Exam 2 Comments

It takes a really bad school to ruin a good student
and
a really fantastic school to rescue a bad student.

Dennis J. Frailey

General Comments

- **Write your answers in a technical/formal style.**
- ***Do not prove by example*. Do not use one instance or one execution sequence to show an algorithm satisfying a property. *This is a repeated mistake.***
- **Present all key elements as grading is based on how many key elements are answered properly.**
- **Use instruction level execution sequences please. *Some just didn't get it!***
- **Justify your answer. For example, if you claim there is a race condition, then show it with *TWO* execution sequences.**

Problem 1(a)

```
int   flag[2];
int   turn = 0 or 1;

Process i

repeat
   flag[i] = REQUESTING;
   while (turn!=i && flag[j]!=OUT_CS)
       ;
   flag[i] = IN_CS;
until flag[j] != IN_CS;
turn = i;

// critical section

turn = j;
flag[i] = OUT_CS;
```

P_0 is in CS if

`flag[0] = IN_CS`
`flag[1] != IN_CS`

P_1 is in CS if

`flag[1] = IN_CS`
`flag[0] != IN_CS`

This is impossible!

The value of `turn` does not matter

Problem 1(b)

```
bool  flag[2];   // global flags

Process 0                 Process 1
---------                 ---------
flag[0] = TRUE;           flag[1] = TRUE;
while (flag[1]) {         while (flag[0]) {
  flag[0] = FALSE;          flag[1] = FALSE;
  while (flag[1])           while (flag[0])
      ;                         ;
  flag[0] = TRUE;           flag[1] = TRUE;
}                         }
                 // in critical section
flag[0] = FALSE;          flag[1] = FALSE;
```

If both processes are executing in a fully synchronized way, they will loop forever.

Thus, the finite decision time condition fails.

Problem 1(c): 1/9

- A ***race condition*** is a situation in which **more than one** processes or threads access a **shared** resource **concurrently**, and the result depends on the **order of execution**.
- Use instruction level execution sequences for your examples.
- You must show sharing in your exe. sequences.
- It takes ***two*** execution sequences to justify the existence of a race condition, because ***you need to show the results depend on the order of execution***.

Problem 1(c): 2/9

```
              int  x, a = 4, b = 5;
exe seq 1     Process 1                   Process 2
 x = 5        x = a
                                             x = b
              ------------------------------------------
exe seq 2                                    x = b
 x = 4        x = a
```

This is not a valid example to show the existence of a race condition because variable `x` is not shared concurrently.

Problem 1(c): 3/9

```
int  Count = 10;
```

Higher-level language statements are not atomic

Process 1 **Process 2**

```
Count++; Count--;
```

`Count` = **9, 10 or 11?**

Only say `Count++` and `Count--` would cause a race condition is inaccurate because the "sharing" and "concurrent access" conditions are not addressed.

Problem 1(c): 4/9

```
int  Count = 10;
```

Process 1	**Process 2**
⋮	⋮
`LOAD  Reg, Count`	`LOAD  Reg, Count`
`ADD   #1`	`SUB   #1`
`STORE Reg, Count`	`STORE Reg, Count`
⋮	⋮

The problem is that the execution flow may be switched in the middle. ***Possible answers are 9, 10 or 11.*** **Show two execution sequences.**

Problem 1(c): 5/9

Process 1			Process 2		
Inst	Reg	Memory	Inst	Reg	Memory
LOAD	10	10			
			LOAD	10	10
			SUB	9	10
ADD	11	10			
STORE	11	11			
			STORE	9	9

overwrites the previous value 11

Problem 1(c): 6/9

Process 1			Process 2		
Inst	Reg	Memory	Inst	Reg	Memory
LOAD	10	10			
ADD	11	10			
			LOAD	10	10
			SUB	9	10
			STORE	9	9
STORE	11	11	*overwrites the previous value 9*		

Problem 1(c): 7/9

- ***You should use instruction level interleaving to demonstrate the existence of race conditions*, because**
  - a) **higher-level language statements are not atomic and can be switched in the middle of execution**
  - b) **instruction level interleaving can show clearly the "sharing" of a resource among processes and threads.**

Problem 1(c): 8/9

```
int  a[3] = { 3, 4, 5};
```

Process 1

```
a[1] = a[0] + a[1];
```

Process 2

```
a[2] = a[1] + a[2];
```

Execution Sequence 1

Process 1	Process 2	Array a[ ]
`a[1]=a[0]+a[1]`		{ 3, 7, 5 }
	`a[2]=a[1]+a[2]`	{ 3, 7, 12 }

There is no concurrent sharing, not a valid example for a race condition.

Execution Sequence 2

Process 1	Process 2	Array a[ ]
	`a[2]=a[1]+a[2]`	{ 3, 4, 9 }
`a[1]=a[0]+a[1]`		{ 3, 7, 9 }

Problem 1(c): 9/9

```
int  Count = 10;
```

Process 1

```
LOAD  Reg, Count
ADD   #1
STORE Reg, Count
```

Process 2

```
LOAD  Reg, Count
SUB   #1
STORE Reg, Count
```

Process 1	Process 2	Memory
LOAD Reg, Count		10
	LOAD Reg, Count	10
	SUB #1	10
ADD #1		10
STORE Reg, Count		11
	STORE Reg, Count	9

variable count is shared concurrently here

Problem 2(a): 1/2

- *AABAB* is possible.

A_1	A_2	B_1	A_3	B_2	Sem *X*	Sem *Y*
					2	0
`Wait(X)`					1	0
`Signal(Y)`					1	1
	`Wait(X)`				0	1
	`Signal(Y)`				0	2
		`Wait(Y)`			0	1
		`Wait(Y)`			0	0
		`Signal(X)`			1	0
		`Signal(Y)`			1	1
AAB finishes here			`Wait(X)`		0	1
			`Signal(Y)`		0	2
				`Wait(Y)`	0	1
				`Wait(Y)`	0	0
				`Signal(X)`	1	0
				`Signal(Y)`	1	1

Problem 2(a): 2/2

- *AABBA* is impossible.

A_1	A_2	B_1	B_2	A_3	Sem X	Sem Y
					2	0
`Wait(X)`					1	0
`Signal(Y)`					1	1
	`Wait(X)`				0	1
	`Signal(Y)`				0	2
		`Wait(Y)`			0	1
		`Wait(Y)`			0	0
		`Signal(X)`			1	0
		`Signal(Y)`			1	1
			`Wait(Y)`		0	0
			`Wait(Y)`		0	-1
			`Signal(X)`			
			`Signal(Y)`			

AAB finishes here

can never reach here

B_2 blocks here

Problem 2(b): 1/8

```
int          Waiting = Newcomers = 0;
int          i;
Semaphore    Mutex = 1, Empty = 0;
// Entry
Mutex.Wait();
if (Newcomers == n || Waiting > 0) {
   Waiting++;
   Mutex.Signal();
      Empty.Wait();
      Mutex.Wait();
         Waiting--;
         Newcomers++;
      Mutex.Signal();
}
else {
   Newcomers++;
   Mutex.Signal();
}
// do business
// Exit
Mutex.Wait();
   Newcomers--;
   if (Newcomers == 0) {
      for (i = 1; i <= Waiting; i++)
         Empty.Signal();
   }
Mutex.Signal();
```

pass-the-baton **means the exiting thread that holds the baton (i.e., critical section) does not release the CS. Instead, it hands the baton (i.e., CS) to the next entering thread.**

In this way, the exiting thread does not call `Signal` and the entering thread does not cal `Wait`.

Thus, we should study the `Mutex.Signal()` calls to identify whether pass-the-baton is possible.

Problem 2(b): 2/8

```
int         Waiting = Newcomers = 0;
int         i;
Semaphore   Mutex = 1, Empty = 0;
// Entry
Mutex.Wait();
if (Newcomers == n || Waiting > 0) {
   Waiting++;
   Mutex.Signal();
      Empty.Wait();
      Mutex.Wait();
         Waiting--;
         Newcomers++;
      Mutex.Signal();
}
else {
   Newcomers++;
   Mutex.Signal();
}
// do business
// Exit
Mutex.Wait();
   Newcomers--;
   if (Newcomers == 0) {
      for (i = 1; i <= Waiting; i++)
         Empty.Signal();
   }
Mutex.Signal();
```

Each `Mutex.Signal()` allows a thread waiting on `Mutex.Wait()` to be released.

If a thread has the baton (`Mutex`) and releases a thread from `Empty`, this released thread can have the baton.

a thread executing here has the baton (`Mutex`)

Problem 2(b): 3/8

```
int         Waiting = Newcomers = 0;
int         i;
Semaphore   Mutex = 1, Empty = 0;
// Entry
Mutex.Wait();
if (Newcomers == n || Waiting > 0) {
   Waiting++;
   Mutex.Signal();
      Empty.Wait();
      Mutex.Wait();
         Waiting--;
         Newcomers++;
      Mutex.Signal();
}
else {
   Newcomers++;
   Mutex.Signal();
}
// do business
// Exit
Mutex.Wait();
   Newcomers--;
   if (Newcomers == 0) {
      for (i = 1; i <= Waiting; i++)
         Empty.Signal();
   }
Mutex.Signal();
```

Because the released thread has the baton, the wait on `Mutex` is not needed!

baton is given to the released thread

Problem 2(b): 4/8

```
1  int        Waiting = Newcomers = 0;
2  int        i;
3  Semaphore  Mutex = 1, Empty = 0;
   // Entry
4  Mutex.Wait();
5  if (Newcomers == n || Waiting > 0) {
6     Waiting++;
7     Mutex.Signal();
8        Empty.Wait();
9        Mutex.Wait();
10          Waiting--;
11          Newcomers++;
12       Mutex.Signal();
13 }
14 else {
15    Newcomers++;
16    Mutex.Signal();
17 }
   // do business
   // Exit
18 Mutex.Wait();
19    Newcomers--;
20    if (Newcomers == 0) {
21       for (i = 1; i <= Waiting; i++)
22          Empty.Signal();
23    }
24 Mutex.Signal();
```

If `Newcomers` ≠ 0, no thread waiting on `Empty` is released, and no baton is passed. So, this thread just goes away:

```
18  Mutex.Wait();
19     Newcomers--;
20     if (Newcomers == 0) {
21        for (i = 1; i <= Waiting; i++)
22           Empty.Signal();
23     }
24a    else
24b       Mutex.Signal();
```

Problem 2(b): 5/8

```
4 Mutex.Wait();
 5 if (Newcomers == n || Waiting > 0) {
 6     Waiting++;
 7     Mutex.Signal();
 8        Empty.Wait();
 9        Mutex.Wait();
10           Waiting--;
11           Newcomers++;
12        Mutex.Signal();
13 }
14 else {
15     Newcomers++;
16     Mutex.Signal();
17 }
   // do business
   // Exit
18 Mutex.Wait();
19     Newcomers--;
20     if (Newcomers == 0) {
21        for (i = 1; i <= Waiting; i++)
22           Empty.Signal();
23     }
24a    else
24b       Mutex.Signal();
```

Is this solution correct?

No, this is an incorrect solution.
If `Newcomers` = 0, the exiting thread signals `Waiting` times. Therefore, `Waiting` number of threads get the baton.
Oops! mutual exclusion is violated.

Only one baton can be passed to maintain mutual exclusion!

Problem 2(b): 6/8

```
Mutex.Wait();
 if (Newcomers == n || Waiting > 0) {
    Waiting++;
    Mutex.Signal();
       Empty.Wait();
       Mutex.Wait();
          Waiting--;
          Newcomers++;
       Mutex.Signal();
 }
 else {
    Newcomers++;
    Mutex.Signal();
 }
 // do business
 // Exit
 Mutex.Wait();
    Newcomers--;
    if (Newcomers == 0) {
       for (i = 1; i <= Waiting; i++)
          Empty.Signal();
    }
    else
       Mutex.Signal();
```

Because only one baton can be passed, by removing the `for` loop, we get it done! ***This is my expectation.***

But, is there something wrong? **The original version allows Waiting threads to go; but, this modified version makes the release sequential.** ***This is the problem.***

Problem 2(b): 7/8

```
Mutex.Wait();
 if (Newcomers == n || Waiting > 0) {
    Waiting++;
    Mutex.Signal();
       Empty.Wait();
       Mutex.Wait();
          Waiting--;
          Newcomers++;
       Mutex.Signal();
 }
 else {
    Newcomers++;
    Mutex.Signal();
 }
 // do business
 // Exit
Mutex.Wait();
    Newcomers--;
    if (Newcomers == 0) {
       for (i = 1; i <= Waiting; i++)
          Empty.Signal();
    }
    else
       Mutex.Signal();
```

release next thread here

Return to the entry section.

A thread gets the baton can have access to `Waiting` and `Newcomers`.

Hence, this released can perform the task that was supposed to be done by the releasing thread in the exit section.

Consequently, the released thread should continue releasing the threads blocked on `Empty`.

Problem 2(b): 8/8

```
Mutex.Wait();
 if (Newcomers == n || Waiting > 0) {
     Waiting++;
     Mutex.Signal();
        Empty.Wait();
        Mutex.Wait();
           Waiting--;
           Newcomers++;
           if (Waiting != 0)
              Empty.Signal();
           else
              Mutex.Signal();
 }
 else {
     Newcomers++;
     Mutex.Signal();
 }
     // do business
     // Exit
 Mutex.Wait();
     Newcomers--;
     if (Newcomers == 0) {
        for (i = 1; i <= Waiting; i++)
           Empty.Signal();
     }
     else
        Mutex.Signal();
```

Here, the group releasing using `for` is changed to cascading releases.

Upon exiting, the release thread who has the baton releases the next thread blocked by `Empty`.

If there is no thread blocked by `Waiting`, then release the baton.

Problem 2(c): 1/2

- Two adults requires six signals to `Center` to leave completely. However, three children can only provide three signals, which are not enough to release both adults.

```
// arrive at the center          // arrive at the center
Enter.Signal();                  Enter.Wait();
Enter.Signal();
Enter.Signal();                  // enter and play

// start child care service      Center.Signal(); // done
                                 // leave center
Center.Wait(); // 1st child done playing
Center.Wait(); // 2nd child done playing
Center.Wait(); // 3rd child done playing
// leave center
```

Problem 2(c): 2/2

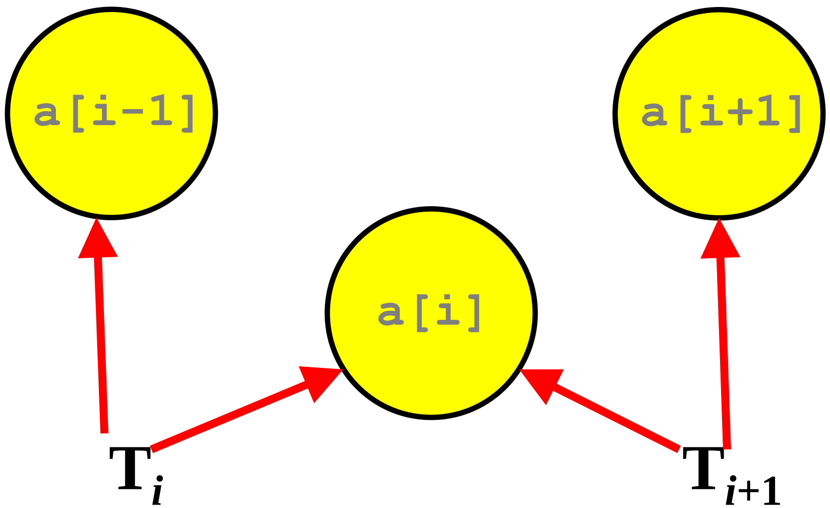
- Three children can only provide 3 signals to `Center`, insufficient to release the 2 adults.

Adult A_1	Adult A_2	Child C_1	Child C_2	Child C_3	Enter	Center
					0	0
S Enter ×3	S Enter ×3				6	0
Wait C	Wait C				6	-2
		Wait Enter			5	-2
		Sig Center			5	-1
Wait C			Wait Enter		5	-2
			Sig Center		5	-1
	Wait C			Wait Enter	4	-1
				Sig Center	4	0
Wait C						

no more signals for this wait

Problem 3(a): 1/3

- Thread T_i accesses data `a[i-1]` and `a[i]`, thread T_{i-1} accesses data `a[i-2]` and `a[i-1]`, and thread T_{i+1} access data `a[i]` and `a[i+1]`.
- This looks like the dining philosophers problem.
- Each `a[i]` requires a semaphore for protection.



Problem 3(a): 2/3

```
Semaphore s[1..n-1] = { 1, 1, …, 1 };
```

Thread$_i$ ($1 \le i \le n$-1)

```
while (not done) {
    s[i-1].Wait();
        in = a[i-1]; // copy a[i-1] to local memory
    s[i-1].Signal(); //   to avoid locking it too long
// use in rather than a[i-1] directly
// it is also possible to have a race condition
//    when testing the value of a[i-1] in an if statement
// it is safer to make a copy than having a race.
// other statements not involving a[i] and a[i-1]
    s[i].Wait();
        a[i] = in;
    s[i].Signal();
}
```

Problem 3(a): 3/3

Some of you did something like this:

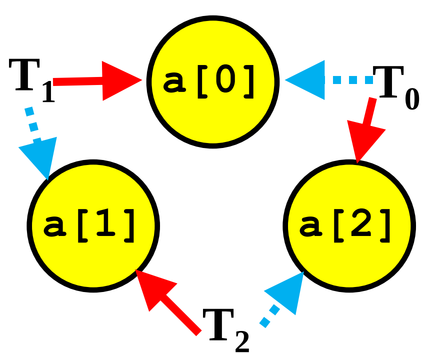
```
Semaphore s[1..n-1] = { 1, 1, …, 1 };
```

Thread$_i$ $(1 \le i \le n\text{-}1)$

left chopstick

right chopstick

```
while (not done) {
    s[i-1].Wait();
        …a[i-1]…;        // lock a[i-1]
        s[i].Wait();     // lock a[i]
          …a[i]…
// other statements not involving a[i] and a[i-1]
        s[i].Signal();  // unlock s[i]
    s[i-1].Signal();    // unlock s[i-1]
}
```



Deadlock can occur.
Did you remember the dining philosophers problem?

Problem 3(b): 1/4

```
Semaphore  Mutex = 1, listProtection = 1; int Count = 0;
```

Searcher

```
while (1) {
   Wait(Mutex);
      Count++;
      if (Count == 1)
         Wait(listProtection);
   Signal(Mutex);
   // do search work
   Wait(Mutex);
      Count--;
      if (Count == 0)
         Signal(listProtection);
   Signal(Mutex);
   // use the data
}
```

Deleter

```
while (1) {
   Wait(listProtection);
   // delete a node
   Signal(listProtection);
   // do something
}
```

this is a writer

this is a reader

Problem 3(b): 2/4

- Thus, searchers and deleters are readers and writers in the readers-writers problem.
- Searchers search concurrently, while deleters must delete exclusively.
- ***Inserters run concurrently with searchers; but, inserters require exclusive access among inserters and deleters.***
- Therefore, inserters are just searchers with exclusive access among all inserters!

Problem 3(b): 3/4

```
Semaphore  Mutex = 1, listProtection = 1; int Count = 0;
Semaphore  insertProtection = 1;

while (1) {
   Wait(insertProtection);
      Wait(Mutex);
         Count++;
         if (Count == 1)
            Wait(listProtection);
      Signal(Mutex);
      // do insertion
      Wait(Mutex);
         Count--;
         if (Count == 0)
            Signal(listProtection);
      Signal(Mutex);
   Signal(insertProtection);
}
```

exclusive use for inserters

Problem 3(b): 4/4

```
Semaphore  Mutex = 1, listProtection = 1; int Count = 0;
Semaphore  insertProtection = 1;

while (1) {
   Wait(Mutex);
      Count++;
      if (Count == 1)
         Wait(listProtection);
   Signal(Mutex);
   Wait(insertProtection);
      // do insertion
   Signal(insertProtection);
   Wait(Mutex);
      Count--;
      if (Count == 0)
         Signal(listProtection);
   Signal(Mutex);
}
```

This one forces inserters to compete with searchers and can cause contention clogging the `Mutex`.

Summary: 1/4

- **Problem 1(a) is very simple. You should get at least *8* points.**
- **Problem 1(b) is a variation of Attempt II. You should get *10* points. So far, you should have *18*pts.**
- **Problem 1(c) is a recycled problem. You should get at least *8* points. So, you should have *26* points.**
- **Problem 2(a) is also easy. I expect you to receive at least *7* pts. Now, you should have *33* points.**
- **Problem 2(b) is a bit tricky, as it requires you to understand the *pass-the-baton* technique fully. I expect you to receive *5* points. So, you should have *38* points so far.**

Summary: 2/4

- **Problem 2(c) is simple enough. Actually, by counting the number of waits and signals, you should have a good guess for how the deadlock can happen. I expect you to get *8* points. Thus, you should have about *46* points.**
- **Problem 3(a) is a variation of the Philosophers problem with `a[i]` and `a[i+1]` being chopsticks *i* and *i*+1. I expected you to get *10* pts. Now, you should have *56* pts.**
- **If you read carefully, Problem 3(b) is variation of the readers-writers problem with a little more complexity. Getting about *10* points would be reasonable. Thus, you should have at least *66* points.**

Summary: 3/4

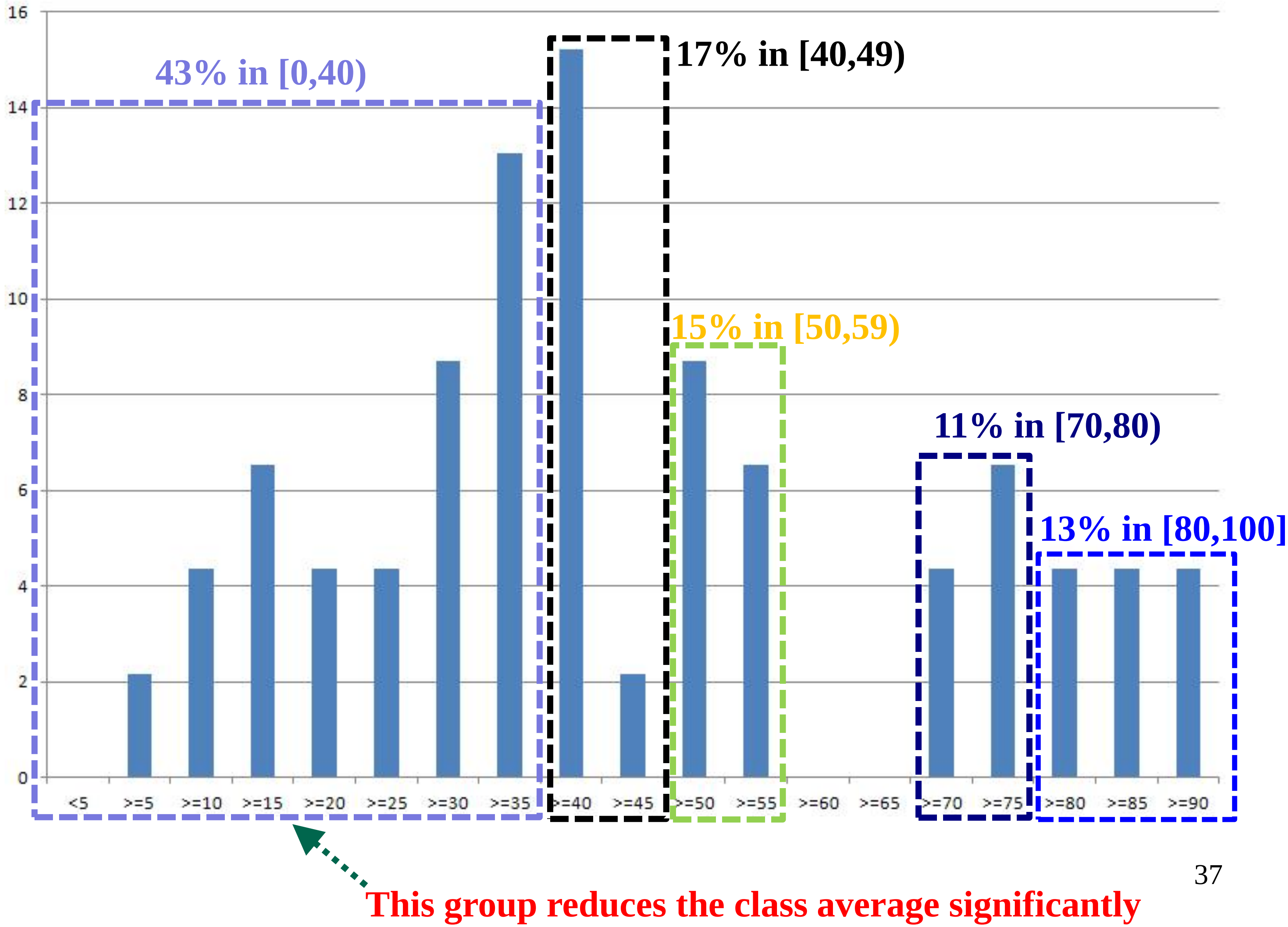
- ***I expected you to receive approximately 60 points as shown below.***

Problem		Possible	Expected	Average	Median
1	A	15	8	9	11
	B	15	10	7	5
	C	10	8	8	8
2	A	10	7	6	7
	B	10	5	3	0
	C	10	8	5	5
3	A	15	10	4	0
	B	15	10	6	5
	Total	100	66	46	42

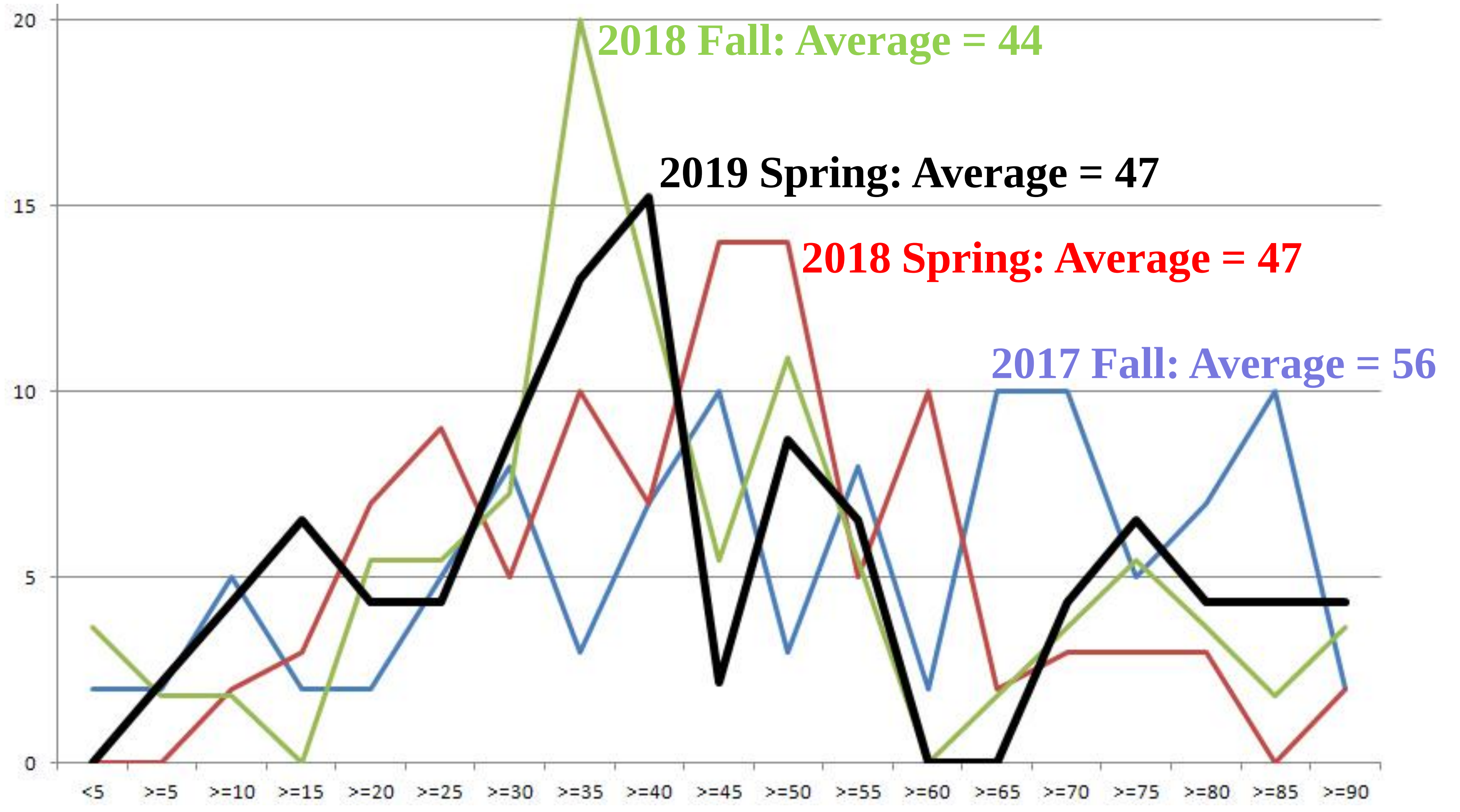
Summary: 4/4

	1a	1b	1c	2a	2b	2c	3a	3b	Class
Min	0	0	0	0	0	0	0	0	9
Max	15	15	10	10	7	10	15	15	97
Median	11	5	8	7	0	5	0	5	42
Avg	9	7	8	6	3	5	4	6	46
St DEV	6	7	3	4	3	4	5	6	24

Grade Distribution

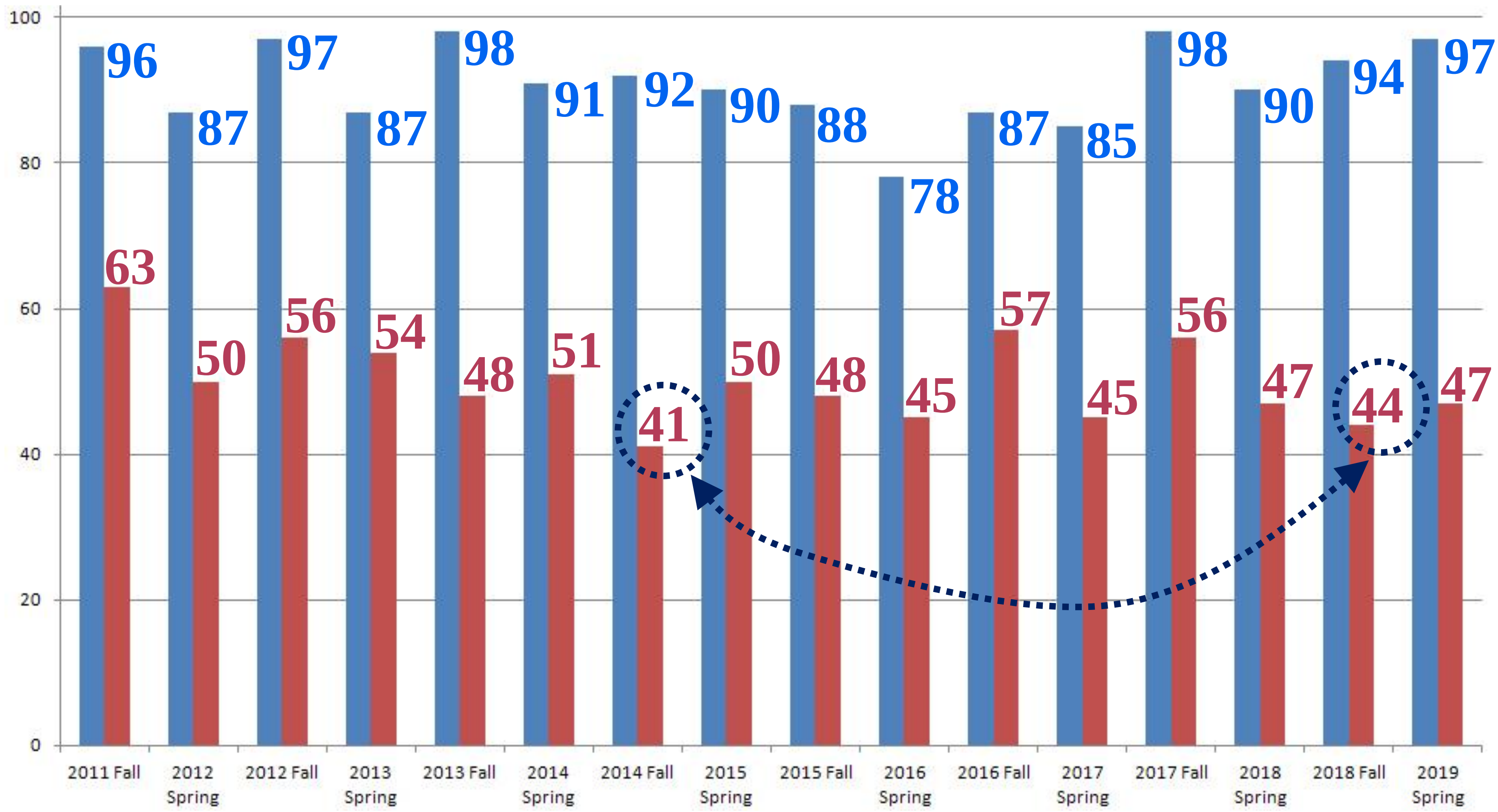


Grade Distribution Compared with Last Few Semesters



Comparison Over the Years

- The following is a comparison of Exam II class average performance recorded over the years.



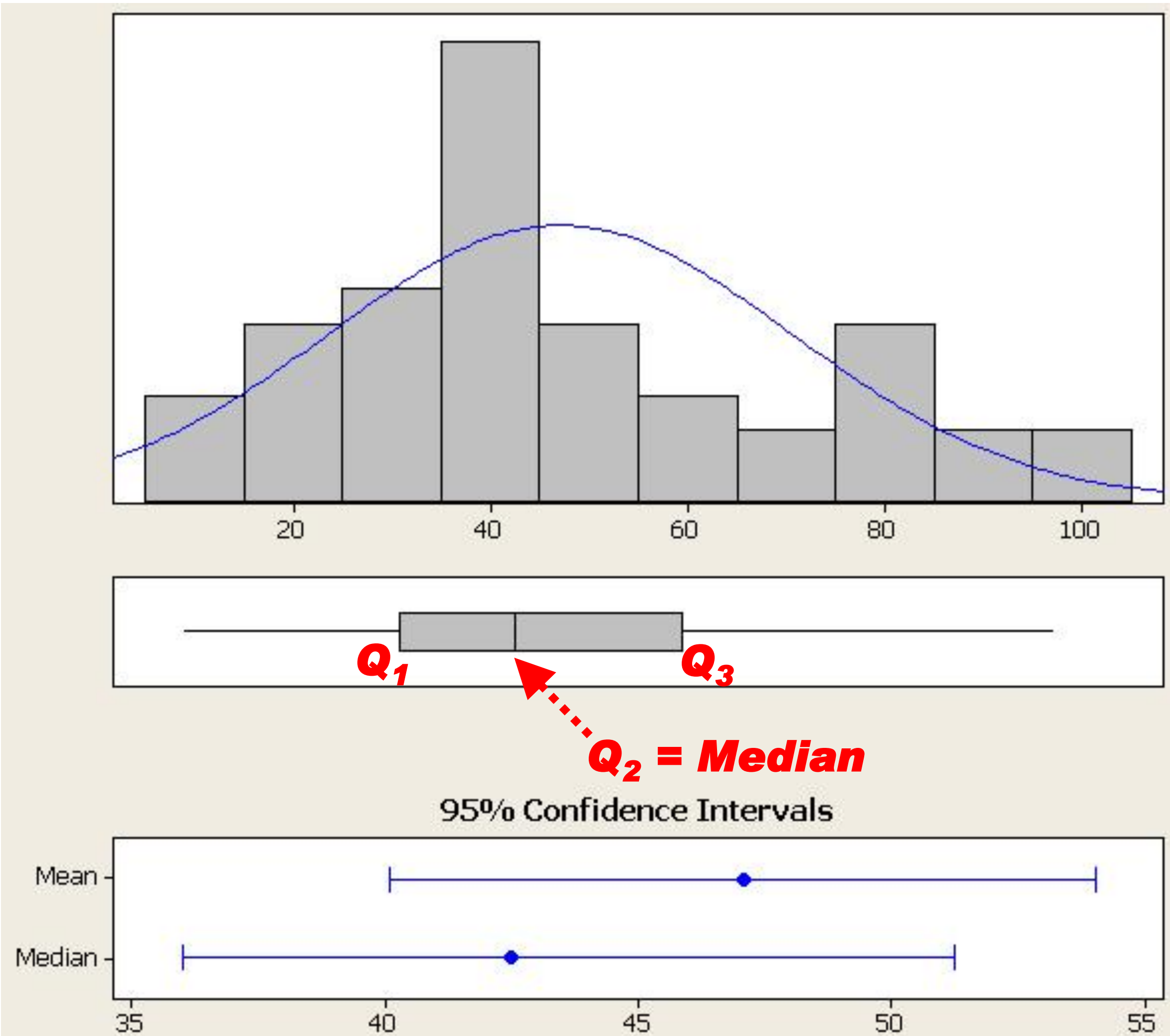
Grade Distributions: 2011-2019

Worst Class and
Worst Evaluation Ever

Grade	11F	12S	12F	13S	13F	14S	14F	15S	15F	16S	16F	17S	17F	18S	18F
A	26	23	12	25	12	19	13	3	16	10	20	14	20	10	25
AB	20	9	4	33	19	21	15	3	6	10	22	7	16	24	4.2
B	11	20	16	17	14	12	9	5	22	12	5	10	14	16	10.4
BC	6	14	16	8	2	21	15	30	12	17	15	12	11	16	10.4
C	9	6	4	6	7	5	9	14	14	8	5	12	9	9	20.8
CD	3	9	8	3	7	5	9	11	6	12	11	12	9	7	2.1
D	6	6	24	0	24	7	4	14	10	21	2	19	5	14	18.8
F	20	14	16	8	14	10	28	22	12	12	20	14	16	5	8.3
Size	35	35	25	36	42	42	47	37	49	52	55	42	56	58	48

Data shown here only include students who completed this class.
Students who did not take the final exam were not included.

More Information



Anderson-Darling Normality Test

A-Squared	0.91
P-Value	0.019
Mean	47.065
StDev	23.501
Variance	552.285
Skewness	0.534085
Kurtosis	-0.590268
N	46
Minimum	9.000
1st Quartile	30.750
Median	42.500
3rd Quartile	59.500
Maximum	97.000

95% Confidence Interval for Mean

40.086	54.044

95% Confidence Interval for Median

36.000	51.248

95% Confidence Interval for StDev

19.492	29.600

Parting Thoughts

- Study our course material effectively.
- Problems usually require the most basic understanding. Their answers are usually short. If you could not finish all problems, make sure you will understand the subjects thoroughly and develop an efficient way to do your work.
- Exams test your level of understanding rather than how much time you spent on preparation.
- Most students improve in the final. Whether you will improve depends on your understanding.
- Make sure your performance is higher than Q_1.

The End