Exam I 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 there is a race condition, then show it with execution sequences.
- **I do not do grade inflation.**

Problem 1(a)i

- Your output should look like the table shown on the right side of this slide if you ran your program on an Intel-based CPU.
- From 0! To 20!, the results are correct.
- 21!, 22! and 24! become negative and 25! is less than 23!

0!	=	1
1!	=	1
2!	=	2
3!	=	6
4!	=	24
5!	=	120
6!	=	720
7!	=	5040
8!	=	40320
9!	=	362880
10!	=	3628800
11!	=	39916800
12!	=	479001600
13!	=	6227020800
14!	=	87178291200
15!	=	1307674368000
16!	=	20922789888000
17!	=	355687428096000
18!	=	6402373705728000
19!	=	121645100408832000
20!	=	2432902008176640000
21!	=	-4249290049419214848
22!	=	-1250660718674968576
23!	=	8128291617894825984
24!	=	-7835185981329244160
25!	=	7034535277573963776

Problem 1(a)ii: 1/3

- The minimum and maximum of `long` are system dependent. See `limits.h` for the details.
- On my MacBook Air and iMac under `gcc`, the minimum and maximum values of the `long int` type, which are the same as the `long long int` type, are `-9223372036854775808` and `9223372036854775807`.
- In general, if a signed integer is represented by k+1 bits with 1 sign bit, then the minimum and maximum are likely to be -2^k-1 and 2^k-1. If the computed result is larger than 2^k-1, only the last k bits would be stored.

Problem 1(a)ii: 2/3

- Suppose a 4-bit register is used for multiplication.
- Multiplying $0110_2 = 6_{10}$ and $0101_2 = 5_{10}$ yields $30_{10} = 11110_2$.
- Because we only use 4-bit registers, the stored result would be the last 4 bits 1110_2.
- Because we use `signed` integers, the first bit is the sign bit (*i.e.*, 0 – positive and 1 – negative), and 1110_2 actually means -2_{10} under the commonly seen **2's complement** system. Note that different computer architectures would produce different results.

Problem 1(a)ii: 3/3

- **4-bit Representation**

2's complement:

1110_2 (negative)

↓

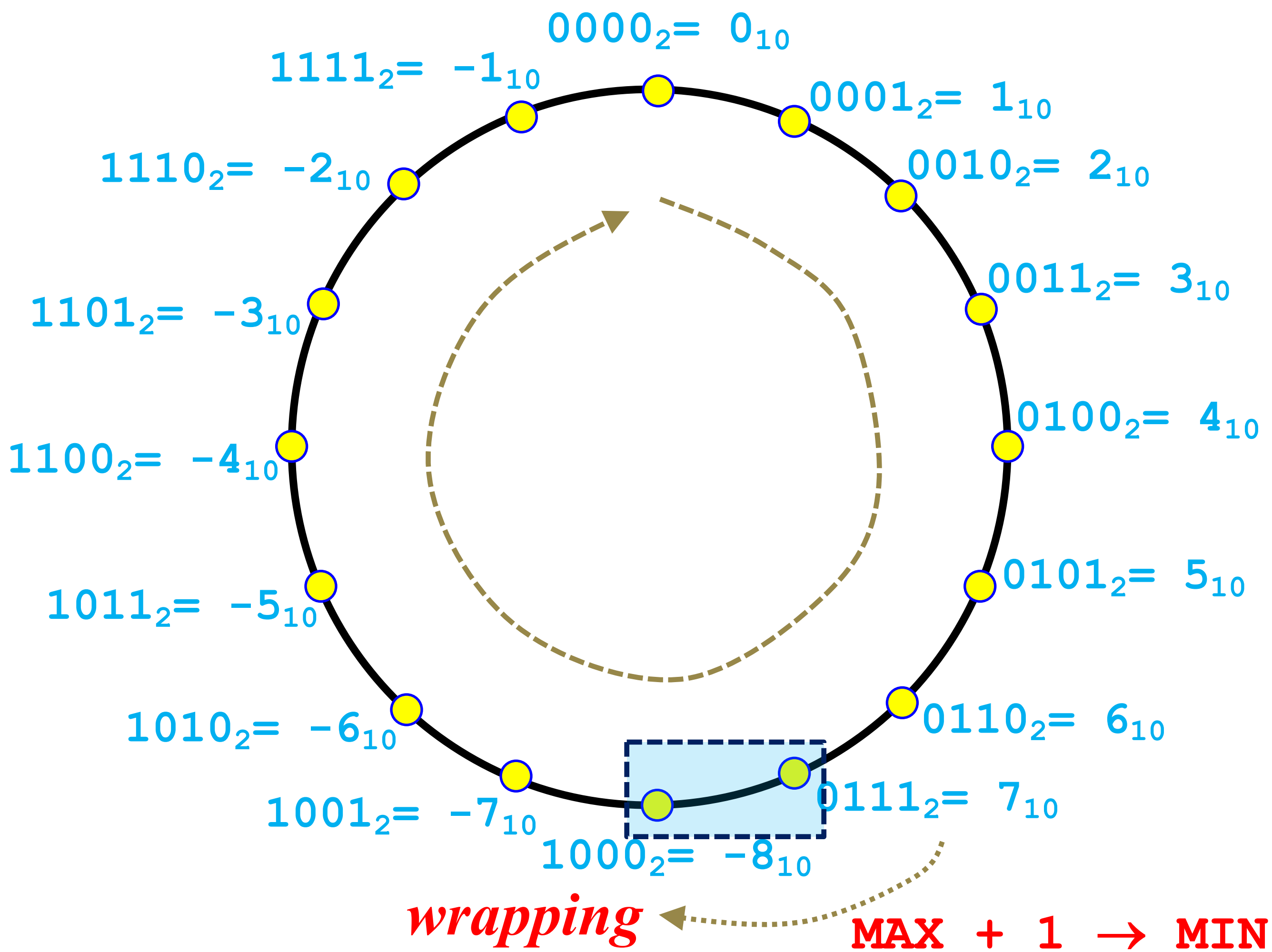
0001_2 (bit flipping)

↓

0010_2 (adding 1)

↓

-2_{10}



Problem 2(a)

- Modern CPUs have two execution modes: the user mode and the supervisor (or system, kernel, privileged) mode, controlled by a mode bit.
- The OS runs in the supervisor mode and all user programs run in the user mode. Some instructions that may do harm to the OS (e.g., I/O and CPU mode change) are privileged instructions, which, for most cases, can only be used in the supervisor mode.
- When execution switches to the OS (resp., a user program), execution mode must be changed to the supervisor (resp., user) mode.

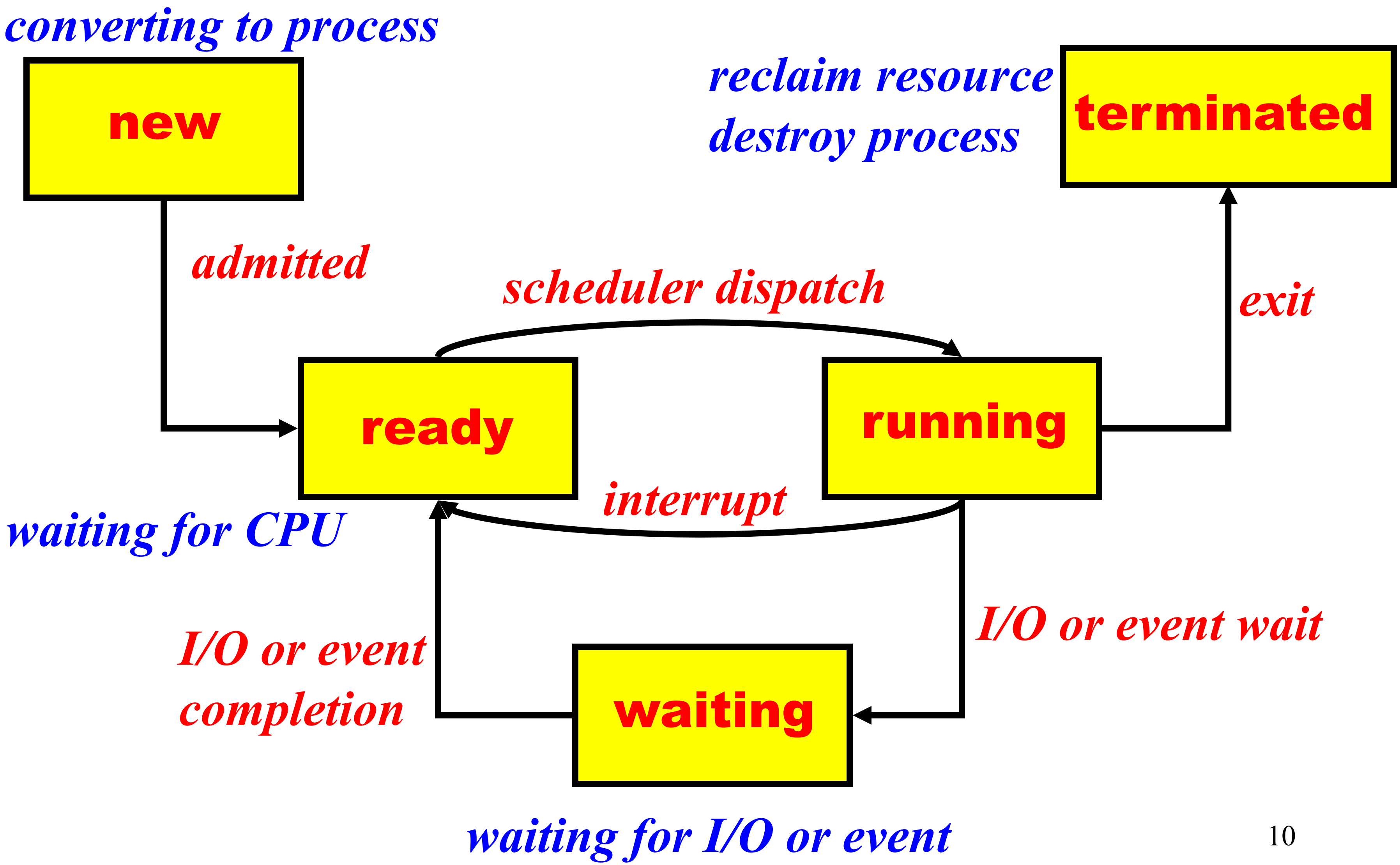
Problem 2(b)

- An **interrupt** is an event that requires OS's attention. It may be generated by hardware (e.g., I/O completion and timer) or software (e.g., system call and division by 0).
- Interrupts generated by software (e.g., division by 0, page fault and system call) are traps.
- Don't forget mode switch.
- Interrupts are **not signals** and are **not called**. Signals have a different meaning in operating systems.

Common Problems

- Interrupts are not machine instructions, not signals, not functions/procedures.
- Signals have a different meaning in OS.
- Interrupts, machine instructions, threads, processes are **NOT** called.
- OS does not call an interrupt. Except for system calls and a few others, interrupts are **not** called to happen.
- Many answered this question by stating the result of an interrupt rather than talking about an interrupt itself.

Problem 3(a)



10

Problem 3(b)

- The **context** of a process is the environment for that process to run properly.
- This includes process ID, process state, registers, memory areas, program counter, files, scheduling priority, etc.
- The sequence of actions are:
  - Control switches back to the OS. Mode switch may be needed.
  - The outgoing process is suspended, and its context saved. Depending on the nature of this context switch, this outgoing process may be moved to the Ready or Waiting state. It could also be moved to the Terminated state if it exits or causes an error.
  - The context of the incoming process is loaded and its state is set to Run.
  - Resume its execution. Mode switch may be needed.

Problem 4(a): 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 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 4(a): 2/9

```
int  x, a = 4, b = 5;
```

	Process 1	Process 2
exe seq 1		
`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 4(a): 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 4(a): 4/9

```
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 4(a): 5/9

First Execution Sequence

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 4(a): 6/9

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 4(a): 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 4(a): 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 <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 4(a): 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
```

variable count is shared concurrently here

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

Problem 5(a)

```
printf("The root process %d, ppid = %d\n\n", getpid(), getppid());
n = atoi(argv[1]);
for (i = 1; i <= n; i++) {
    if ((pid = fork()) == 0) {              // left child
        printf("My ID = %d My PPID = %d\n", getpid(), getppid());
        exit(0);                            // left child must exit
    }
    else {                                  // parent
        if ((pid = fork()) == 0) {          // right child
            printf("My ID = %d My PPID = %d\n", getpid(), getppid());
        }                                   // must keeps creating
        else {
            wait(NULL);                     // parent must wait for
            wait(NULL);                     //    both children
            exit(0);                        // parent exits
        }
    }
}
```

Problem 5(b): 1/2

- Obvious cases are as follows (i.e., 2, 3 and 4):

Process 1	Process 2	x in memory
	`x = 2*x`	0
`x++`		1
`x++`		2

Process 1	Process 2	x in memory
`x++`		1
	`x = 2*x`	2
`x++`		3

Process 1	Process 2	x in memory
`x++`		1
`x++`		2
	`x = 2*x`	4

The final result cannot be greater than 4, because `x=2*x` can only double the result of Process 1.

Problem 5(b): 2/2

- Non-obvious cases are as follows (i.e., 0 and 1):

Process 1	Process 2	x in memory
	`LOAD x`	0
	`MUL #2`	0
`x++`		1
`x++`		2
	`SAVE x`	0

Process 1	Process 2	x in memory
	`LOAD x`	0
	`MUL #2`	0
`x++`		1
	`SAVE x`	0
`x++`		1

Problem 5(c): 1/2

```
int   status[2];    // status of a process
int   turn;         // initialized to either 0 or 1
```

P_0

```
status[0]=COMPETING;
while (status[1]==COMPETING) {
  status[0]=OUT_CS;
  repeat until (turn==0);
  turn = 0;
  status[0] = COMPETING;
}
```

P_1

```
status[1] = COMPETING;
while (status[0]==COMPETING) {
  status[1]=OUT_CS;
  repeat until (turn==0 || turn==1);
  turn = 1;
  status[1] = COMPETING;
```

Before entering `while`, `status[ ]` is COMPETING

When loops back `status[ ]` is set to COMPETING

P_0 enters its critical section

iff `status[0]` is COMPETING and `status[1]` not COMPETING

P_1 enters its critical section

iff `status[1]` is COMPETING and `status[0]` not COMPETING

If both P_0 and P_1 are in their critical section, `status[0]` (and `status[1]`) must be COMPETING and not COMPETING at the same time.

Problem 5(c): 2/2

```
int   status[2];    // status of a process
int   turn;         // initialized to either 0 or 1
```

P_0

```
status[0]=COMPETING;
while (status[1]==COMPETING) {
  status[0]=OUT_CS;
  repeat until (turn==0);
  turn = 0;
  status[0] = COMPETING;
}
```

P_1

```
status[1] = COMPETING;
while (status[0]==COMPETING) {
  status[1]=OUT_CS;
  repeat until (turn==0 || turn==1);
  turn = 1;
  status[1] = COMPETING;
```

turn plays no role here.

REASON:

- If a process sets status[ ] to COMPETING and finds the other status[ ] being non-COMPETING, this process enters its critical section.
- In this case, the process never sets its status[ ] to OUT_CS and turn to 0 or 1.

Hence, you should not use turn and OUT_CS in your argument.

Class Performance

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

Problem		Possible	Expected	Class Average	Class Median
1	a(i)	3	2	1	1
	3(ii)	7	5	3	4
2	a	10	8	8	9
	b	10	8	6	7
3	a	10	8	8	10
	b	10	8	6	7
4	a	10	8	6	7
5	a	10	7	4	4
	b	15	10	8	9
	c	15	10	4	0
	Total	100	74	55	51

50 points expected

from class slides directly

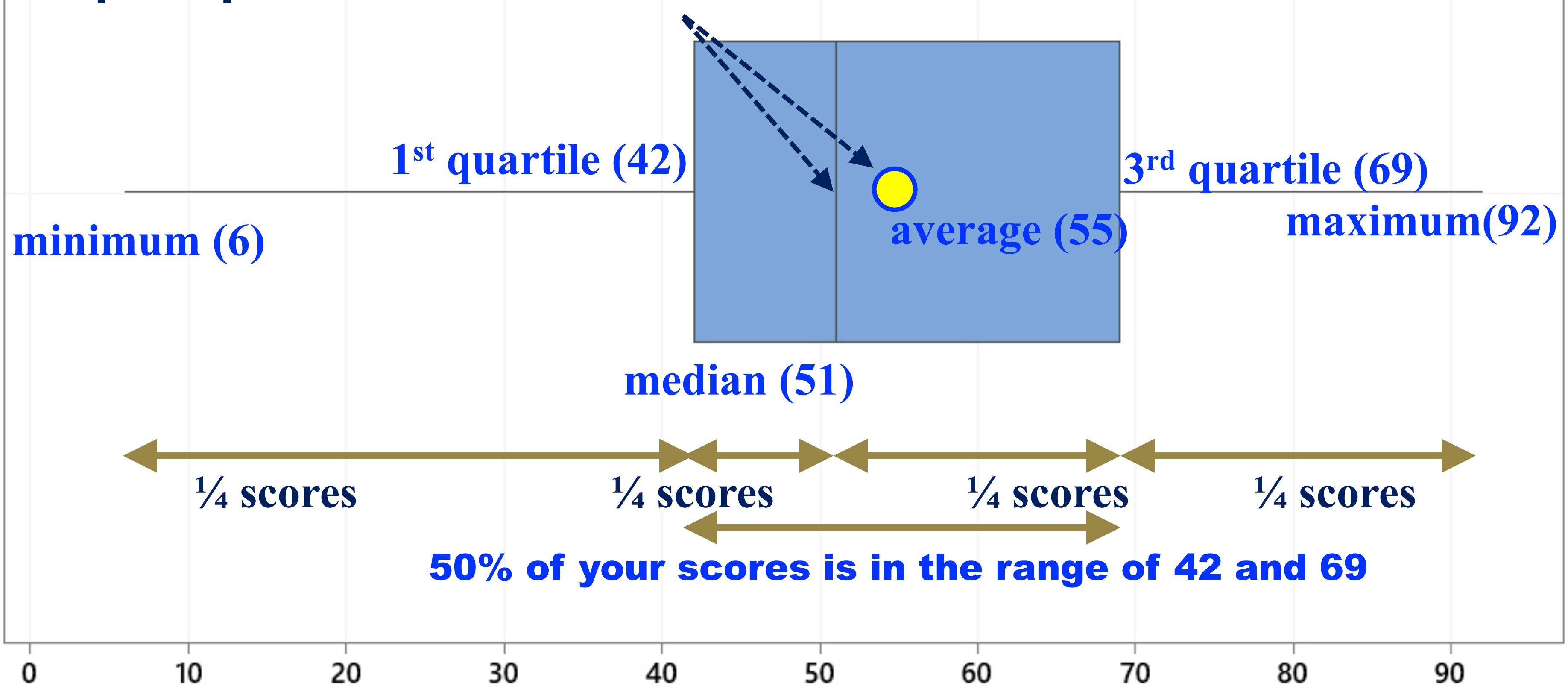
Grade Distribution Problem-Wise

	1a	1b	2a	2b	3a	3b	4a	5a	5b	5c	Class
Min	0	0	0	0	0	0	0	0	0	0	6
Max	3	6	10	10	10	10	10	10	15	15	92
Median	1	4	9	7	10	7	7	4	9	0	51
Avg	1	3	8	6	8	6	6	4	8	4	55
St DEV	1	2	3	3	3	3	3	3	5	6	20

- **Problems 2a, 2b, 3a, 3b and 4a are from our course slides.**
- **Problem 1 is an exercise stated in Programming Assignment I.**
- **Problem 5a tests whether you know `fork()` properly.**
- **Problem 5b tests whether you can use machine instruction interleaving.**
- **Problem 5c is a simple problem using prove-by-contradiction.**

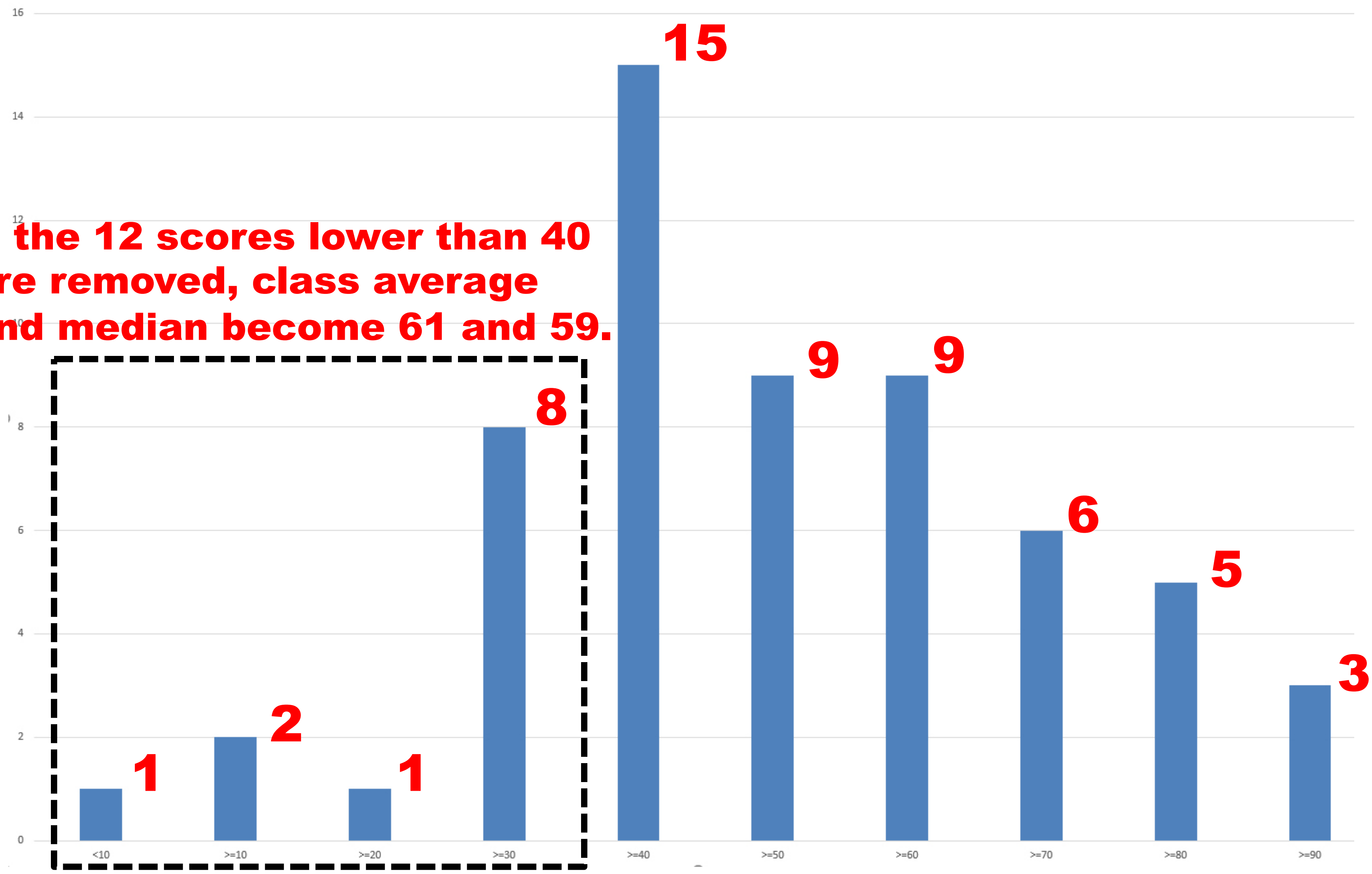
Boxplot

Average > Mean →
top half performed better than the lower half



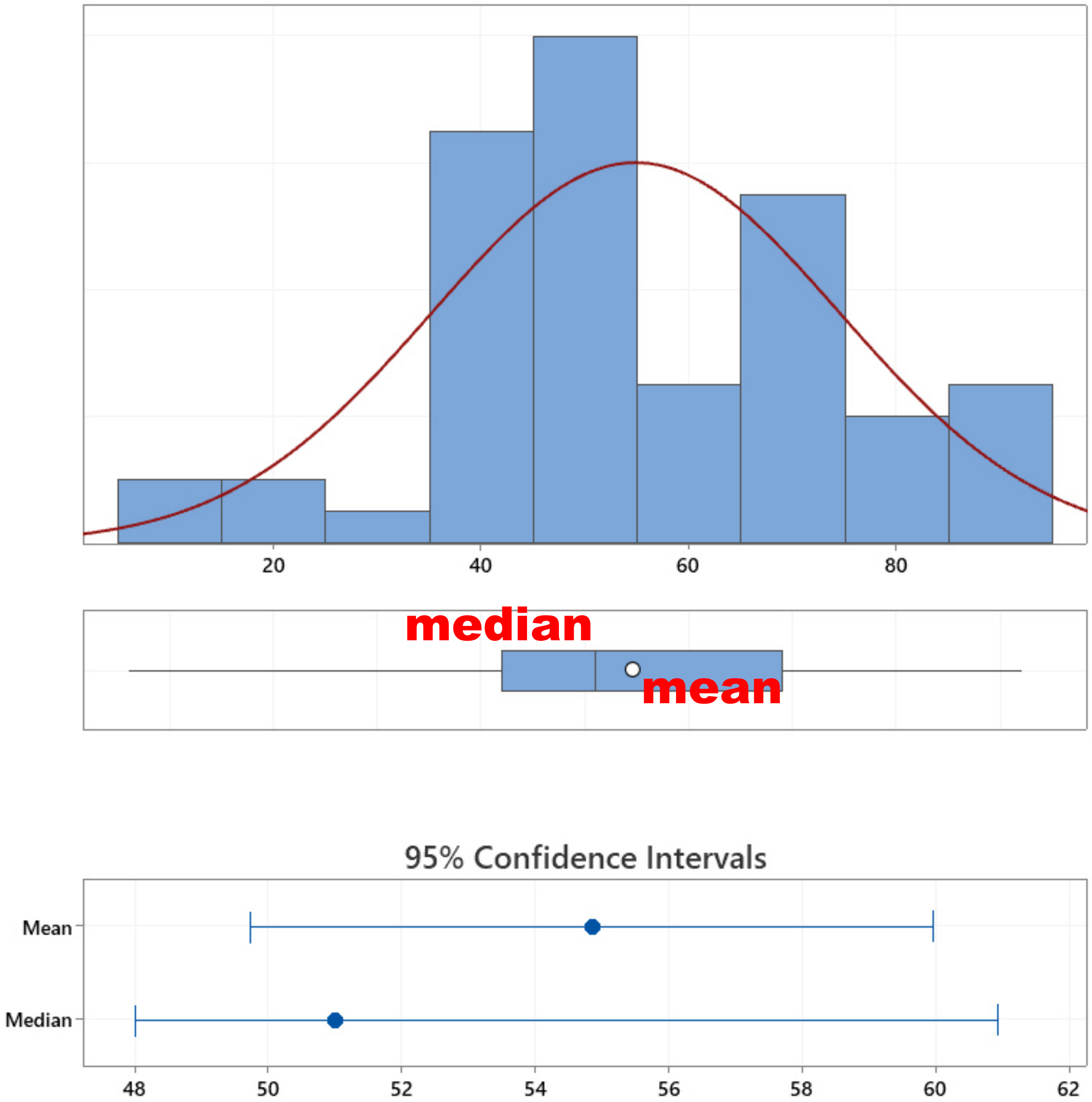
There were no outliers

Grade Distribution



If the 12 scores lower than 40 are removed, class average and median become 61 and 59.

Boxplot


20
40
60
80
median
mean
95% Confidence Intervals
Mean
Median
48
50
52
54
56
58
60
62

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.**
- If execution sequences are needed, always provide valid ones. Otherwise, you will receive a **ZERO**.
- 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!
- **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