

Computational thinking

Computational thinking

1. Decomposition

Can I divide this into sub-problems?

2. Pattern recognition

Can I find repeating patterns?

3. Abstraction

Can generalise this to make an overall rule?

4. Algorithm design

Can I design the programming steps for any of this?

Applying computational thinking

1. Start with a small human solvable version of the problem.
2. Keep working through the 4 computational thinking prompts until you devise a solution.
3. Test your proposed solution works on larger, real-world versions of the problem.
4. Adapt and repeat the process as necessary.

Example: Sorting numbers

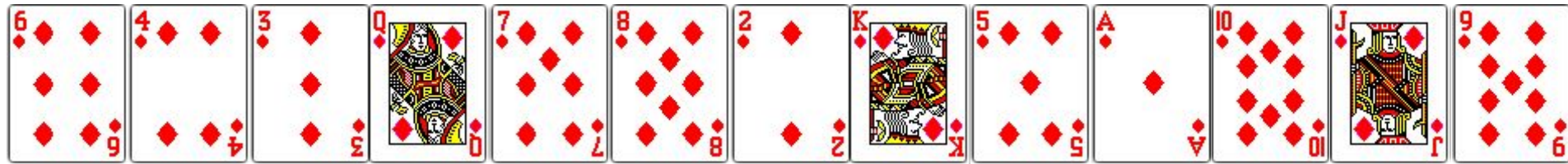
Challenge: Create a program that will sort a list of numbers into ascending order.

16	23	80	18	24	77	52	44	36	1	39	41	45	14	63	70	11	5
63	36	24	60	34	4	41	51	27	18	93	97	51	58	17	78	40	22
67	60	47	52	12	35	32	32	47	49	90	92	14	41	31	28	43	78
42	15	48	21	45	33	26	92	98	6	81	41	31	37	100	72	62	56
24	85	12	70	5	92	31	2	11	100	53	45	62	58	61	67	58	12
90	6	95	20	32	89	90	64	99	29	78	90	98	38	47	5	69	79
18	46	35	84	93	40	68	55	73	47	9	56	48	24	42	11	10	43
32	17	48	66	21	65	24	35	94	74	63	87	92	93	91	93	49	48
61	88	24	75	35	63	4	80	44	11	44	88	64	78	85	72	82	31
47	64	38	55	82	92	48	98	2	21	100	40	51	39	62	12	74	96
62	19	53	9	66	92	67	24	25	39	77	27	73	20	27	48	56	89

Getting started

First step: Create a human solvable version of the problem.

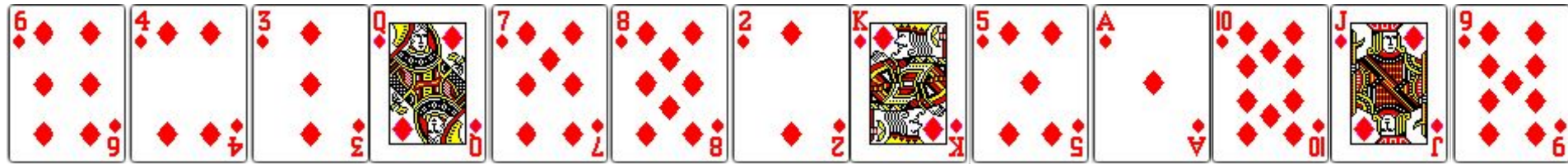
An enormous infinitely large set of numbers is a little overwhelming to think about. We know how to sort playing cards though.



Have a go

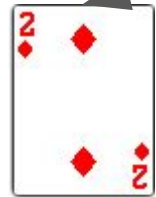
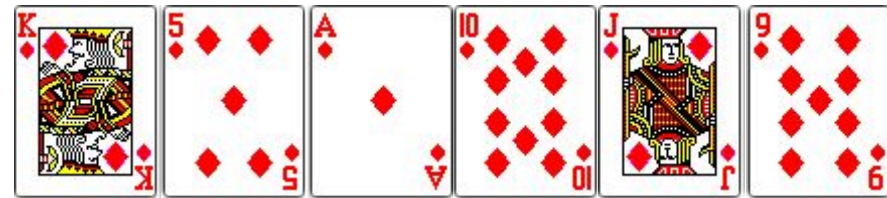
Task: 5 minutes with the person next to you. Write a set of steps for someone who had never sorted values, what would you write?

Remember: Your ultimate goal is not how to sort the given set of cards as presented, but to sort any possible set of numbers.

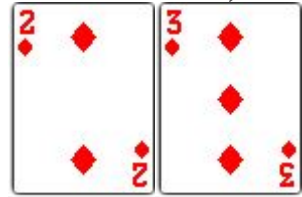
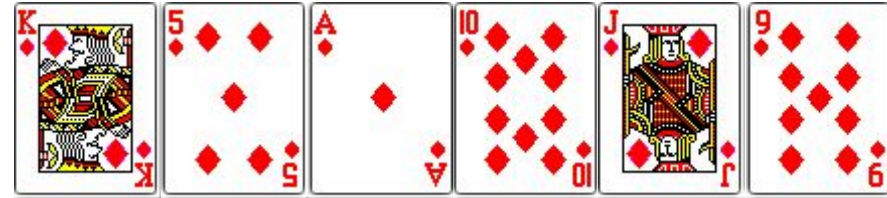
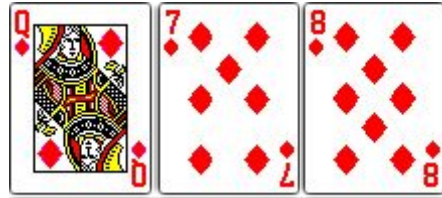
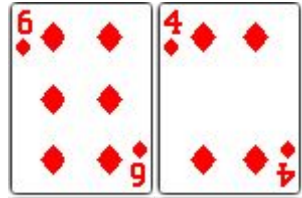


* assume Jack = 11, Queen = 12, King = 13, Ace = 14

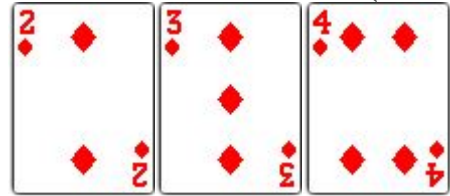
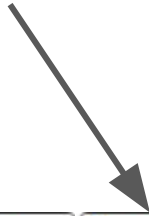
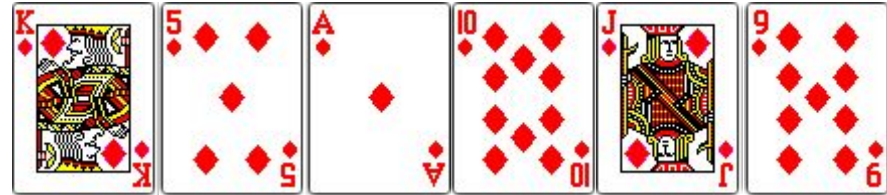
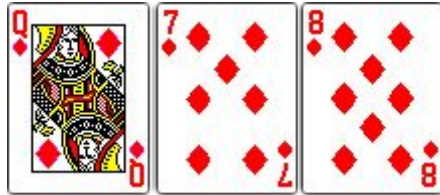
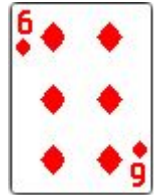
The human process...



The human process...



The human process...



But what did we actually do?

Find the 2

Move it to the sorted set

Find the 3

Move it to the sorted set

Find the 4

Move it to the sorted set

... etc

But what did we actually do?

Find the 2

Move it to the sorted set

Find the 3

Move it to the sorted set

Find the 4

Move it to the sorted set

... etc

The principle of **Abstraction** requires us to remove unnecessary complexity so we can create a general model or rule.

This sequence of steps is currently specific to our sample problem. How can we generalise it?

What did we actually do when we found the 2?

But what did we actually do?

Find the lowest value

Move it to the sorted set

Find the lowest value

Move it to the sorted set

Find the lowest value

Move it to the sorted set

... etc

But what did we actually do?

Find the lowest value

Move it to the sorted set

Find the lowest value

Move it to the sorted set

Find the lowest value

Move it to the sorted set

... etc

The principle of **Pattern recognition** requires us to identify repeating patterns which this clearly is.

But what did we actually do?

While unsorted values remain:

- Find the lowest value

- Move it to the sorted set

We have taken our original one “overwhelming” problem and broken it down into three smaller sub-problems.

This is **Decomposition** at work!

I can now solve each of these separately. I continue decomposing my larger problem into smaller problems until I reach chunks I know how to program.

Turning this into code...

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]

# While unsorted values remain
    # Find the lowest value
    # Move it to the sorted set
```

Note: For the purpose of this exercise, we are going to behave as though `sort()`, `min()` and `max()` do not exist. These are fairly trivial functions so if you can't write your own alternatives to them there is a problem.

Turning this into code...

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]

# While unsorted values remain
while len(values) > 0:
    # Find the lowest value
    # Move it to the sorted set
```

← This is fairly easy, so doesn't need further decomposition.

What about the other steps?

Turning this into code...

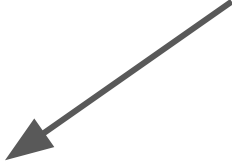
```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]

# While unsorted values remain
while len(values) > 0:

    # Find the lowest value
    lowest = find_lowest_value( values )

    # Move it to the sorted set
```

This function doesn't exist yet, but I know this is what I need it to do. I can solve it later.



I now realise I don't have a "sorted set" yet, so I need to create it.

Turning this into code...

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]
result = []
```

For my solution



```
# While unsorted values remain
while len(values) > 0:

    # Find the lowest value
    lowest = find_lowest_value( values )

    # Move it to the sorted set
```

Turning this into code...

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]
result = []

# While unsorted values remain
while len(values) > 0:

    # Find the lowest value
    lowest = find_lowest_value( values )

    # Move it to the sorted set
    result.append( values[lowest] )
```

Problem. This adds a value to my results but does not remove it from the source values.

I now realise that “move” is a two step process. More decomposition required.

Turning this into code...

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]
result = []
```

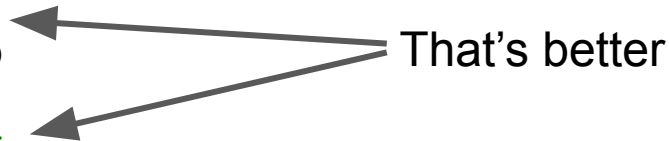
```
# While unsorted values remain
while len(values) > 0:
```

```
    # Find the lowest value
    lowest = find_lowest_value( values )
```

```
    # Add it to the sorted set
    result.append( values[lowest] )
```

```
    # Remove it from the source set
```

That's better



Turning this into code...

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]
result = []

# While unsorted values remain
while len(values) > 0:

    # Find the lowest value
    lowest = find_lowest_value( values )

    # Add it to the sorted set
    result.append( lowest )

    # Remove it from the source set
    values.remove( lowest )
```

That's better



Turning this into code...

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]
result = []

# While unsorted values remain
while len(values) > 0:

    # Find the lowest value
    lowest = find_lowest_value( values )

    # Add it to the sorted set
    result.append( lowest )

    # Remove it from the source set
    values.remove( lowest )
```

Now we are just left with
this....

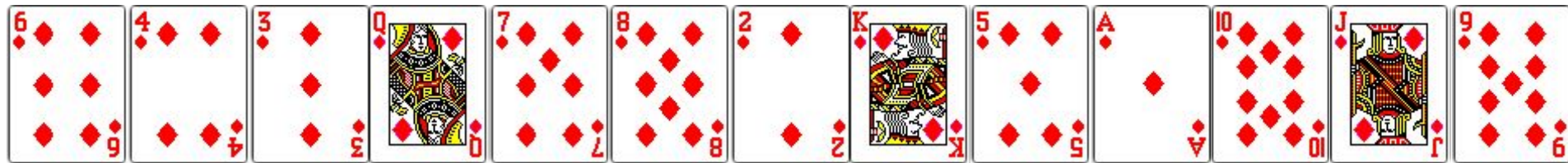
Use the same steps if you
have to...

Finding the lowest value

How did we identify the 2 as the lowest value?

Decomposition

Identify the steps.



Finding the lowest value

Start with the first value

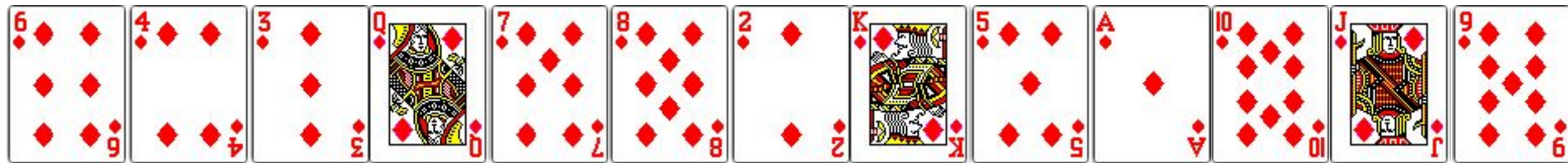
Is the 2nd value lower? If so, remember it as the lowest

Is the 3rd value lower? If so, remember it as the lowest

Is the 4th value lower? If so, remember it as the lowest

Abstraction

Create a generalised model.



Pattern recognition.

Identify repetition.

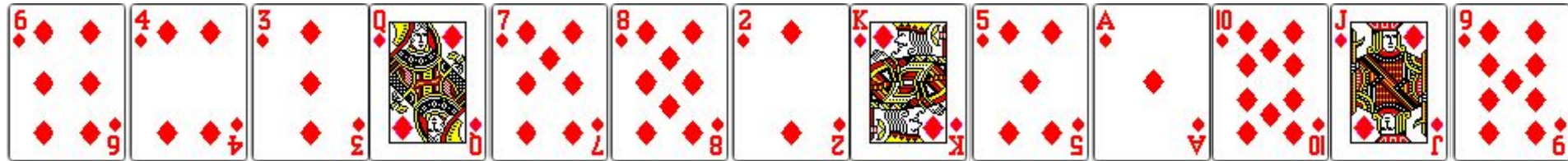
Finding the lowest value

Start with the first value

Look at the next card. Is it lower? If so, remember it as the lowest

Look at the next card. Is it lower? If so, remember it as the lowest

Look at the next card. Is it lower? If so, remember it as the lowest



Algorithm design

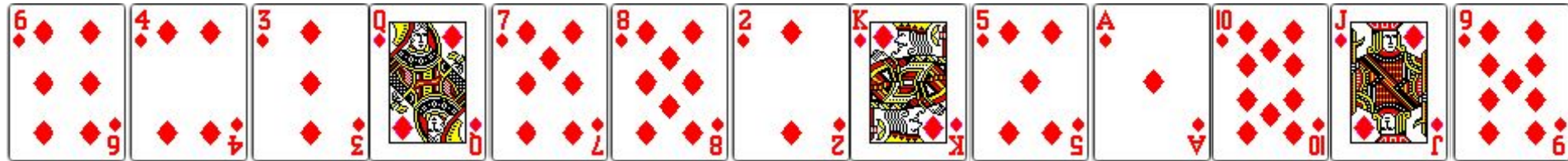
Document our sequence of steps.

Finding the lowest value

Start with the first value

For all the values in our problem:

Look at the next value. Is it lower? If so, remember it as the lowest



Done!

Finding the lowest value

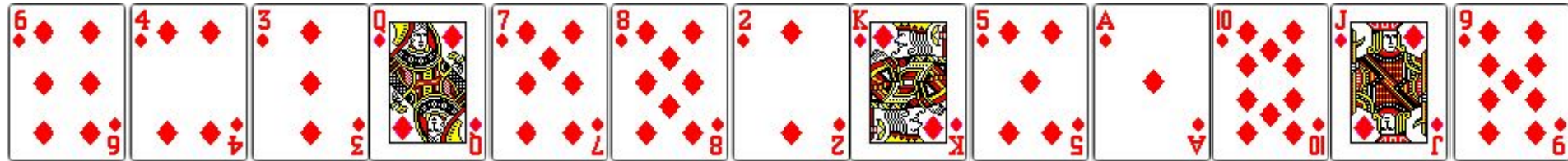
Start with the first value

For all the values in our problem:

Look at the next value

Is it lower? If so,

remember it as the lowest



Turning this into code...

```
def find_lowest_value( numbers ):  
  
    # Start with the first value  
    lowest = numbers[0]  
  
    # For all the values in our problem  
    for n in numbers:  
  
        # Is it lower? If so...  
        if n < lowest:  
  
            # Remember it as the lowest  
            lowest = n  
  
    # Return the lowest we found  
    return lowest
```

```
def find_lowest_value( numbers ):  
    # Start by treating the first value as the lowest  
    lowest = numbers[0]  
    # Inspect every value in our set of numbers  
    for n in numbers:  
        # If a particular value is lower than what we already found  
        if n < lowest:  
            # Save it as our new lowest value  
            lowest = n  
    # Return the lowest we found  
    return lowest
```

```
# The problem
```

```
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]  
results = []
```

```
# While unsorted values remain
```

```
while len(values) > 0:  
    # Find the lowest value  
    lowest = find_lowest_value( values )  
    # Add it to the sorted set  
    results.append( lowest )  
    # Remove it from the source set  
    values.remove( lowest )
```

Solved!

Returning to these steps

1. Start with a small human solvable version of the problem.
2. Keep working through the 4 computational thinking prompts until you devise a solution.
3. **Test your proposed solution works on larger, real-world versions of the problem.**
4. Adapt and repeat the process as necessary.

Does our solution work for decimals? negatives? strings?

Review: Computational thinking

1. Decomposition

Can I divide this into sub-problems?

2. Pattern recognition

Can I find repeating patterns?

3. Abstraction

Can generalise this to make an overall rule?

4. Algorithm design

Can I design the programming steps for any of this?

Some other suggestions for beginners

1. Just start (A blank screen can be overwhelming)
2. Don't start at the start (It's not a novel)
3. Start with something you know (Such as the UI)
4. Don't be afraid to Google (Prioritise forums such as stackoverflow)
5. Test & print a lot

Your turn!

Problem 1: Create an age calculator. Prompting the user for (1) their birthdate and (2) the current date, calculate their age.

Problem 2. Create a money change calculator. Given a set of possible coins available to the shop attendant, and an amount of change they must provide the customer, calculate how many of each coin they should give the customer.

For example if a country has 1cent, 5cent, 10cent, & 50cent coins and you need to provide 67cents of change, the clerk would give 1x50cent, 1x10cent, 1x5cent, and 2x1cent coins.