How many memory slots are needed for cards for Simple Sort for 7 cards? (don’t forget swapping!)

A. 7  
When counting, ignore things like markers and dividers

B. 8  
Simple Sort Reminder:
1. Place the unsorted cards in the top row
2. Repeat steps 3 through 6 until no unsorted card remains
3. Initially mark the first (leftmost) card
4. Working right from the second card, if the current card is smaller than the marked card, move the marker to the current card
5. Move the marked card to the sorted hand
6. Put a “Max” (upside down) card in the empty unsorted slot
7. Stop

C. 14

D. 15

E. Other
How many memory slots are needed for cards for Selection Sort for 7 cards (don’t forget swapping!)

A. 7
B. 8
C. 14
D. 15
E. Other

When counting, ignore things like markers and dividers

B. 8

Selection sort reminder:

1. Deal the unsorted cards
2. Put a divider at the left of the unsorted cards
3. Repeat steps 4 through 7 until one unsorted card remains
   4. Initially mark the first (leftmost) card
   5. For each card to the right of the second card:
      If the card is smaller than the marked card,
      move the marker to the current card
   6. Swap the marked card with the first unsorted card
      (just to the right of the divider)
   7. Advance the divider to the right one card
8. Stop
How many memory slots are needed for cards for Insertion Sort for 7 cards (don’t forget swapping!)

A. 7
B. 8
C. 14
D. 15
E. Other

When counting, ignore things like markers and dividers

Insertion sort reminder

1. Deal the unsorted cards
2. Put a divider after the first card
3. Repeat steps 4 through 6 until no card is to the right of the divider
4. Select the first card to the right of the divider
5. Swap this card to the left until it arrives at the correct sorted position
6. Advance the divider to the right one card
7. Stop
Simple sort performed roughly twice as badly as either Selection Sort or Insertion Sort.
How about time?

- There are lots of things that we can look at that take time
  - Copies
  - Swaps
  - Comparisons
- Let’s concentrate on comparisons
On average, Selection Sort and Insertion sort use roughly half as many comparisons as Simple Sort.
Quick Sort and Merge Sort are even faster!
Their (purple) plots lie close to the x axis.
That’s a big difference!

- On average, Selection Sort and Insertion sort have roughly half as many comparisons and use roughly half as much space as Simple Sort.
- Think about what that looks like when you have a lot of cards.

![Graph showing the comparison between Simple Sort, Selection Sort, and Insertion Sort.](http://www.ugrad.cs.ubc.ca/~cs100)
Learning Goals

• CT Building Block: Students will be able to define the difference between hardware, applications, and the operating system.

• CT Building Block: Students will be able to give examples of a problem for which there are different algorithms, give examples of cases where one algorithm works better than the other, and reason about which algorithm is likely to work better overall.

• CT Building Block: Students will be able to evaluate and compare algorithms in terms of its efficiency (time and space requirements).
From algorithms to code:
How do programs work?
Learning Goals

• [CT Building Block] Explain what a variable is in computer programming.

• [CT Building Block] Be able to trace through code using sequences of instructions, variables, loops, and conditional statements in short programs specified in a visual programming language such as Snap, or in other clearly expressed processes (which may or may not be computer related)

Note: Look carefully: it says be able to trace code, not write code. In order to help you do this, you will write a small amount of code in lab. You will not, however, be asked to write code on exam.
This is *not* a programming course.

- That being said, we do want you to understand a little about how programs work.
- We’ll cover a small amount of basic concepts in class and you’ll work on a visual language in lab – Snap!

(http://snap.berkeley.edu/snapsource/snap.html)
At the highest level

- Programs are a way of encoding algorithms in a precise enough way for computers to understand the instructions
Programmers use a high level language like Snap, Scratch, Python, C++, Java, Racket. These languages may look very different.
Underneath, they all do the same thing

Another program called a **compiler** or an **interpreter** takes a high level language and translates it into something that looks about the same, regardless of which high level language is used.

```plaintext
.code
demomain:
    REPEAT 20
    switch rv(nrandon, 9) ; generate a number between 0 and 8
    mov ecx, 7
    case 0
        print "case 0"
    case ecx ; in contrast to most other programming languages,
        print "case 7" ; the Masm32 switch allows "variable cases"
    case 1 .. 3
        .if eax==1
            print "case 1"
        .elseif eax==2
            print "case 2"
        .else
            print "cases 1 to 3: other"
        .endif
    ...
```
Another program called an assembler translates this into machine language.

Low-level machine languages are at the level of the computer’s hardware.

```
load 3, 7000  ← Load data #7000 from cache into register 3
load 4, 7001  ← Load data #7004 from cache into register 4
add 3, 4, 3  ← Add registers 3 and 4 and put in register 3
store 3, 7002 ← Store the data from register 3 in slot #7002
jmp 20  ← Go ahead in the instructions by 20
```

A simple diagram of a half adder is shown, with inputs A and B, and outputs Sum and Carry.
But many of the concepts are the same

• We’ll cover just enough so that you have an idea of how programs work
• If you want to learn more, you can take a class like CPSC 103 or CPSC 110
• For now, we’ll just stick with some examples like…
Blackjack!

- We’re going to model what needs to be done to play blackjack.
- To do that, we need to make sure that everyone understands how blackjack works.
- In a minute, we’ll play a couple of rounds of blackjack in groups to make sure that everyone knows the rules.
  - There are more complex rules – we’re keeping it very simple.
Blackjack as we know it (simplified from the real world)

Ground rules:
• In blackjack, there is a dealer and one player.
• The player places a bet to get into the game.
• Face cards are worth 10. Aces can be worth 1 or 11.
• The goal is to get as close to 21 points as possible.
• Anything over 21 points loses.
• The dealer wins all ties.

Play:
• The dealer and the player each start with 2 cards; the dealer has 1 face down.
• The player is allowed to hit (ask for more cards) as many times as long as the total is under 22
• The player then stands
• The dealer will then take as many hits as necessary to get to more than 16 points
Great! Now we’re ready to design a small Blackjack application

- Let’s take a look at one that some TAs and I put together.
- Note: we’re not going to look at ALL the code, just enough to get the idea of some key concepts.
Most things that we need to keep track of, we track with *variables* (named quantities).

Variable: where on the x axis is the mouse?  
Variable: where on the y axis is the mouse?
Variables have values

Value of mouse x

Value of mouse y
When we sorted cards, each slot was a variable

Unsorted
Simple sort
Sorted

So were the markers and dividers
We can use variables in our code

- Some variables are built in (e.g., “answer” is the answer to a question in Snap)
- You can make your own variables

Variables are in orange. Things that are black writing on white are constants – actually that value
Order!

- Programs will execute exactly in the order that’s given
- So if we assign values to variables, they’ll set one value after another after another
Snap! Makes it easy to show non-text things

For example we might need pictures of cards and a dealer

In your group, list non-text things that we might need to show
We call our screen our “stage”. “Things” we add are called sprites.

A sprite is an object you can move on a larger scene.