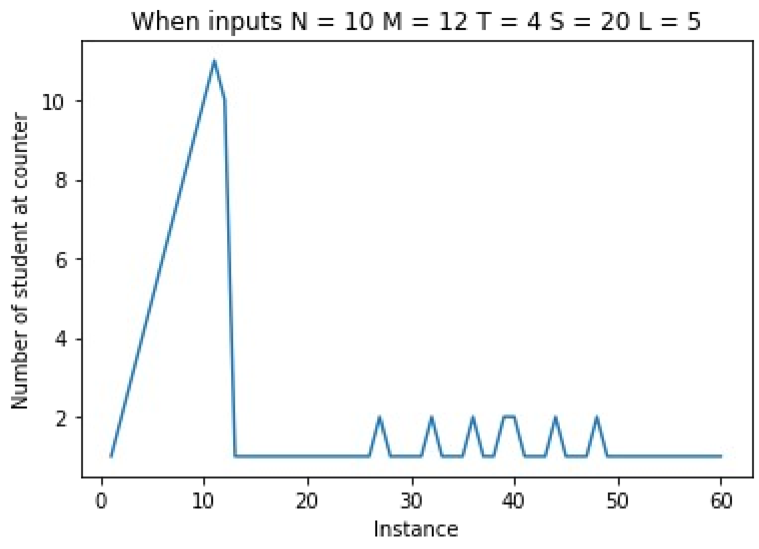
- First we change **cooker number N** and keep same remaining. When input N = 6 M = 12 T = 4 S = 20 L = 5 graphs shows as:

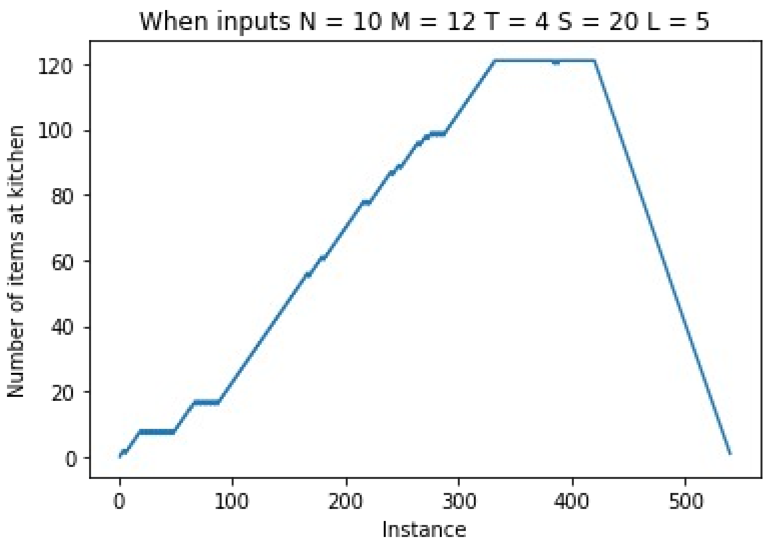


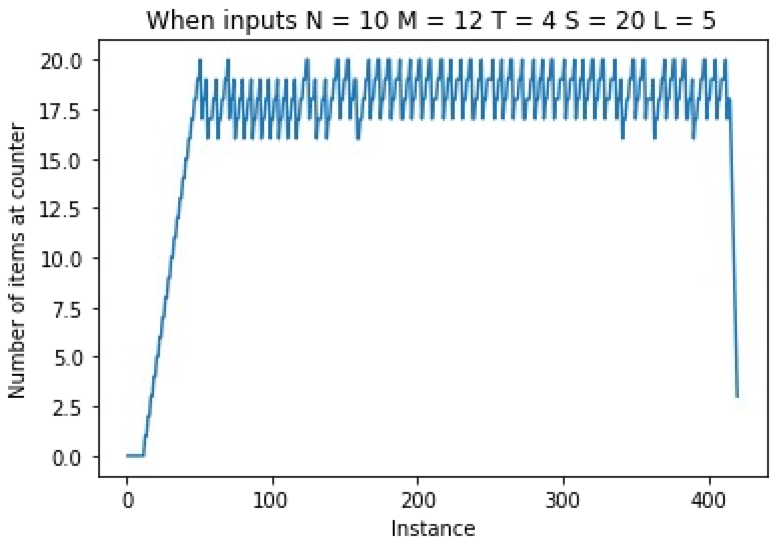




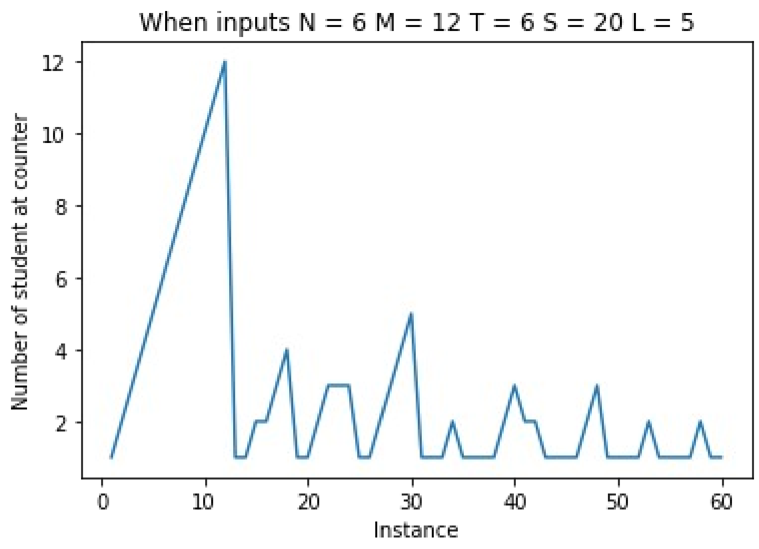
- If we change **N** to 10 i.e input when N = 10 M = 12 T = 4 S = 20 L = 5

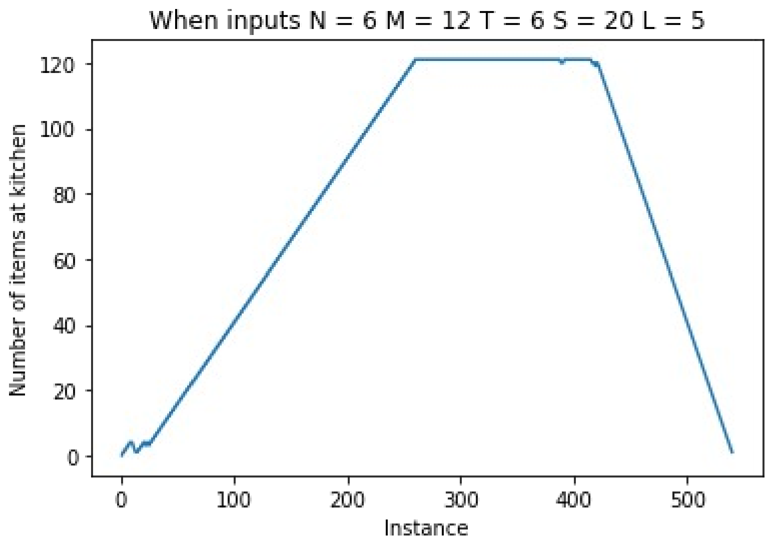


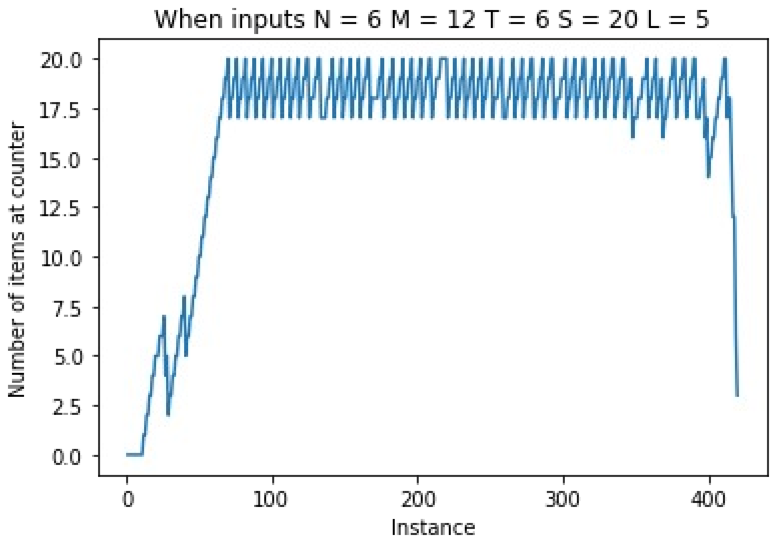




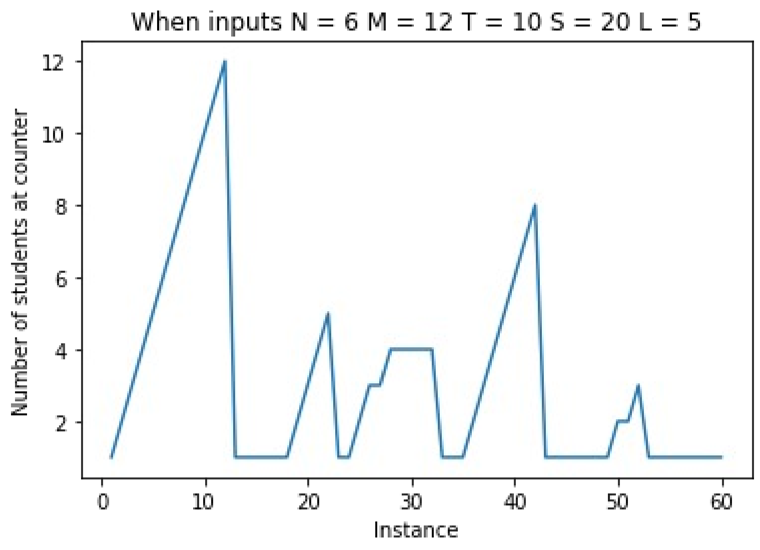
- Then, If we change **table size T** and keep others same we will have following graphs. If input is N = 6 M = 12 T = 6 S = 20 L = 5

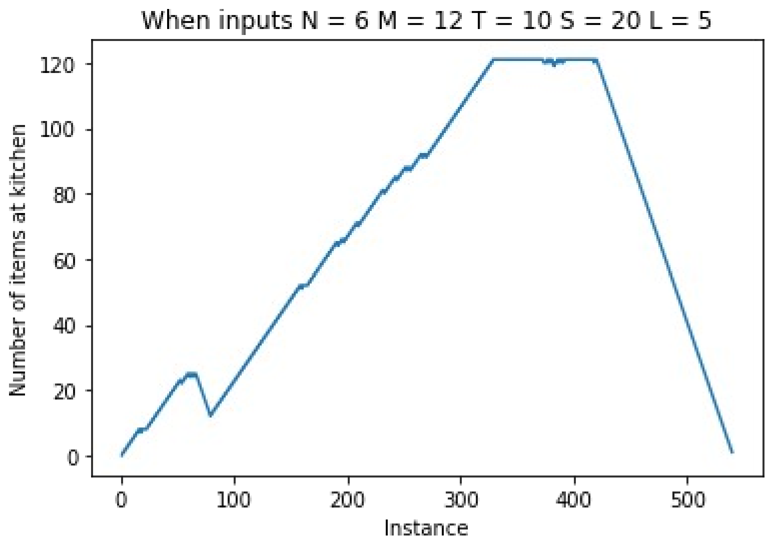


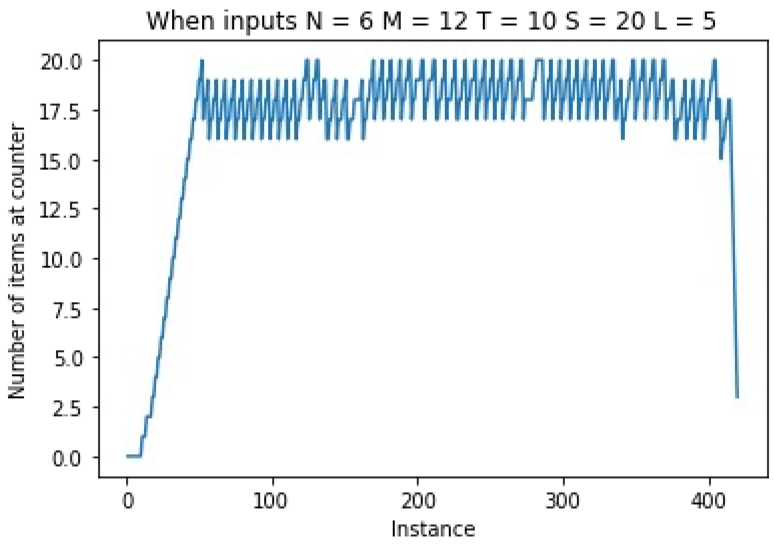




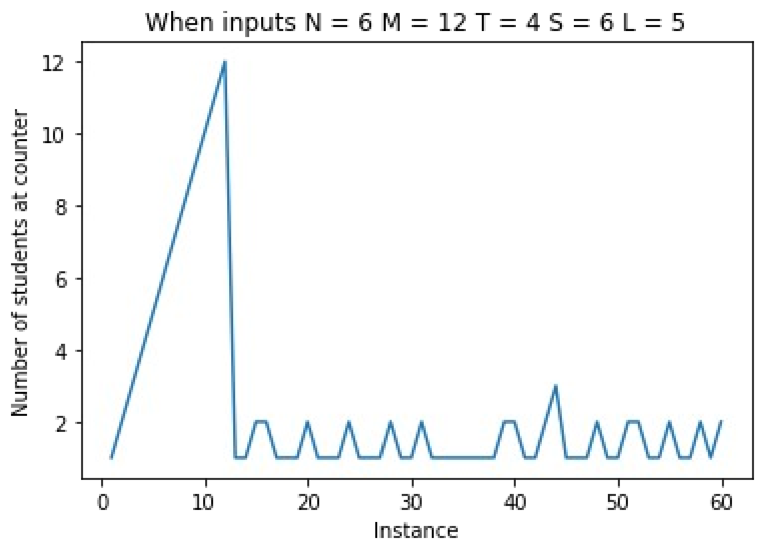
- If we change **table size T** to 10 i.e when input N = 6 M = 12 T = 10 S = 20 L = 5

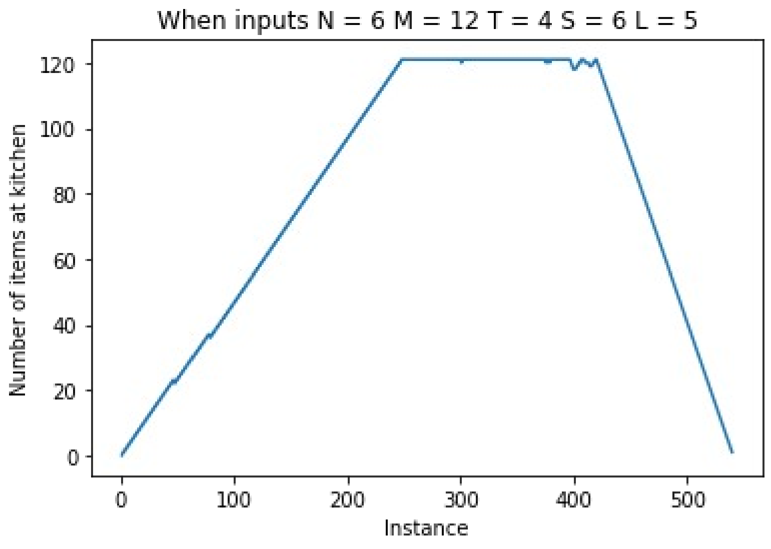


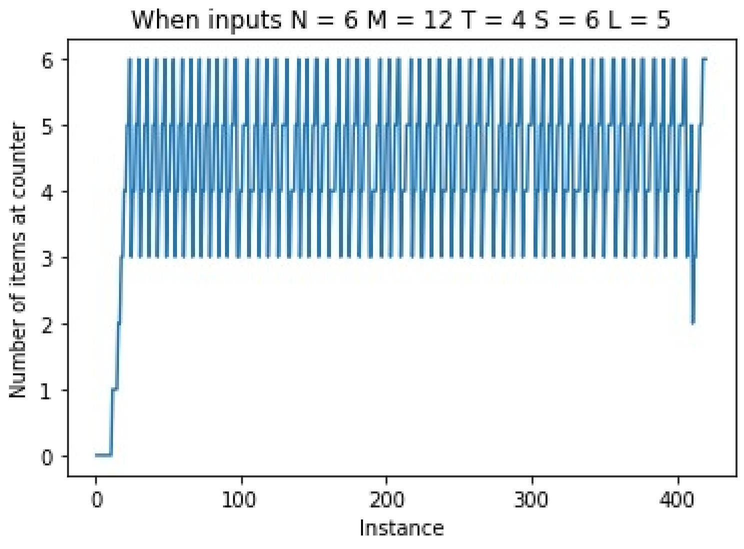




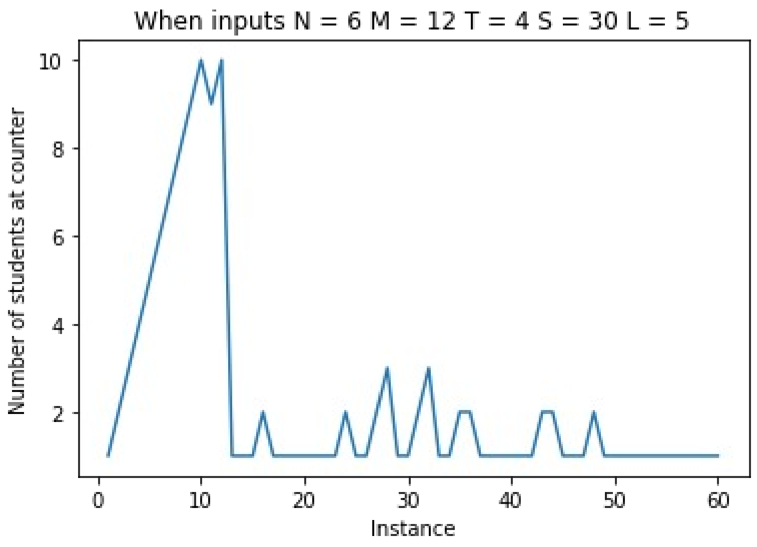
- Now, we keep all remaining inputs same and change **counter size S**. First we apply inputs when N = 6 M = 12 T = 4 S = 6 L = 5

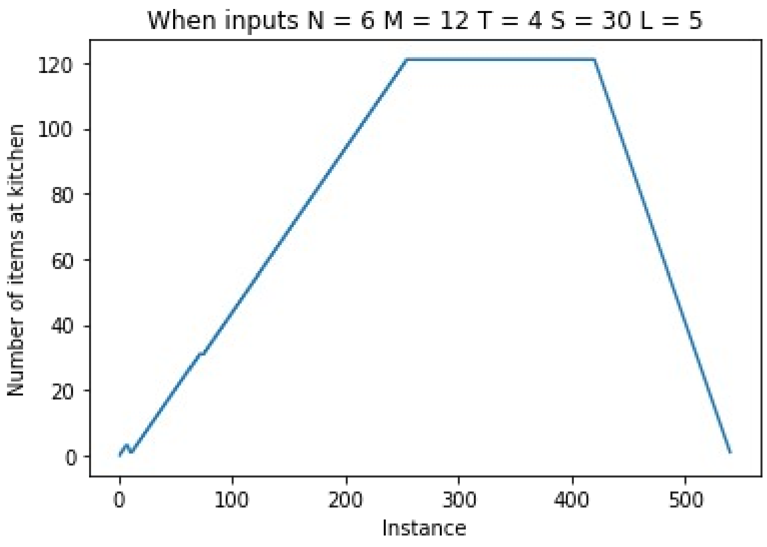


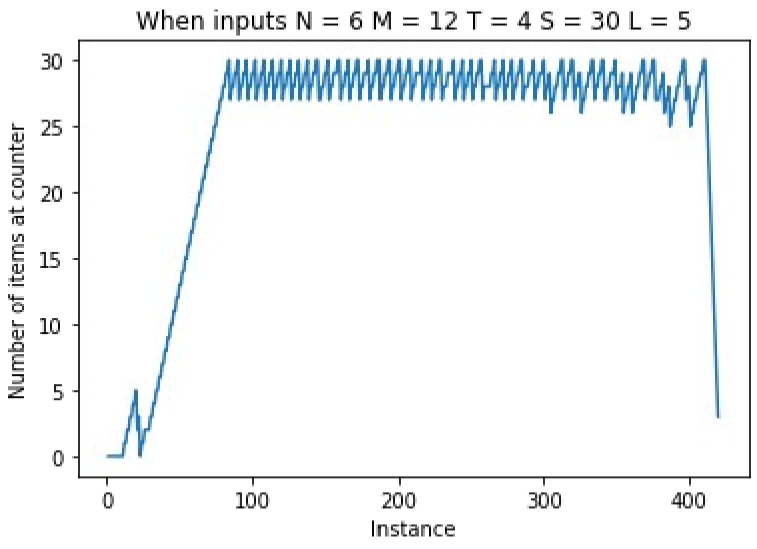




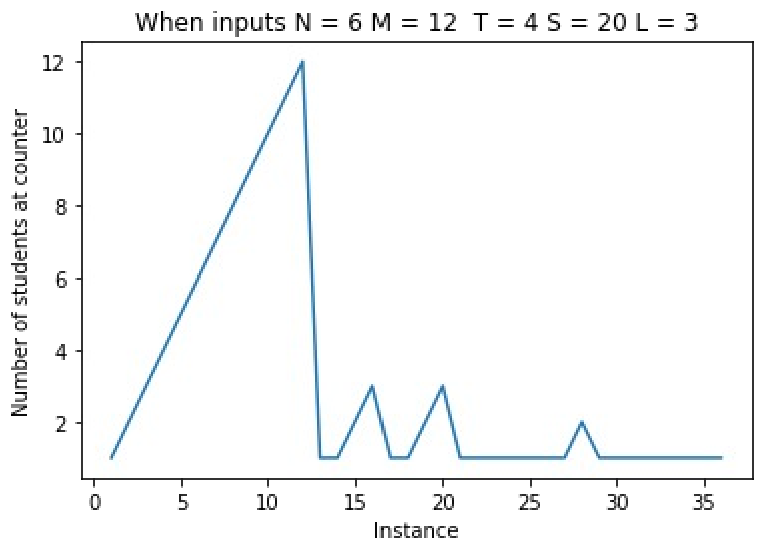
- If we change **S** to 30 i.e when inputs N = 6 M = 12 T = 4 S = 30 L = 5

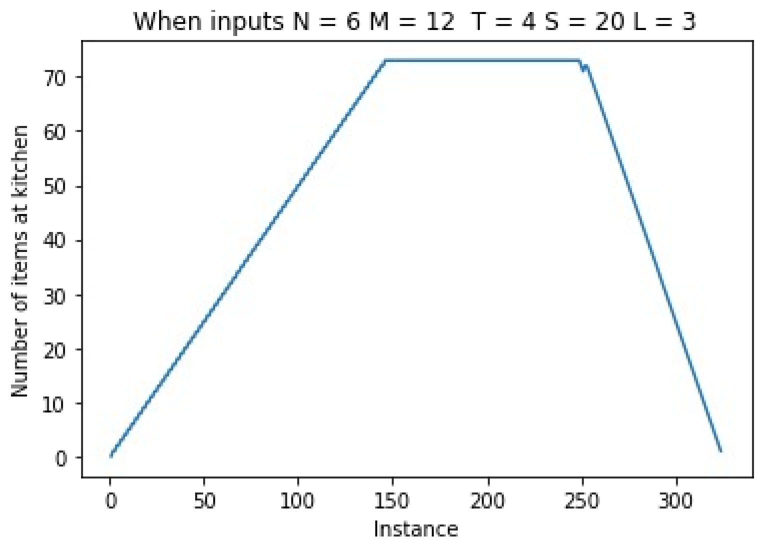


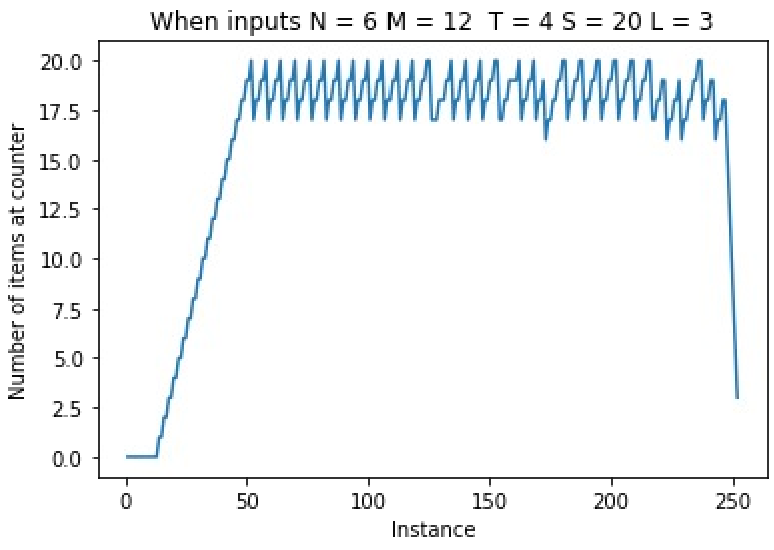




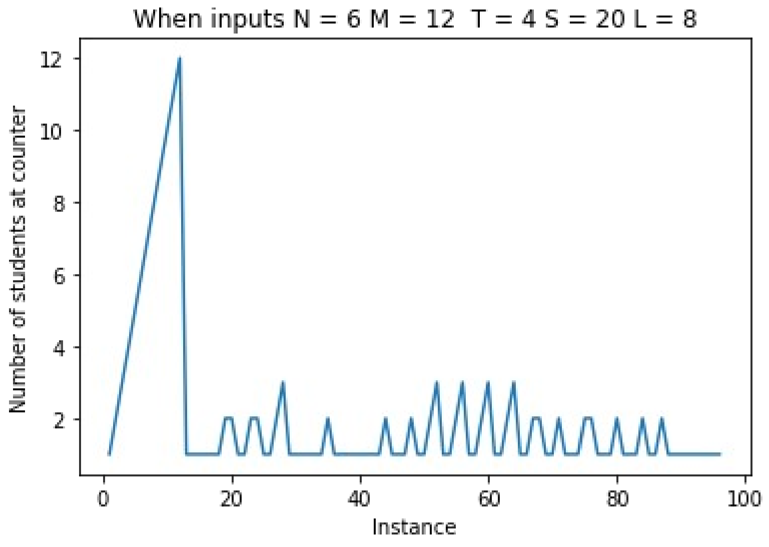
- Now we keep all inputs same except we change **turn time** L. First we make turn time 3 i.e when inputs N = 6 M = 12 T = 4 S = 20 L = 3:

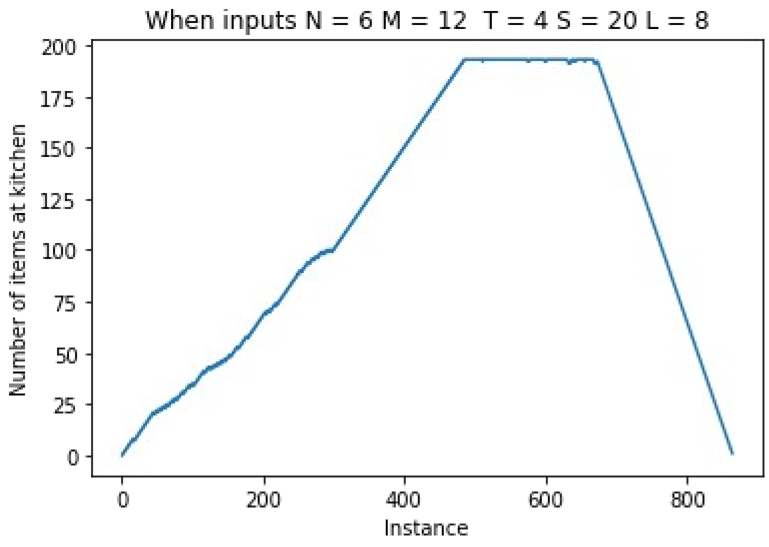


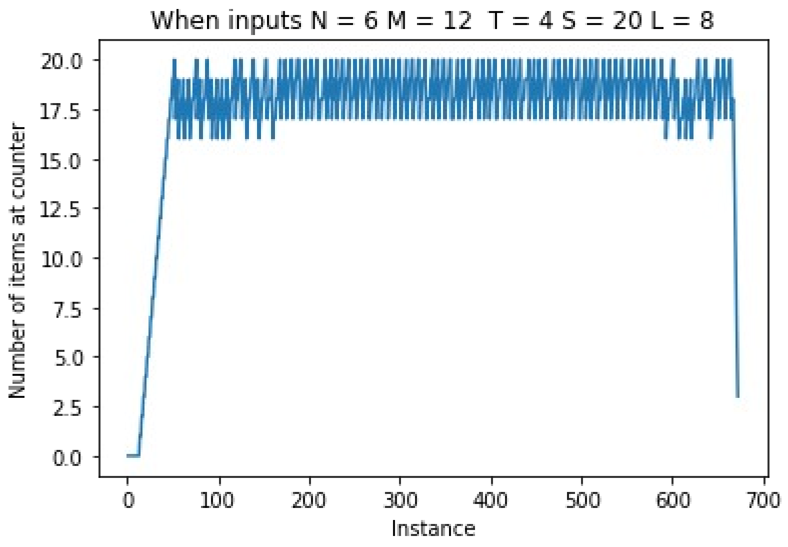




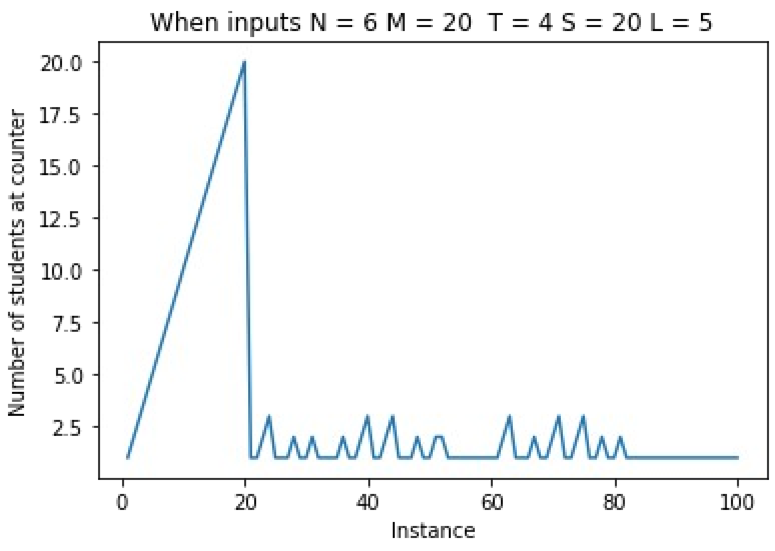
- Now if we change **turn time L** to 8 i.e when input N = 6 M = 12 T = 4 S = 20 L = 8:

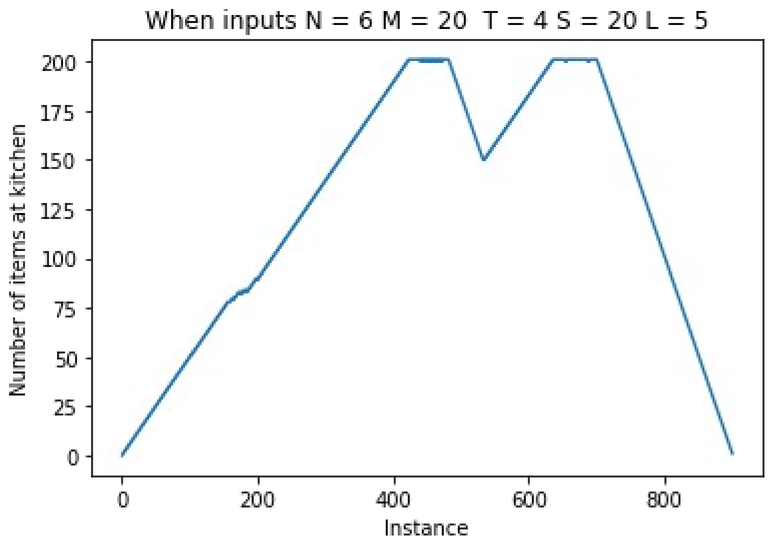


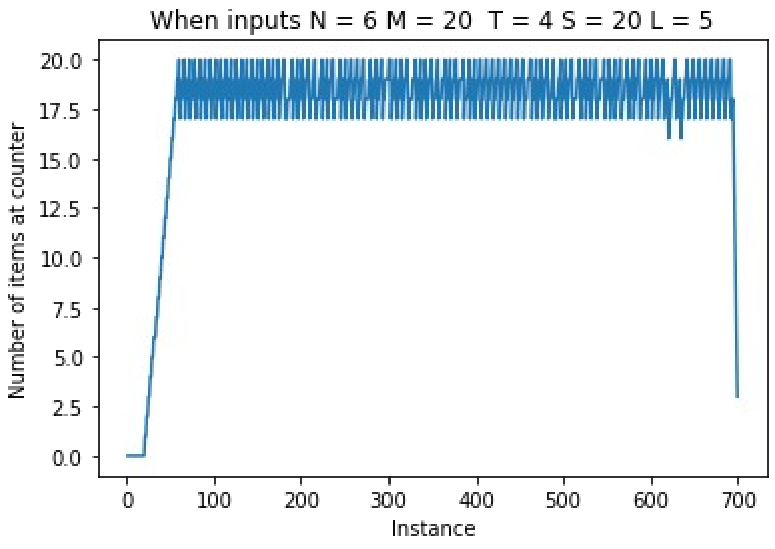




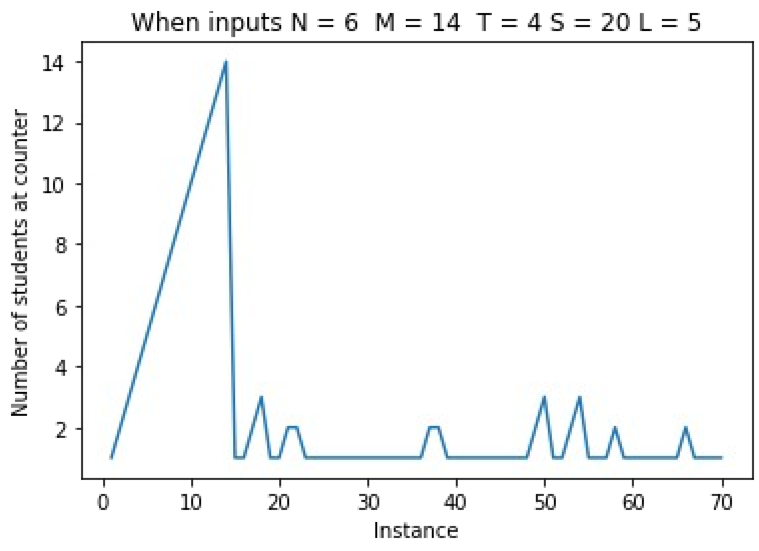
- Lastly, we keep all inputs constant and we change **student number M**. First we make M = 20 and others same i.e input is N = 6 M = 20 T = 4 S = 20 L = 5

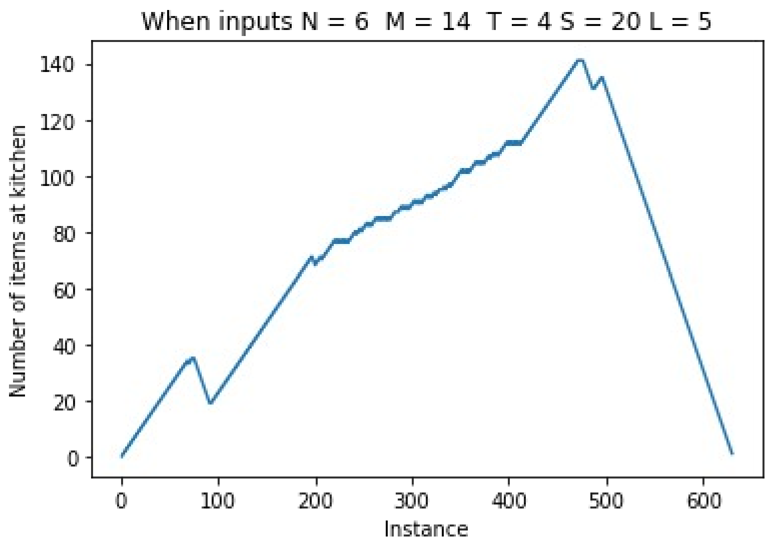


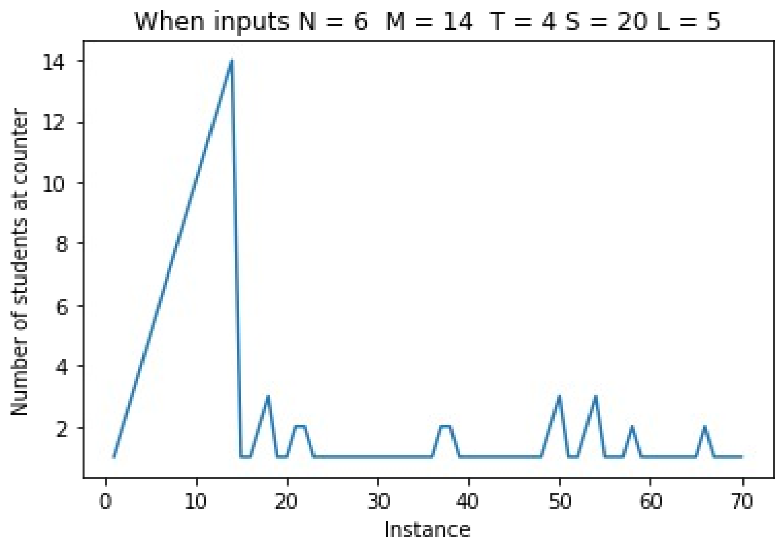




- Then we keep values same but we change **student number M** to 14 i.e input is N = 6 M = 14 T = 4 S = 20 L = 5


When inputs N = 6 M = 14 T = 4 S = 20 L = 5
Number of students at counter
Instance


When inputs N = 6 M = 14 T = 4 S = 20 L = 5
Number of items at kitchen
Instance


When inputs N = 6 M = 14 T = 4 S = 20 L = 5
Number of students at counter
Instance