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.
- Avoid the use of imprecise and non-professional wording and language as computer science is an exact science and we must learn to communicate in a professional way.
- Present all key elements as grading is based on how many key elements are answered properly.
- Justify your answer. For example, if you claim bounded violation is violated, then show it with an execution sequence. **Don't make a vague claim without a good justification.**
- **I do not do grade inflation.**

Problem 1(a): 1/2

```
int  flag[2];  // initialized to OUT_CS
int  turn;     // initialized to 0 or 1
```

Process *i* (P_i), *i* = 0 or 1

```
// Enter Protocol
repeat
   flag[i] = REQUEST;
   while (turn != i && flag[j] != OUT_CS)
      ;
   flag[i] = IN_CS;
until flag[j] != IN_CS;
turn = i;
```

Critical Section

```
// Exit Protocol
turn = j;
flag[i] = OUT_CS;
```

- Process P_i exits its `repeat-until` loop sees `flag[j]` being not `IN_CS`, and right before that P_i sets `flag[i]` to `IN_CS`.
- By the same reason, if P_j is in its critical section, `flag[j]` is `IN_CS` and `flag[i]` is not `IN_CS`.
- If P_i and P_j were both in the critical section, `flag[i]` would be `IN_CS` and not `IN_CS` at the same time.

Problem 1(a): 2/2

```
int  flag[2];  // initialized to OUT_CS
int  turn;     // initialized to 0 or 1
```

Process *i* (P_i), *i* = 0 or 1

```
// Enter Protocol
repeat
   flag[i] = REQUEST;
   while (turn != i && flag[j] != OUT_CS)
      ;
   flag[i] = IN_CS;
until flag[j] != IN_CS;
turn = i;
```

Critical Section

```
// Exit Protocol
turn = j;
flag[i] = OUT_CS;
```

- **Variable `turn` is not used in the reasoning.**
- **If P_i and P_j are both in the critical section, they execute the statement `turn = i`.**
- **If P_i executes this statement first followed by P_j, the value of `turn` is `j`.**
- **If P_j executes this statement first followed by P_i, the value of `turn` is `i`.**
- **Hence, `turn` will not have two values at the same time.**

Problem 1(b): 1/2

```
int status[2];
int turn;      // initialized to 0 or 1
```

Process 0

```
status[0] = COMPETING;
do {
  while (turn != 0) {
    status[0] = OUT_CS;
    if (status[turn] == OUT_CS)
      turn = 0;
  }
  status[0] = IN_CS;
} while (status[1] == IN_CS);
```

Critical Section

```
status[0] = OUT_CS;
```

Process 1

```
status[1] = COMPETING;
do {
  while (turn != 1) {
    status[1] = OUT_CS;
    if (status[turn] == OUT_)CS)
      turn = 1;
  }
  status[1] = IN_CS;
} while (status[0] == IN_CS);

status[1] = OUT_CS;
```

- **Variable `turn` is set only once in the `if` statement and is not reset when exits the critical section.**
- **Suppose P_0 sets `turn` to 0 and enters the critical section. Because P_0 does not reset `turn`, P_0 may come back and re-enter the critical section.**
- **This may repeat again and again, and bounded waiting fails.**

Problem 1(b): 2/2

	P_0	P_1	turn	status[0]	status[1]	Comment
1			0			
2	s[0]=C	s[1]=C	0	C	C	Entering
3	while	while	0	C	C	P_0 breaks while
4	s[0]=IN		0	IN	C	P_0 about to enter
5		s[1]=OUT	0	IN	OUT	P_0 about to enter
6	P_0 enters its critical section					
7	s[0]=OUT	if	0	OUT	OUT	P_0 enters CS as turn = 0
8	P_0 comes back					
9	s[0]=C	while loops back	0	C	OUT	P_0 entering
10		if	0	C	OUT	if is false
11	while	while	0	C	OUT	P_1 loops back
12	s[0]=IN		0	IN	OUT	P_0 about to enter
13	P_0 enters its critical section					

```
status[0] = COMPETING;
do {
  while (turn != 0) {
    status[0] = OUT_CS;
    if (status[turn] == OUT_CS)
      turn = 0;
  }
  status[0] = IN_CS;
} while (status[1] == IN_CS);
```

```
status[1] = COMPETING;
do {
  while (turn != 1) {
    status[1] = OUT_CS;
    if (status[turn] == OUT_)CS)
      turn = 1;
  }
  status[1] = IN_CS;
} while (status[0] == IN_CS);
```

Problem 1(c): 1/10

- A **race condition** is a situation in which **more than one** process or thread 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 concurrent sharing in your execution 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): 2/10

```
              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): 3/10

```
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): 4/10

```
int  Count = 10;
```

Process 1

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

Process 2

```
   ⋮
LOAD  Reg, Count
SUB   #1
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): 5/10

First Execution Sequence

Process 1 Inst	Process 1 Reg	Process 1 Memory	Process 2 Inst	Process 2 Reg	Process 2 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): 6/10

Second Execution Sequence

Process 1 Inst	Process 1 Reg	Process 1 Memory	Process 2 Inst	Process 2 Reg	Process 2 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): 7/10

- **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): 8/10

```
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 <u>concurrent sharing</u>, 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): 9/10

```
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 1(c): 10/10

- The following execution sequence is not acceptable, because `count++` and `count--` are higher level language statements mixed with machine instructions. These statements apply to memory and have immediate impact.

```
int count = 0;

Process 1               Process 2
LOAD count              LOAD count
count++                 count--
SAVE count              SAVE count
```

Problem 2(a): 1/2

- The following is an obvious solution.

```
Semaphore  S1 = 1, S2 = 0, S3 = 0;
```

Thread 1

```
while (1) {

   S1.Wait();
      cout << "1";
   S2.Signal();

}
```

Thread 2

```
while (1) {
// do something
   S2.Wait();
      cout << "2";
   S3.Signal();
   S2.Wait();
      cout << "2";
   S1.Signal();
// do something
}
```

Thread 3

```
while (1) {

   S3.Wait();
      cout << "3";
   S2.Signal();

}
```

Problem 2(a): 2/2

- If you insist that ***Thread 2*** can only have one statement to print 2, here is another solution.
- After printing 2 the first time, the printing process goes “*forward*” to Thread 3. Then, the next time, the printing process goes “*backward*”.

```
semaphore  S1 = 1, S2 = 0, S3 = 0;
```

Thread 1

```
while (1) {
   S1.Wait();
      cout << “1”;
   S2.Signal();
}
```

Thread 2

```
int Forward = TRUE;
while (1) {
   S2.Wait();
      cout << “2”;
      if (Forward)
         S3.Signal();
      else
         S1.Signal();
      Forward = !Forward;
}
```

Thread 3

```
while (1) {
   S3.Wait();
      cout << “3”;
   S2.Signal();
}
```

Problem 2(b)

- If `Wait()` is not atomic, multiple threads can call `Wait()` and increase the `counter` at the same time. Race condition can happen.

Mutual Exclusion

```
semaphore  S=1;

S.Wait();
   Critical Section
S.Signal();
```

	P_0	P_1	count	Comment
1			1	=1 for M.Ex
2	`S.Wait()`	`S.Wait()`	1	Both call
3	`LA count`	`LA count`	`Reg=1, 1`	Non-Atomic
4	`SUB #1`	`SUB #1`	`Reg=0, 1`	Register is 0
5	`SA count`	`SA count`	0	`count` is 0
6	Both processes enter their critical sections			

Problem 2(c): 1/2

- We assume the weirdo (philosopher 5) always picks his right chopstick first followed by his left one, and all normal ones pick their left first.

```
if the weirdo has his right chopstick then
    if the weirdo has his left chopstick then
        the weirdo eats and there is no deadlock.
    else // weirdo's left is taken by philosopher 4 as his right
        philosopher 4 eats.  no deadlock.
else // the weirdo does not have his right because philosopher 1 has it as his left
    //      weirdo's left is available
    if philosopher 1 has his right then
        philosopher 1 eats and there is no deadlock
    else // philosopher 1's right is taken by philosopher 2 as his left
        if philosopher 2 has his right then
            philosopher 2 eats and there is no deadlock
        else // philosopher 2's right is taken by philosopher 3 as his left
            if  philosopher 3 has his right then
                philosopher 3 eats and there is no deadlock
            else // philosopher 3's right is taken by philosopher 4 as his left
                philosopher 4 eats as he can use weirdo's left as his right
```

Problem 2(c): 2/2

weirdo has his right

philosopher 1 cannot eat

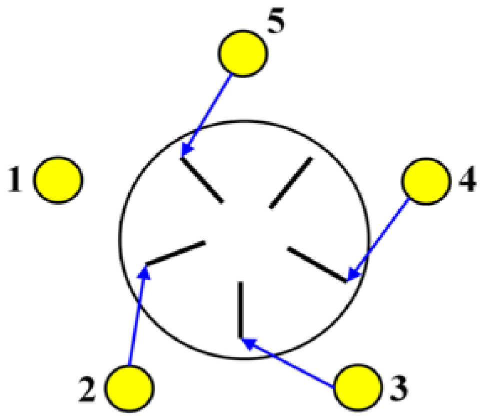


Figure (a)

if weirdo also has his left, weirdo eats

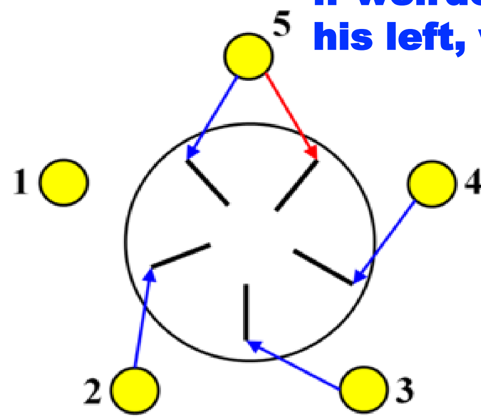


Figure (b)

if weirdo does not have his left, it is taken by philo 4 as his right and philo 4 eats

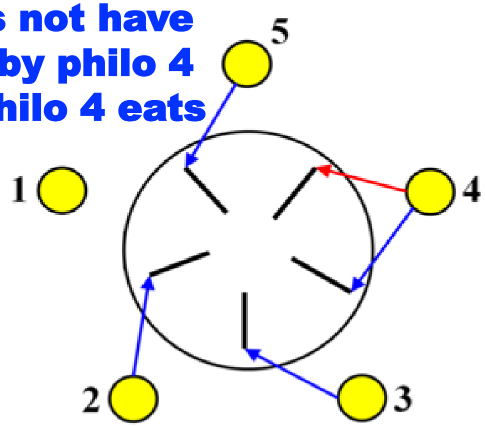


Figure (c)

if weirdo does not have his right, this chop is taken by philo 1, ... Philo 4 eats

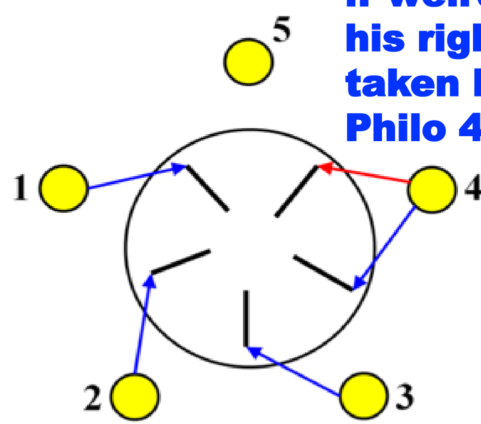


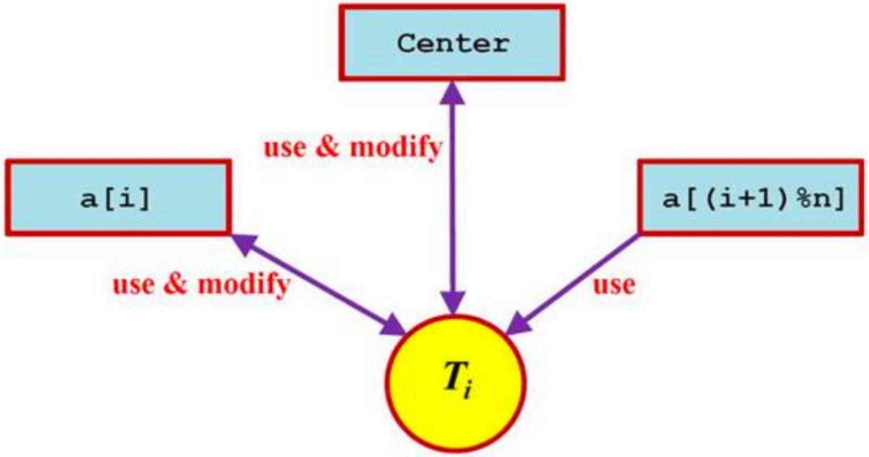
Figure (d)

Problem 3(a): 1/5

- The following is the basic code:

```
while (1) {
    a[i] = f(a[i], a[(i+1)%n]);
    Center = a[i] + Center;
}
```

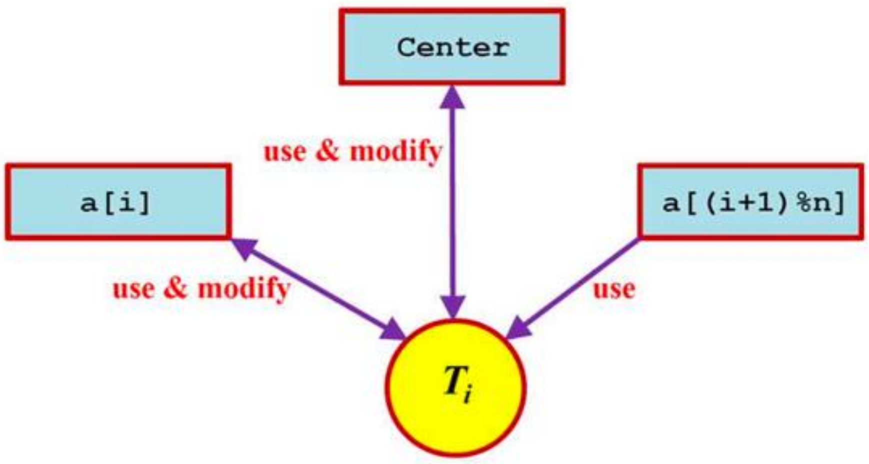
- Thus, thread T_i uses `a[(i+1)%n]` and modifies `a[i]` and `Center`.
- This is similar to the dining philosophers problem.



Problem 3(a): 2/5

- We need a semaphore for each `a[i]`. T_i needs two semaphores for `a[i]` and `a[(i+1)%n]` to access `a[i]` and `a[(i+1)%n]`.
- Because `Center` is accessed by all threads, we also need a semaphore to protect `Center`.

```
semaphore S_Center = 1;
semaphore S_a[n] = { 1, 1, ….., 1};
```



Problem 3(a): 3/5

```
semaphore S_Center = 1;
semaphore S_a[n] = { 1, 1, …., 1};

while (1) {
    S_a[(i+1)%n].Wait();
        Local = a[(i+1)%n];
    S_a[(i+1)%n].Signal();
    fx = f(a[i], Local);

    S_a[i].Wait();
        a[i] = fx;
    S_a[i].Signal();

    S_Center.Wait();
        Center = fx + Center;
    S_Center.Signal();
}
```

copy a[(i+1)%n] **to** Local

Because f() **does not modify** a[i] **and** Local**, no lock needed.**

update a[i]

update Center

Problem 3(a): 4/5

```
semaphore S_Center = 1;
semaphore S_a[n] = { 1, 1, ….., 1};

while (1) {
     // other irrelevant computation
     S_a[(i+1)%n]).Wait();
       S_a[i].Wait();
         S_Center.Wait();
           a[i] = f(a[i], a[(i+1)%n]);
           Center = a[i] + Center;
         S_Center.Signal();
       S_a[i].Signal
     S_a[(i+1)%n].Signal
     // other irrelevant computation
}
```

This implementation serializes all threads, no concurrency at all. Only one thread can modify `a[]` (not OK) and `Center` (OK).

Problem 3(a): 5/5

```
semaphore S;

while (1) {
    // other irrelevant computation
    S.Wait();
      a[i] = f(a[i], a[(i+1)%n]);
      Center = a[i] + Center;
    S.Signal();
    // other irrelevant computation
}
```

This solution is even worse because there is no concurrency.

Problem 3(b): 1/2

- **All men can use the bathroom as long as there is a man using it. Aren't the man threads readers in the readers-writers problem?**
- **By the same reason, all women can use the bathroom as long as there is a woman using it. Therefore, all woman threads form another "reader" threads in the readers-writers problem.**
- **In conclusion, we have two groups of readers, and while one group of readers is using the bathroom the other group is blocked.**
- **What we need? Duplicate the reader thread, one for men and the other for women.**

Problem 3(b): 2/2

```
int       MaleCounter = 0, FemaleCounter = 0;
Semaphore MaleMutex = 1, FemaleMutex = 1;
Semaphore BathRoom = 1;
```

```
while (1) {
   // working

   MaleMutex.Wait();
      MaleCounter++;
      if (MaleCounter == 1)
         BathRoom.Wait();
   MaleMutex.Signal();

   // use the bathroom

   MaleMutex.Wait();
      MaleCounter--;
      if (MaleCounter == 0)
         BathRoom.Signal();
}
```

```
while(1) {
   // working

   FemaleMutex().Wait();
      FemaleCounter++;
      if (FemaleCounter == 1)
         BathRoom.Wait();
   FemaleMutex.Signal();

   // use the bathroom

   FemaleMutex.Wait();
      FemaleCounter--;
      if (FemaleCounter == 0)
         BathRoom.Signal();
}
```

if I am the first man/woman, yield the bathroom

if I am the last man/woman, yield the bathroom

Class Performance

- **I expected you to receive approximately 70 points as shown below.**

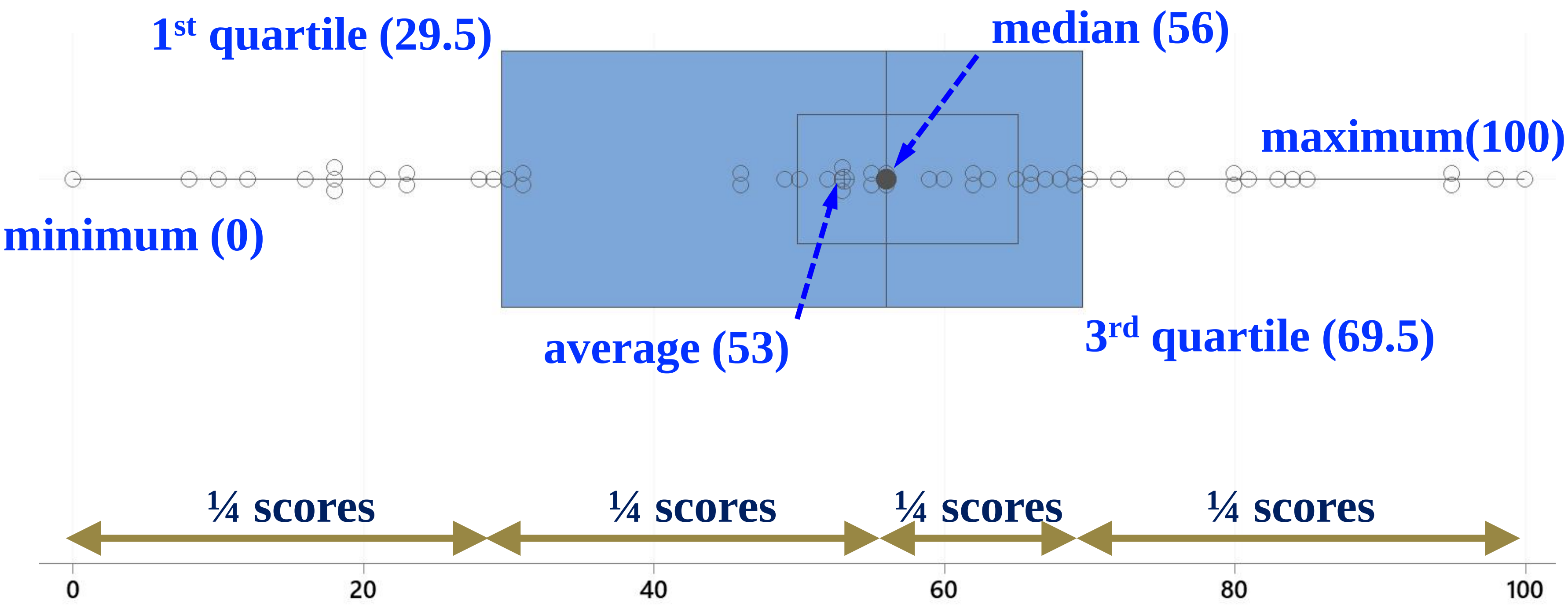
Problem		Possible	Expected	Class Average	Class Median
1	a	15	10	10	13
	b	15	7	5	0
	c*	10	8	7	8
2	a	10	8	7	10
	b	10	8	7	8
	c	10	8	5	5
3	a	15	10	5	0
	b	15	10	8	11
	Total	100	69	53	56

Grade Distribution Problem-Wise

	1a	1b	1c	2a	2b	2c	3a	3b	Class
Min	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Max	**15**	**15**	**10**	**10**	**10**	**10**	**15**	**15**	**100**
Median	**13**	**0**	**8**	**10**	**8**	**5**	**0**	**11**	**56**
Avg	**10**	**5**	**7**	**7**	**7**	**5**	**5**	**8**	**53**
St DEV	**6**	**7**	**3**	**4**	**4**	**3**	**6**	**7**	**26**

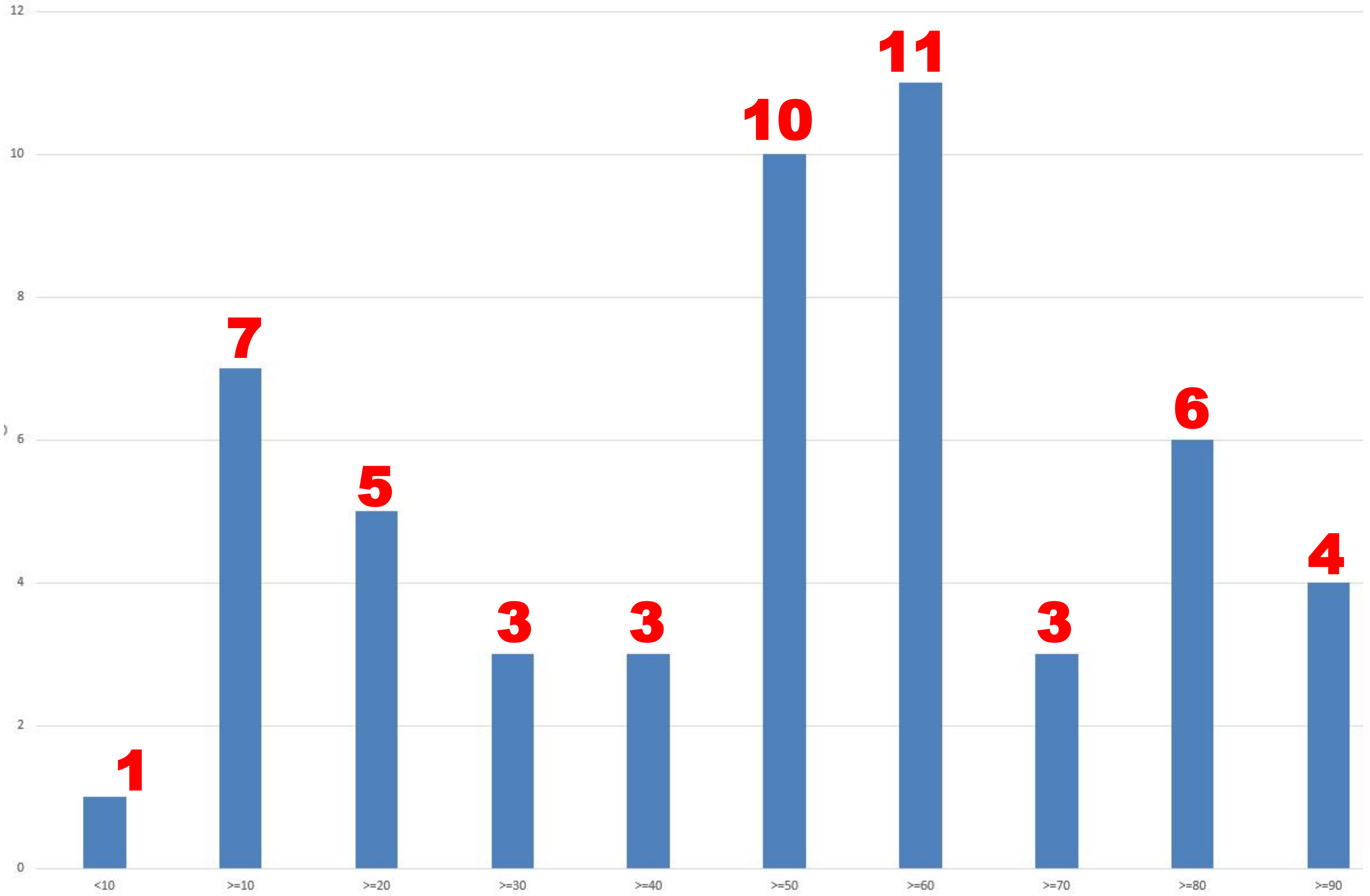
- **Problem 1a is a problem similar to Attempt II**
- **Problem 1b is a little more difficult, but you have a hint**
- **Problem 1c is a "recycled" problem from EXAM I**
- **Problem 2a, 2b and 2c were exercises assigned in class**
- **Problem 3a is similar to the philosophers problem and 3b is a variation of the readers-writers problem.**

Boxplot



1st quartile (29.5)
median (56)
maximum(100)
minimum (0)
average (53)
3rd quartile (69.5)
¼ scores
¼ scores
¼ scores
¼ scores
0
20
40
60
80
100

Grade Distribution



My Findings

- **Many of you did not study the slides carefully. Even the easiest problems were answered poorly/incorrectly.**
- **Some just provide an answer or value without elaboration. I am not supposed to finish your answer for you. Whenever a justification and/or elaboration is needed, please do it. Use correct wording.**
- **Please study harder, ask questions, and make sure you understand the subjects.**
- **Your grade is proportional to the quality of your answers and is not proportional to the time you spent!**
- **In my experience the Final is usually easier because difficult topics are spread thin.**
- **Again, I do not do grade inflation.**

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

Dennis J. Frailey

The End