Unit 4: Computational thinking, problem solving, & programming

The Big Bang Theory - The Friendship Algorithm
https://www.youtube.com/watch?v=k0xgjUhEG3U

1. Computational thinking

There are several different models for conceptualising computational thinking. Ultimately their goal is to give you thinking tools to help you devise an algorithm for a problem. They are intended as thinking steps and prompt questions to guide you.

Google and many others advocate the following four prompts:

- Abstraction - Ignore the irrelevant to create a model that represents the problem
- Decomposition - Break a big problem into its constituent parts.
- Pattern recognition - Identify what pieces have in common, and what remains distinct
- Algorithm design - Formulate a series of repeatable steps that solve the problem

The IB course mandates you be aware of the following six prompts. In reality you can use whatever works best for you, but you should ensure you are at least aware of these terms as they do occasionally appear in a Paper 1 question.

- Thinking procedurally
- Thinking logically
- Thinking abstractly
- Thinking ahead
- Thinking concurrently
- Thinking recursively

1.1 Thinking procedurally

Thinking procedurally refers to turning the solution to your problem into a set of steps that can be followed. These steps should be reproducible, so that if correctly followed, will generate the solution every time. Each step may, if required, be divided into sub-procedures with their own lists of steps.

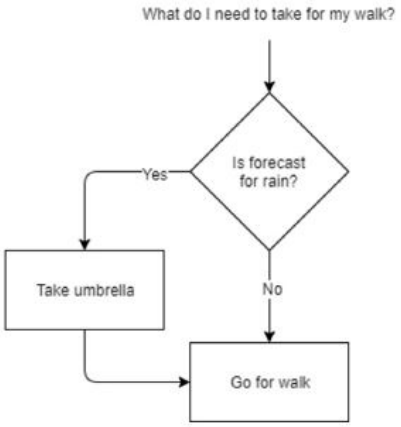
The intent of this thinking skill is to take a divide and conquer approach by taking the one overwhelming whole and break it down into manageable pieces. Additionally the sequencing (ordering) of steps is usually an important consideration.

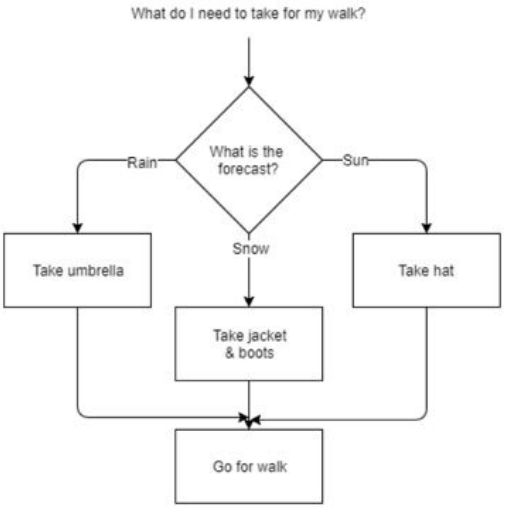
A common everyday example of thinking procedurally is a recipe for a meal.

1.2 Thinking logically

Thinking logically is all about making decisions, and formalising the conditions that will affect those decisions. There are generally three steps:

- Identify when decision making is required.
- Identify what is the decision that needs to be made.
- Identify the conditions which will form the basis of each decision.





1.3 Thinking abstractly

Being able to create a meaningful model, or way to represent, a real world “thing” in a way that contains everything that is relevant, and nothing that isn’t.

Real world examples:

- Maps – An abstraction containing pertinent information about a physical place. Different maps contain different information dependent on their purpose.
- Daily planners – An abstraction to represent hours, days, weeks and months in a simple manner that allows us to stay organised.
- Schools – People are abstracted into groups such as teachers, students, year 1, year 2 which makes them easier to organise.

When you take a real-world situation and are writing a program or algorithm for it, you are creating an abstraction: a way to represent that situation within the computer. As such you will make decisions about how that abstraction should be created such as what variables you need, what you will call them, what data type they will be, and how your algorithm will behave in response.

To succeed at thinking abstractly, you need to be able to take a problem and identify the parts that are relevant to your solution.

1.4 Thinking ahead

Thinking ahead is all about pre-planning and attempting to anticipate future needs.

What are the pre-conditions to solving the problem?

- What has to be in place, or known, in order to be able to solve the problem? ie: what are the inputs going to be? Prepare sample testing data to test the algorithm with

What are the post-conditions of the problem?

- What will be in place, or known, after the problem has been correctly solved? ie: what are the outputs going to be?

Anticipate exceptions to the rule

- What are the likely exceptions we are going to need to deal with? How do we want to handle them?
- Test anticipated exceptions to verify your responses work as intended.

Examples of thinking ahead in daily life:

- Shopping lists. You know you want to bake a cake, so you make sure you have all the necessary ingredients ahead of time.
- Preheating an oven. You know you're going to need it in a few minutes, start getting it warm now.
- Grabbing books/files from your locker and putting them into your bag for the lessons until the next break.
- Gantt charts.
- The cache on a computer is an example of thinking ahead.

1.5 Thinking concurrently

Concurrency dealings with multiple things happening at the same time. Sometimes a computing problem may involve multiple threads running simultaneously.

A non-computing example is the GANTT chart where multiple processes occur simultaneously. For example project managing the construction of a new house.

GPU's are a popular and powerful way to do multithreaded programming on computers.

Syllabus note: Students will not be expected to construct an algorithm related to concurrent processing (but you may be expected to be able to interpret/understand/recognise one presented to you)

1.6 Thinking recursively (HL)

See thinking recursively 😛

(Recursive thinking will be taught within unit 5 for HL students)

1.7 Computational thinking walk through

I actually prefer to use the four thinking skills approach. I think it is simpler and yet gets the same point across. A reminder, the four step computational thinking approach is...

1. Decomposition - Can I divide this into sub-problems?
2. Pattern recognition - Can I find repeating patterns?
3. Abstraction - Can I generalise this to make an overall rule?
4. Algorithm design - Can I design the programming steps for any of this?

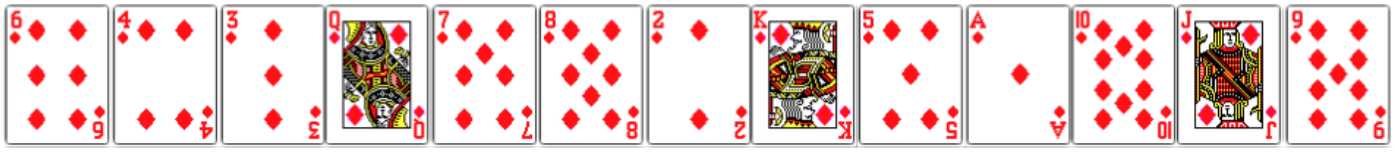
To illustrate what these concepts are, it is best to walk through an example. To get started, these are the steps you are going to follow:

1. Start with a small, human solvable version of the problem.
2. Look at your problem, looking to identify where you can use any of the 4 computational thinking prompts outlined above until you devise a solution.
3. Test your proposed solution on larger, real-world versions of the problem.
4. Adapt and repeat until your program is complete.

A good example is to consider how to create a program that will sort a list of numbers (such as the following) into ascending order.

16 23 63 36 67 60 42 15 24 85 90 6 18 46 32 17 61 88 47 64 62 19 80 18 24 60 47 52 48 21 12 70 95 20 35 84 48 66 24 75 38 55 539 24 77 34 4 12 35 45 33 5 92 32 89 93 40 21 65 35 63 82 92 66 92 52 44 36 41 51 27 32 32 47 26 92 98 31 2 11 90 64 99 68 55 73 24 35 94 4 80 44 48 98 2 67 24 25 1 39 18 93 49 90 6 81 100 53 29 78 479 74 63 11 44 21 100 40 51 39 62 12 39 77 27 73 20 27 48 115 40 22 43 78 62 56 58 12 69 79 10 43 49 48 82 31 74 96 56 89

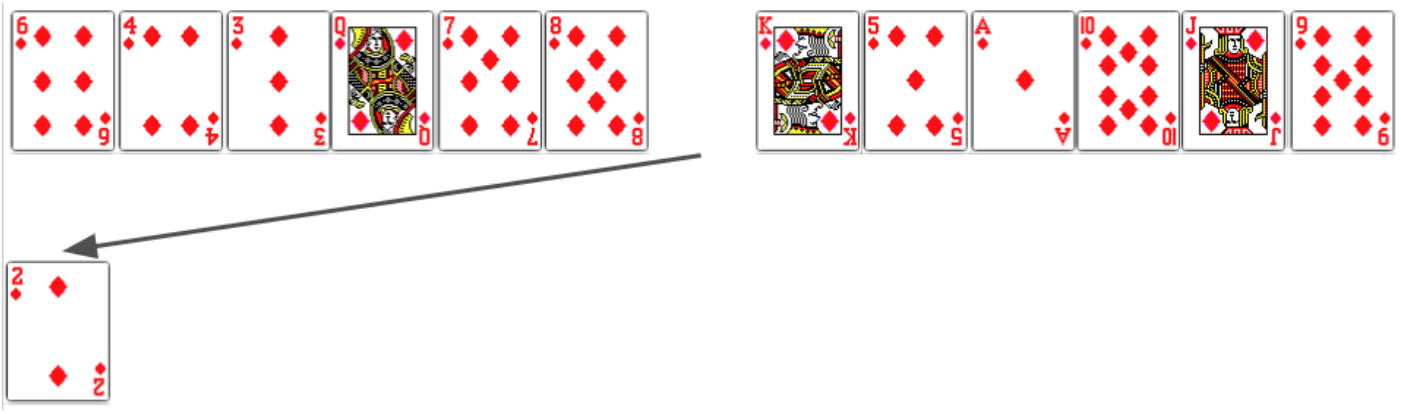
So, by applying step 1, let's create a small, human solvable version of the problem. This is important because an enormous infinitely large set of numbers is too overwhelming to think about. We do know how to sort playing cards though, so let's start with that.



If you had to write a set of instructions for someone who had never sorted cards before, what would you write? I suggest you attempt to write a set of instructions yourself before proceeding any further.

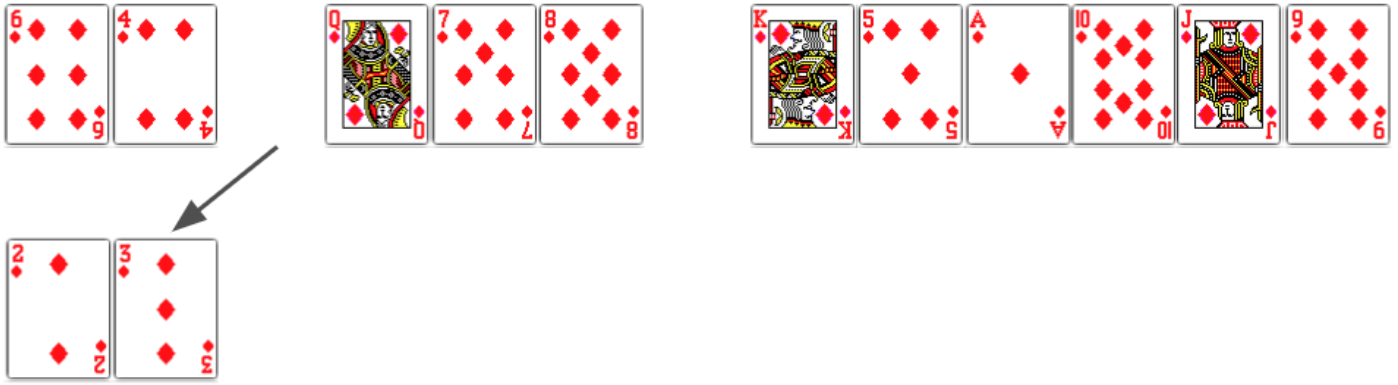
This is my attempt at documenting the human process of sorting cards... see how it compares to what you came up with...

First action...

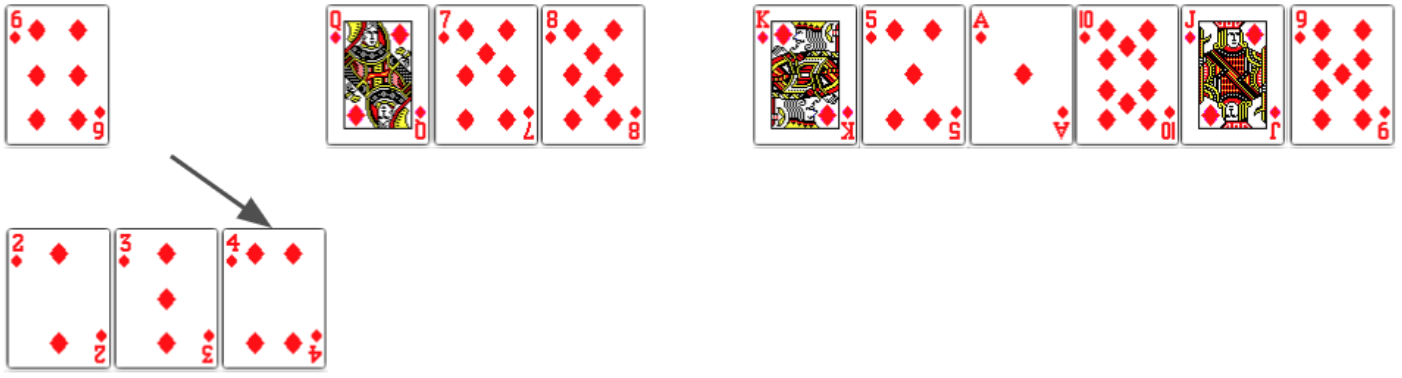


Second action..

.



Third action...



And so on...

So what did I actually do? I could write my instructions like this...

```
1. Find the 2, move it to the sorted set
2. Find the 3, move it to the sorted set
3. Find the 4, move it to the sorted set
...etc
```

The concept of **abstraction** requires removal of unnecessary complexity so we can create a general model or rule. This sequence of steps is currently very specific to our sample problem. Remember, the ultimate goal is to sort any set of numbers, not just a set of cards, so the procedure needs to be rewritten in generic terms. For instance, refer to the first card or last card, rather than the "2 of diamonds". After all, what happens when we want to sort 1'000'000 numbers of different sizes? We want a rule we can use in all scenarios if possible.

With this in mind, I rewrite my instructions to this...

```
1. Search through the values, find the lowest, move it to the sorted set
2. Search through the values, find the lowest, move it to the sorted set
3. Search through the values, find the lowest, move it to the sorted set
...etc
```

While this is an improvement, there is still more we need to do. Looking at our four concepts of computational thinking again, the one that should jump out at you is the idea of **pattern recognition**. There is clearly a repeating pattern in our set of instructions, so let's fix that. When using a repetition construct in programming, we generally need to ensure we are clear about the terminating scenario (when do we start and/or stop the repetition?), so let's make sure to specify that.

```
while unsorted values remain:
    search through the values, find the lowest, move it to the sorted set
```

If I'm observant, I can easily spot I've actually got three steps happening at once as well... so let's simplify this so that there is one clear step per line.

```
while unsorted values remain:
    search through the values
    find the lowest
    move it to the sorted set
```

We have now taken our original "overwhelming problem" and broken it down into three smaller sub-problems. This is **decomposition** at work! I can now try to solve each of these separately, continuing to decompose into smaller and smaller chunks until I end up with pieces that I know how to convert into programming code.

The process of actually documenting this into a set of accurate instructions is **algorithm design**.

It's time to start attempting to solve this with code. I will take my recipe of instructions and turn them into Python code comments. I will then use those comments to help me design my program.

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

# while unsorted values remain:
    # search through the values
    # find the lowest
    # move it to the sorted set
```

When I look at this, I realise I know how to do the loop, so I don't need to decompose that any further. It's a while loop that will run while the number of items in the `values` list is greater than zero...

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

# while unsorted values remain:
while len(values) > 0:
    # search through the values
    # find the lowest
    # move it to the sorted set
```

Now I think about it a little further, I realise what I don't have is a place to store my sorted set. I realise that's just another list, so I will add an empty list to my program. I also know that to add a value to a list uses the append function, so I'll add that straight away too.

```
# 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:
    # search through the values
    # find the lowest
    # move it to the sorted set
    result.append( lowest )
```

Notice that I'm not writing my code in a linear, top-down fashion? It's ok to jump around your code, add lines above where you are working if you realise you need something later. Designing a program beyond the absolutely trivial is never a top-down process. Don't think you have to know everything you have to write at the top of your program before you continue on. Start with what you know and move around as required.

So how to search through the values? I know a `for` loop will let me look at every value if I do something like...

```
for number in values:
   print(number)
```

And I know that `if` statements will let me compare values. So, it occurs to me that I could add a `for` loop to inspect every value, and if that particular number is lower than any I've seen so far, I could treat it as my lowest. Then as the program keeps looping, if it sees a new lower number, that can become the lowest, and so forth.

```
# 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:
    # search through the values
    lowest = ?????
    for number in values:
        if number < lowest:
            lowest = number
    # find the lowest
    # move it to the sorted set
    result.append( lowest )
```

Ok, I know that a variable has to be created before I can run the loop, but what should I set it to? If I gave `lowest = 0`, then it would be lower than all my numbers already so it wouldn't work. What if I did `lowest = 9999`? The problem is I want a general rule that could work for any set of numbers, so what if the lowest happens to be bigger than that?

The solution is to just start with the first value from the list. This is a bit of a programmer's instinct that will come with practice.

```
# 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:
    # search through the values, find the lowest
    lowest = values[0]
    for number in values:
        if number < lowest:
            lowest = number
    # move it to the sorted set
    result.append( lowest )
```

I notice that my `for` loop and `if` statement section take care of two jobs, so I've rearranged my comments to match.

There's one last problem however. I haven't actually "moved" a number into the sorted set, I "append" or "add" a number to the sorted set. Moving is actually a two step process. I need to remove it from the original list and then add it to the new list. It's ok if you didn't spot this mistake until you ran your program... That's the whole point of running tests on your code, to help spot what you are still missing.

Let's add that final step...

```
# 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:
    # search through the values, find the lowest
    lowest = values[0]
    for number in values:
        if number < lowest:
            lowest = number
    # move it to the sorted set
    values.remove( lowest )
    result.append( lowest )

print(result)
```

And, that's it. Our program is done.

I will preface that this is not a perfect sort algorithm at all. It is quite an inefficient algorithm. That said, it largely gets the job done and will suffice for our purposes of illustrating the computational thinking process.

Some final thoughts when facing a daunting programming problem:

1. **Just start**. A blank screen can be scary, so put something, anything, on screen. It doesn't matter if you end up deleting it all later, just start coding!
2. **Don't start at the start**. As discussed earlier, jump around your code. Start writing the bit you can figure out and go from there.
3. **Start with something you know**. This might be the user interface, perhaps some print statements or the input commands.
4. **Don't be afraid to Google**. When you do search online prioritise results from sites that are reputable for programmers such as stackoverflow.com.
5. **Test and print a lot**. You won't get it all in one hit. Add a thousand `print()` statements into your code to see what different variables are doing, or when different lines run.

Good luck and remember to have fun! Programming can be frustrating at times, but it is extremely rewarding as well when you persevere with a problem long enough to finally figure it out.

1.8 Computational thinking exercise

Your turn.

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.

Your finalised program should use an array of coin values, and accept user input of the amount of change needed to supply. An example run through might look like…

```
How much change do you have to give? 67
To return 67c, give the customer...
1 x 50c coins
0 x 20c coins
1 x 10c coins
1 x 5c coins
2 x 1c coins
```