Fill in the correct circles & squares completely…like this: ● (select ONE) ■ (select ALL that apply)
What’s that Smell? Oh, it’s Potpourri! (2 pts each for 1-6, low score dropped)

**Question 1:** What is $101_2 + 213_2$? (select ONE)

\[101_2 = 1^2 + 0^2 + 1^0 = 110_2 \quad \text{and} \quad 213_2 = 2^3 + 1^3 + 3^0 = 610_2 \]

\[101_2 + 213_2 = 513_2 \]

\[\boxed{\begin{array}{ccccccccccccc}
122_{16} & 8_{16} & 9_{16} & A_{16} & B_{16} & C_{16} & D_{16} & E_{16} & F_{16} & 10_{16} & 11_{16} & 12_{16} & 13_{16}
\end{array}}\]

**Question 2:** Which of the following was in the Human-Computer Interaction lecture? (select ONE)

- Solid planning by good design teams always beats “enlightened trial and error”
- In fact, the quote was exactly the opposite. “Enlightened trial and error out-performs the planning of flawless intellect.” –David Kelly
- The digital pen was invented after 2000.
- None of these

**Question 3:** Which of the following was in the Saving the World with Computing lecture? (select ONE)

- After using higher resolution models, some said earlier “doomsday” global warming trends were too extreme.
- Dan Martin, a computational scientist at Lawrence Berkeley National Lab, argued that his and his colleagues’ work showed that ice cliffs might simply be a product of running a computer model of ice physics at a too-low resolution.
- One of the projects at the lab was to use computers to study 

**Question 4:** Which of the following was in the Limits of Computing lecture? (select ONE)

- The “knapsack problem” is all about geometry – how to fit the most different-shaped boxes in a knapsack.
- The “knapsack problem” is about maximizing profit by a thief who has an option of many boxes with various weights and values; the key is the knapsack has a weight limit, not a size/geometry limit.
- Alan Turing proved all problems are decidable; people used to believe it wasn’t.
- They recently proved that $P=NP$, sharing this amazing discovery was the main point of the lecture.
**Question 5:** Based on the *Artificial Intelligence (AI)* lecture, which one is easiest for a computer? (select ONE)

- Detecting a stop sign from an image captured by a camera.
  
  **Nope,** this is done through a complex process, typically by a neural network trained to recognize stop signs.

- Translating an essay from one language to another.
  
  **Nope,** this requires understanding both of the languages and common mappings for any idiosyncratic phrases.

- Having a conversation in English with a person.
  
  **Nope,** this requires a remarkable amount of understanding of the English language (done through Natural Language Processing algorithms), determining what response to say and how to say it. Try talking to any state-of-the-art system (Alexa, Siri, Google) and you’ll see how far we have to go here.

  - Winning a game of Chess.
    
    Sure, this is just search – we showed how to write a game solver in class with a few lines of Python code for the 10-to-0-by-1-and-2 game.

  - Opening a door.
    
    **Nope,** if you consider all the possible handles, grasp requirements, and forces at play.

**Question 6:** Which one of these WAS NOT one of the participants of the *Alumni Panel*? (select ONE)

- An indie game developer who advised to take CS classes outside of CS dept.

- A software engineer at YouTube who advised you not to avoid challenge.

- A backend engineer at Lisnr who advised you to delete your social media.

- A data visualization consultant who advised taking personal finance courses.

  - None of these, they were all on the panel!

  This question was meant to be a “gimme” if you attended the Alumni Panel lecture.
(The `Bool` block on the right is used for Questions 7 & 8; 2,3 pts)

**Question 7:** Fill in the blanks so the predicate is the same as the original `Bool` block. (select ONE from each)

When does the original block return `true`? If B is `true` and A is equal to it (also `true`) – so A=`true`, B=`true`. What if B is `false`? Then B is not equal to `true`, so that rightmost `=` is `false` but what if A is equal to it? Then `bool` would return `true`. So we have A=`false`, B=`false` also. Therefore it turns out that `bool` simply returns whether A = B!

How do we make A=B in an if/then/else template? Let’s see, if A is `true`, the “then” part of the if would be followed. We want to return `true` if B is `true` also, so the first slot is simply B.

If A is `false`, we want to return `true` when B is also `false` (same as B) so therefore we return “not B”.

**Question 8:** Fill in the blanks so the predicate is the same as the original `Bool` block. (select ONE from each)

When we were describing the values that cause the original block to return true, we said: “A=`true`, B=`true` but also A=`false`, B=`false`”. If we say it with just Boolean phrases, we would say: “(A=`true` AND B=`true`) OR (A=`false` AND B=`false`)”, and one way to say “X = `false`”, is just write “not X”. Thus, we get the final answer of “(A=`true` AND B=`true`) OR (not A AND not B)”
Question 9: *Roll up for the magical mystery block, step right this way...* (4 pts)

What does **mystery** (below) report, if B is a counting number (i.e., 1, 2, 3, …)? (select ONE)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

One way to solve this is to see what happens for very small B.
If B = 1 (the loop happens once), then A gets set to A+A → 2A.
If B = 2 (the loop happens twice), then the first time A gets set to A+A → 2A, but then A gets doubled again to 4A after the second loop.
From this, it’s apparent that A will get doubled with every loop, so we can see that on the 3rd loop it’s 8A, 4th loop it’s 16A, etc. Therefore, the return value is simply $2^B\times A$, and because × is commutative, it’s also A×$2^B$.

Question 10: *It’s a mad, mad, mad, mad world...* (4 pts)

Two people (A and B) were told to find a hidden paper bag, which initially has $10 in it. Each one was told that when they found it, they would look at the contents, then wait a random amount of time within another week before returning to the spot, burning the paper bag (and whatever was in it) to ashes, and replacing it with an identical paper bag and contents to the one they initially saw, except:

- **Person A**: The $ amount has been increased by 2. E.g., if the paper bag initially had $200, they would swap it with a paper bag with $202 inside. If the paper bag had $200 and €1000, they would swap it with a paper bag that had $202 and €1000 inside.

- **Person B**: They would convert half of the $ to € (assume the exchange rate is the same, $1 = €1). E.g., if the paper bag initially had $200, they would swap it with a paper bag that had $100 and €100.

What are possible contents of the paper bag at the end, after both had done their swap? (select ALL that apply by filling in the box completely) for a value of dollars ($) in the column and euros (€) in the row.)

This is a concurrency (race conditions) problem. Let’s see what possible cases come about:
1. **A** finds the bag, does the "+$2" swap (making it $12), then **B** finds that bag and does the "half $→ €" swap (making it $6,€6).
2. **B** finds the bag, does the "half $→ €" swap (making it $5,€5), then **A** finds that bag and does the "+$2" swap (making it $7,€5).
3. **A** and **B** both find the original bag, each prepares their separate replacement bags (**A** has $12, while **B** has it $5,€5).
   a. **A** puts their $12 bag first, leaves, then **B** burns that bag and replaces it with their $5,€5 bag.
   b. **B** puts their $5,€5 bag first, leaves, then **A** burns that bag and replaces it with their $12 bag.
**Question 11: Did we assign all student ID numbers correctly?** (9 pts)

There’s been a problem with the student ID system and campus is not sure everyone has a unique number! Here are 3 algorithms to find out whether all student IDs (SIDs) are unique or not. For all problems, assume the number of students (N) is a power of 2 and really big. (How big?) Really big. Also, “clock time” is the actual elapsed time if you used a clock or stopwatch to time the running of the algorithm.

**Algorithm I – “Musical Chairs” algorithm**
1. The music starts, everyone mills around the room and when the music stops, they find a random partner and compare SIDs.
2. If anyone matches their SIDs with their partner, they yell “NOT UNIQUE!!” and stop.
3. If not, they wait for the music to start again, and repeat 1-3.
4. This continues for exactly N rounds, and if nobody has yelled “NOT UNIQUE”, they all yell “UNIQUE”.

**Algorithm II – “Recursive Halving” algorithm**
1. All people in the room pair up and compare SIDs with their partner.
2. If anyone matches their SIDs with their partner, they yell “NOT UNIQUE!!” and stop.
3. Otherwise, there will be a smaller and a larger SID. All the smaller SID students hold hands and go find a new room together, and all the larger SID students hold hands and go find a new room.
4. Step 1-3 continues until there are N rooms occupied with one person in each room, at that point they all yell “UNIQUE” and stop.

**Algorithm III – “Down the Line” algorithm**
1. Everyone lines up, and the first person is designated as the “unique tester”.
2. The “unique tester” person goes down the line, comparing their SID to that of each new person, 1-on-1.
3. If ever the SIDs match, they yell “NOT UNIQUE!!” and stop.
4. If they get to the end and haven’t matched, they go to the back of the line and the new first person is designated as the “unique tester”, and they repeat steps 2-4.
5. If that initial first person ever ends up being first again, the yell “UNIQUE” and stop.

---

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Yes</th>
<th>No</th>
<th>Reason</th>
<th>$0</th>
<th>$5</th>
<th>$6</th>
<th>$7</th>
<th>$10</th>
<th>$12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Musical Chairs</td>
<td>0</td>
<td>1</td>
<td>There is nothing in the algorithm that mandates partner swapping, so each round could have the same partners meeting, which doesn’t allow us to “learn” anything new. Therefore, two people could have the same SID but never meet to confirm it, so if it says “UNIQUE”, we really can’t trust it.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Recursive Halving</td>
<td>0</td>
<td>1</td>
<td>The algorithm is peculiar, since once the initial pairings happen (say into the LEFT group and the RIGHT group), every LEFT person only meets one person in the RIGHT group. So it’s possible that there are many duplicate SIDs, every other person in the RIGHT group could be the same as a single SID in the LEFT group and we’d never catch it. Therefore, if it says “UNIQUE”, we really can’t trust it.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Down the Line</td>
<td>1</td>
<td>0</td>
<td>This works! That is, this guarantees that every person will eventually check their SID with every other person (twice, actually, once with each partner being the “unique tester”, so it’s a little inefficient by a factor of 2), so that by the end, if it says “UNIQUE”, we can certainly trust it.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Constant</th>
<th>Logarithmic</th>
<th>Linear</th>
<th>Quadratic</th>
<th>Cubic</th>
<th>Exponential</th>
</tr>
</thead>
<tbody>
<tr>
<td>Musical Chairs</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Recursive Halving</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Down the Line</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

Reasoning:
- Musical Chairs has $N$ rounds where there are $N/2$ simultaneous comparisons each round. Therefore, that’s $N^2/2$ comparisons, which is Quadratic (since constants don’t matter).
- Recursive Halving has log $N$ rounds where there are $N/2$ simultaneous comparisons each round. Therefore, that’s $N/2 \cdot \log N$ comparisons. We know that’s bigger than Linear, and we know log $N$ is smaller than $N$ (it’s own category, after all), so $N/2 \log N$ is smaller than Quadratic. So its complexity of “$N \log N$” (again, we drop constants) falls between categories we’re told to pick the bigger one, therefore it’s also Quadratic.
- Down the Line has $N$ people each doing exactly $N-1$ comparisons, so it’s $N*(N-1) = N^2 - N$ which is Quadratic, (since we drop smaller terms).

So they’re all Quadratic, cool!

c) If the SID comparisons between different pairs of people could happen at the same time, how much clock time (NOT running time) would each algorithm take in the WORST case? (select ONE per row)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Constant</th>
<th>Logarithmic</th>
<th>Linear</th>
<th>Quadratic</th>
<th>Cubic</th>
<th>Exponential</th>
</tr>
</thead>
<tbody>
<tr>
<td>Musical Chairs</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Recursive Halving</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Down the Line</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

Reasoning:
- Musical Chairs has $N$ rounds where there are $N/2$ simultaneous comparisons each round. These $N$ rounds make for Linear clock time.
- Recursive Halving has log $N$ rounds where there are $N/2$ simultaneous comparisons each round. These log $N$ rounds make for Logarithmic clock time.
- Down the Line has $N$ people each doing exactly $N-1$ comparisons, so it’s $N*(N-1) = N^2 - N$ which is a Quadratic, number of comparisons (since we drop smaller terms), and only one comparison happens at a time, so it’s Quadratic clock time.
Consider the following code below. We’re now to going to zoom in on pixels affected by calls to \texttt{Fun}; the stage is always cleared before the calls below, the sprite always starts in the center facing up, and the pen is in the \textit{center} of the sprite.

Your job is to shade in (completely!) \textit{all} the pixels that will be colored in after calls to \texttt{Fun} with \texttt{n} set to 1, 2 & 3. \textit{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.
A rook is a piece in the game of chess that can move any number of squares vertically or horizontally. We put a rook somewhere on integer coordinates in the first quadrant \((0 \leq x \leq \infty, 0 \leq y \leq \infty)\) and put a spell on it so that it can only move toward the origin (i.e., either down or left). Author to calculate how many different paths there are home given an \((x, y)\) starting point. E.g., the rook at \((3, 2)\) could get back to \((0, 0)\) any one of 10 ways, and the number of paths for each starting square in \((0 \leq x \leq 3, 0 \leq y \leq 2)\) is below.

Author your solution below AND then completely fill in the boxes above to correspond to the solution you wrote below.

paths from \((x, y)\):
- if \(x = 0\) or \(y = 0\)
  - report 1
- else
  - report paths from\((x-1,y)\) + paths from\((x, y-1)\)
**Question 14: Hey, let's call the function \( f \) on our \textbf{DATA}! Uh oh, on second thought...** (7 pts, 1 each)

Ever wanted an "undo" in life? Well, finally we have it! Given a particular function \( f \) whose domain is anything, we have invented a miracle function \( \text{undo } f \) that can always undo whatever \( f \) did. I.e., this is always true for any \( \text{input} \):

\[
\text{undo } f \: \text{input} = \text{input}
\]

Given the following function:

\[
\text{identity } \text{input}
\]

...which of the following are ALWAYS equivalent to the list \( \text{DATA} \)?

We did the first one for you. (select ALL that apply)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Yes, always equivalent to ( \text{DATA} )</th>
<th>Not always equivalent to ( \text{DATA} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{undo } ) ( f ) ( \text{DATA} )</td>
<td>✗</td>
<td>☑</td>
</tr>
<tr>
<td>Imagine if ( f ) and ( \text{undo } f ) were the functions above; it wouldn't work the other way...</td>
<td>☑</td>
<td>✗</td>
</tr>
<tr>
<td>( f ) ( \text{undo } ) ( f ) ( \text{DATA} )</td>
<td>☑</td>
<td>✗</td>
</tr>
<tr>
<td>Doesn't work for the “Thunk” pair above since ( f ) doesn't return a list that can be mapped over...</td>
<td>☑</td>
<td>✗</td>
</tr>
<tr>
<td>( \text{undo } f ) ( \text{map } ) ( \text{identity } ) ( \text{over } f ) ( \text{DATA} )</td>
<td>☑</td>
<td>✗</td>
</tr>
<tr>
<td>Doesn't work for the “Thunk” pair above since ( \text{undo } f ) doesn't work on lists, but on functions...</td>
<td>☑</td>
<td>✗</td>
</tr>
<tr>
<td>Since we know ( \text{undo } (f(\text{input})) = \text{input} ), this doesn't do anything to all the elements of ( \text{data} ), so it's just ( \text{data} ).</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>( \text{map } ) ( \text{undo } f ) ( \text{over } f ) ( \text{DATA} )</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>Doesn't work for the “Thunk” pair above since ( f ) doesn't return a list that can be mapped over...</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>Each element gets ( f(\text{input}) ) then ( \text{undo } (f(\text{input})) = \text{input} ), so this doesn't do anything!</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>Doesn't work for the “Thunk” pair above since ( f ) doesn't return a list that can be mapped over...</td>
<td>☐</td>
<td>☑</td>
</tr>
</tbody>
</table>
Question 15: Joshua: “How about a nice game of checkers?” (4 pts)

You are trying to write code to generate a “N x N” checkerboard in which the bottom-left-most “origin” pixel is always turned on (black). E.g., calling it with N on values 1 through 5 should result in the following five images.

<table>
<thead>
<tr>
<th>Checker(1)</th>
<th>Checker(2)</th>
<th>Checker(3)</th>
<th>Checker(4)</th>
<th>Checker(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Checker(1)" /></td>
<td><img src="image2" alt="Checker(2)" /></td>
<td><img src="image3" alt="Checker(3)" /></td>
<td><img src="image4" alt="Checker(4)" /></td>
<td><img src="image5" alt="Checker(5)" /></td>
</tr>
</tbody>
</table>

What Boolean expression should go in the if to accomplish this? (Select ONE)

- $x + y \mod 2 = 0$
- $x + y \mod 2 = 1$
- $x \times y \mod 2 = 0$
- $x \times y \mod 2 = 1$
- $x \mod 2 = 0$ and $y \mod 2 = 0$
- $x \mod 2 = 0$ and $y \mod 2 = 1$
- $x \mod 2 = 1$ and $y \mod 2 = 0$
- $x \mod 2 = 1$ and $y \mod 2 = 1$
- $x \mod 2 = 0$ or $y \mod 2 = 0$
- $x \mod 2 = 0$ or $y \mod 2 = 1$
- $x \mod 2 = 1$ or $y \mod 2 = 0$
- $x \mod 2 = 1$ or $y \mod 2 = 1$
We recreated an interpreter script. For each, indicate what the right answer should be.

```python
>>> 1 + '2'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
○ 3 ○ '3' ○ 12 ○ '12' ● Error ○ None of these
```

```python
>>> [1] + ['2']
[1, '2']
○ [3] ○ ['3'] ○ [12] ○ ['12'] ○ [1,2] ○ ['1','2'] ● Error ○ None of these
```

```python
>>> 'CAL'[1:2]
'O'  ○ 'CA' ○ 'AL' ○ 'C' ○ 'A' ○ Error ○ None of these
```

```python
>>> A = [3]
>>> B = A
>>> A.append(2)
>>> B
○ [32] ○ [3] ○ [3,2] ○ Error ○ None of these
```

We'd like to write a `backwards` function that would have the following behavior:

```python
>>> school = 'cal'
>>> backwards(school)
'lac'
>>> def backwards(school):
...     return 'lac'
...     return 'lac'

Is it possible to write `backwards`?
● yes ○ no
```

```python
>>> [n+1 for n in range(2,4) if n != 3]
[3, 5]
○ [3,5] ○ [4,5] ○ [3] ○ [4] ○ Error ○ None of these
```

```python
>>> ['n+1' for n in range(2,4) if n != 3]
['n+1']
○ ['3', '5'] ○ ['4', '5'] ○ ['3'] ○ ['4'] ○ Error ○ None of these
```

```python
>>> ['HA','AB'] if n != 'A']
['HA','AB'] ○ ['HA','AB'] ○ ['HA','AB'] ○ ['HAHA','ABAB'] ○ Error ○ None of these
```

```python
>>> def increm(n):
...     return n+1
>>> def double(n):
...     return n+n
>>> def square(n):
...     return n*n
>>> D = {1: increm, 2: double, 3: square}
>>> [D[i](i) for i in [3,1,2]]
[9, 2, 4]
```