UC Berkeley's CS10 Fall 2019 Midterm Answers

Question 1: What was shared with you in the *Testing + HW3* lecture? (select ONE)

○ You should try to use *mutable data* whenever possible, since it makes testing easier.
FALSE! This was one of the points of the lecture, that *mutable* data structures make testing difficult, since each run of the code mutates your data structures, and you always have to reset them to the initial state before re-running the tests. Also, if your code crashes halfway through for some reason, it isn't always easy to reset them.

[✓ You should try to use *immutable data* whenever possible, since it makes testing easier.]
TRUE! Since the data is immutable, you can just use our testing block, with no worry about "resetting" your data structures.

○ Putting all your code in one monolithic, top-level script *is a good thing*, since it makes testing easier.
FALSE! If it works, great, but if there's a problem where is it? You want to use abstraction and functional decomposition to break your problem down into smaller sub-problems which themselves can be tested.

○ You can prove that a block *with state and finite inputs* is correct by testing it on all the possible inputs.
FALSE! The finite inputs helps a lot, but the state makes it impossible to prove that it'll always work. The state elements (say, global variables) may play nice when you're testing it, but who knows what their value will be in the future?

Question 2: What was shared with you in the *Computing & the Environment* lecture? (select ONE)

○ E-waste is worth $62.5 billion/yr, so *first-world nations are competing to receive and process it*.
FALSE! The e-waste is produced by first-world nations, but almost universally exported to third-world nations.

○ Thanks to streaming services (e.g., Netflix and Spotify) *overall global e-waste emissions is decreasing*.
FALSE! This one was tricky – while streaming services HAVE contributed to reduced e-waste (since there's no need for CDs and DVDs much anymore), overall global e-waste continues to grow as developing countries get richer and start purchasing e-devices (then throwing them away) at an increased pace.

○ Researchers are using old cell phones for *bio-acoustic monitoring of the ocean*, to hear whales.
FALSE! They were using it for bio-acoustic monitoring of the forest, not the ocean.

[✓ Researchers are using old cell phones as a *low-power, low-cost distributed computing cluster*.]
TRUE! This was directly from a slide in lecture.

Question 3: What was shared with you in the *Computers in Education* lecture? (select ONE)

○ Using Judah Schwartz' definitions, *Snap! would be a Microworld*.
FALSE! Snap! is would be classified as a *Tool*.

○ cMOOCs are "classroom-style" MOOCs, where lectures from the world's best lecturers are emphasized.
FALSE! The "c" in cMOOC stands for "connectivist", where lectures are de-emphasized, and learning from your peers is paramount.

○ Prof Harvey: "The most important use of computers in education is web search to access information."
FALSE! He believed it was "iclicker-like technologies", since the multiple-choice "i-clicker" standardized testing has changed what passes for knowledge.

○ Sir Ken Robinson believes that we should have more standardized testing, since that brings more efficiency.
FALSE! Much of the video had him railing *against* standardized testing, and standardization, saying we should go "a whole other way"

[✓ None of these]

Question 4: What was shared with you in the *Concurrency* lecture? (select ONE)

○ *Amdahl's law* predicted the number of transistors on a chip would double every two years.
FALSE! That's Moore's law.

○ *Moore's law* said that the maximum speedup is a function of the percent of serial code you have.
FALSE! That's Amdahl's law.

○ *Time sharing* is a technique to allow multiple CPUs to share the work for a same task (a single thread).
FALSE! Time sharing allows a single CPU to mask as multiple CPUs by giving small rotating slices of time to each thread.

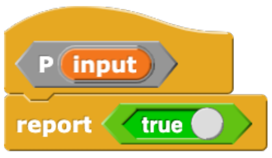
○ If four self-driving cars arrive at a four-way stop simultaneously, and nobody moves, that's a *race condition*.
FALSE! That would be deadlock.

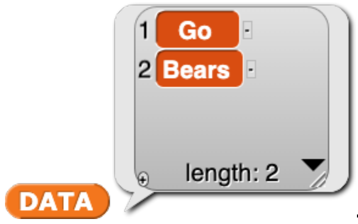
[✓ None of these]

Question 5: If the max speedup with ∞ cores is 5x, what percentage of the code is <u>serial</u>? (select ONE)
Max speedup with ∞ cores is 1/SerialFraction, so 5 = 1/s → s = 1/5 = 20%

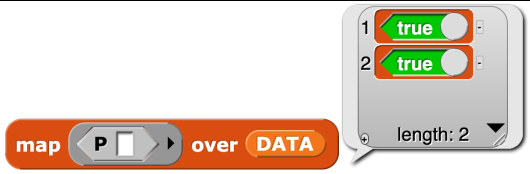
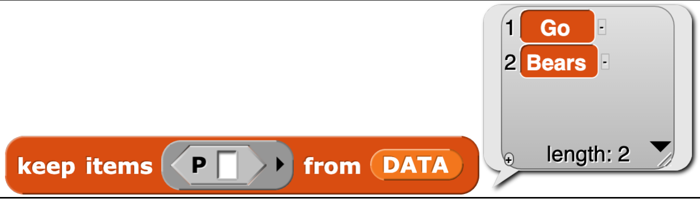
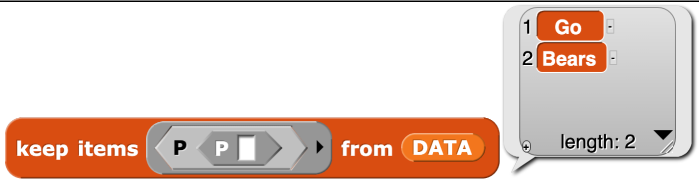
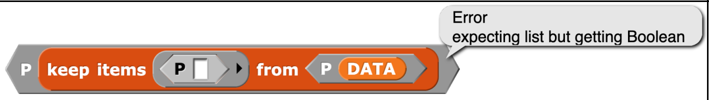
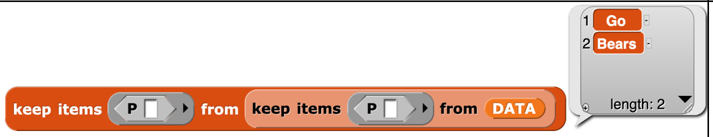
Question 6: What is $12_{16} \div 11_2$? (select ONE)
$12_{16} = 1 * 16^1 + 2 * 16^0 = 1 * 16 + 2 * 1 = 18$
$11_2 = 1 * 2^1 + 1 * 2^0 = 1 * 2 + 1 * 1 = 3$
$18 \div 3 = 6$

Question 7: ***Two keeps are better than one! (…or are they? Bwahaha...)***

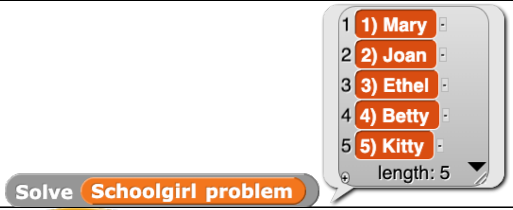
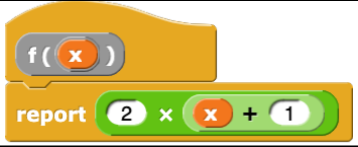
You are given the following predicate 
P (input)
report (true)

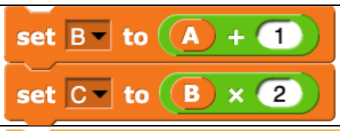
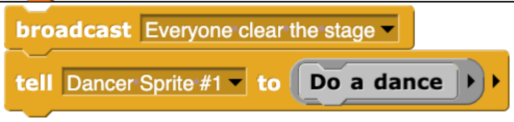
and list 
DATA
1 Go
2 Bears
length: 2

.
What do the following expressions return?

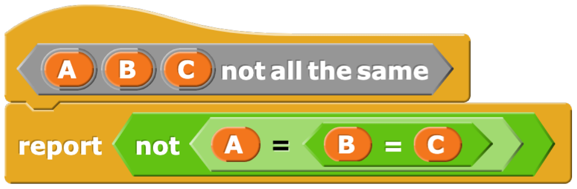
 map (P [ ]) over DATA 1 true 2 true length: 2	map returns a new list in which every element of the input has had P applied to it. P is a function that ignores its input and reports true. So we get (true true).
 keep items (P [ ]) from DATA 1 Go 2 Bears length: 2	keep returns a new list in which every element of the input is checked against P, and if P returns true, then the element is kept, otherwise it's not kept. Since P always returns true, all elements are kept.
 keep items (P (P [ ])) from DATA 1 Go 2 Bears length: 2	Here each element is passed through P twice. The innermost P ignores the word, and always returns true which is passed through the second P which ignores the input (true) and always returns true. Thus they're all kept.
 P (keep items (P [ ]) from (P DATA)) Error expecting list but getting Boolean	Snap*!* always evaluates expressions inside out, so the first things that's called is P(data) which returns true. That is passed to keep, which is expecting its second argument to be a list, and when it's not, it gives the error shown.
 keep items (P [ ]) from (keep items (P [ ]) from DATA) 1 Go 2 Bears length: 2	This is two applications of keep. We saw (from the second row in this table) that keep<P()>from(DATA) returns DATA, so this is just DATA passing twice through the filter that isn't really a filter at all since it lets every list pass through untouched!

Question 8: Match each programming paradigm with the description. There should be only one per row and one per column, so if some rows match more than one column, adjust it so they all work.

 Solve Schoolgirl problem 1 1) Mary 2 2) Joan 3 3) Ethel 4 4) Betty 5 5) Kitty length: 5	**Declarative** – this was exactly the the example we showed in class. It's also functional (since it a call to a function), but the function definition below is better.
 f(x) report (2 × (x + 1))	**Functional** – Here we're defining a simple f(x)=2(x+1) function.

 `set B to (A + 1)` `set C to (B × 2)`	**Imperative/Sequential** – Do this(set B) then that (set C), with mutation.
 `broadcast Everyone clear the stage` `tell Dancer Sprite #1 to (Do a dance)`	**Object-Oriented Programming (OOP)**, since one sprite is both sending a message to ALL sprites (through broadcast), as well as sending a message to a particular sprite to run its “Do a dance” method (here a command).

You author the following (possibly buggy) code because you want to return **true** when **A**, **B**, *and* **C** are not all the same. That is, return **false** only when **A**, **B**, *and* **C** are all **true** or all **false**.

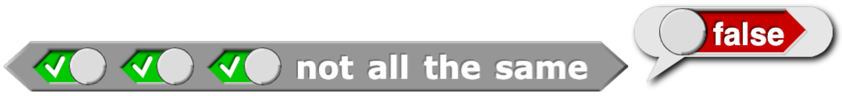


```
(A) (B) (C) not all the same
report not ((A) = ((B) = (C)))
```

For the following cases, choose the appropriate values for **A**, **B** and **C**. *(There may be multiple right answers)*
(For each (a)-(d), select ONE per row, or select “Impossible to achieve!” if it can’t be done)

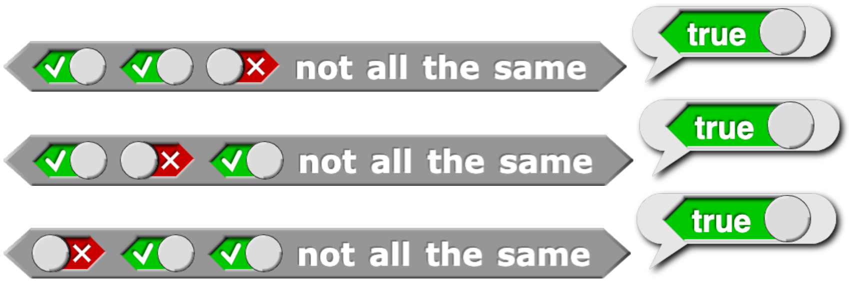
e)

is supposed to return **false**, and *does* return **false**.



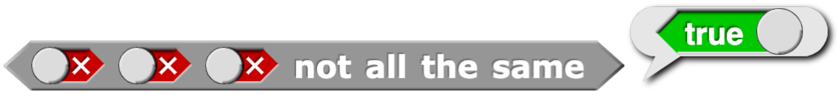
f)

is supposed to return **true**, and *does* return **true**.



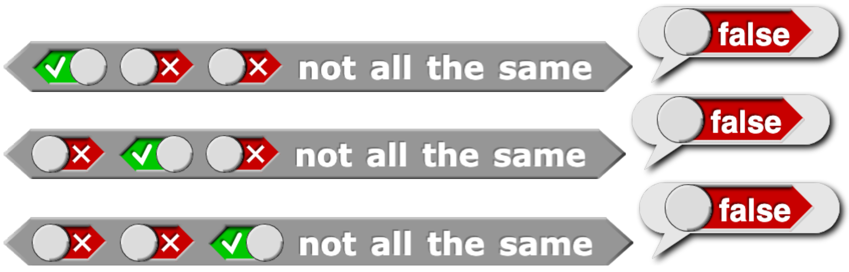
g)

is supposed to return **false**, but returns **true**.



h)

is supposed to return **true**, but returns **false**.



Question 10: *Folks, please line up by SID, smallest to largest…* (12 pts, 3*2+6*1 pt each)

Students are asked to stand and line up by student IDs (SIDs), **smallest in the front and largest in the back**. Here are 3 algorithms to find out if they are or not. For all problems, assume the number of students (N) is a power of 2 minus 1 (e.g., 2^1-1=**1**, 2^2-1=**3**, 2^3-1=**7**, 2^4-1=**15**, 2^5-1=**31**, …) and *really* big. (How big?) Really big. Also, "clock time" is actual elapsed time if you *used a stopwatch to time the algorithm*, **and SIDs are unique.**

Algorithm I – "Everyone" algorithm

6. All at once, everyone (but the person in the back) writes their SID on a piece of paper, puts it in their left hand and hands it to the person behind them over their right shoulder.
7. Everyone (but the person in the front) takes the paper being handed to them in their right hand.
8. The first person sits down.
9. If anyone notices the given SID is more than theirs, they yell "NOT IN ORDER", otherwise they sit down.
10. If the person in back sees everyone sitting (including themselves), they yell "IN ORDER"

Algorithm II – "Divide and Conquer" algorithm

5. You walk to the middle student of the consecutive standing students, and make sure their number is greater than the SID of the person directly in front of them and less than the SID of the person directly behind them (whether standing or seated), skipping the comparison if there is nobody there.
6. If any of these are out of order (front bigger than back), you yell "NOT IN ORDER" and stop.
7. Otherwise, have that student sit down. If there are no remaining standing students, yell "IN ORDER"!
8. Otherwise, go to the consecutive group of standing students in front of the one seated, and ask a friend – you have an infinite amount of non-student friends – to replicate what you're doing for a similar consecutive group of standing students behind the one just seated, and both of you go to step 1.

Algorithm III – "Random" algorithm

11. If no students are standing, you yell "IN ORDER" and stop.
12. You choose a random student from those who are standing.
13. You make sure that SID is bigger than the SID of the person directly in front (whether standing or seated) and less than the SID of the person directly behind them (whether standing or seated), skipping the comparison if there is nobody there.
14. If any of these are out of order (front bigger than back), you yell "NOT IN ORDER" and stop.
15. Otherwise, you ask that student to sit down and go to step 1.

a) Is each *correct* (i.e., always return the correct value, not error or run forever)? (select ONE per row)

Algorithm	Yes	No	Rationale
Everyone	●	○	All of these algorithms work because they check everyone (eventually). "Everyone" has all the people working in parallel with all but the person in the front doing the check for them and their front neighbor, "Divide and Conquer" works to split and split and split (etc) to check everyone with their person in front and back, and "Random" randomly takes a student from the set of standing students until nobody is standing. Often random algorithms aren't guaranteed to work, but this one does since it eventually moves everyone from the set of standing folks to the set of sitting folks.
Divide and Conquer	●	○	
Random	●	○	

b) In the WORST case, what's the *number of comparisons* (NOT running time)? If it's actually between two categories, pick the bigger category. E.g., N^4 is bigger than *cubic*, so pick *exponential*. (select ONE per row)

Algorithm	Linear	Rationale
Everyone	●	Every algorithm has every person do either one check with the person in front ("Everyone") or two checks, one for the person in front and one for the person in back ("Divide and Conquer" and "Random"). So that's either 1*N or 2*N comparisons total, which are all linear.
Divide and Conquer	●	
Random	●	

c) How *much clock time* (NOT running time) would each take in the WORST case? (select ONE per row)

Algorithm	Constant	Logarithmic	Linear	Rationale
Everyone	●	○	○	The algorithm works in exactly 4 steps, because step 3 has everyone working at the same time.
Divide and Conquer	○	●	○	The algorithm works in $\log_2(n+1)$ clock time because it works in a divide-and-conquer way with 1 comparison in the first iteration, 2 comparisons (in parallel) in the second iteration, 4 comparisons (in parallel) in the third, etc. 31 people takes $\log_2(32) = 5$ iterations, as demonstrated below 0000000000000001000000000000000 0000000100000001000000010000000 0001000100010001000100010001000 0101010101010101010101010101010 1111111111111111111111111111111
Random	○	○	●	The algorithm works in (worst case) N steps, since every iteration asks one person to compare with their neighbors and sit down, and nothing is happening in parallel.

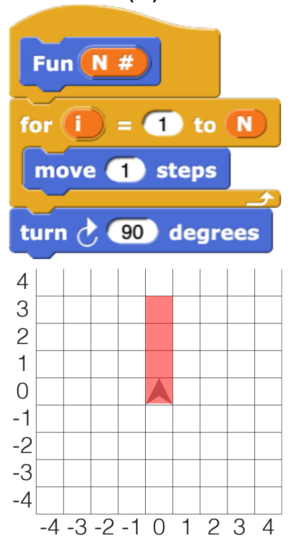
Question 11: どうもありがとうミスターロボット ***Dōmo arigatō, Mr. Roboto…*** (16 pts, 2+2+2+10)

(Clarification: if the sprite were at (0,0) and moved 2 steps up, it would be at (0,2) and all pixels along the line from (0,0) through (0,2) would be shaded; 3 pixels in total.)

For (a), (b), (c), and (d) **we start with the pen down, the sprite in the middle of the grid, facing up,** as shown. Your job is to shade in (completely!) *all* the pixels that will be colored in after Fun 3.

…and here is how we would have written each if we were texting our friend our code (note the indenting):

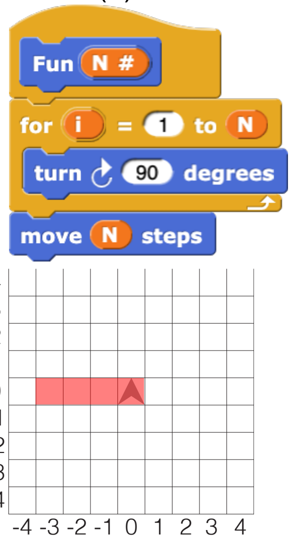
(a)



```
Fun N #
for i = 1 to N
    move 1 steps
turn 90 degrees
```

```
Fun(N)
for i = 1 to N
--> move 1 steps
turn right 90
```

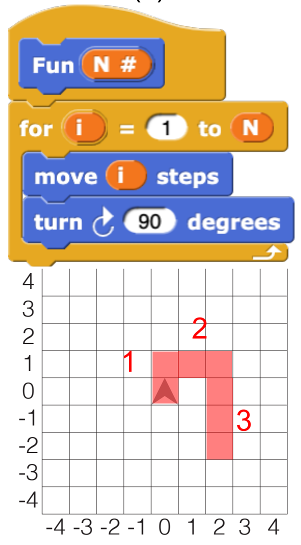
(b)



```
Fun N #
for i = 1 to N
    turn 90 degrees
move N steps
```

```
Fun(N)
for i = 1 to N
--> turn right 90
move N steps
```

(c)

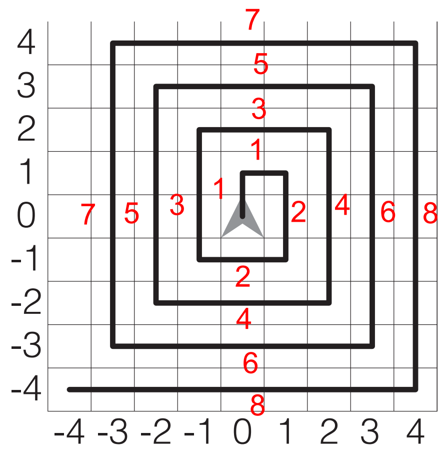


```
Fun N #
for i = 1 to N
    move i steps
    turn 90 degrees
```

```
Fun(N)
for i = 1 to N
--> move i steps
--> turn right 90
```

d) Well, all of those were pitiful attempts at writing code that would have the sprite spiral outward perfectly, like the picture below. If **N** were big enough, **Fun(N)** would eventually shade every pixel.
Write the code for **Fun(N)** that does this in the lines below, using the "text your friend" style we show above. Make sure to use arrows to indent the inside part of any **for** loop you use. You might not need all the lines.

```
Fun(N)
for i = 1 to N
--> move i steps
--> turn right 90
--> move i steps
--> turn right 90
```


4
3
2
1
0
-1
-2
-3
-4
-4 -3 -2 -1 0 1 2 3 4
7
5
3
1
1
2
3
4
5
6
7
8
2
4
6
8