Searching

Searching

- You have a list.
- How do you find something in the list?

- Basic idea: go through the list from start to finish one element at a time.

Linear Search

- Idea: go through the list from start to finish

5	2	3	4

- Example: Search for 3

5	2	3	4

3 not found, move on

5	2	3	4

3 not found, move on

5	2	3	4

Found 3.

Linear Search

Idea: go through the list from start to finish

```
# equivalent code
for i in [5, 2, 3, 4]:
    if i == 3:
        return True
```

Linear Search

Implemented as a function:

```
def linear_search(value, lst):
    for i in lst:
        if i == value:
            return True
    return False
```

What kind of performance can we expect?

Large vs small lists?

Sorted vs unsorted lists?

$$O(n)$$

Can we do better?

Of course la!

Searching

IDEA:

If the elements in the list were sorted in order, life would be much easier.

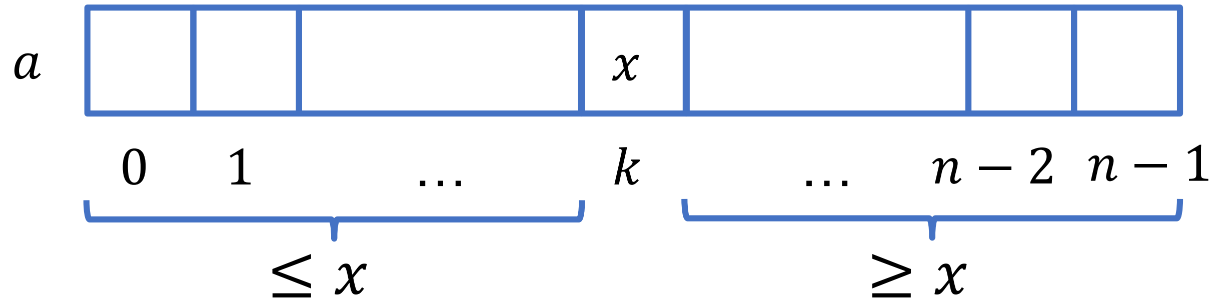
Why?

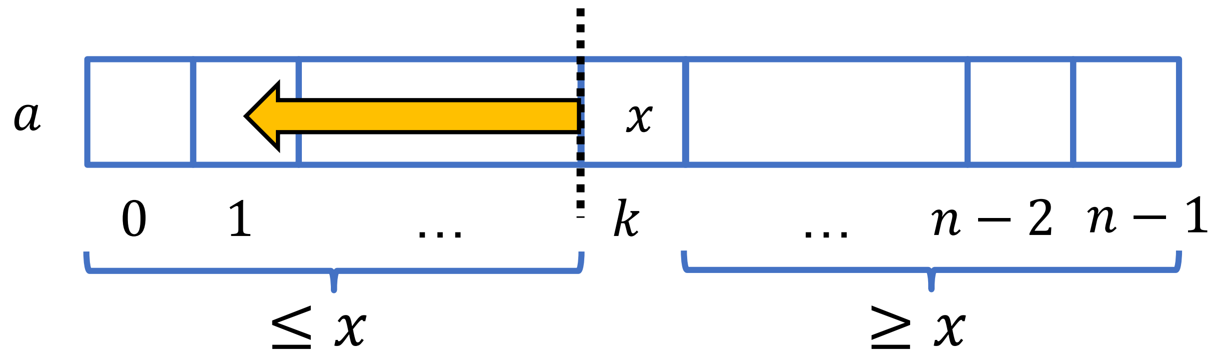

VITAL GRAF
0221 482-5494

IDEA

If list is sorted, we can "divide-and-conquer"

Assuming a list is sorted in ascending order:





if the k^{th} element is larger than what we are looking for, then we only need to search in the indices $< k$

Binary Search

1. Find the middle element.
2. If it is what we are looking for (key), return True.
3. If our key is smaller than the middle element, repeat search on the left of the list.
4. Else, repeat search on the right of the list.

Binary Search

Looking for 25 (key)

5	9	12	18	25	34	85	100	123	345

Find the middle element: 34

5	9	12	18	25	34	85	100	123	345

Not the thing we're looking for: $34 \neq 25$

5	9	12	18	25	34	85	100	123	345

$25 < 34$, so we repeat our search on the left half:

5	9	12	18	25	34	85	100	123	345

Binary Search

Find the middle element: 12

5	9	12	18	25	34	85	100	123	345

25 > 12, so we repeat the search on the right half:

5	9	12	18	25	34	85	100	123	345

Find the middle element: 25

5	9	12	18	25	34	85	100	123	345

Great success: 25 is what we want

5	9	12	18	25	34	85	100	123	345

Binary Search

"Divide and Conquer"

In large sorted lists, performs much better than linear search <u>on average</u>.

Binary Search

Algorithm (assume sorted list):

1. Find the middle element.
2. If it is we are looking for (key), return True.
3. A) If our key is smaller than the middle element, repeat search on the left of the element.
   B) Else, repeat search on the right of the element.

Binary Search

```
def binary_search(key, seq):
    if seq == []:
        return False
    mid = len(seq) // 2
    if key == seq[mid]:
        return True
    elif key < seq[mid]:
        return binary_search(key, seq[:mid])
    else:
        return binary_search(key, seq[mid+1:])
```

Binary Search

```python
def binary_search(key, seq): # seq is sorted
    def helper(low, high):
        if low > high:
            return False
        mid = (low + high) // 2 # get middle
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Binary Search

Now let’s try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high):
        if low > high:
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 1. Find the middle element.

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high):
        if low > high:
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

key → 11

helper(0, 10-1)

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 0, 9
        if low > high:
            return False
        mid = (low + high) // 2 # mid=4
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 1. Find the middle element.

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 0, 9
        if low > high:
            return False
        mid = (low + high) // 2 # mid=4
        if key == seq[mid]: # 11 == 25
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 2. If it is what we are looking for, return True

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 0, 9
        if low > high:
            return False
        mid = (low + high) // 2 # mid=4
        if key == seq[mid]: # 11 == 25
            return True
        elif key < seq[mid]: # 11 < 25
            return helper(low, mid-1) # helper(0, 4-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is smaller, look at left side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=4
        if key == seq[mid]: # 11 == 25
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is smaller, look at left side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=1
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 1. Find the middle element

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=1
        if key == seq[mid]: # 11 == 9
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 2. If it is what we are looking for, return True

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=1
        if key == seq[mid]: # 11 == 9
            return True
        elif key < seq[mid]: # 11 < 9
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is smaller, look at left side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=1
        if key == seq[mid]: # 11 == 9
            return True
        elif key < seq[mid]: # 11 < 9
            return helper(low, mid-1)
        else:
            return helper(mid+1, high) # helper(1+1, 3)
    return helper(0, len(seq)-1)
```

Step 3b. Else look at right side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3b. Else look at right side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3b. Else look at right side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 1. Find the middle element

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=2
        if key == seq[mid]: # 11 == 12
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 2. If it is what we are looking for, return True

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=2
        if key == seq[mid]: # 11 == 12
            return True
        elif key < seq[mid]: # 11 < 12
            return helper(low, mid-1) # helper(2, 2-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is smaller, look at left side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 2, 1
        if low > high:
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is smaller, look at left side

Binary Search

Now let's try searching for 11:

5	9	12	18	25	34	85	100	123	345

```
def binary_search(key, seq):
    def helper(low, high): # 2, 1
        if low > high: # 2 > 1
            return False
        mid = (low + high) // 2
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Key cannot be found. Return False

Binary Search

- Each step eliminates the problem size by half.
  - The problem size gets reduced to 1 very quickly
- This is a simple yet powerful strategy, of halving the solution space in each step
- What is the order of growth?

$$O(\log n)$$

Wishful Thinking

We assumed the list was sorted.

Now, let's deal with this assumption!

Sorting

Sorting

- High-level idea:
  1. some objects
  2. function that can order two objects

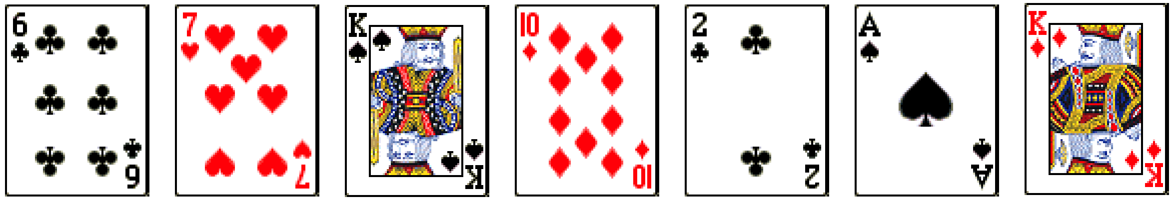
  $\Rightarrow$ order all the objects

How Many Ways to Sort?

Too many. ☺

Example

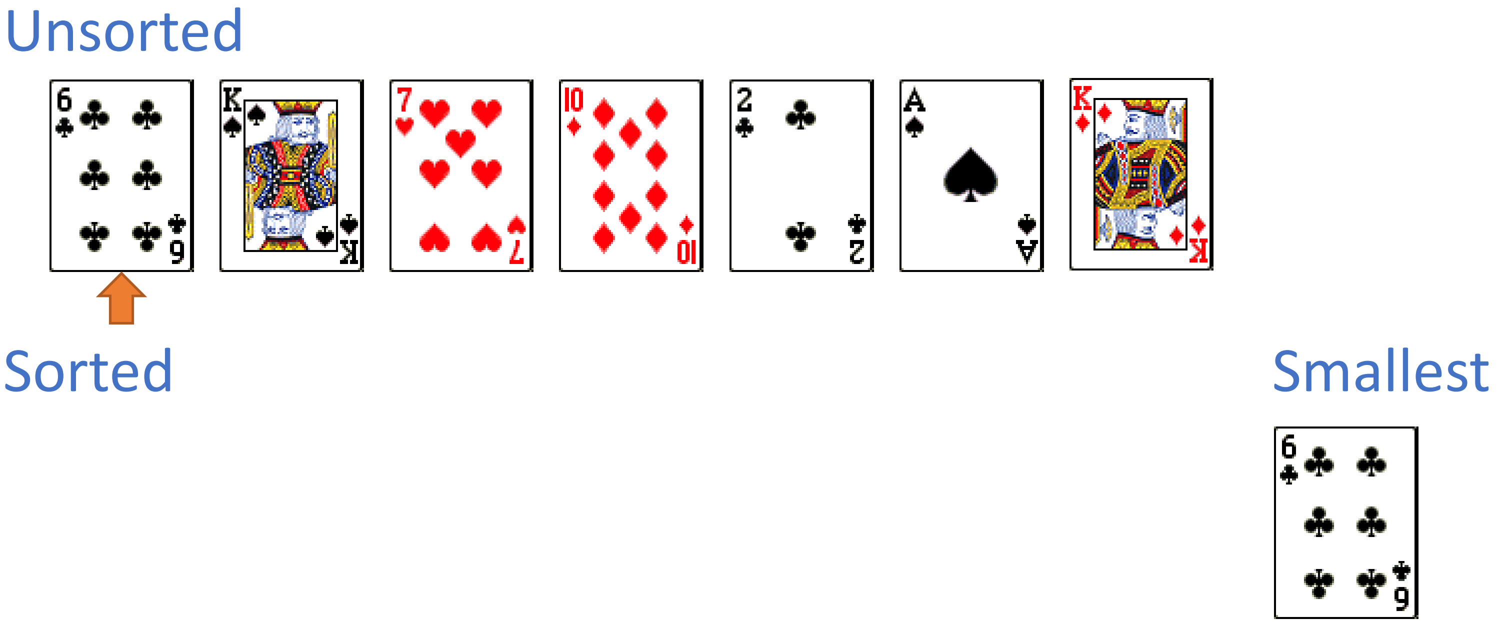
Let’s sort some playing cards?



What do you do when you play cards?

Obvious Way

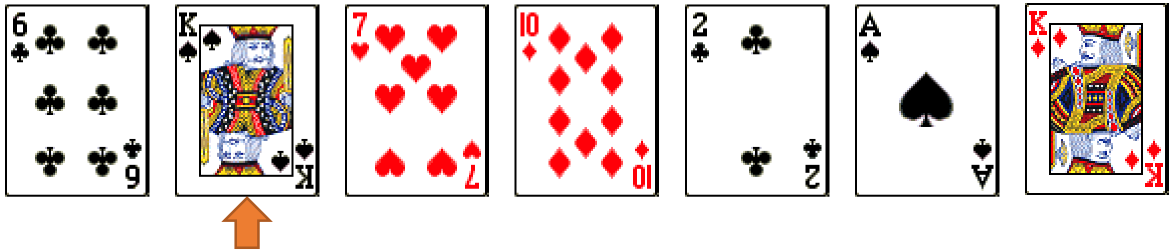
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.



Obvious Way

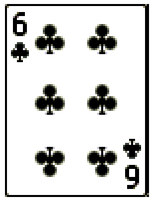
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

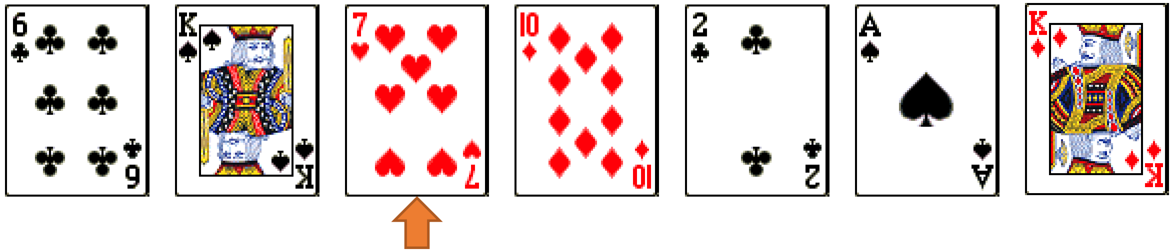
Smallest



Obvious Way

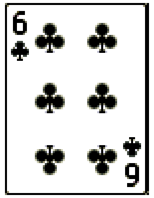
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

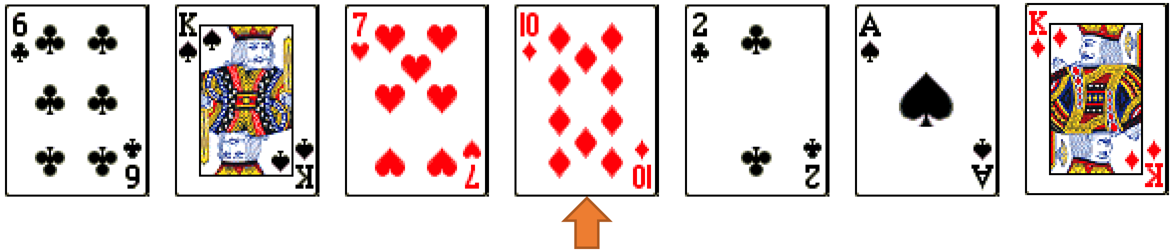
Smallest



Obvious Way

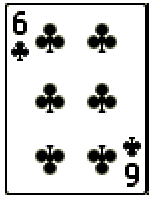
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

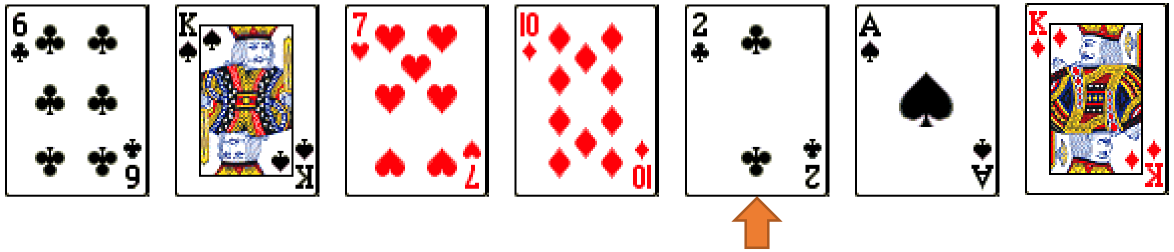
Smallest



Obvious Way

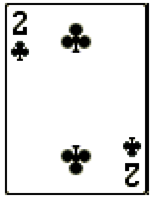
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

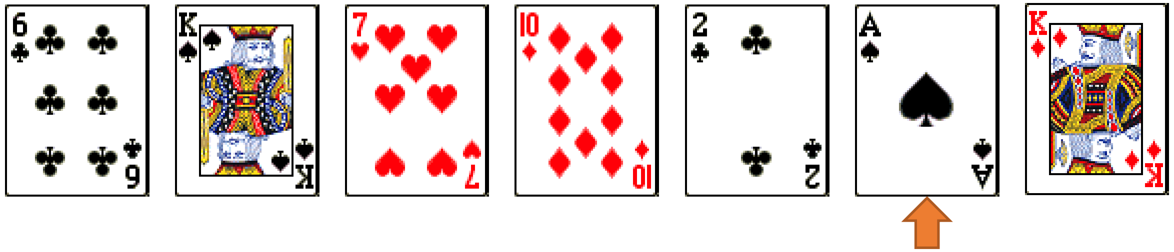
Smallest



Obvious Way

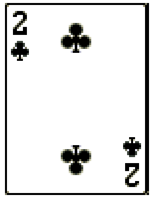
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

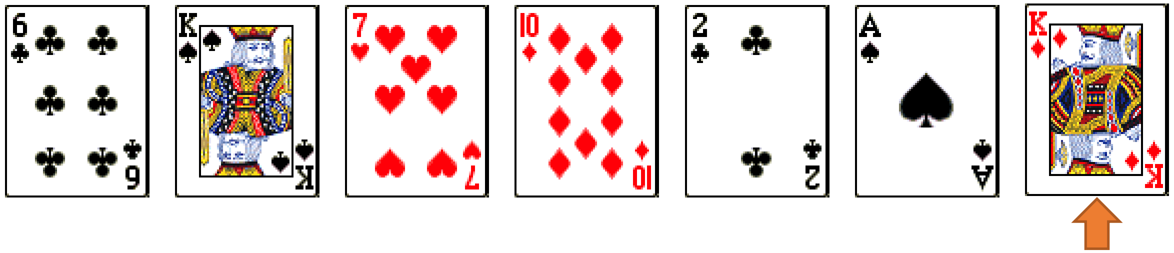
Smallest



Obvious Way

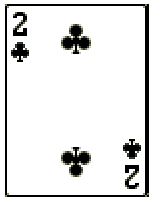
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

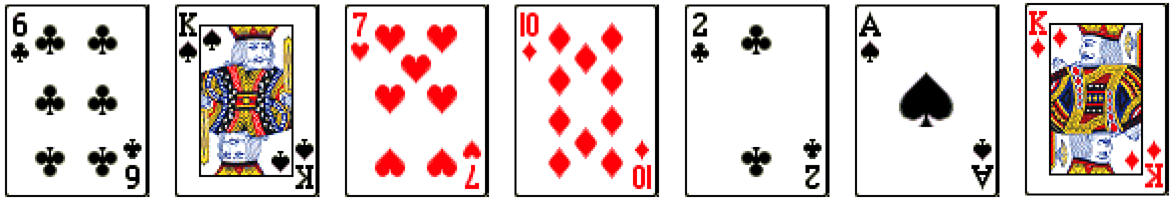
Smallest



Obvious Way

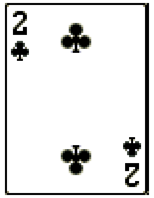
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

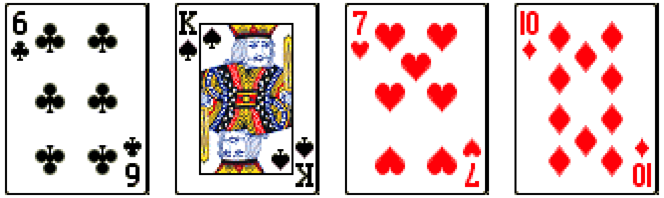
Smallest

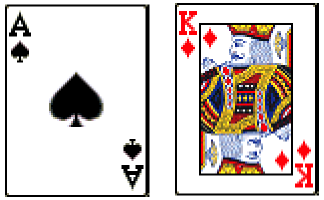


Obvious Way

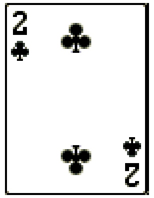
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted





Sorted



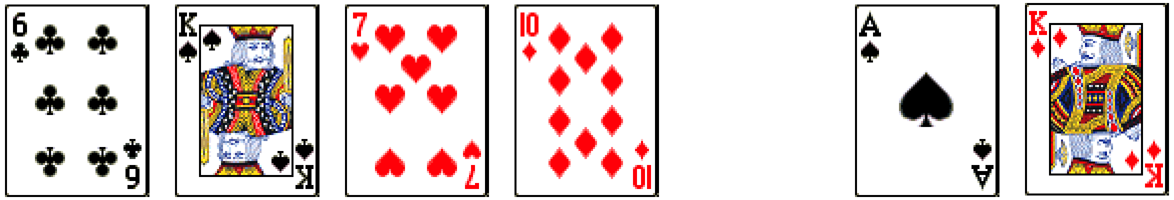
Smallest

Obvious Way

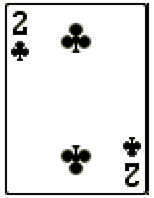
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Repeat

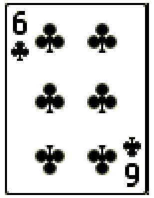
Unsorted



Sorted

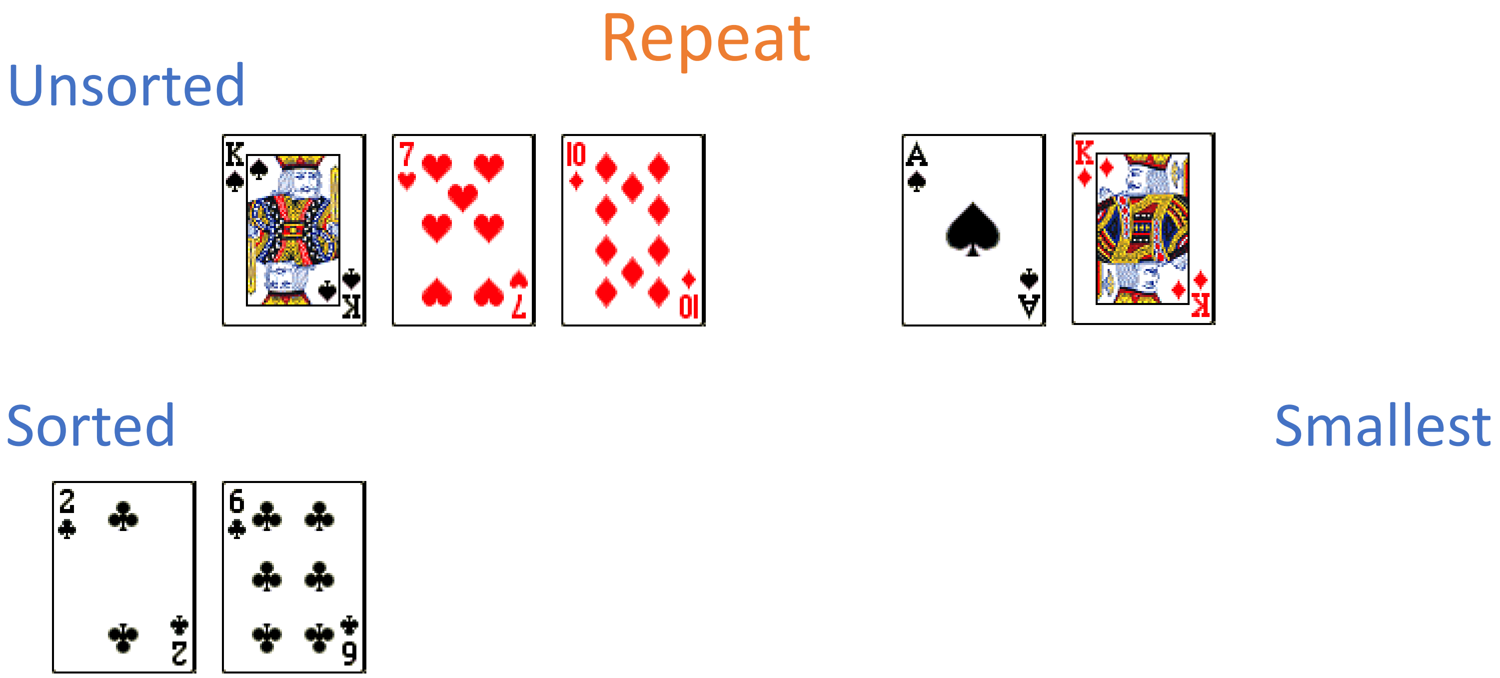


Smallest



Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

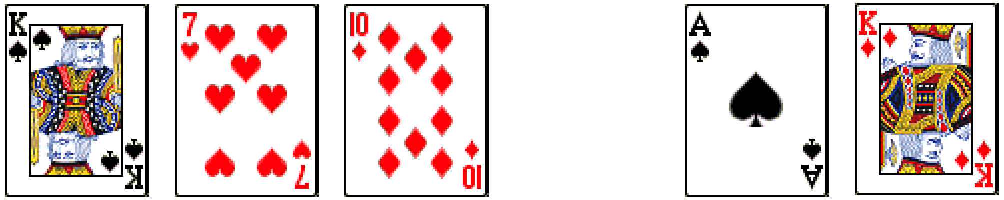


Obvious Way

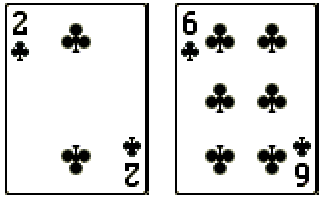
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Repeat

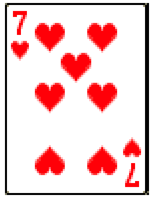
Unsorted



Sorted



Smallest

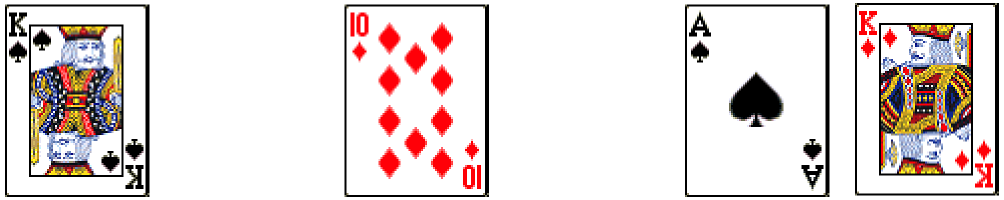


Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

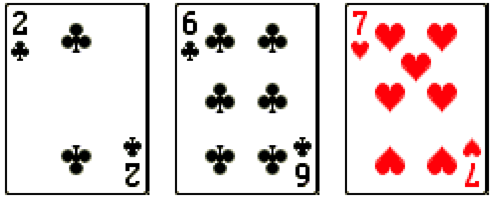
Repeat

Unsorted



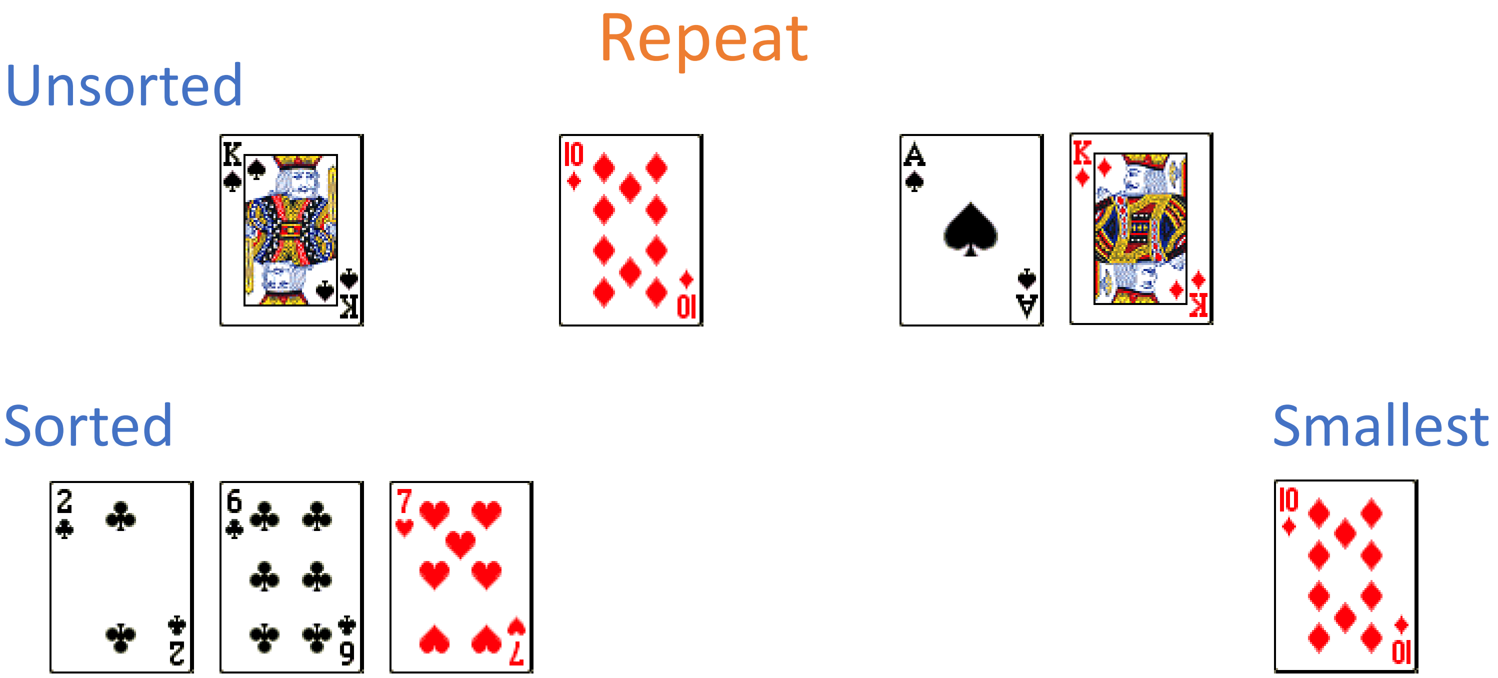
Sorted

Smallest



Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

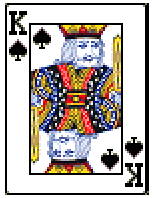


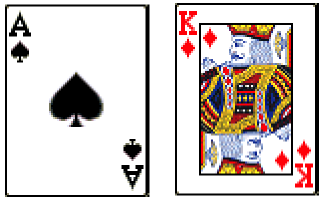
Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

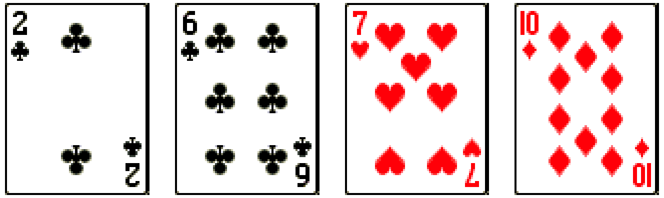
Repeat

Unsorted





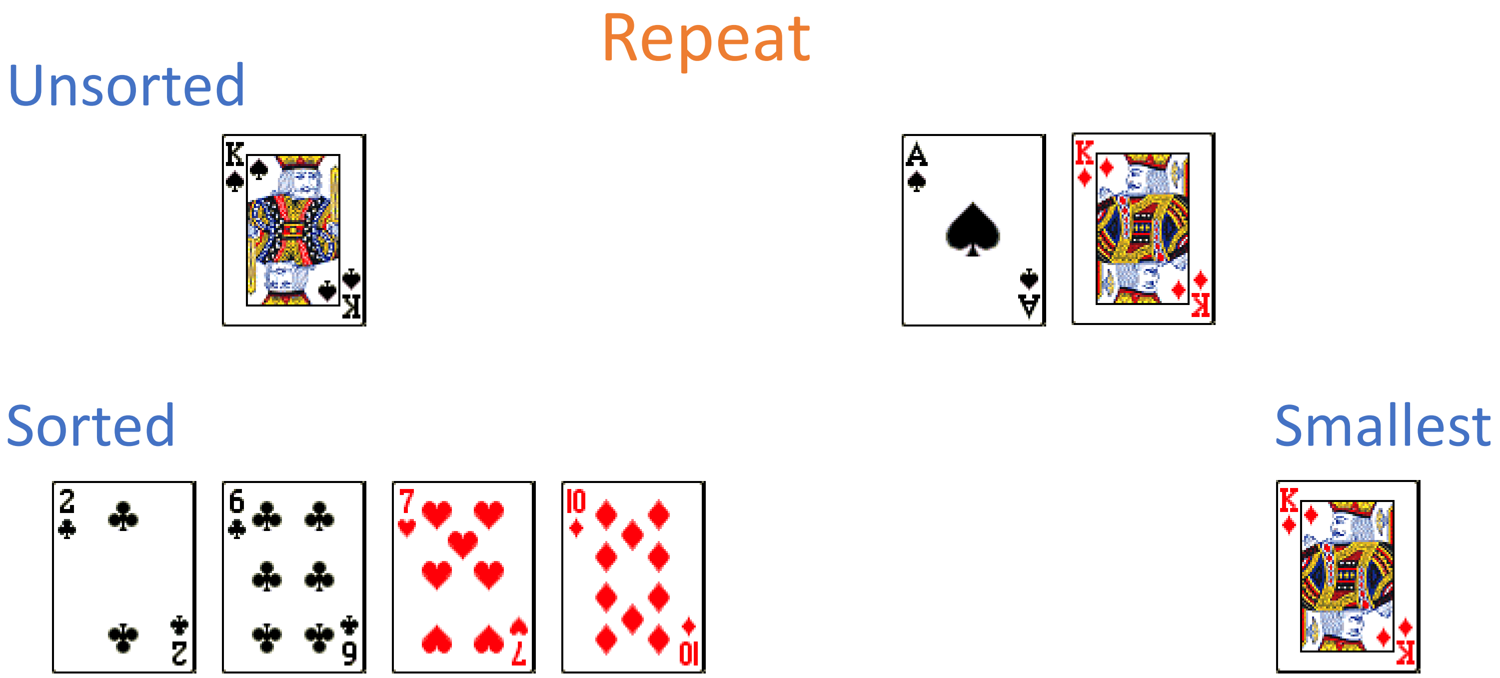
Sorted



Smallest

Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

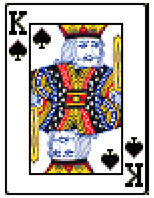


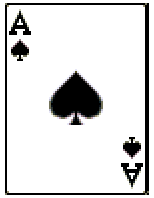
Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Repeat

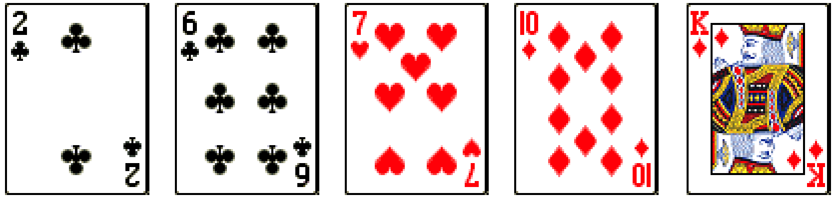
Unsorted





Sorted

Smallest

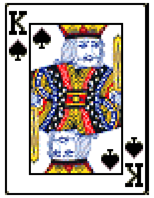


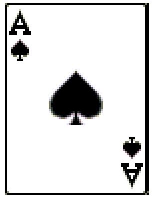
Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

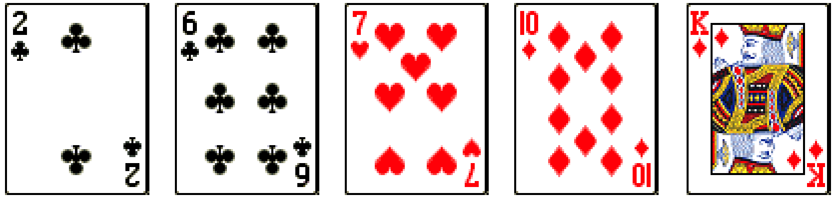
Repeat

Unsorted

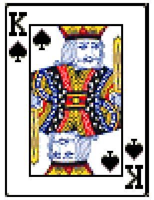




Sorted



Smallest

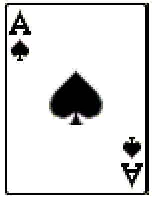


Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

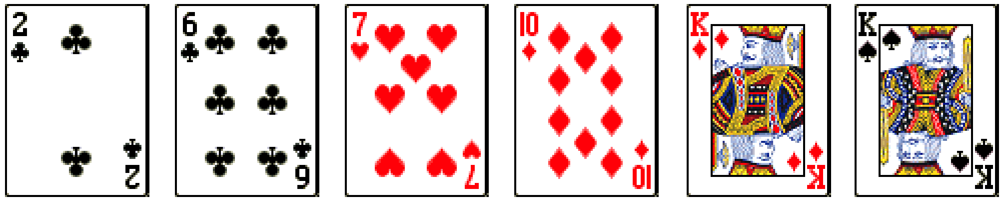
Repeat

Unsorted



Sorted

Smallest

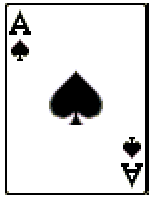


Obvious Way

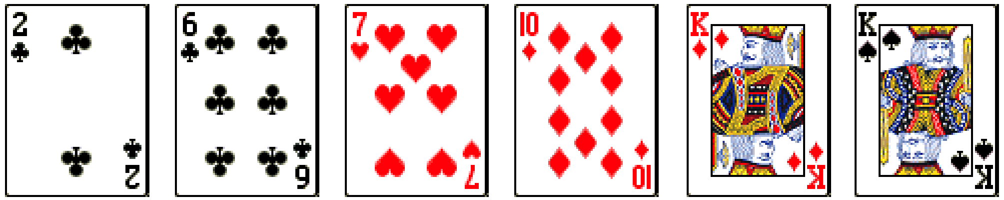
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Repeat

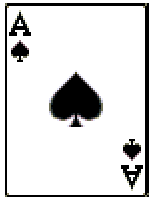
Unsorted



Sorted



Smallest



Obvious Way

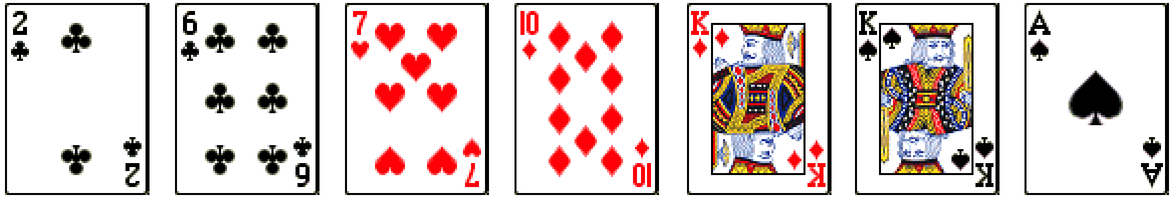
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Done

Unsorted

Sorted

Smallest



There is actually a name for this:

Selection Sort!

Let’s Implement it!

```
a = [4,12,3,1,11]
sort = []

while a:   # a is not []
    smallest = a[0]
    for element in a:
        if element < smallest:
            smallest = element
    a.remove(smallest)
    sort.append(smallest)
    print(a)
```

Output

```
[4, 12, 3, 11]
[4, 12, 11]
[12, 11]
[12]
[]
print(a)
[]

print(sort)
[1, 3, 4, 11, 12]
```

Order of Growth?

- Time:

Worst	$O(n^2)$
Average	$O(n^2)$
Best	$O(n^2)$

- Space: $\cancel{O(n)}$ $O(1)$

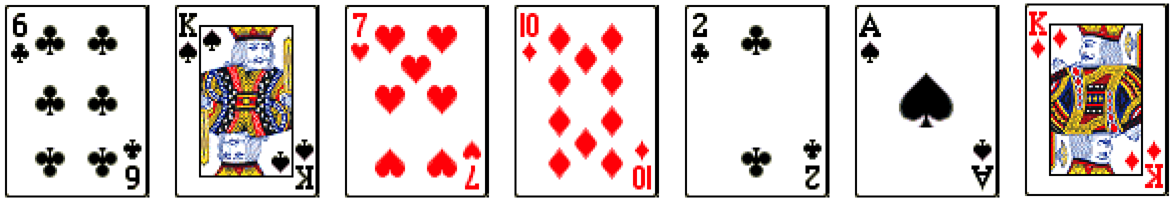
Let’s try something else…

suppose you have a friend

Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

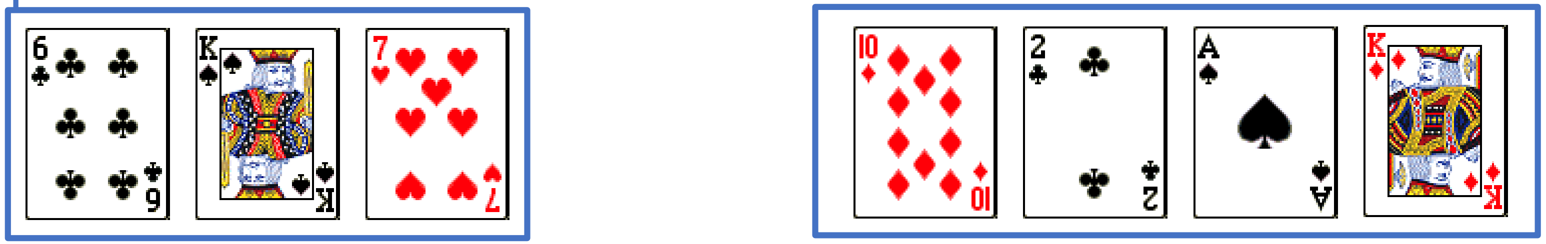
Split into halves



Doing it with a friend

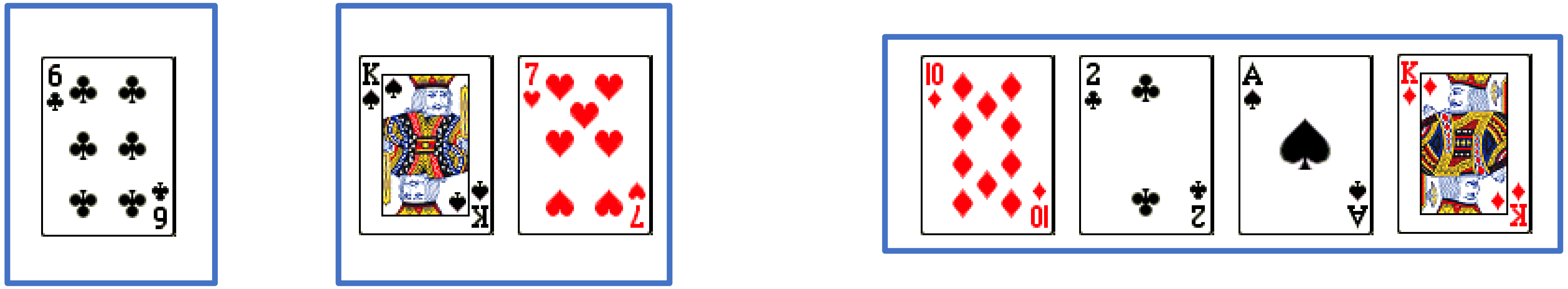
- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

Split into halves



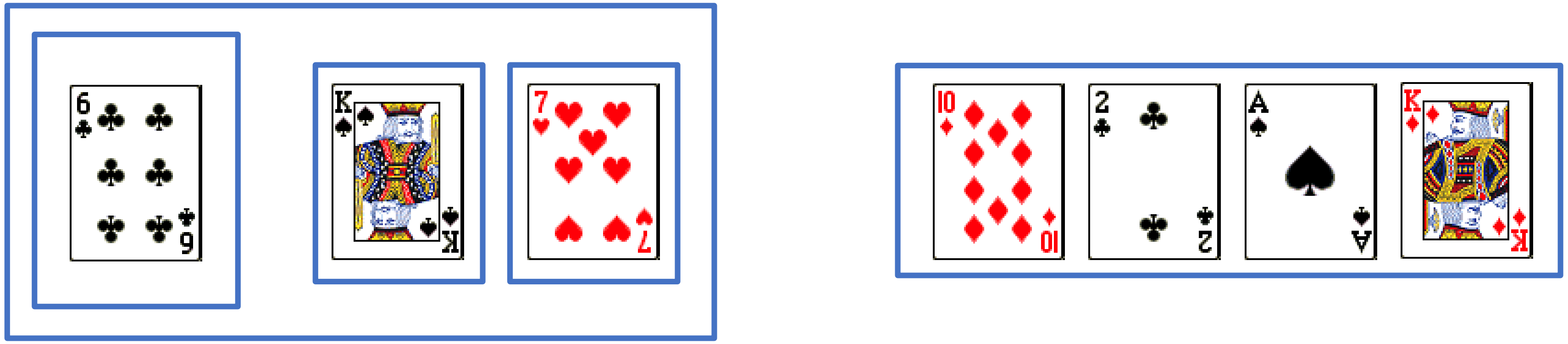
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



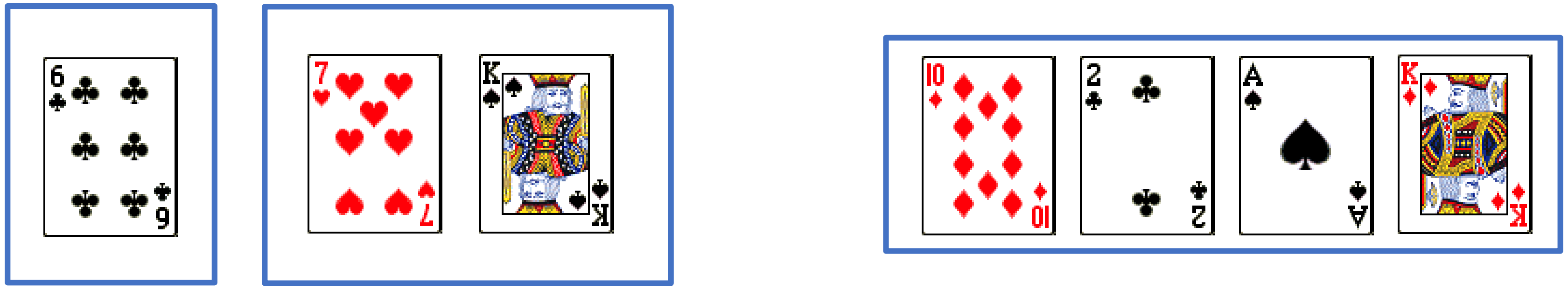
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



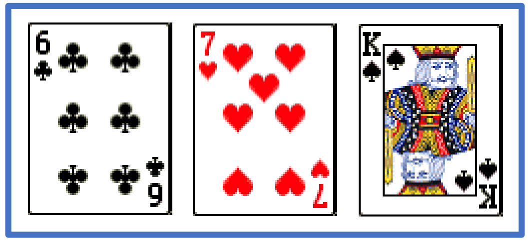
Doing it with a friend

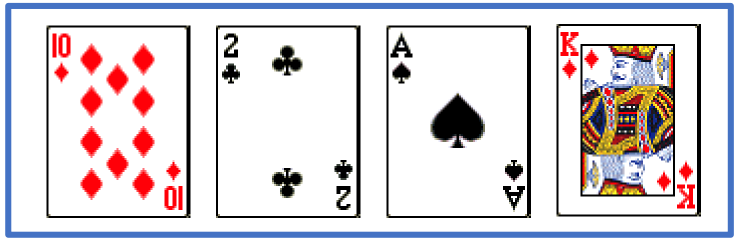
- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



Doing it with a friend

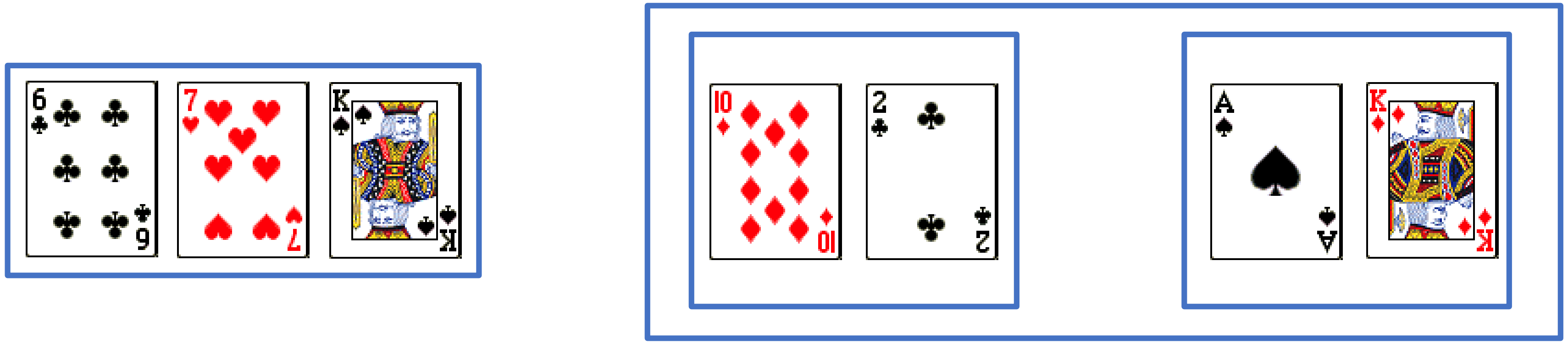
- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.





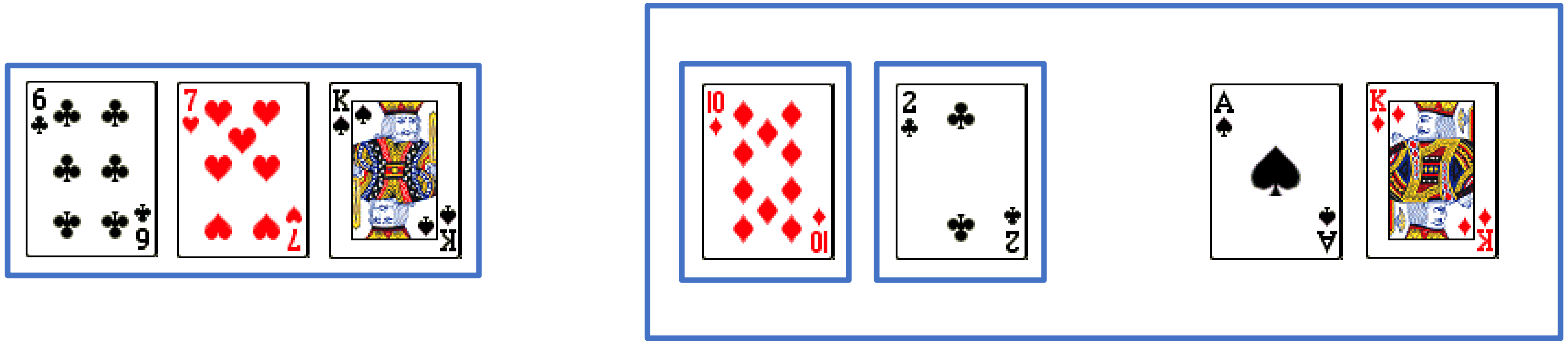
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



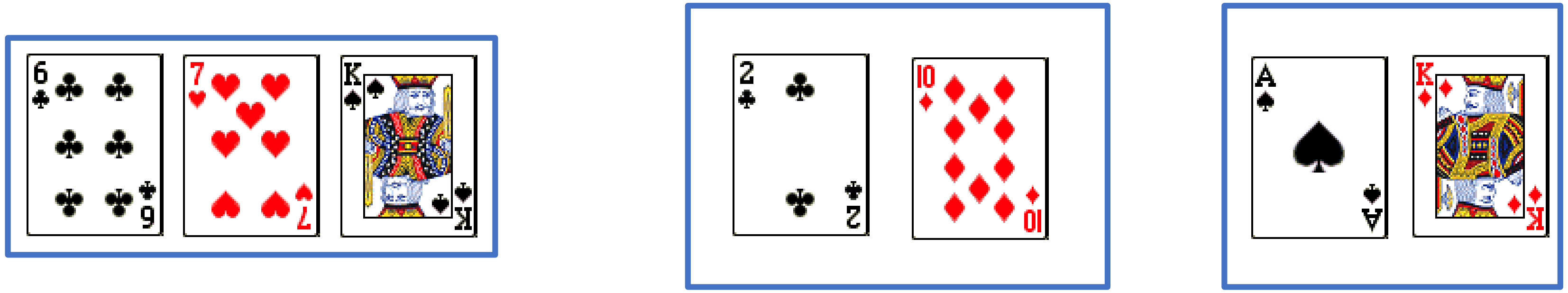
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



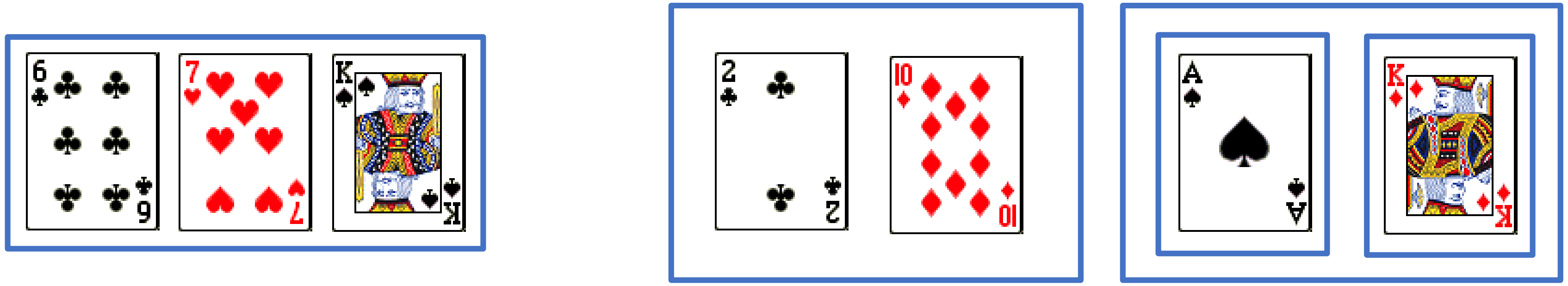
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



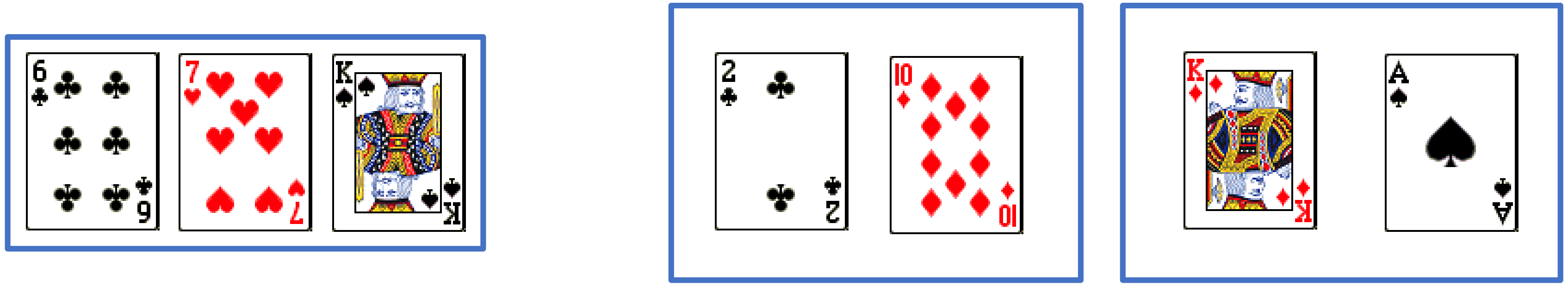
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



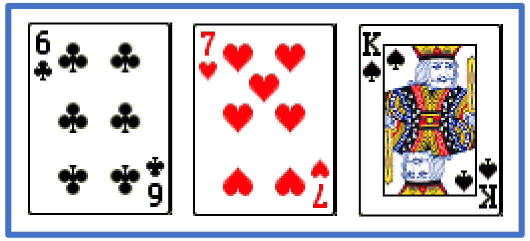
Doing it with a friend

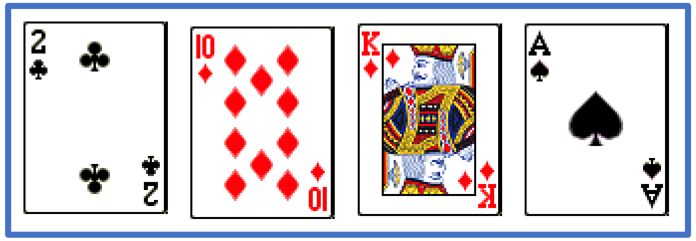
- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



Doing it with a friend

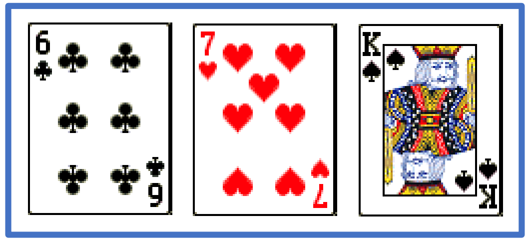
- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

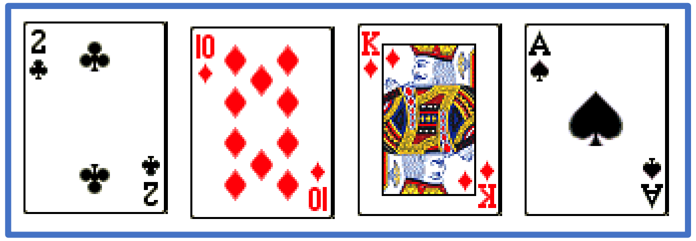




Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



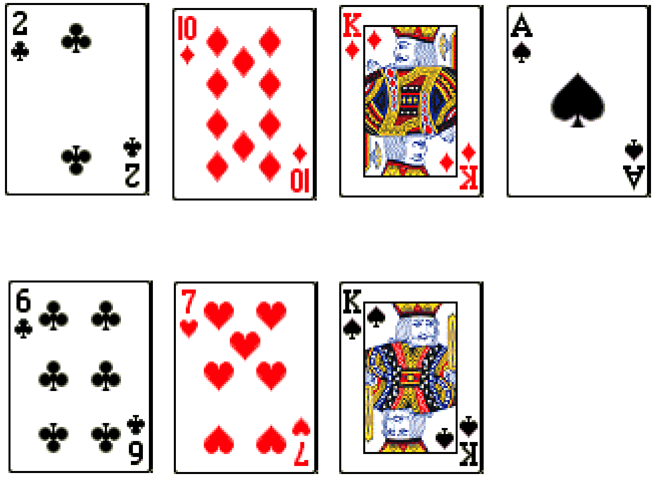


How to combine the 2 sorted halves?

Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

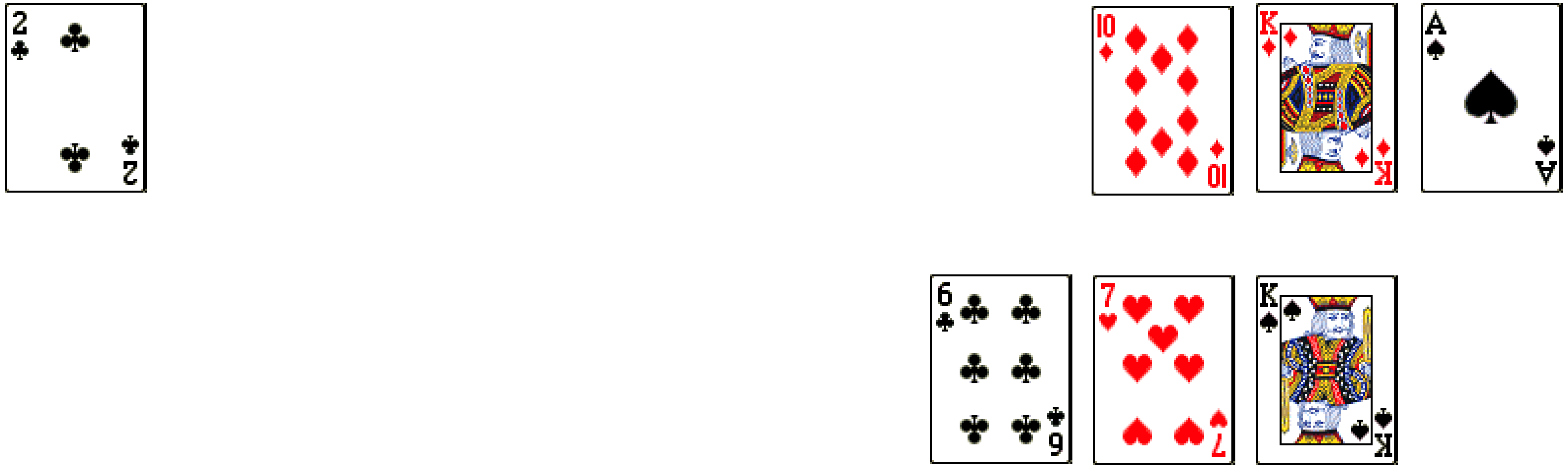
Compare first elements



Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

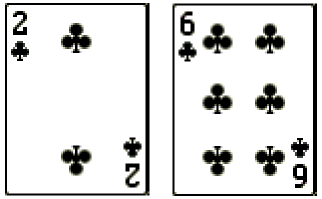
Compare first elements

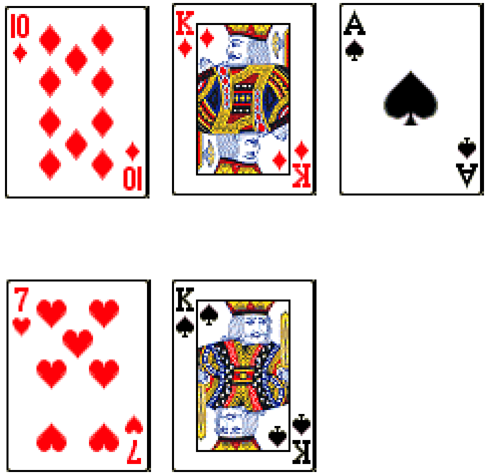


Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

Compare first elements

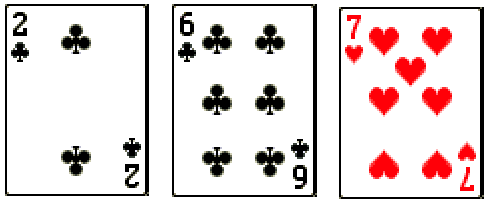


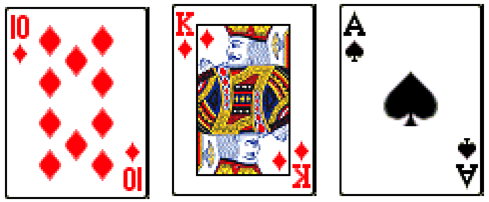


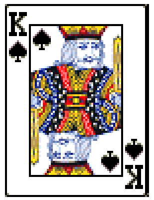
Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

Compare first elements



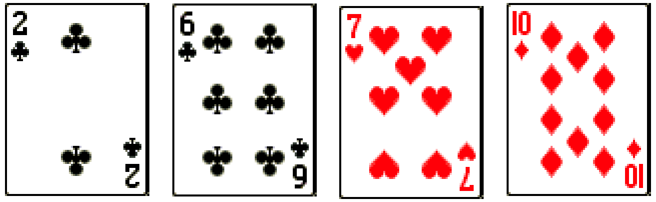


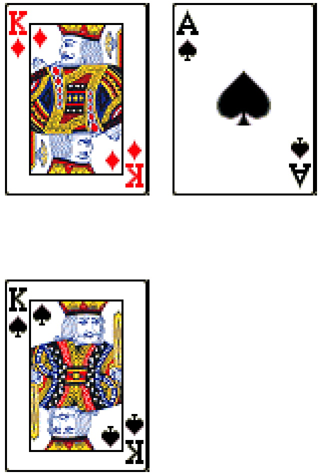


Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

Compare first elements

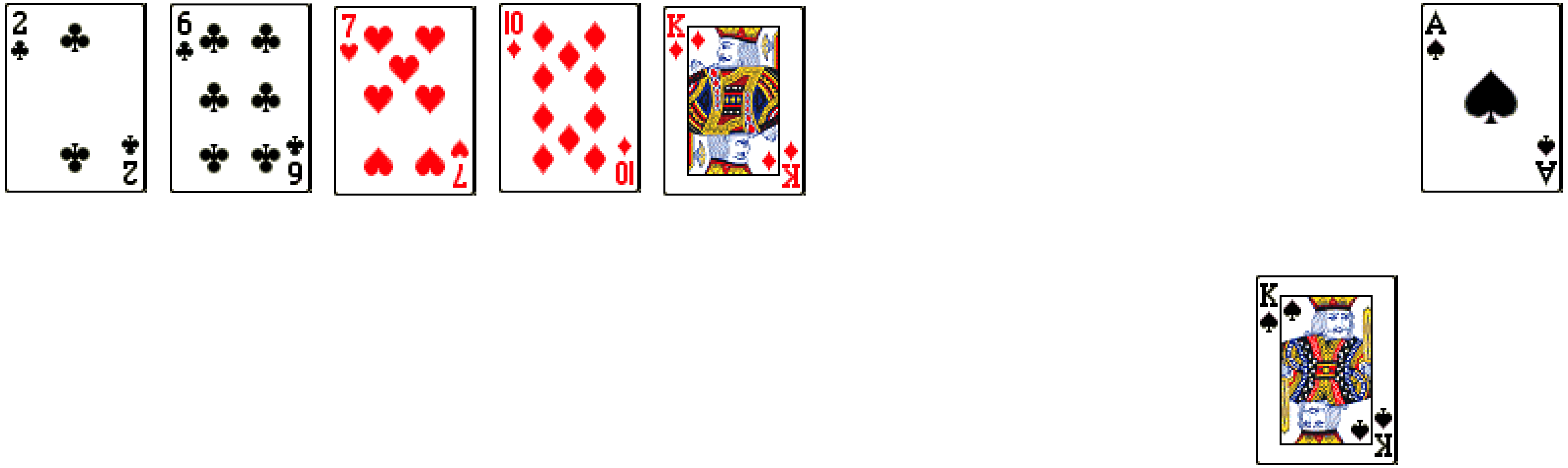




Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

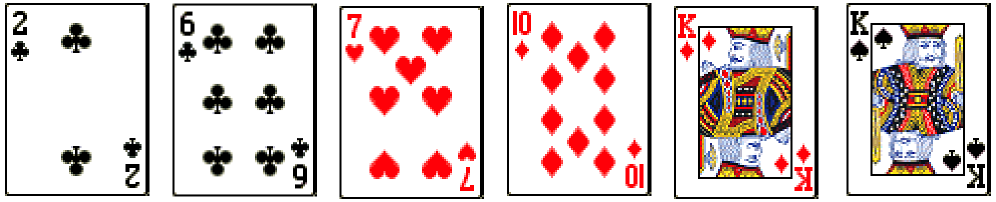
Compare first elements

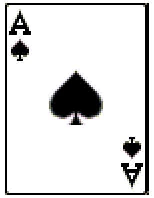


Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

Compare first elements

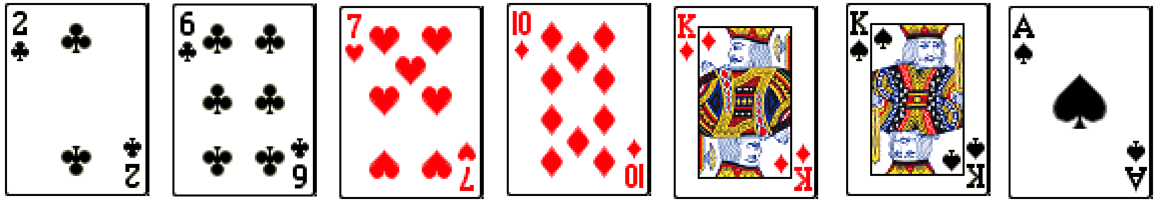




Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

Compare first elements



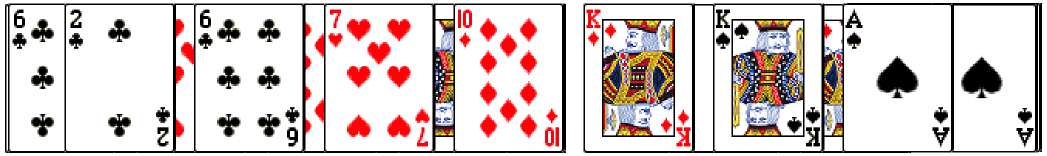
There is also a name for this:

Merge Sort!

Let’s Implement It!

First observation: RECURSION!

- Base case: n< 2, return lst
- Otherwise:



  - Divide list into two
  - Sort each of them
  - Merge!

Merge Sort

```
def merge_sort(lst):
    if len(lst) < 2:  # Base case!
        return lst
    mid = len(lst) // 2
    left = merge_sort(lst[:mid])  #sort left
    right = merge_sort(lst[mid:]) #sort right
    return merge(left, right)
```

How to merge?

How to merge?

- Compare first element
- Take the smaller of the two
- Repeat until no more elements

Merging

```
def merge(left, right):
    results = []
    while left and right:
        if left[0] < right[0]:
            results.append(left.pop(0))
        else:
            results.append(right.pop(0))
    results.extend(left)
    results.extend(right)
    return results
```

Order of Growth?

- Time:

Worst	$O(n \log n)$
Average	$O(n \log n)$
Best	$O(n \log n)$

- Space: $O(n)$

No need to memorize

Sort Properties

In-place: uses a small, constant amount of extra storage space, i.e., $O(1)$ space

Selection Sort: No (Possible)

Merge Sort: No (Possible)

Sort Properties

Stability: maintains the relative order of items with equal keys (i.e., values)

Selection Sort: Yes (maybe)

Merge Sort: Yes

How Many Ways to Sort?

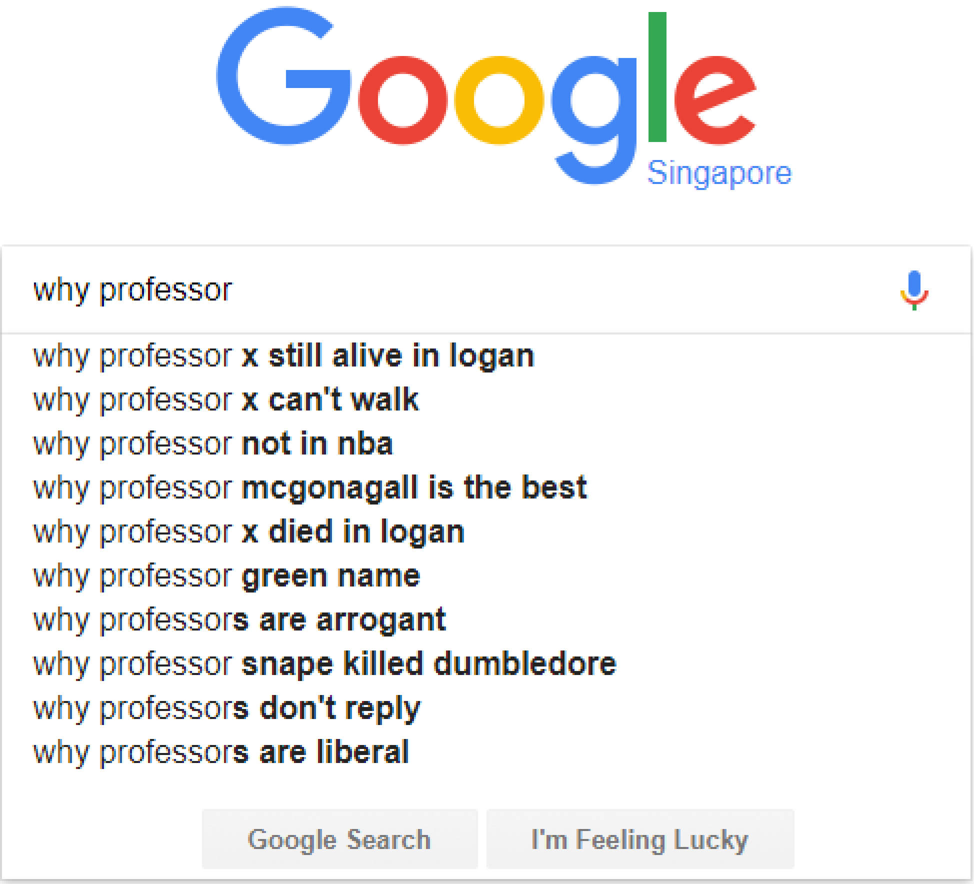
Too many. ☺

Summary

- Python Lists are mutable data structures
- Searching
  - Linear Search
  - Binary Search: Divide-and-conquer
- Sorting
  - Selection Sort
  - Merge Sort: Divide-and-conquer + recursion
  - Properties: In-place & Stability

Ok, now I know how to guess a number quickly

Google?


Singapore
why professor
why professor x still alive in logan
why professor x can't walk
why professor not in nba
why professor mcgonagall is the best
why professor x died in logan
why professor green name
why professors are arrogant
why professor snape killed dumbledore
why professors don't reply
why professors are liberal
Google Search
I'm Feeling Lucky

How much data does Google handle?

- About 10 to 15 Exabyte of data
  - 1 Exabyte(EB)= 1024 Petabyte(PB)
  - 1 Petabyte(PB) = 1024 Terabytes(TB)
  - 1 Terabyte(PB) = 1024 Gigabytes(TB)
    - = 4 X 256GB iPhone
- So Google is handling about 60 millions of iPhones

How fast is my desktop?

```
import time

def create_list(n):
    start_time = time.time()
    l = [i for i in range(n)]
    end_time = time.time()
    print('Duration = ' +
          str(end_time-start_time) + ' s')

create_list(1000)
create_list(1000*1000)
create_list(1000*1000*100)
```

Return the time in seconds since the epoch (1970/1/1 00:00) as a floating point number.

Create 100M of numbers, estimated to be 400MB of data

Output:

```
Duration = 0.0 s
Duration = 0.04769182205200195 s
Duration = 6.02686572074890l s
```

Let's calculate

- 400M of data needs 6 seconds
- 15 Exabyte of data needs how long?
  - 15 EB = 15 x 1024 x 1024 x 1024 x 1024 MB
  - To search through 15EB of data.....
    - 7845 years.....

- If we do it with Binary Search
  - $\log_2$ (15EB) = 43 steps!!!!!

The Power of Algorithm