Christopher Flippen
Professor Ghosh
CMSC 312 - Operating Systems
17 April 2020

Assignment 3 - Multi-Process, Multi-Threaded Print Server Report

Parts 1 and 2: Code Outline, Explanation of Logic, and List of Implemented Components

List of Implemented Parts of the Assignment:

This program has functioning producers, consumers, shared memory, and semaphores.

Counting semaphores implemented for assignment 2 are used for the “full_sem” and “empty_sem” semaphores. These counting semaphores function the same way as in the assignment 2 bounded-buffer problem by keeping track of if the print queue is full or empty.

The printer queue is implemented using a singly linked list of structs in shared memory created with mmap. Two other shared memory segments are used: one for the counting semaphores and one storing an int which counts how many jobs total will be created by the producers.

The job structs contain their own sem_t type mutexes (using mutexes inside the jobs rather than for the list as a whole is listed as extra credit on the assignment).

A signal handler is used to close the program cleanly at any time with CTRL C. Several tests of using “ps -aux | grep flippenc” and “ipcs | grep flippenc” showed no loose processes and open shared memory segments after running and CTRL C-ing the code.

The program calculates and prints the amount of time it spent running the producer and consumer functions (total run time not including the initialization and destruction of variables).

Note: I pair-programmed a large amount of this assignment with Chelsea Greco, so much of the logic in our two programs is very similar, but our code is still our own. We also used the same base document for creating our charts of run times of our program, so the format of our charts is the same, but our data is from our own programs.

Known Issues:
The program can sometimes segfault or deadlock if more than ~20-30 producers are used. The program seems to run very consistently with up to 20-30 producers, but it becomes less and less consistent with more than that. Segfaults and deadlocks seem to be more common if there are a

large number of producers with a small number of consumers such as 30 producers with 2 consumers. The opposite problem does not occur, for example, 10 producers and 5000 consumers seems to work consistently.

Detailed explanations of the functions are below:

The Job Struct:
This structure stores job requests inside a struct call job.
This struct contains:

An int size - representing the job size (used as the time in which the consumer spends consuming the job)
An int id - representing which producer created this job
A sem_t mutex - the extra credit component of the assignment in which individual jobs have mutex semaphores instead of the whole queue being controlled by one mutex
A job pointer next - pointer to the next element in the linked list

The Main Function:
The main function starts by getting the parameters from the command line - the program exits and prints a message if there are too few or too many parameters
Next, the program declares the signal handler for SIGINT so that it can stop cleaning even if CTRL C is pressed while it is initializing the shared memory

Next, it creates the shared memory segments:
One segment is semShared which holds the 2 counting semaphores
The two counting semaphores are initialized using counting_sem_init

The second segment is totalNumJobsShared which holds how many jobs the producers will create (this is an integer that allows the producer to know when it can stop).
This int is initialized to 0.

The third segment is queue_ptr which holds the printer queue. This segment is made using mmap rather than shmat and shmget like the other two because mmap made working with pointers easier. The printer queue uses a linked list structure and contains two separate job queues. One of these is the "empty" job list pointed to by free_head. The other list is the "full" job list pointed to by job_head. When a job is created, instead of it being malloced into the shared memory, it is taken from the empty list and filled with data. When a job is consumed, its data is removed and it is moved back into the empty list. I got this idea of using two lists in mmap from (https://stackoverflow.com/questions/31450163/share-linked-list-between-child-and-parent-in-c/31460186#31460186). To initialize the job queue, the main method fills the "empty" list while making the "full" list initially NULL. Initializing the jobs in the empty list involves linking them together, setting their size and id values to -1, and initializing their mutex semaphores with sem_init. This code does not use a buffer_index for the SJF or FCFS versions. In SJF, jobs are added into their appropriate locations in the queue and in FCFS, jobs are appended to the end of the queue. The use of free_head and job_head makes buffer_index unnecessary.

Next, the start time of the program is obtained using clock_gettime. This is stored as a global variable that is used to calculate the running time. In one of the live lectures, Professor Ghosh said to leave out initialization time from the running time, so the start time is obtained here rather than at the beginning of main.

Next, the forking begins in a for loop. The index of this loop (i) is used to count which producer is being made. This value is sent as a parameter to the producer function, allowing the producer functions to label who made each job.

Once the forking of the producers finishes, the parent process starts creating the threads for the consumers. Once the consumers finish, pthread_join is used on the terminated threads. Next, a SIGINT is raised to activate the signal handler. It is unlikely that the consumers will all finish on their own because there are usually multiple consumers waiting on the counting sem empty_sem when all of the jobs become finished. To fix this, the first consumer to finish raises a SIGINT to call the signal handler to deal with terminating the other threads. This works because if any consumer finishes, it must mean that all of the jobs are finished (this is explained more in the consumer section). Because the program usually ends with a consumer calling the signal handler, to prevent repeated code and to guarantee the program always finishes the same way, the main function calls the signal handler as well.

The Signal Handlers:

This program uses 2 signal handlers
The first signal handler is child_SIGINT_handler which is used in the child process. The only code in this handler is just to tell the child to exit to prevent any zombies. When a producer is created, it is set to use this one

The second and more complicated signal handler is parent_SIGINT_handler which is used by the parent process. This handler basically does the opposite of the initialization that the main method did. First, this handler uses clock_gettime to get the ending time of the program (since we only want the time the program is actually making and consuming jobs rather than the time including initialization and destruction of variables). Next, the shared memory segments are closed using shmdt and shmctl for the semaphore and total_num_jobs shared memory segments and munmap for the printer queue. Next, the counting semaphores are destroyed. Next, the running time of the program is printed. Finally, a SIGTERM signal is raised to terminate any remaining threads.

The FCFS and SJF Producers:
Both versions of the producer function start the same way. The only difference is in the lines for adding jobs to the list.
First, the signal handler for the child function is set to child_SIGINT_handler so that the child will know to exit itself if a SIGINT signal arrives. Next, pointers to the different pieces of data in shared memory are set up. Next, srand() is called to initialize the random number generator. The producer uses its pid as part of the seed for srand() to prevent producers from possibly using the same seed. Next, the number of jobs that this producer will make is decided using rand, resulting in the producer making between 1 and 30 jobs and this number of jobs is added to the shared memory value for the total number of jobs.
Before producing jobs, counting_sem_wait is called on full to make sure the queue has room.

Now, a for loop starts that loops through numJobs times.
Using rand() again, a random delay for this job is calculated (around 0 to 2 seconds). Because sleep() only takes ints as input, usleep() is used.
A random size between 100 and 999 is calculated for this job.

In both the SJF and FCFS code, the logic for the case of there being no jobs in the queue is the same (the ordering is different, but the logic is still the same). The head of the “free” list is obtained, its mutex is locked, it is filled with the appropriate data with j->next pointing to NULL since this is the only element in the queue. The job’s mutex is unlocked, the heads of the “free” and “full” lists are updated, a message describing the job is printed, and the empty counting sem is decremented.

If the queue is not empty, the code is different for the SJF and FCFS. The logic for FCFS appears in case 2 of SJF as described below.
SJF has multiple cases:

1. The job to be added will be the new smallest in the list (it is < the first element)
2. The job to be added will be the new largest in the list (>= all elements)
3. The job to be added will be somewhere in the middle of the list (someElement < this< someElement)

Case 1: An if statement checks if case 1 holds. Case 1 is handled similarly to if the list was previously empty in terms of how mutex semaphores are used. The main difference is instead of j->next being NULL, it points to the previous head of the list.

If this is not case 1, a while loop runs through the list to find where this job should be added.

Case 2: If j->next is NULL, the job will be added to the end of the list. The logic for this case is the same as the logic used by FCFS since in both this case and FCFS, the job is being added to the end of the list. This process uses the same logic as adding to an empty list since it makes the new element's next pointer NULL, the only difference is that we had to scroll through the list first.

Case 3: If j->next is not NULL, the job will be added somewhere in the middle of the list. This code is very similar to the other 2 cases except that it updates 2 elements' next pointers (the new element and the element pointing to the new element).

In both SJF and FCFS, once the producer finishes looping through all of its jobs, it prints a message and exits itself.

The Consumer:
Both the SJF and FCFS code for the consumer is the same. Both consumers dequeue from the front of the list (in SJF this will be the job with the smallest size and in FCFS this will be whichever job is oldest).
The consumer starts by setting up pointers to the shared memory segments.
Two variables, currentID and currentSize are declared and will store the producer id and size of the jobs that it consumes so they can be printed
A sleep(1) call is used before the main while loop of the consumer to allow for the production and consumption of jobs to be slightly more staggered. Without this sleep statement, the consumers frequently consume jobs immediately as they are added to the queue since the consumer runs faster than the producer. With this sleep statement, the output becomes more interesting and demonstrates that the consumer can handle the queue containing jobs before it starts. This sleep statement also ensures that the producers have time to initialize before the consumer. If the consumer finished initializing first, it could see 0 total jobs and instantly close.

The main while loop for the consumer runs while *total_num_jobs_ptr > 0. This variable points to the shared memory segment used to count how many jobs the producers will make. When a producer is created, it adds the number of jobs that it will make to this value and this value is decremented every time a job is consumed. *total_num_jobs_ptr is only 0 before the producers are initialized and after all jobs have been consumed.

Before manipulating the queue, the consumer uses counting_sem_wait on empty to make sure the list isn't currently empty. Next, the heads of the "full" and "empty" job lists are locked if they are not NULL. There are two cases that can occur here: either the free list is NULL or not, depending on if the list of "full" jobs is completely filled.
If the free list is empty, then a new head must be created for it and this head will point to NULL.
If the free list is not empty, then we will update its head to be the newly consumed job and the new head will point to the old head.

In both cases, the head of the “full” list is taken, its values are stored into currentId and currentSize, then its values are set to -1 and it points to the next element of the “empty” job list (either NULL or an element depending on the case).
Finally, the locked head semaphores are unlocked, the list heads are updated, a message is printed, *total_num_jobs_ptr is decremented to represent the consumed job, the full counting sem is decremented, and the consumer sleeps for an amount of time proportional to the size of the consumed job.

This while loop will continue to run until the producers are done producing and the print queue is done. The consumer that consumes this last job will exit the while loop and raise a SIGINT. As described earlier, this is because the other consumers will be stuck on the counting_sem_wait(empty) line and calling the signal handler will close them as well as the rest of the program.

Print_List
This function is not used in the final code and was created for debugging. It was useful for visualizing how the list worked, so it was left in the code.

Part 3: Sample Runs of the Code:
The sample runs of the code are located in two text files submitted with this report: Assignment3SampleRunSJF.txt and Assignment3SampleRunFCFS. These runs involve a lot of text, so they are easier to read in .txt form.

Part 4: Charts Demonstrating Run Time of the Code
This data in spreadsheet form can be found at:
https://docs.google.com/spreadsheets/d/1PD6bPSVMGtZaystiFbOhXQsVSbYY61bn2Vefyl70IyM/edit?usp=sharing

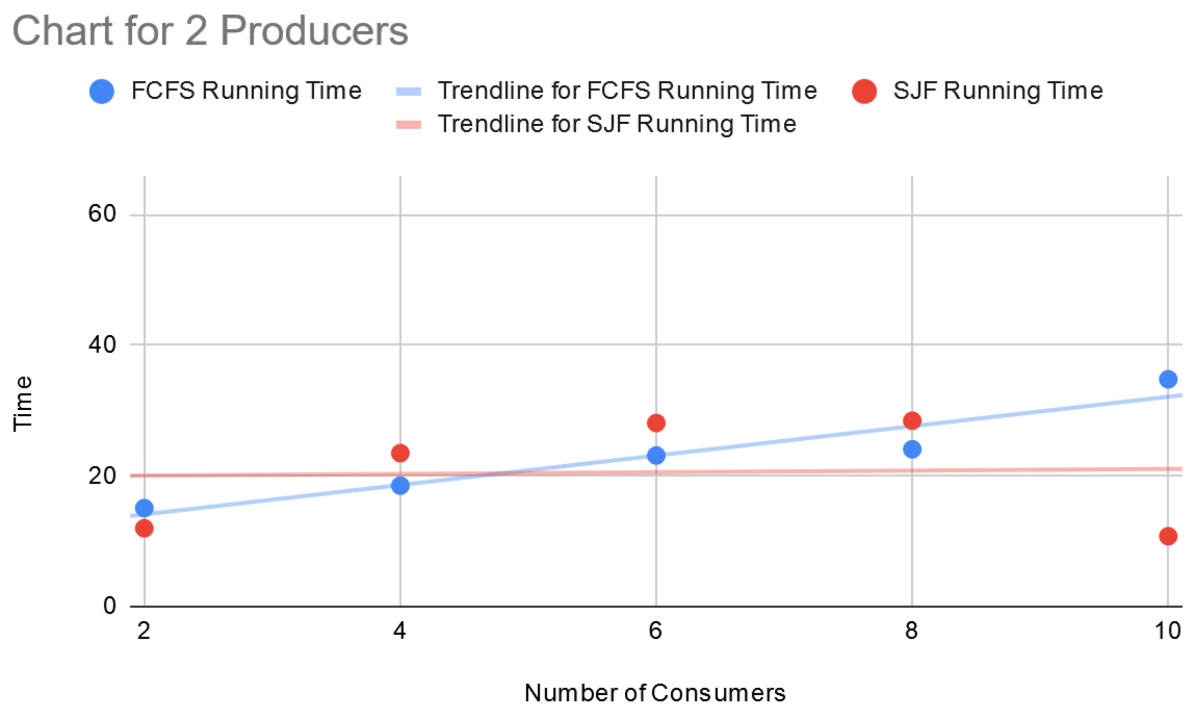


Chart for 2 Producers
FCFS Running Time
Trendline for FCFS Running Time
SJF Running Time
Trendline for SJF Running Time
Time
60
40
20
0
2
4
6
8
10
Number of Consumers

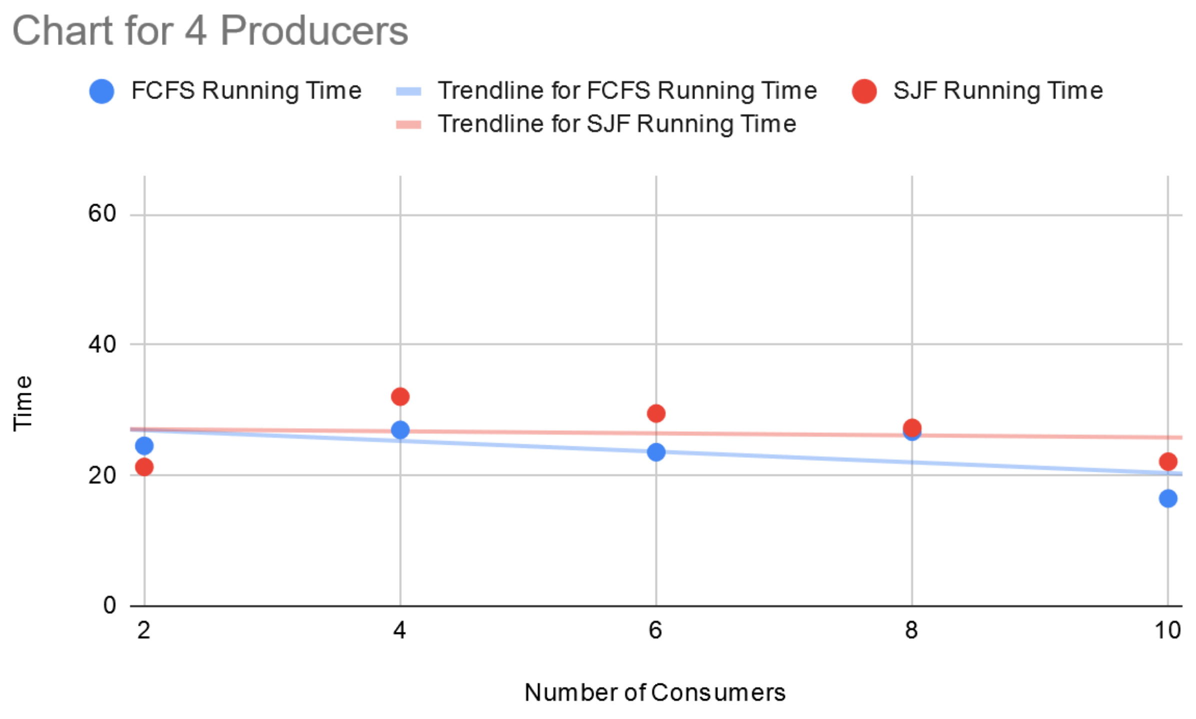


Chart for 4 Producers
FCFS Running Time
Trendline for FCFS Running Time
SJF Running Time
Trendline for SJF Running Time
Time
60
40
20
0
2
4
6
8
10
Number of Consumers

Chart for 6 Producers

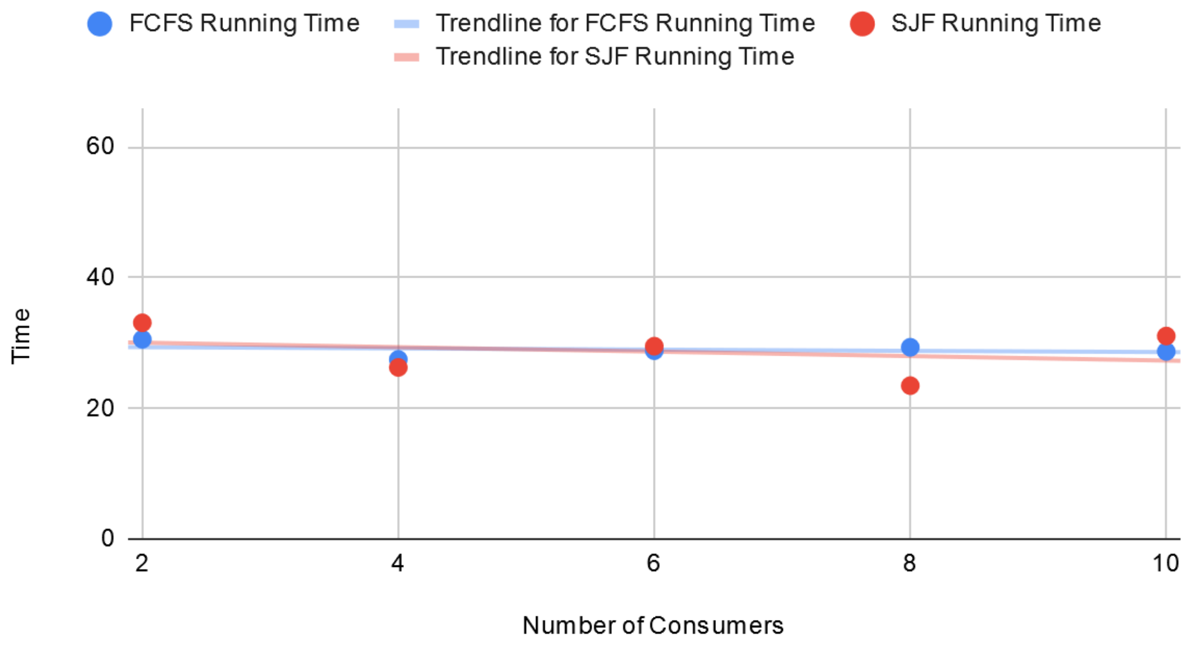
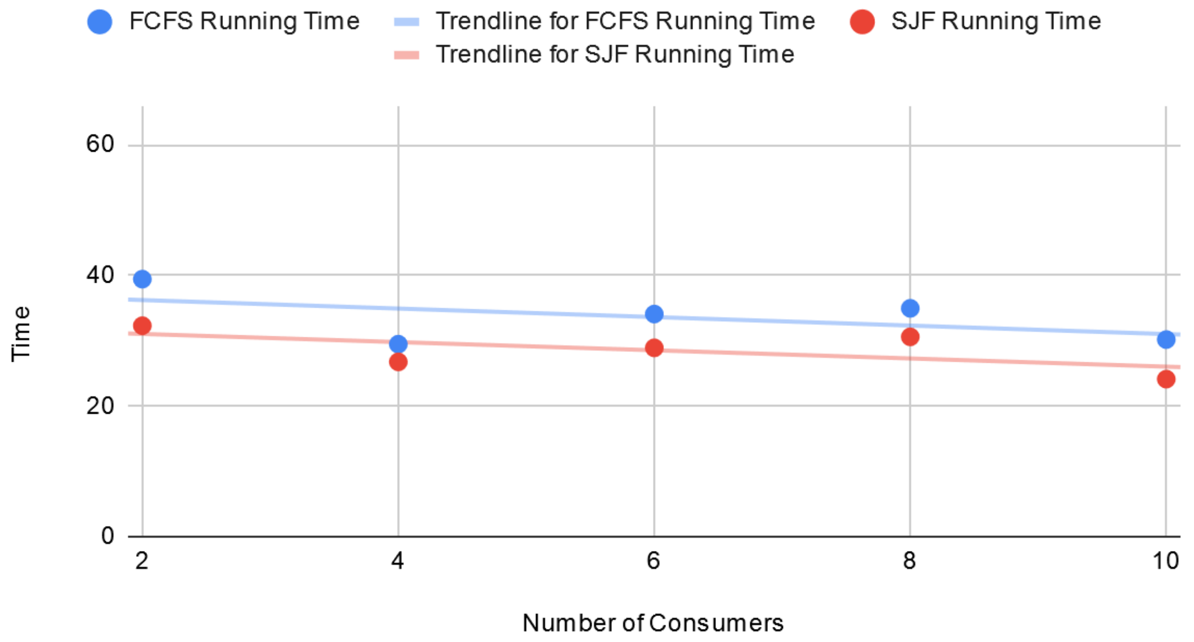
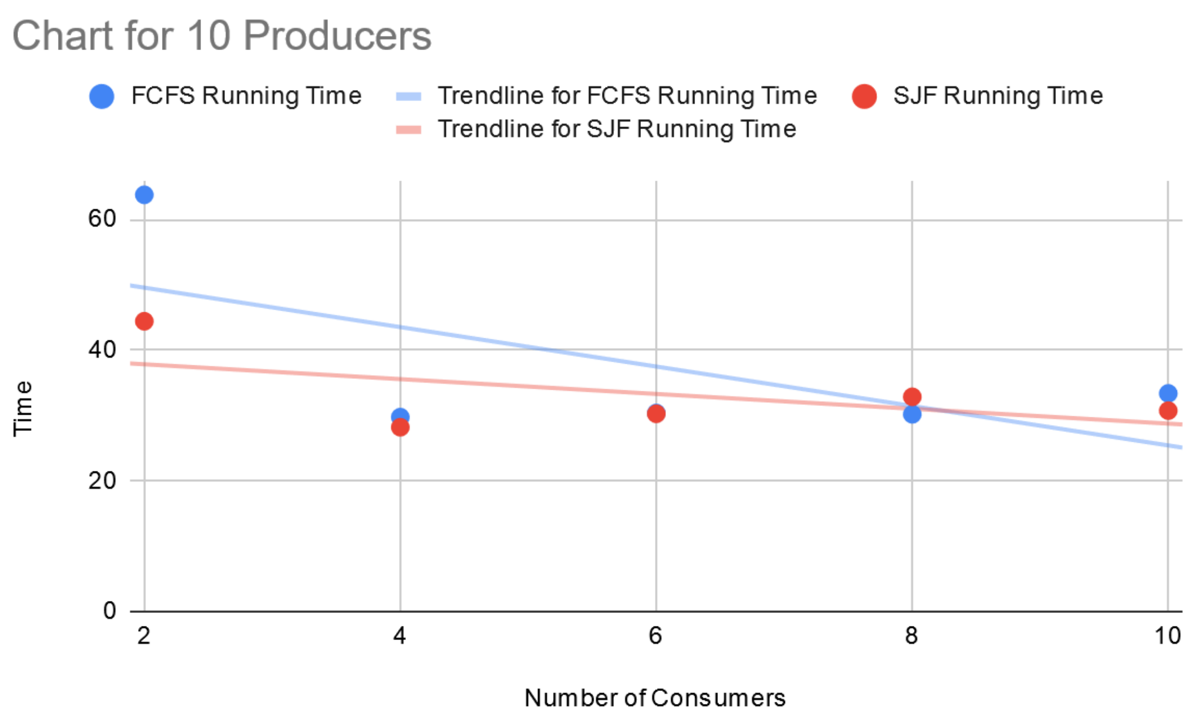


Chart for 8 Producers





Data in table form:

Producers:	Consumers:	FCFS Running Time	SJF Running Time
2	2	15.074432	11.981184
	4	18.5038	23.490758
	6	23.127761	28.095945
	8	24.077931	28.4501
	10	34.797668	10.765296
4	2	24.560974	21.315753
	4	26.993305	32.089228
	6	23.578784	29.510708
	8	26.727437	27.308558
	10	16.489018	22.134461
6	2	30.581438	33.105063
	4	27.493594	26.269885
	6	28.832094	29.523995
	8	29.355988	23.480896
	10	28.71852	31.077648
8	2	39.444089	32.311596
	4	29.477486	26.768098
	6	34.095163	28.917005
	8	34.971865	30.603176
	10	30.18426	24.128726
10	2	63.798656	44.439022
	4	29.766786	28.219863
	6	30.391596	30.27523
	8	30.201596	32.876455
	10	33.392786	30.757093