COMP 110

Big-O Notation

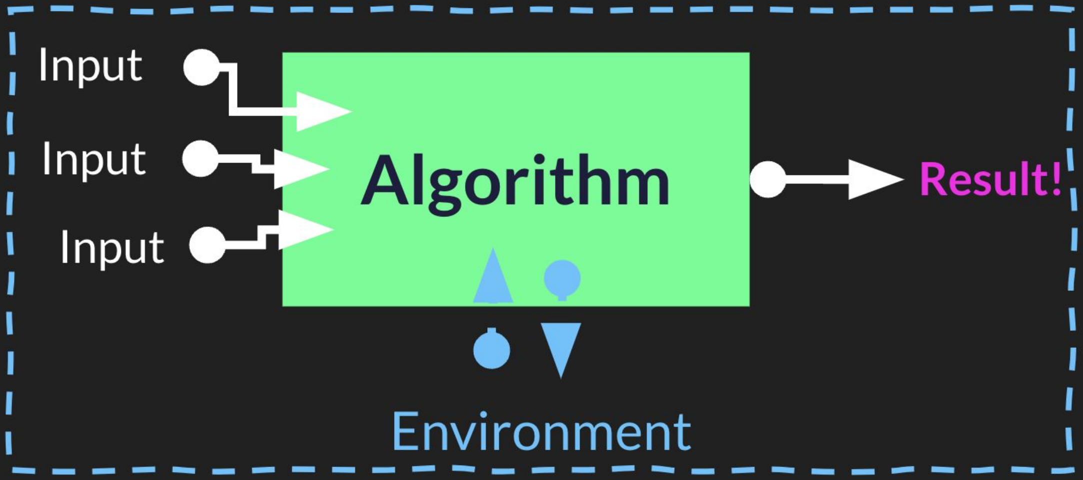
Recall: Algorithms

Input is data given to an algorithm

An **algorithm** is a series of steps

An algorithm **returns** some **result**

An algorithm *may* be influenced by its **environment** and it *may* produce side-effects which influence its environment.



What is an algorithm?

- A set of steps to solve a general problem
- Finite
- Can handle a problem of arbitrary size

How do we measure how “good” an algorithm is?

- Is it correct? How precise is it?
- How easy is it to implement?
- How long does it take to implement?
- How much computer memory does it take?

Why do we care about computation speed?

- Security: Cryptography works because encrypted information takes *too long* to decipher!
- User Experience: Users don’t want to work with a slow application!
- Big Data: We want to be able to feed as much data as possible into our systems, but we need a way to *efficiently* do that!

Running time: how long does an algorithm take to run?

- Empirical analysis: write the code and test how long it takes to run!
  - Weaknesses:
    - You have to write the code for the whole algorithm and run it to see how long it will take
    - Different computers with different specs will have different runtimes
- Rather than using empirical analysis, computer scientists commonly consider the **number of operations (steps)** an algorithm requires
  - 1 operation == 1 step

Measurements We Use

Ω Best case (lower bound):

- Minimum number of operations (running time) required for the algorithm to execute

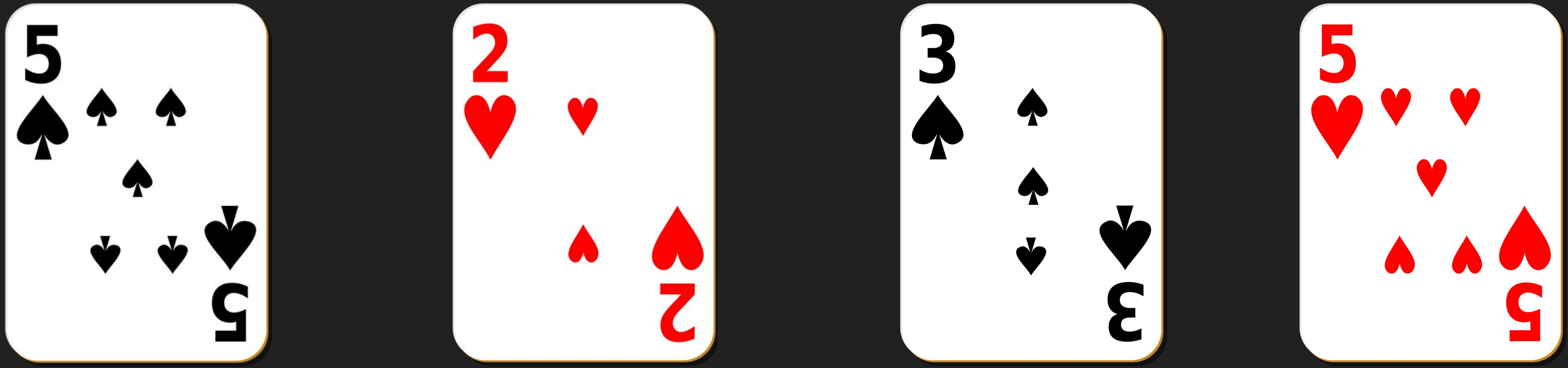
Θ Average case:

- Average running time among several different inputs

O Worst case (upper bound) ✨:

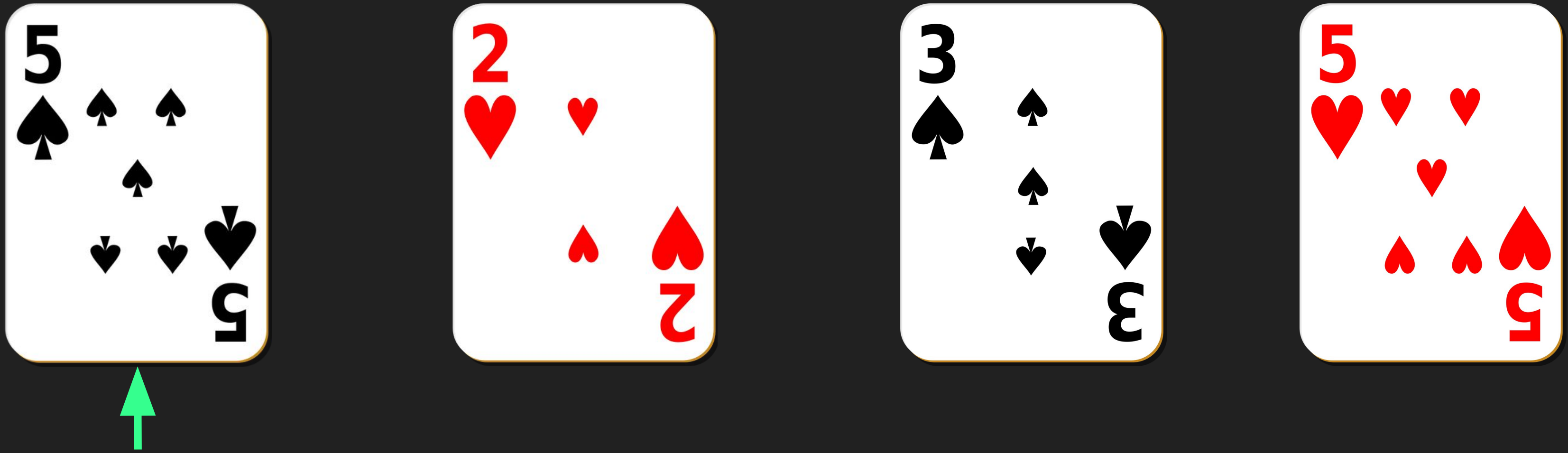
- The maximum running time given an input
  - How does the number of operations grow as an input grows?
- Important to understand how our algorithm will perform in the *worst* case
  - Prepare for the worst case. If an input ends up requiring fewer operations, great!!

Returning to Finding the Lowest Card in a Deck

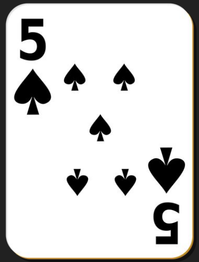


- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

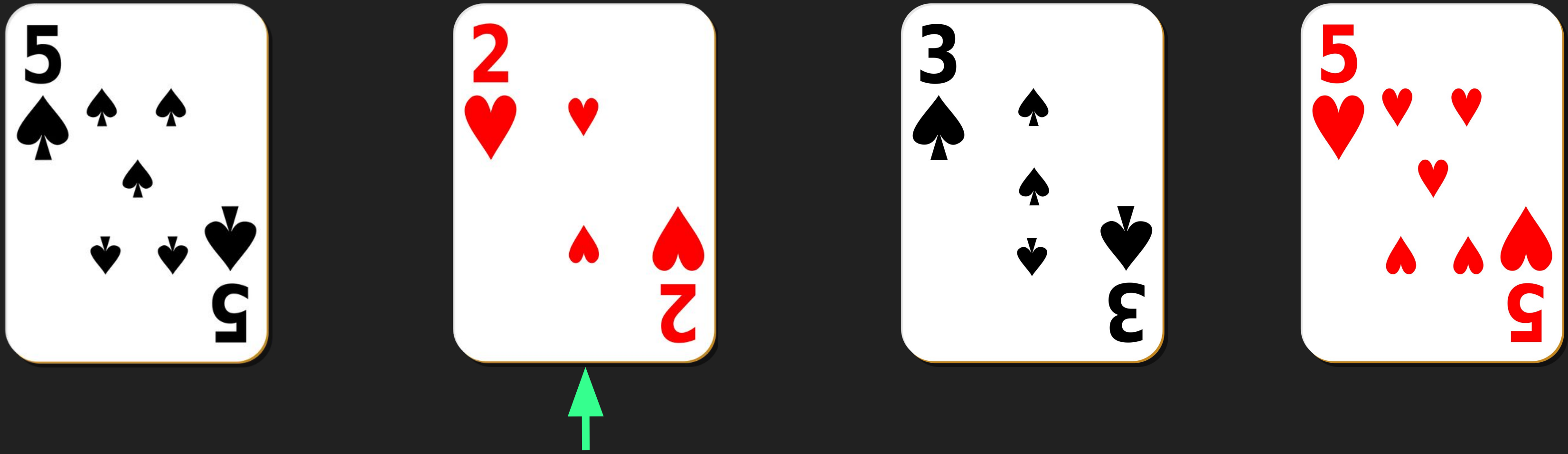
Finding the Lowest Card



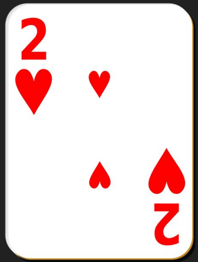
Low card:



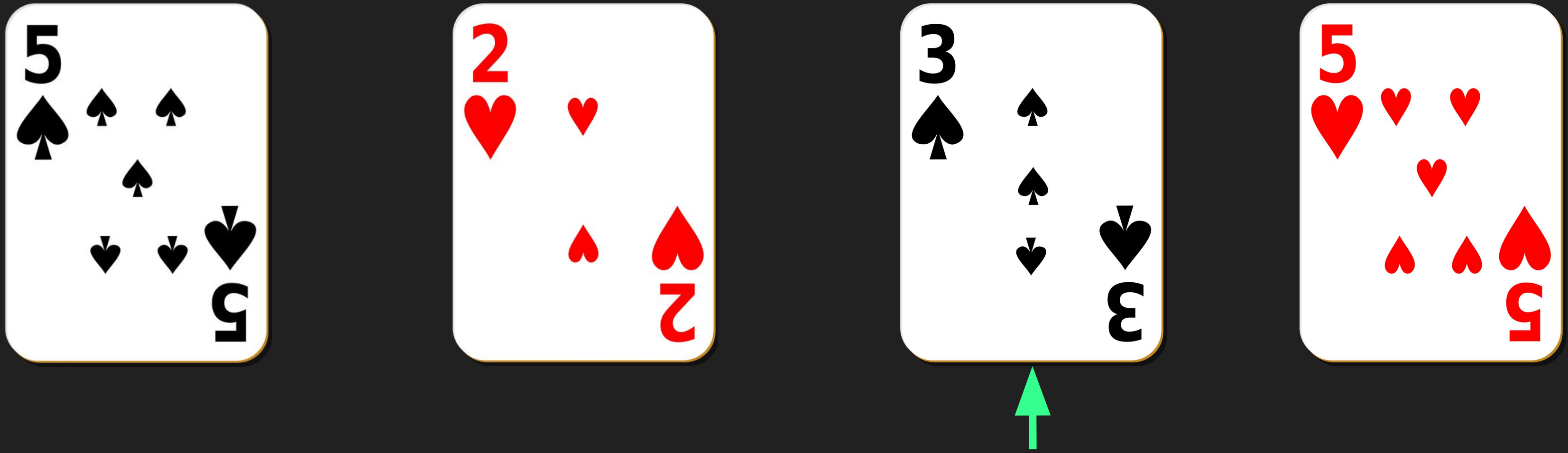
Finding the Lowest Card



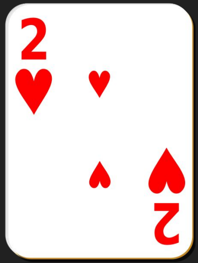
Low card:



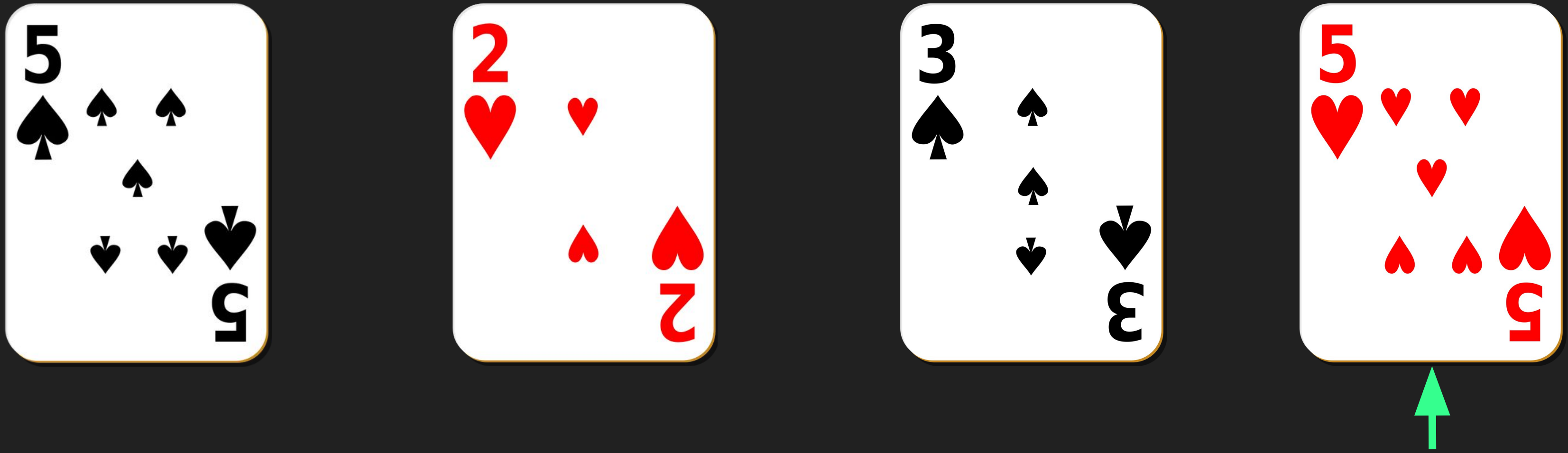
Finding the Lowest Card



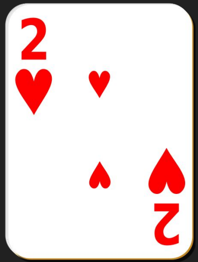
Low card:



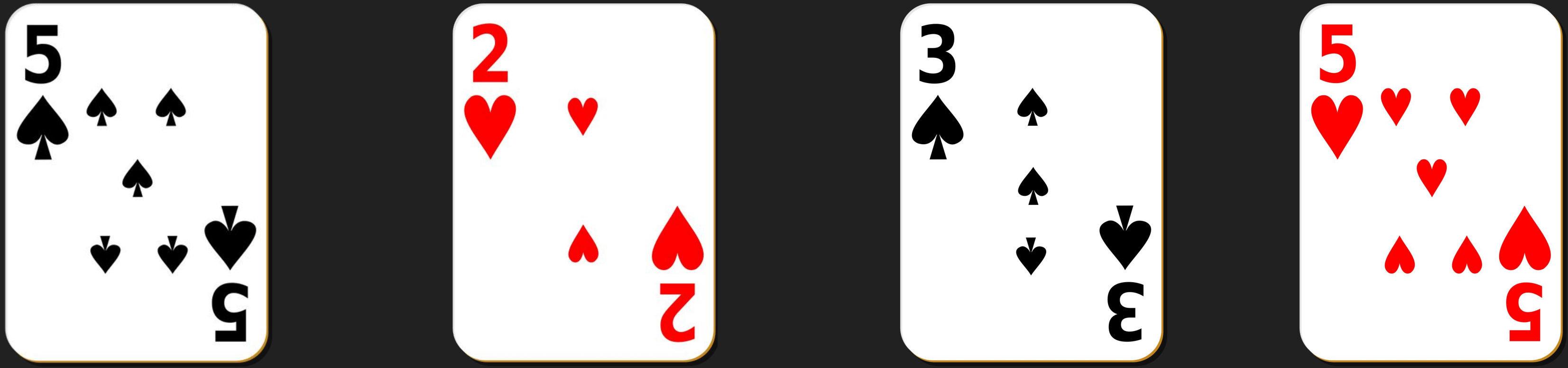
Finding the Lowest Card



Low card:

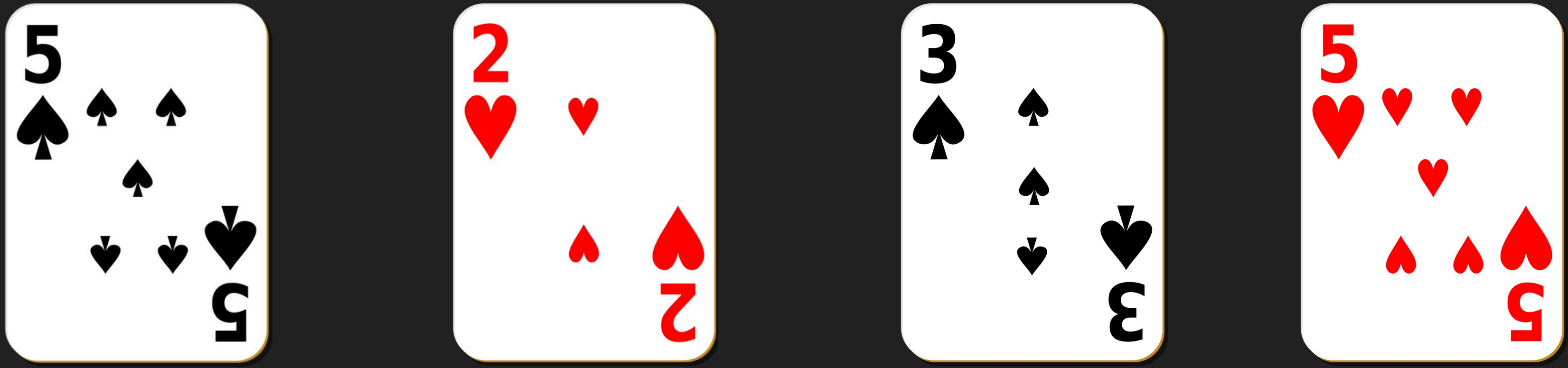


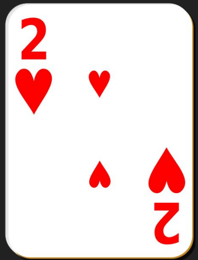
Finding the Lowest Card



Low card:

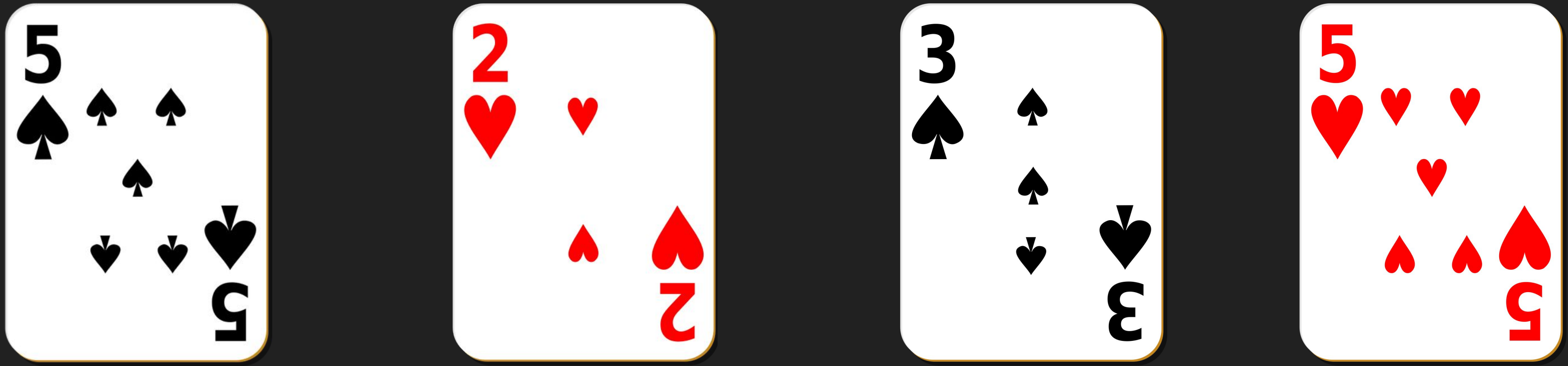
Finding the Lowest Card



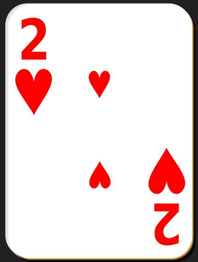
Low card: 

4 actions for
input of 4 cards.

Finding the Lowest Card



Low card:



4 actions for
input of 4 cards.
→
n actions for
input of size n.

Finding the Lowest card

- In this approach, we always have to check every card in the deck, so our runtime will always be approximately *n* where *n* is the size of the deck.

$$\text{Finding the minimum} \in O(n)$$

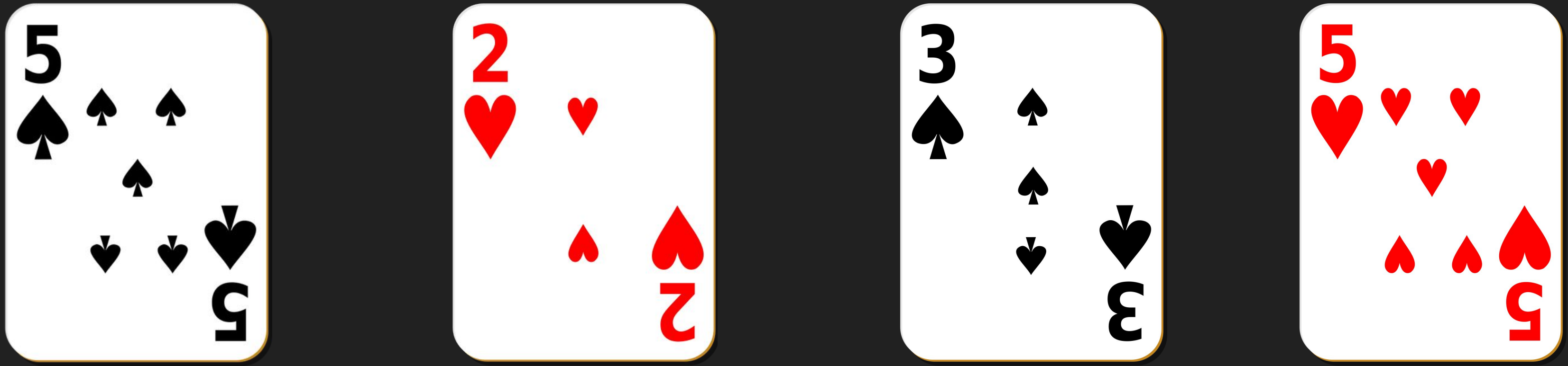
$$\text{Finding the minimum} \in \Omega(n)$$

$$\text{Finding the minimum} \in \Theta(n)$$

Speed vs. Memory

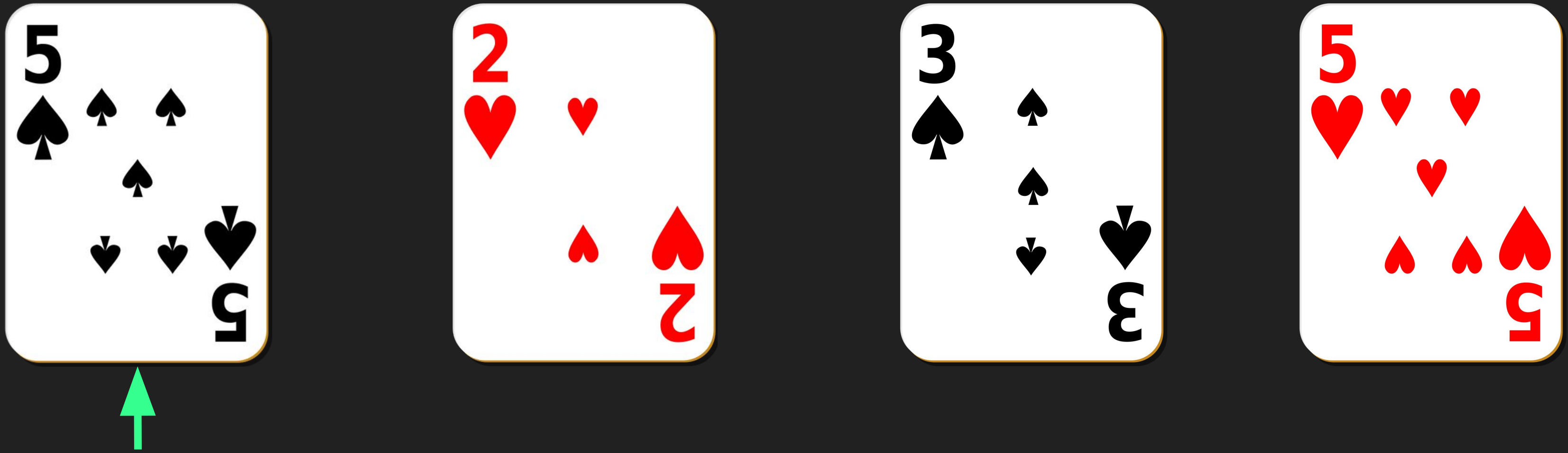
- Sometimes you can make a tradeoff between speed and memory.
- E.g. storing a value rather than computing it repeatedly.

New Example: Finding a specific card.

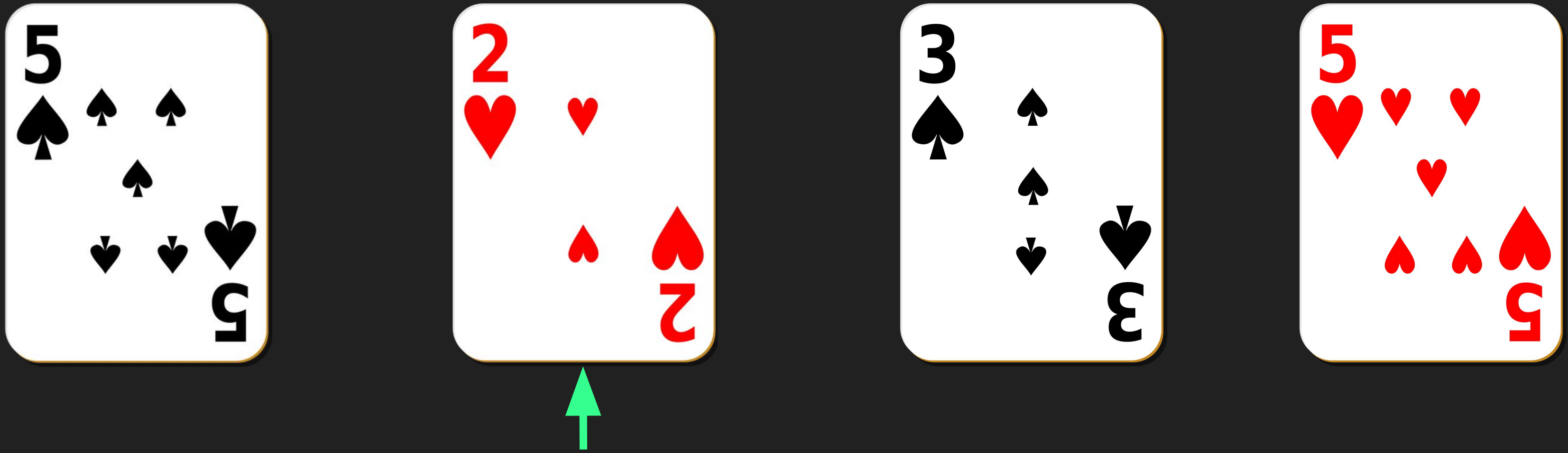


- Go from left to right
- The first time you see your card, exit!

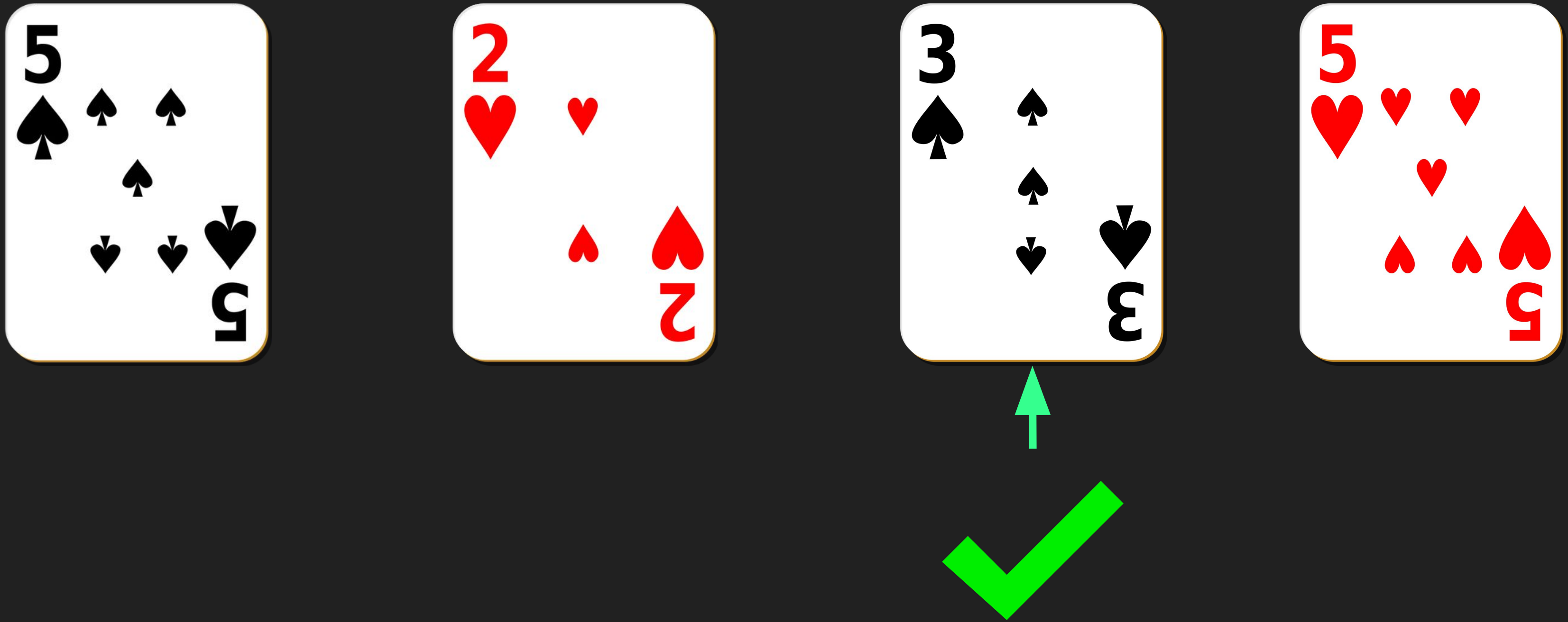
Finding 3



Finding 3



Finding 3



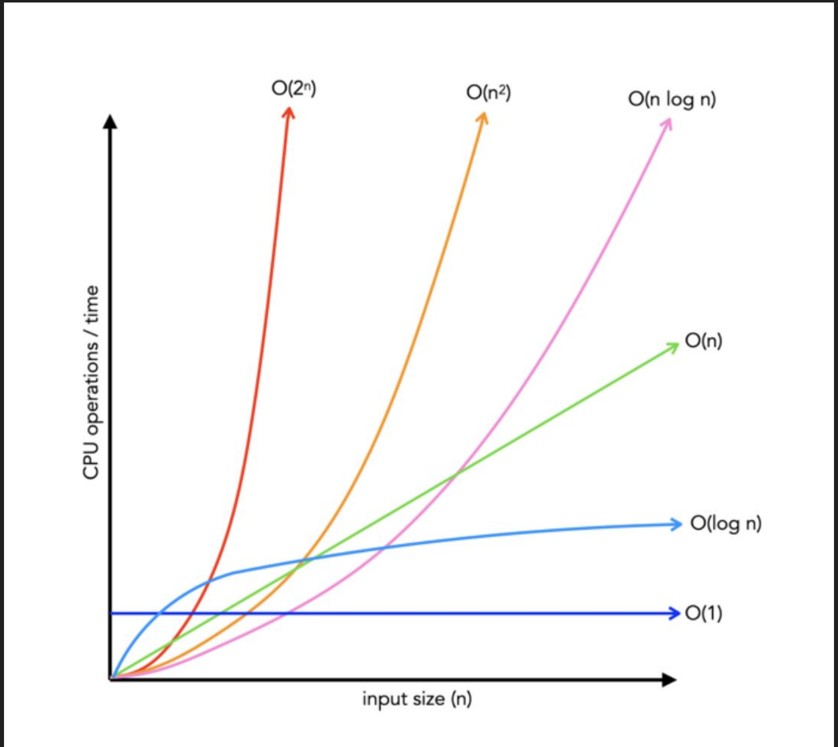
Worst Case

What is the worst case input for this algorithm? (What will make us look at the *most* cards before exiting?)

What is the Big-O (worst case) runtime in terms of deck size *n*?

Common Runtimes

- O(1) - Constant
- O(n) - Linear
- O(n^2) - Quadratic
- O(x^n) - Exponential (BAD)



Source

Dictionaries vs. Lists

- There are runtime considerations for dictionaries/hash tables and lists!
- Dictionaries:
  - Faster lookup: “x in d” ~ O(1)
  - Slower iteration (theoretically)
- Lists:
  - Slower lookup: “x in l” ~ O(n)
  - Faster iteration (theoretically)
- There are many other pros/cons to dictionaries vs. lists, which you will see in other languages/future courses.

Search Algorithms

Selection Sort

Outer loop: Loop over list (everything up to pointer is sorted, everything else is not). Once you reach the end of the list, you're done!

Inner loop: Loop over list to find minimum. Swap the object at outer pointer with the minimum.

```
while idx1 < len(l):
    # Do stuff
    while idx2 < len(l):
        # Do more stuff
```

Outer Loop

Inner Loop

Insertion Sort

Outer loop: Loop over list (everything up to pointer is sorted, everything else is not). Once you reach the end of the list, you're done!

Inner loop: Swap the object at the pointer backwards until it's in the correct position

```
while idx1 < len(l):
    # Do stuff
    while idx2 < len(l):
        # Do more stuff
```

Outer Loop

Inner Loop

Algorithm Analysis

- Runtime: $O(n^2)$
- Memory Usage: $O(n)$