

# CS10 Midterm Exam - Summer 2019

\_\_\_\_\_  
*Your Name (first, last)*

\_\_\_\_\_  
*ID Card Number*

\_\_\_\_\_  
*Your TA's Name*

← \_\_\_\_\_  
*Name of person on left (or aisle)*

\_\_\_\_\_ →  
*Name of person on right (or aisle)*

Fill in the correct circles & squares completely...like this: ● (select ONE) ■ (select ALL that apply)

## **Questions 1-4: What's that Smell? It's Potpourri! (3 points each, 12 points total)**

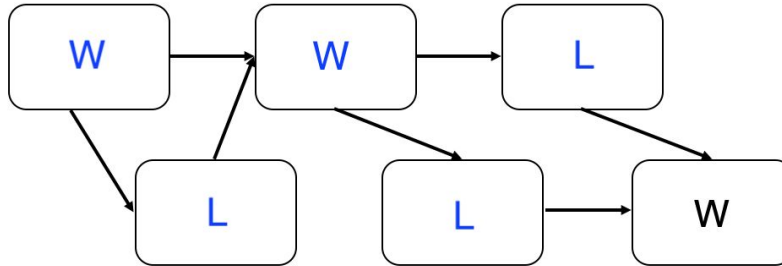
**Question 1:** Which of the following points was NOT discussed in the Environment & Computing Lecture?

- We have plenty of information on where and how our electronic waste is disposed of.
- It is possible to repurpose old cell phones into bio-acoustic monitors and servers.
- People who are most affected by eWaste pollution also tend to be most dependent on it.
- Planned Obsolescence is the practice of creating devices with artificially limited lifespans.
- Streaming and subscription-based services can help reduce the amount of eWaste we produce.

**Question 2:** Which of the following is NOT true from the Social Media and Virtual Communities lecture?

- Many new technologies are initially met with fear and suspicion: current attitudes towards the rise of AI are not new.
- Google gets much of its big data from Google Home, which listens to your conversations and uses the data from them.
- Big Data is not only used to drive advertisements, but also to (hyper) customize technology to users' needs and lifestyles.
- All of the above are true from the Social Media and Virtual Communities lecture.

**Question 3:** Below is a state tree for an imaginary game. Circle the appropriate outcome for each of the states. We've filled the rightmost state out for you.



**Question 4:** Which of the following is true about the way we measure the runtime of a block?

- Since different computers run at different speeds, we measure the runtime of a block in terms of the number of steps it takes, not the amount of time it takes.
- Since time is very precise, we measure the runtime of a block in terms of the amount of time it takes to run as opposed to the number of steps it takes.
- We usually measure the best-case runtime of a block.
- None of the above.

**Question 5: Sign and Magnitude (3 points each, 9 points total)**

So far, we have only learned how to represent positive numbers in binary. However, there are schemes to represent negative numbers in binary as well. Consider the sign and magnitude scheme: given a binary number, the leftmost bit represents the sign (if it's 1, the number is negative, if it's 0, the number is positive), and the rest of the bits represent the number itself. For example,  $0b1100_2$  would be  $-0b0100_2$ .

- a. In the box to the right, write out the representation for -19 using sign and magnitude.

110011

First, we can find the representation for 19 in binary. 19 in binary is  $0b10011$ . Since -19 is negative, the first sign and magnitude digit must be 1. Therefore, we can represent -19 using sign magnitude by putting a 1 in front of 10011, giving us 110011.

- b. What is the **minimum** number (in decimal) we can represent using 5 digits in sign and magnitude?

-15

The minimum number will be the most negative number. Since the first digit represents the sign, we want the first digit to be 1, representing a negative number. Since we want the number that is most negative, we should maximize the other digits by having them all be 1's. The resulting sign and magnitude number is 11111. In decimal, this is -15.

- c. What is the **maximum** number (in decimal) we can represent using 5 digits in sign and magnitude?

15

To find the maximum number, we want a positive number, so the first digit should be 0. Since we want the largest positive number, the remaining four digits, should be 1's. The resulting sign and magnitude number is 01111. In decimal, this is 15.

**Question 6: Speeding Up (3 points)**

If we have a program that is 50% parallelizable, how many cores would we need to make it run 4 times as fast? Write your answer in the box to the right. If it isn't possible to achieve such speedup, you can write "Impossible."

Impossible


From Amdahl's Law, the maximum speedup of a program is  $\frac{1}{1-p}$ . Here, we have  $p = \frac{1}{2}$ . The maximum speedup is therefore  $\frac{1}{1-1/2} = 2$ , and we cannot achieve 4 times speedup.

**Question 7: Wow, these numbers are p-norm-ous! (4 points)**

In mathematics, the **p-norm** is a function of a list of numbers. The output is a single number, the **p-norm**.

To calculate the **p-norm** of a list with  $n$  items:

- For each number in the list, find the absolute value and then raise the value to the power  $p$ .
  - $|item\_1|^p, |item\_2|^p, \dots, |item\_n|^p$
- Sum the resulting values by adding them all together.
  - $|item\_1|^p + |item\_2|^p + \dots + |item\_n|^p$
- Raise this number to the power  $1/p$ .
  - $(|item\_1|^p + |item\_2|^p + \dots + |item\_n|^p)^{(1/p)}$

Example: the **2-norm** of the list  is 5:

- $|3|^2 = 9, |-4|^2 = 16$  (find the absolute value of each item and raise to the power  $p$ )
- $9 + 16 = 25$  (sum the resulting values)
- $25^{(1/2)} = 5$  (raise this number to the power  $1/p$ )

Write a function that calculates the **p-norm** of a list. This function has two inputs: (1) a number  $p$  and (2) a list of **numbers**. Fill in the blanks to complete the function. Write this function using higher-order functions. **Do not** use any loops or recursion for this question.

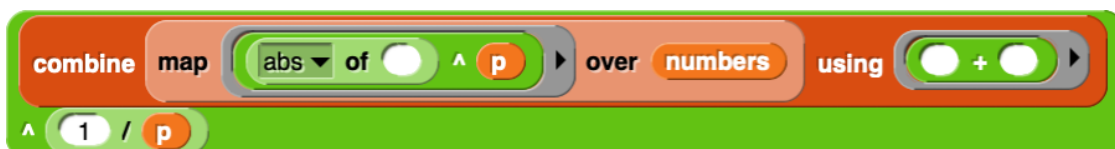
Hint: These blocks can be used to find the absolute value and to perform exponentiation, respectively.

abs of

^

(p) norm of (numbers):

report



The code block consists of the following elements from top to bottom:

- A **combine** block.
- A **map** block containing:
  - An **abs** block.
  - An **of** block.
  - An **^** block.
  - A **p** block.
- An **over** block containing a **numbers** block.
- A **using** block containing a **+** block.
- A **^** block.
- A **1** block.
- A **/** block.
- A **p** block.

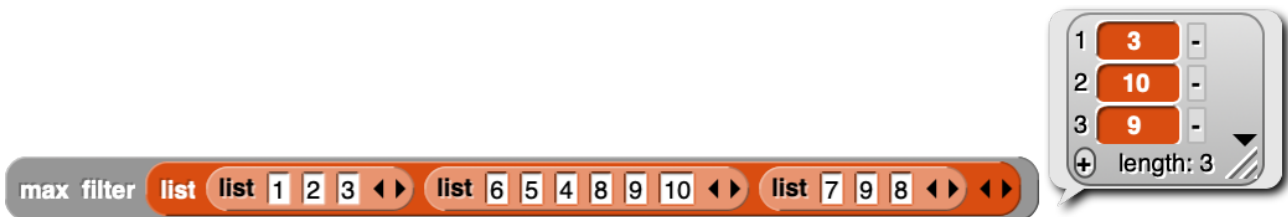
**Question 8: Lists (of Lists) (4 + 3 + 2 = 9 points)**

- a. Write a function called ***max filter*** which takes in a (2-dimensional) list of lists of numbers and reports a new (1-dimensional) list of numbers. Each value in the output list will be the maximum value from the list at the corresponding index in the input list.

For example:

- If item 1 of the input list is **list 1 2 3**, item 1 of the output list should be 3.
- If item 2 of the input list is **list 6 5 4 8 9 10**, item 2 of the output list should be 10.
- If item 3 of the input list is **list 7 9 8**, item 3 of the output list should be 9.

Here is an example call to the function:



Write this function in the box below. **If you are able to write it using only higher-order functions (no loops, no recursion), you will receive 2 bonus points.**

Hint: You may use the function **max of** and **and**, which reports the maximum of two numbers. You may assume that we have given you this function and do **not** need to write it yourself.

Possible solution:

max filter (input):

```

script variables result max value
set result to list
for each sublist in input
  set max value to item 1 of sublist
  for each number in all but first of sublist
    set max value to max of max value and number
  add max value to result
report result
  
```

Only using HOFs: add report and input list

max filter (input):

```

report map combine using max of and over input
  
```

Fun fact: this process is an example of what is called **convolution** and is a popular technique used in neural networks for artificial intelligence and machine learning!

- b. Now, fill in the **test** block below to test the block you wrote in part (a). We left the three inputs blank. Fill in the blanks by writing what should go in each blank on the lines below. Your test case must be different from the example we showed above.

```

test 1 with input(s) 2 and expected output 3
  
```

Using the example from part a (students are required to create their own example):

- max filter
- list list 1 2 3 list 6 5 4 8 9 10 list 7 9 8
- list 3 10 9

- c. Your friend has created a new representation of boards in Snap! They want to be able to update an item of the board using the **replace** block. However, there seems to be a bug because the result of the following code is **not** what they expected!

Here is their code:

```

set BOARD to list
script variables row
set row to list [ ] [ ] [ ] [ ] [ ]
repeat 5
  add row to BOARD
  replace item 2 of item 3 of BOARD with 1
  
```

As a reminder, here is an example of how boards are displayed in Snap!:



In the grid below, write out what **BOARD** would look like after running your friend's code: Hint: Remember that your friend's code is buggy and the result of the block is **not** what they initially expected!

	1			
	1			
	1			
	1			
	1			

**BOARD** consists of 5 references to the same list, **row**. If we replace item 2 of item 3 of **BOARD** with 1, the second item of **row** will be set to 1. Since each of the 5 items of **BOARD** references this list **row**, all 5 rows of **BOARD** will have a 1 at index 2.

**Question 9: Full of Lists (4 points each, 16 points total)**

A **full list** is defined as a list that contains all of the numbers from 1-10. It may contain other numbers, but as long as each number from 1-10 appears at least once (each number may appear multiple times), the list is a **full list**.

- [1, 2, 3, 4, 5] **is not a full list** because it doesn't contain the numbers 6-10.
- [1, 10, 4, 2, 5, 3, 6, 9, 8, 7, 13] **is a full list** because it contains all of the numbers from 1-10.
- [10, 10, 9, 8, 8, 7, 6, 5, 4, 3, 2, 1] **is a full list** because it contains all of the numbers from 1-10.

Below, we've written various implementations of blocks that checks if a given list of positive integers is a **full list**. Some of these blocks, however, may be buggy. For each of the blocks below, indicate an input that would report the incorrect value if passed into our block (if the block works correctly, just write "correct") and the runtime of the block. **You may assume that the input list only contains positive integers!**

You may assume most Snap! blocks run in constant time. However, the following blocks do not:

- The **contains** block runs in linear time in terms of the size of its input list.
- **map**, **keep**, and **combine** run their input function for every item of their input list.

A.



Buggy Input: [1] With this input, the block will report True, even though this is not a full list.

Runtime:  constant  logarithmic  linear  quadratic  exponential

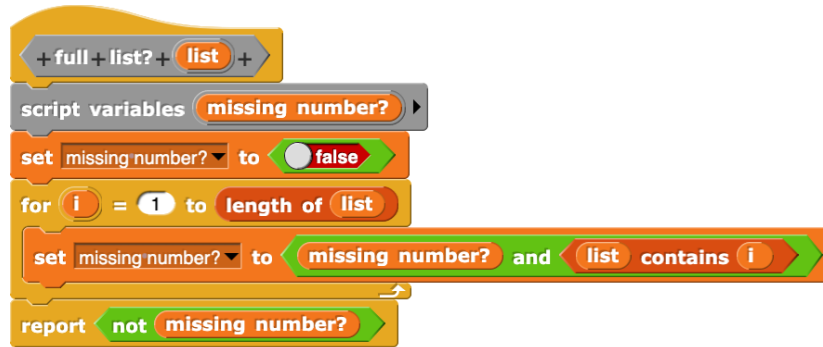
The **map** block will run **contains** for the numbers 1 to 10. Since **contains** runs in linear time, **map** will run a linear function 10 times. Therefore, the **map** block takes linear time.

**map** reports a list of 10 booleans, which will be the input to **combine**. **combine** will then run a constant function over a list of length 10. Therefore, the **combine** block takes constant time.

Since we have a linear runtime, followed by a constant runtime, the overall runtime of this block is linear.



B.

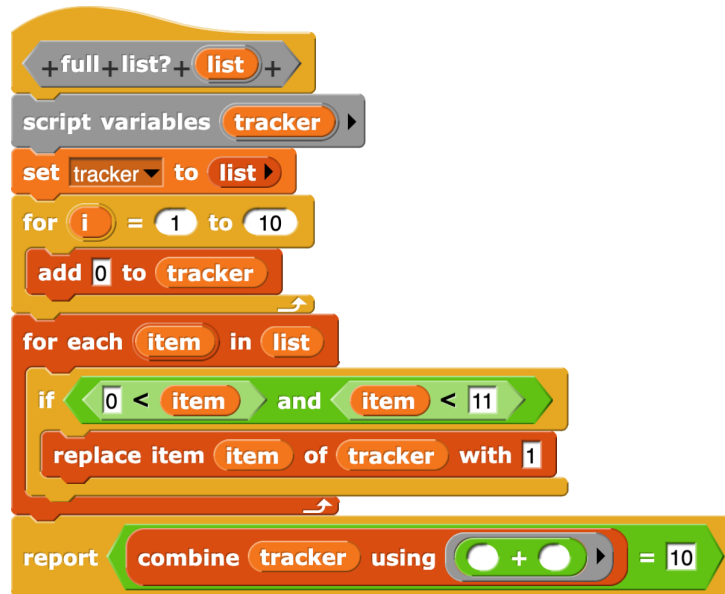


Buggy Input: [1] With this input, the block will report True, even though this is not a full list.

Runtime:  constant  logarithmic  linear  quadratic  exponential

Let  $n$  be the length of the input list. The for loop will run  $n$  times. Inside the for loop, **missing number?** will always be False. As explained in part D, when the first input to **and** is False, Snap! will short-circuit and evaluate the entire expression as False without checking the second input. Therefore, **contains** will never be run. As a result, each loop iteration takes constant time, and the overall runtime is linear.

C.



Buggy Input: correct

Runtime:  constant  logarithmic  linear  quadratic  exponential

The first for loop does not depend on the input size and is therefore constant. The second for loop iterates through every *item* in *list* and performs constant operations inside. The second for loop is therefore linear. **combine** will run a constant function over a list of length 10, and will therefore take constant time.

We have a constant runtime, followed by a linear runtime, followed by a constant runtime. Therefore, the overall runtime of the block is linear.

D.



As a review from discussion, in Snap!, the and and or blocks do something called “short circuiting,” meaning that if they already know the output of the expression from the first input, they won’t bother checking the rest of their inputs. For example:



Buggy Input: [correct](#)

Runtime:  constant  logarithmic  linear  quadratic  exponential

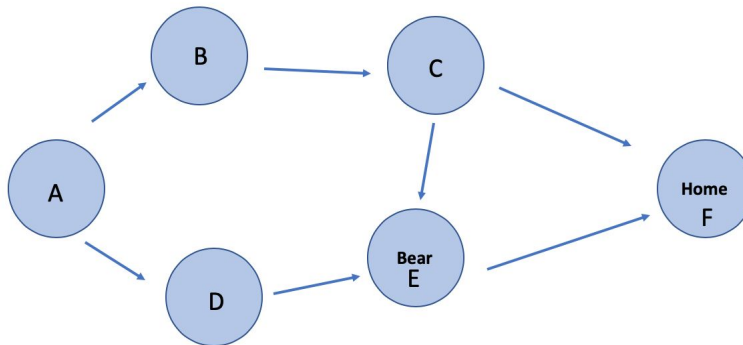
There will be 11 recursive calls: **full list recursive** (list) (10), **full list recursive** (list) (9), ..., **full list recursive** (list) (0). Except for  $n = 0$ , each of these recursive calls may run **contains**, which takes linear time. Therefore, **contains** will be run at most 10 times. Therefore, this block will run a linear function at most 10 times, and the overall runtime is therefore linear.

**Question 10: Go Bears! (4 points)**


You are lost in the woods and want to make sure you get **Home** safely. However, you want to be careful to avoid the bears! A location is safe if there is **at least one** path from the location to **Home**, without any bears along the path. A path starts at a location and will always lead to **Home** (i.e. there are no dead-ends). You cannot go backwards along a path. There will never be a bear at **Home**, but there may be bears in the other locations.

Write a block called **safe path?** which takes in a location and tells you if there is a path from the location to **Home** without any bears.

Below is an example forest and some important blocks. Your **safe path?** block should work for any forest, not just this example.



Block	Description	Calls using the example forest
	Takes in a location and reports if there is a path from the location without any bears.  (for you to implement)	
	Takes in a location and reports if it is home.  (for you to use in your solution)	
	Take in a location and reports if there is a bear.  (for you to use in your solution)	
	Takes in a location and reports a list of all the locations you can go to next.  (for you to use in your solution)	

		
--	--	---

safe path? (location):

```

if is home? location
  report true
else
  if has bear? location
    report false
  else
    report
      combine map safe path? over neighbor locations location using
        or
  
```

**Question 11: Another Homework Assignment? (3 points)**

What is the code below an example of?

```

when clicked
  set assignment to Homework
  set section to Discussion
  broadcast begin class!
  
```

```

when I receive begin class!
  wait until assignment = Homework and section = Lab
  complete assignment Homework
  set assignment to Worksheet
  set section to Discussion
  
```

```

when I receive begin class!
  wait until assignment = Worksheet and section = Discussion
  complete assignment Worksheet
  set assignment to Homework
  set section to Lab
  
```

- race condition     
  deadlock     
  livelock     
  none of the above